Nauka programowania w języku Java

v. 0.7.1, 27-05-2020

(C) Przemysław Kruglej
2020

kontakt:

przemyslaw.kruglej@gmail.com

www:

https://kursjava.com

https://przemyslawkruglej.com

https://craftsmanshipof.software

Spis Treści

1 Rozdział I – Wstęp....11
1.1 Dla kogo przeznaczony jest kurs?....12
1.2 O kursie....13
1.2.1 Przykłady, odpowiedzi do pytań i rozwiązania zadań....13
1.3 O autorze....14
1.4 Formatowanie użyte w kursie....15
1.5 Dlaczego Java?....16
1.6 Jak tworzy się własne programy?....17
1.7 Instalacja Java i przygotowanie środowiska programistycznego....18
1.7.1 Katalog z przykładami....18
1.7.2 Edytor tekstu – Notepad++....19
1.7.3 Instalacja Java....20
1.7.3.1 Instalacja Java Development Kit....21
1.7.3.2 Ustawienie lokalizacji Java w zmiennej PATH....21
1.7.3.3 Sprawdzenie poprawności instalacji....23
1.8 Pierwszy program....25
1.8.1 Krok pierwszy – kod źródłowy – Witaj Świecie!....25
1.8.2 Krok drugi – kompilacja kodu Java....25
1.8.3 Krok trzeci – uruchamiamy nasz program....28
1.9 Analiza programu Witaj Świecie!....29
1.9.1 Pierwsza linia – definicja klasy Hello World....29
1.9.2 Druga linia – definicja metody main....29
1.9.3 Trzecia linia – wypisanie tekstu na ekran....30
1.10 Biblioteka standardowa Java....31
1.11 Podsumowanie....33
1.11.1 Podstawy....33
1.11.2 Kompilacja i uruchamianie kodu Java....33
1.11.3 Przykład utworzenia, kompilacji, i uruchomienia programu Java....33
1.11.4 Język Java....34
1.12 Pytania....35
1.13 Zadania....37
1.13.1 Wypisz imię....37
1.13.2 Brak końcowego znaku }....37
1.13.3 Zakładka do JavaDoc....37
2 Rodział II – Komentarze i formatowanie kodu....38
2.1 Komentarze....39
2.1.1 Rodzaje komentarzy....39
2.1.2 Zagnieżdżanie komentarzy....41
2.1.3 Komentarze w stringach....42
2.1.4 Kiedy używać komentarzy?....42
2.2 Formatowanie kodu i najlepsze praktyki....43
2.2.1 Najlepsze praktyki i konwencje....43
2.2.1.1 Jedna instrukcja na linię....44
2.2.1.2 Stosowanie wcięć....44
2.2.1.3 Nazwy-wielbłądy (czyli Camel Case)....45
2.2.1.4 Nazwy klas zaczynamy wielką literą....46
2.2.1.5 Nie trzymamy zakomentowanego kodu....46
2.2.1.6 Nazwy obiektów oraz komentarze po angielsku....46
2.3 Podsumowanie....47
2.3.1 Komentarze....47
2.3.2 Formatowanie kodu i najlepsze praktyki....47

2.4 Pytania....48
2.5 Zadania....48
2.5.1 Dopisz komentarze....48
2.5.2 Brak main....48
3 Rozdział III – Zmienne....49
3.1 Czym są zmienne?....50
3.1.1 Definiowanie zmiennych....50
3.1.2 Wypisywanie na ekran wartości zmiennych....51
3.1.3 Przykład – liczenie pola i obwodu koła....52
3.2 Reguły nazw w języku Java....53
3.2.1 Nadawanie nazw....55
3.3 Typy podstawowe....57
3.3.1 Literały....58
3.3.2 Typy całkowite i zmiennoprzecinkowe....58
3.3.3 Typ boolean....60
3.3.4 Typ char....60
3.4 Używanie zmiennych....61
3.5 Stałe....63
3.6 Podstawy zmiennych – zadania....64
3.6.1 Dodawanie liczb....64
3.6.2 Obwód trójkąta....64
3.6.3 Aktualna data....64
3.6.4 Liczba miesięcy w roku....64
3.6.5 Inicjały....64
3.7 Operatory w programowaniu....65
3.7.1 Operatory arytmetyczne....66
3.7.1.1 Operatory dodawania, odejmowania i mnożenia....66
3.7.1.2 Operatory dzielenia i reszty z dzielenia....66
3.7.1.3 Rzutowanie....67
3.7.1.4 Zmiana priorytetów operatorów za pomocą nawiasów....68
3.7.1.5 Plus jako operator konkatenacji....68
3.7.2 Operatory przypisania....69
3.7.2.1 Pomocnicze operatory przypisania....70
3.7.3 Operatory jednoargumentowe....72
3.8 Typ String i wczytywanie danych od użytkownika....74
3.8.1 Typ String....74
3.8.2 Wczytywanie danych od użytkownika....75
3.9 Podsumowanie....77
3.9.1 Zmienne....77
3.9.2 Nazwy....78
3.9.3 Typy....79
3.9.4 Stałe....79
3.9.5 Operatory....80
3.9.6 Typ String i wczytywanie danych od użytkownika....81
3.10 Pytania....83
3.11 Zadania....85
3.11.1 Obwód trójkąta z pobranych danych....85
3.11.2 Pobrane słowa w odwrotnej kolejności....85
3.11.3 Liczba znaków w słowie....85
3.11.4 Wynik rzeczywisty....85

3.11.5 Wielkie litery....85
3.11.6 Pole koła o podanym promieniu....85
4 Rozdział IV – Instrukcje warunkowe....86
4.1 Podstawy instrukcji warunkowych....87
4.1.1 Składnia instrukcji warunkowych....87
4.1.2 Instrukcje w instrukcjach warunkowych....89
4.1.3 Formatowanie instrukcji warunkowych....91
4.2 Operatory relacyjne....92
4.3 Typ boolean....94
4.4 Warunki instrukcji if....97
4.5 Operatory warunkowe i operator logiczny !....99
4.5.1 Operator logiczny !....100
4.6 Tablica prawdy operatorów warunkowych....102
4.7 Nawiasy i priorytety operatorów warunkowych....104
4.8 Short-circuit evaluation....107
4.9 Zagnieżdżanie instrukcji warunkowych....109
4.10 Bloki kodu i zakres zmiennych....112
4.10.1 Bloki kodu w instrukcjach warunkowych....114
4.11 Instrukcja switch....116
4.11.1 Użycie break w instrukcji switch....117
4.12 Trój-argumentowy operator logiczny....120
4.13 Podsumowanie....122
4.13.1 Instrukcje warunkowe if....122
4.13.2 Operatory relacyjne i typ boolean....123
4.13.3 Operatory warunkowe....124
4.13.4 Bloki kodu i zakres zmiennych....126
4.13.5 Instrukcja switch....126
4.13.6 Trój-argumentowy operator logiczny....127
4.14 Pytania....128
4.15 Zadania....132
4.15.1 Czy liczba podzielna przez trzy....132
4.15.2 Czy można zbudować trójkąt....132
4.15.3 Wypisz największą z dwóch liczb....132
4.15.4 Wypisz największą z trzech liczb....132
4.15.5 Zamień liczbę na nazwę miesiąca....132
4.15.6 Sprawdź imię....132
4.15.7 Czy pełnoletni....132
4.15.8 Czy rok przestępny....132
5 Rozdział V – Pętle....133
5.1 Czym są pętle?....134
5.2 Pętla while....135
5.2.1 Przykład: wypisywanie ciągu liczb....136
5.2.2 Przykład: wypisywanie gwiazdek....138
5.3 Pętla do...while....140
5.3.1 Przykład: dodawanie kolejnych liczb....141
5.4 Pętle nieskończone....144
5.5 Pętla for....145
5.5.1 Instrukcje inicjalizujące i kroku....147
5.6 Zakres (scope) zmiennych w pętlach....148
5.7 Instrukcje break oraz continue....149

5.7.1 Instrukcja break....149
5.7.2 Instrukcja continue....151
5.8 Ten sam kod z użyciem różnych pętli....153
5.8.1 Wyświetlanie kwadratu podanej liczby....153
5.8.2 Wypisanie kolejnych liczb parzystych....154
5.9 Zagnieżdżanie pętli....156
5.9.1 Użycie break i continue w pętlach zagnieżdżonych....158
5.10 Typ String i metoda charAt oraz pętle....160
5.10.1 Porównywanie znaków zwracanych przez charAt....162
5.11 Podsumowanie....164
5.12 Pytania....166
5.13 Zadania....167
5.13.1 While i liczby od 1 do 10....167
5.13.2 Policz silnię....167
5.13.3 Palindrom....167
5.13.4 Wypisz największą liczbę z podanych....167
5.13.5 Zagnieżdżone pętle....167
5.13.6 Kalkulator....167
5.13.7 Choinka....168
6 Rozdział VI – Tablice....169
6.1 Czym są tablice?....170
6.2 Definiowanie i używanie tablic....171
6.2.1 Odnoszenie się do elementów tablicy....173
6.2.2 Domyślne wartości w tablicach....174
6.2.3 Sprawdzanie liczby elementów tablicy....175
6.3 Użycie pętli z tablicami....177
6.4 Tablice wielowymiarowe....179
6.4.1 Inicjalizacja tablic wielowymiarowych....181
6.5 Pętla for-each....182
6.6 Porównywanie tablic i zmiana rozmiaru tablic....183
6.6.1 Zmiana rozmiaru tablicy....184
6.7 Podsumowanie....187
6.8 Pytania....190
6.9 Zadania....192
6.9.1 Co druga wartość tablicy....192
6.9.2 Największa liczba w tablicy....192
6.9.3 Słowa z tablicy wielkimi literami....192
6.9.4 Odwrotności słów w tablicy....192
6.9.5 Sortowanie liczb....192
6.9.6 Silnia liczb w tablicy....192
6.9.7 Porównaj tablice stringów....192
7 Rozdział VII – Metody....193
7.1 Czym są metody?....194
7.1.1 Do czego potrzebne są metody?....196
7.1.2 Podsumowanie podstaw metod....197
7.1.3 Pytania do podstaw metod....198
7.1.4 Zadania do podstaw metod....198
7.1.4.1 Metoda wypisująca Witajcie!....198
7.1.4.2 Metoda odejmująca dwie liczby....198
7.2 Zakres i wywoływanie metod, zmienne lokalne....199

7.2.1 Zakres metod....199
7.2.2 Wywoływanie metod....200
7.2.3 Zmienne lokalne....202
7.2.4 Czas życia zmiennych lokalnych....204
7.2.5 Podsumowanie zakresu i wywoływania metod, zmiennych lokalnych....205
7.2.6 Pytania do zakresu i wywoływania metod, zmiennych lokalnych....207
7.3 Wartości zwracane przez metody....208
7.3.1 Słowo kluczowe return....208
7.3.2 Używanie wartości zwracanych przez metody....211
7.3.2.1 Przypisanie wyniku metody do zmiennej....211
7.3.2.2 Rezultat metody jako argument innej metody....212
7.3.2.3 Metoda użyta w instrukcji warunkowej....213
7.3.2.4 Nieużywanie wyniku metody....214
7.3.3 Nieosiągalne ścieżki wykonania i ścieżki bez return....215
7.3.3.1 Nieosiągalny kod....216
7.3.4 Void, czyli niezwracanie wartości....217
7.3.5 Podsumowanie do zwracania wartości....218
7.3.6 Pytania do zwracania wartości....221
7.3.7 Zadania do zwracania wartości....222
7.3.7.1 Metoda podnosząca do sześcianu....222
7.3.7.2 Metoda wypisująca gwiazdki....222
7.4 Argumenty metod....223
7.4.1 Modyfikacja argumentów metod....224
7.5 Metody typu String....226
7.5.1 Przypomnienie metod length i charAt oraz indeksów znaków....226
7.5.2 Przykłady użycia metod typu String....227
7.5.2.1 toLowerCase, toUpperCase....227
7.5.2.2 endsWith, startsWith, contains....228
7.5.2.3 equals, equalsIgnoreCase....230
7.5.2.4 replace, substring, split....232
7.6 Dokumentowanie metod....235
7.7 Podsumowanie, pytania i zadania do argumentów metod i metod typu String....237
7.7.1 Argumenty metod....237
7.7.2 Metody typu String....238
7.7.3 Dokumentowanie metod....239
7.7.4 Pytania....240
7.7.5 Zadania....243
7.7.5.1 Metoda zwracająca ostatni znak....243
7.7.5.2 Metoda czyPalindrom....243
7.7.5.3 Metoda sumująca liczby w tablicy....243
7.7.5.4 Metoda zliczająca znak w stringu....243
7.8 Przeładowywanie metod....244
7.8.1 Typ zwracany przez metodę a przeładowywanie metod....246
7.8.2 Nazwy parametrów a przeładowywanie metod....247
7.8.3 Podsumowanie przeładowywania metod....248
7.8.4 Pytania do przeładowywania metod....249
7.8.5 Zadania do przeładowywania metod....250
7.8.5.1 Metoda porównująca swoje argumenty....250
8 Testowanie kodu....251
8.1 Wstęp do testowania....252

8.2 Pierwsze testy....253
8.2.1 Testy w osobnych metodach....254
8.2.2 Informowanie tylko o błędnym działaniu....254
8.2.3 Więcej przypadków testowych....255
8.2.4 Duplikacja kodu....257
8.3 Given .. when .. then....259
8.4 Dlaczego testy jednostkowe są ważne?....260
8.5 Testowalny kod....261
8.6 Co charakteryzuje dobre testy jednostkowe?....264
8.7 Przykłady testów jednostkowych....265
8.7.1 Wartość bezwzględna....265
8.7.2 Metoda sprawdzająca, czy tablica zawiera element....266
8.8 TDD – Test Driven Development....269
8.9 Podsumowanie....270
8.10 Pytania....272
8.11 Zadania....273
8.11.1 Testy czyParzysta....273
8.11.2 Testy sprawdzania znaku liczby....273
8.11.3 Testy zwracania indeksu szukanego elementu....273
9 Rodział IX – Klasy....274
9.1 Czym są klasy i do czego służą?....275
9.1.1 Przykład pierwszej klasy z polami....276
9.1.2 Użycie pierwszej klasy....277
9.1.3 Nazewnictwo klas....279
9.1.4 Jak tworzyć nowe instancje (obiekty) klas?....279
9.1.5 Metoda toString....280
9.1.6 Podsumowanie....283
9.1.7 Pytania....285
9.1.8 Zadania....286
9.1.8.1 Klasa Osoba....286
9.2 Różnice między typami prymitywnymi i typami referencyjnymi....287
9.2.1 Przechowywane wartości....287
9.2.2 Tworzenie....291
9.3 Modyfikatory dostępu....292
9.3.1 Modyfikatory dostępu public oraz private....292
9.3.2 Modyfikatory public oraz private – przykład – klasa Ksiazka....293
9.3.2.1 Użycie pól i metod publicznych klasy Ksiazka....294
9.3.2.2 Próba użycia pola i metody private klasy Ksiazka spoza tej klasy....294
9.3.2.3 Dostęp do prywatnych pól oraz metod w klasie Ksiazka....295
9.3.3 Kiedy stosować modyfikatory dostępu....297
9.4 Podsumowanie i pytania – typy prymitywne i referencyjne, modyfikatory dostępu....299
9.4.1 Typy prymitywne i referencyjne....299
9.4.2 Typy prymitywne i referencyjne – pytania....301
9.4.3 Modyfikatory dostępu....302
9.4.4 Modyfikatory dostępu – pytania....305
9.5 Pola klas....307
9.5.1 Pola klas a zmienne definiowane w metodach....307
9.5.1.1 Zapamiętywanie wartości....307
9.5.1.2 Inicjalizacja i domyślne wartości typów prymitywnych....308
9.5.1.3 Wartość null....310

9.5.1.4 Brak wymagania definicji przed użyciem....316
9.5.2 Gettery i settery oraz enkapsulacja....318
9.5.2.1 this....321
9.5.3 Konwencje dotyczące pisania setterów i getterów....323
9.5.4 Podsumowanie....324
9.5.4.1 Pola klas....324
9.5.4.2 Wartości domyślne i null....325
9.5.4.3 Gettery i settery oraz this....326
9.5.4.4 Konwencje dotyczące pisania setterów i getterów....328
9.5.5 Pytania....329
9.5.6 Zadania....331
9.5.6.1 Klasa Punkt....331
9.6 Konstruktory....332
9.6.1 Domyślny konstruktor....333
9.6.2 Przeładowanie konstruktora....335
9.6.3 Inicjalizacja pól finalnych w konstruktorach....338
9.6.4 Podsumowanie....342
9.6.5 Pytania....345
9.6.6 Zadania....348
9.6.6.1 Klasa Adres....348
9.6.6.2 Klasa Osoba z konstruktorem....348
9.7 Equals – porównywanie obiektów....349
9.7.1 Porównywanie wartości zmiennych....349
9.7.2 Porównywanie obiektów za pomocą equals....353
9.7.2.1 Sygnatura metody equals....354
9.7.2.2 Typ Object i krótko o dziedziczeniu....354
9.7.2.3 Implementacja metody equals w klasie Osoba....356
9.7.2.4 Kontrakt equals....365
9.7.2.5 Equals – przykład z tablicą....367
9.7.2.6 Krok po kroku – pisanie metody equals....372
9.7.3 Podsumowanie....374
9.7.4 Pytania....376
9.7.5 Zadania....377
9.7.5.1 Klasa Punkt z equals....377
9.7.5.2 Klasa Figura z equals....377
9.8 Referencje do obiektów....378
9.8.1 Przesyłanie i modyfikowanie obiektów w metodach....379
9.8.2 Współdzielenie obiektów....382
9.8.2.1 Osobne obiekty typu Punkt....386
9.8.2.2 Tworzenie kopii obiektów....387
9.8.2.3 Kiedy współdzielić obiekty?....388
9.8.3 Stałe referencje....388
9.8.4 Obiekty niemutowalne (immutable objects)....390
9.8.4.1 Zalety i wady obiektów niemutowalnych....395
9.8.4.2 Jak zapewnić niemutowalność obiektów....395
9.8.4.3 Niemutowalny typ String....396
9.8.5 Pamięć programu – stos i sterta....397
9.8.6 Czas życia obiektów utworzonych w metodach....399
9.8.7 Podsumowanie różnic typów prymitywnych i referencyjnych....400
9.8.8 Podsumowanie....401

9.8.9 Pytania....406
9.8.10 Zadania....408
9.8.10.1 Immutowalna Ksiazka i Biblioteka....408
9.9 Metody i pola statyczne....409
9.9.1 Pola statyczne....410
9.9.2 Metody statyczne....411
9.9.3 Dlaczego metoda main jest statyczna?....414
9.9.4 Kiedy stosować pola i metody statyczne?....415
9.9.5 Podsumowanie....417
9.9.6 Zadania....419
9.9.6.1 Klasa użyteczna Obliczenia....419
9.10 Pakiety i importowanie klas....420
9.10.1 Pakiety klas....420
9.10.1.1 Konwencja nazewnicza pakietów klas....424
9.10.2 Importowanie klas....425
9.10.2.1 Importy statyczne – static import....428
9.10.2.2 Lokalizacja klas – classpath....430
9.10.2.3 Kiedy nie trzeba stosować instrukcji import....432
9.10.3 Dostęp domyślny (default access) i klasy niepubliczne....433
9.10.3.1 Dostęp domyślny....433
9.10.3.2 Kiedy stosować dostęp domyślny?....435
9.10.3.3 Modyfikator dostępu protected....436
9.10.3.4 Niepubliczne klasy....436
9.10.4 Podsumowanie....437
9.10.4.1 Pakiety klas....437
9.10.4.2 Importowanie klas....439
9.10.4.3 Dostęp domyślny....442
9.10.5 Pytania....444
10 Rozdział X – Dziedziczenie i polimorfizm – rozdział w przygotowaniu....447
11 Rozdział XI – Wyjątki....448
11.1 Czym są wyjątki?....449
11.1.1 Stack trace....450
11.2 Przykład obługi sytuacji wyjątkowej....452
11.3 Korzystanie z try..catch..finally....453
11.3.1 Zakres zmiennych definiowanych w bloku try....454
11.3.2 Metoda getInt i obsługa wyjątków....456
11.4 Definiowanie i rzucanie wyjątków....458
11.4.1 Definiowanie własnych wyjątków....459
11.4.2 Przerywanie wykonania bloku kodu przez wyjątki....462
11.4.3 Rzucanie wyjątków i nieosiągalny kod....464
11.4.4 Rzucanie wyjątków a wartość zwracana z metody....464
11.4.5 Rzucanie wyjątków w try, catch, i finally....465
11.4.6 Ponowne rzucanie wyjątku....467
11.4.7 Treść wyjątku, stack trace i inne pola i metody....467
11.5 Wyjątki Checked & Unchecked....470
11.5.1 Jak rozpoznać wyjątki Checked i Unchecked?....471
11.5.2 Dlaczego istnieją dwa rodzaje wyjątków?....472
11.5.3 Sprawdzanie wyjątków rzucanych przez metodę....472
11.5.4 Jak sprawdzić rodzaj wyjątku....473
11.5.5 Błędy kompilacji podczas braku obsługi wyjątków Checked....474

11.5.6 Błąd Error....474
11.6 Łapanie wyjątków....476
11.6.1 Łapanie wyjątków za pomocą klasy bazowej....477
11.6.2 Łapanie kilku wyjątków za pomocą znaku |....478
11.6.3 Kolejność łapania wyjątków ma znaczenie....478
11.7 Pomijanie łapania wyjątków....480
11.7.1 Silent catch....481
11.7.2 Pomijanie try..catch w metodzie main....481
11.8 Try with resources....483
11.9 Wady i zalety wyjątków....487
11.9.1 Dlaczego używać mechanizmu wyjątków?....487
11.9.2 Wady wyjątków....487
11.10 Podsumowanie....489
11.10.1 Podstawy wyjątków....489
11.10.2 Łapanie wyjątków....489
11.10.3 Rodzaje wyjątków....491
11.10.4 Definiowanie i rzucanie wyjątków....492
11.10.5 Sprawdzanie rzucanych wyjątków i ich rodzaju....494
11.11 Pytania....495
11.12 Zadania....499
11.12.1 Silnia z obsługą ujemnych liczb....499
11.12.2 Klasa Adres z walidacją danych....499
11.12.3 Liczba znaków w pliku....499
11.12.4 Implementacja stosu....499

1 Rozdział I – Wstęp

W tym rozdziale:

- opowiemy sobie o założeniach kursu,
- poznamy używane w kursie formatowanie tekstu,
- dowiemy się, dlaczego warto używać języka Java,
- zainstalujemy Javę w naszym systemie,
- napiszemy, uruchomimy, oraz przeanalizujemy, nasz pierwszy program.

1.1 Dla kogo przeznaczony jest kurs?

Kurs przeznaczony jest dla osób, które nie miały do tej pory styczności z programowaniem lub miały w niewielkim stopniu, a chciałyby spróbować programowania w Javie lub myślą o sprawdzeniu swoich sił w zawodzie programisty. Wiedza zawarta w kursie nie wystarcza, jednakże, aby móc rozpocząć pracę jako programista, ale jest solidnym punktem startowym, po którym będzie można kontynuować naukę. Treść tego kursu to wymagane minimum, *pierwszy krok* na drodze do pracy jako programista.

Jeżeli chcesz nauczyć się programowania, aby mieć przyjemność z pisania własnych programów, to po przerobieniu kursu będziesz miał solidne ku temu podstawy. Jeżeli myślisz o pracy jako programista, to przygotuj się na długą i interesującą przygodę.

1.2 O kursie

Ten kurs ma na celu nauczenie Cię podstaw programowania, jak i podstaw języka Java. Po skończeniu kursu, będziesz miała/miał solidne podstawy, które będą wstępem do Twojej przygody z programowaniem.

Z założenia, w początkowej fazie kursu nie są wyjaśnione wszystkie koncepty związane z programowaniem w Javie. Na razie chcemy skupić się na podstawach, bez wdawania się w, na przykład, zawiłości klas, na naukę których przeznaczymy dużo czasu w dalszej części kursu.

Większość podstaw dotyczących programowania będzie podobna do innych języków takich jak Python czy C# – jeżeli nauczymy się, dla przykładu, instrukcji warunkowych i pętli w jednym języku, to będziemy potrafili z nich korzystać w innych językach (po ewentualnym uprzednim sprawdzeniu składni tych instrukcji).

Kurs został przygotowany dla osób korzystających z systemu Windows, ale nie będzie to przeszkadzać użytkownikom innych systemów – będą oni jedynie musieli w inny sposób zainstalować Javę i skonfigurować swoje środowisko.

Każdy rozdział kończy się częścią bądź wszystkimi z poniższych sekcji:

- ***Podsumowanie*** – jest to streszczenie rozdziału, zawierające "esencję" zagadnienia omawianego w danym rozdziale wraz z przydatnymi przykładami. Warto tutaj wrócić, jeżeli będziemy chcieli sobie szybko odświeżyć informacje o danym temacie.
- ***Pytania*** – sprawdzają zrozumienie i znajomość zagadnień omówionych w danym rozdziale.
- ***Zadania*** – do wykonania w ramach ćwiczeń.

Uwaga! Nauka programowania wiąże się z wymaganiem pisania bardzo dużej ilości kodu – samo czytanie kursu nie wystarczy, aby nauczyć się programować. Poza zadaniami opisanymi na końcu rozdziałów warto próbować pisać własne programy i przykłady. **Nic tak nie zwiększa zrozumienia i umiejetności programowania, jak pisanie kodu.**

Podziękowania dla Michała Justyńskiego za recenzję kursu.

1.2.1 Przykłady, odpowiedzi do pytań i rozwiązania zadań

Odpowiedzi do pytań i rozwiązania do zadań oraz ich omówienie znajdują się na stronie kursu:

https://kursjava.com/odpowiedzi-na-pytania-i-zadania

Dodatkowo, można je znaleźć także na GitHubie, razem z przykładowymi programami używanymi w tym kursie:

https://github.com/przemyslaw-kruglej/kursjava_przyklady

Odpowiedzi na pytania i rozwiązania do zadań dostępne są także w formacie pliku PDF, który można znaleźć na powyższej stronie GitHub.

1.3 O autorze

Programowaniem zainteresowałem się mając 14 lat i od tej pory stało się ono moją pasją. Jako programista pracowałem na kilkunastu projektach przez ostatnie 10 lat, pracując z różnymi technologiami i zdobywając cenne doświadczenie.

Kurs stworzyłem po to, by w przystępny sposób nauczać podstaw programowania. Lubię uczyć oraz dzielić się wiedzą. Chciałbym dzięki mojemu kursowi (oraz przyszłym kursom) ułatwić rozpoczęcie przygody z programowaniem osobom, które nie miały z nim do tej pory styczności.

Zależy mi, aby kurs był najwyższej jakości i abyś, Drogi Czytelniku, jak najwięcej się dzięki niemu nauczył. Dlatego, jeżeli cokolwiek będzie dla Ciebie niejasne bądź niezrozumiałe, nie wahaj się poinformować mnie o tym w komentarzu lub napisz maila na przemyslaw.kruglej@gmail.com. Będę wdzięczny za wszelkie uwagi i pomysły, jak usprawnić kurs.

1.4 Formatowanie użyte w kursie

W kursie używane będzie następujące formatowanie:

1. Treści akapitów:
   a) Istotne informacje zaznaczone będą w ten sposób.
   b) Techniczne pojęcia zaznaczone będą `w ten sposób`.
   c) Elementy tekstu związane z kodem Java będą formatowane zgodnie z przyjętym formatowanie składni języka Java, np. **`słowo kluczowe`**, `14`, `tekst`.
   d) Klawisze do skrótów klawiaturowych będą zaznaczone następująco: **`Enter`**.
2. Nazwy plików z kodami źródłowymi zawartymi w treści kursu będą zaznaczone w następujący sposób (plików należy szukać w katalogu danego rozdziału):

Plik: ToJestPrzykladowaNazwaPliku.java

3. Kod źródłowy będzie prezentowany w poniższy sposób. Zaznaczenie w formie `// (1)` będzie służyło jako odniesienie do wyjaśnienia kodu:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!"); // (1)
  }
}
```

4. Wynik uruchomienia programu będzie prezentowany jak poniżej. Tekst na białym tle to dane wprowadzone przez użytkownika z klawiatury:

```
Prosze podac promien kola:
10
Pole kola o promieniu 10 wynosi: 314.0
```

5. Stosowane będą poniższe ramki do zaznaczenia różnego rodzaju informacji:
   a) Najlepsze praktyki związane z programowaniem – konieczne, by jakość naszego kodu była jak najwyższa:

Przykład ramki z opisem różnych najlepszych praktyk związanych z programowaniem.

Warto się do nich stosować, ponieważ pomoże to w pisaniu kodu lepszej jakości.

   b) W ramkach, jak zaprezentowano poniżej, umieszczane będą informacje, na które trzeba zwrócić szczególną uwagę:

Pamiętaj, aby nie dzielić przez zero!

   c) Mniej istotne informacje, będą zaznaczone poniższą ramką:

To jest pierwszy rozdział.

   d) Dodatkowo, w jeszcze jeden, pokazany niżej sposób, zaznaczane będą ważne informacje:

Kluczowa informacja.

1.5 Dlaczego Java?

Java to obiektowy język programowania (*OOP – Object Oriented Programming*) stworzony w 1995 przez Sun Microsystems, kilka lat temu kupiony przez korporację Oracle. Java przez lata rozprzestrzeniła się na większość systemów operacyjnych i urządzeń i jest jednym z najbardziej popularnych języków programowania.

Ideą Javy jest to, że można jej używać na różnych platformach – laptopach, telefonach i innych urządzeniach oraz w różnych systemach, jak np. Windows, Linux, czy Android.

Java jest cały czas rozwijana i z wersji na wersję oferuję coraz więcej funkcjonalności – aktualna wersja to 12, która została wydana w marcu 2019 roku. Obecnie, zazwyczaj wymaganą wersją Java w ofertach pracy jest 1.8 – chociaż do kompilacji i uruchamiania programów napisanych w języku Java będziemy korzystać z najnowszej wersji Javy, to zakres tego kurs skupia się na wersji 1.8.

Zmiany wprowadzone do najnowszych wersji Javy skupiają się na bardziej zaawansowanych aspektach programowania w tym języku i nie są wymagane w trakcie nauki podstaw programowania w Javie. Do nauki podstaw, zakres funkcjonalności oferowany przez Javę 1.8 w zupełności wystarcza.

Warto być programistą Java, ponieważ zapotrzebowanie na programistów Java na rynku jest bardzo duże i są to dobrze płatne oferty pracy. Jednakże, sama Java nie wystarczy – wymagania znajomości m.in. popularnego frameworku Spring, SQLa, Gita, czy też podstaw technologii frontendowych, można znaleźć w bardzo wielu ofertach pracy dla Javowców. Ten kurs skupia się na podstawach programowania w Javie i tematyka wykorzystania innych technologii nie jest w nim poruszana.

Spring jest bardzo dobrze udokumentowany, a dodatkowo, na oficjalnej stronie, można znaleźć bardzo wiele darmowych tutoriali pokazujących, jak używać Springa: https://spring.io/guides

SQL, czyli Structured Query Language, to język służący do pobierania i manipulowania danymi w bazie danych.

Git to system kontroli wersji, który pozwala na wersjonowanie kodu źródłowego.

Frontend to ta część aplikacji, którą widzi użytkownik. Backend to to, co dzieje się pod spodem – tutaj właśnie znajduje zastosowanie Java.

1.6 Jak tworzy się własne programy?

Za chwilę napiszemy nasz pierwszy programy w Javie.

Najpierw jednak spójrzmy jak (w uproszczeniu) w ogóle tworzy się własne programy:

1. My, jako programiści, piszemy kod źródłowy programu w zrozumiałym dla nas języku – na przykład w Javie.
2. Komputer nie rozumie napisanego przez nas kodu – nasz kod musi:
   - zostać *przeształcony* na format rozumiany przez komputer

   lub

   - zostać *zinterpretowany* i wykonany przez inny program.

Proces przekształcania kodu zrozumiałego dla programisty na kod zrozumiały przez komputer nazywamy *kompilacją*, a programy, którę kompilację wykonują, *kompilatorami*.

Przykładem języka, którego kod źródłowy jest kompilowany do *kodu maszynowego*, zrozumiałego przez komputer, jest język C++, prekursor takich języków jak Java czy C#. Program skompilowany w systemie Windows nie będzie działał w systemie Linux.

Z kolei przykładem języka interpretowanego jest język Ruby.

Język Java leży pośrodku obu powyższych sposobów – programista Java:

a) pisze kod źródłowy w języku Java,

b) kompiluje swój kod źródłowy do *kodu pośredniego*, nazywanego *bytecodem*,

c) uruchomia w programie o nazwie ***Maszyna Wirtualna Java*** (**JVM**) swój skompilowany kod, który Maszyna Wirtualna Java zinterpretuje i wykona.

Aby móc kompilować programy napisane w Javie (punkt drugi powyżej), potrzebujemy programu o nazwie *javac* (**java** **c**ompiler). Z kolei, by uruchamiać programy napisane w Javie, potrzebujemy programu o nazwie *java* (jest to wspomniana wyżej Maszyna Wirtualna Java – JVM).

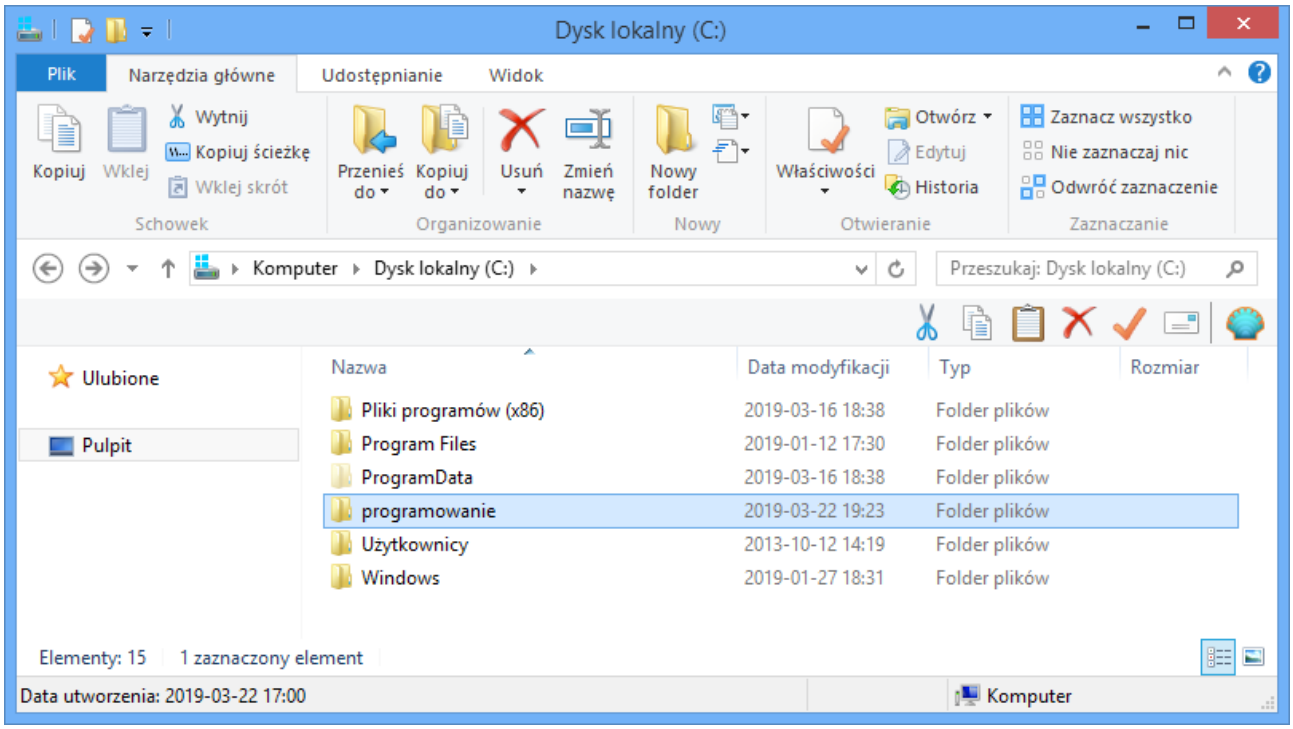
Za chwilę przejdziemy przez każdy z trzech powyższych punktów, ale najpierw przygotujemy nasze środowisko programistyczne.

1.7 Instalacja Java i przygotowanie środowiska programistycznego

Zanim będziemy mogli kompilować i uruchamiać programy napisane w języku Java, musimy przygotować nasze środowisko programistyczne.

1.7.1 Katalog z przykładami

W trakcie kursu będziemy pisali wiele programów. Stwórz na dysku swojego komputera katalog, w którym będziesz je przechowywać. Katalog może się nazywać, na przykład, *programowanie*, a utworzyć go możesz, na przykład, na dysku C:



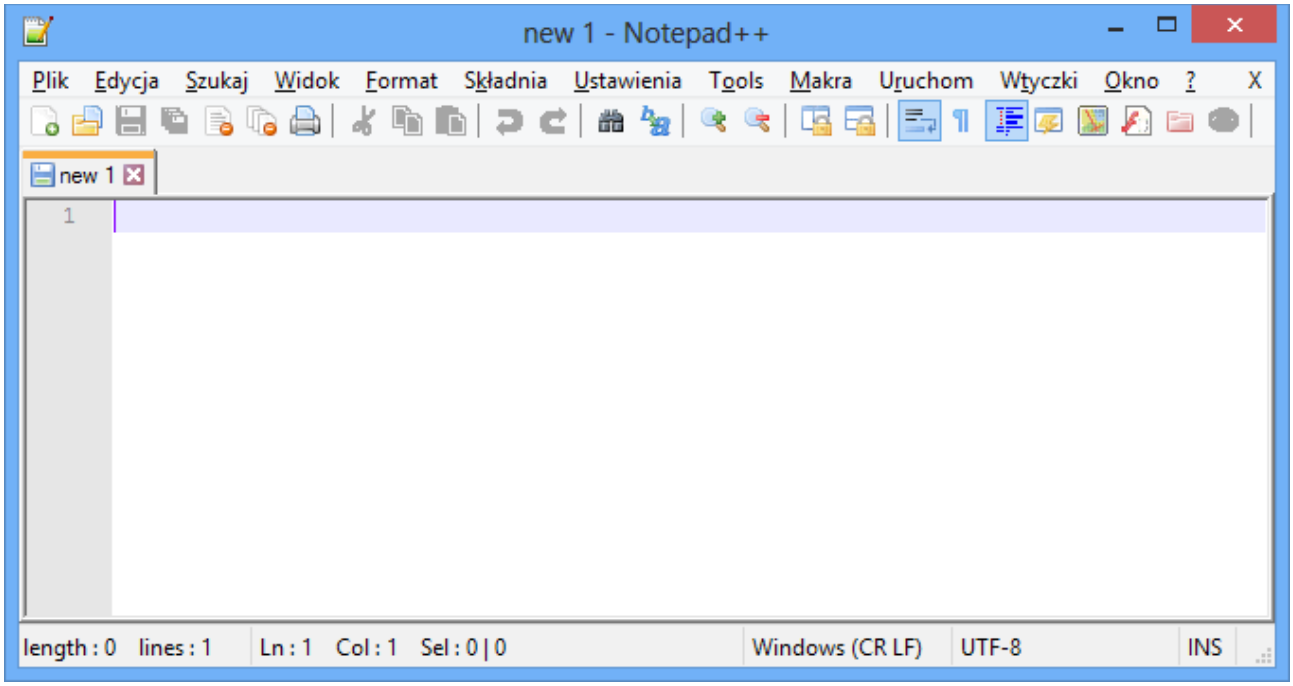
1.7.2 Edytor tekstu – Notepad++

Następnie, zainstalujemy program *Notepad++*, w którym będziemy pisać przez pewien czas nasze kody źródłowe.

Program Notepad++ jest darmowym, rozbudowanym notatnikiem, który m. in. koloruje składnię kodu, co ułatwia jego pisanie i czytanie. Pobierz i zainstaluj program Notepad++ ze strony:

http://notepad-plus-plus.org/download

Po zainstalowaniu i uruchomieniu Notepad++, powinnaś/powinienieś zobaczyć okno programu podobne do tego zaprezentowanego na obrazku poniżej:

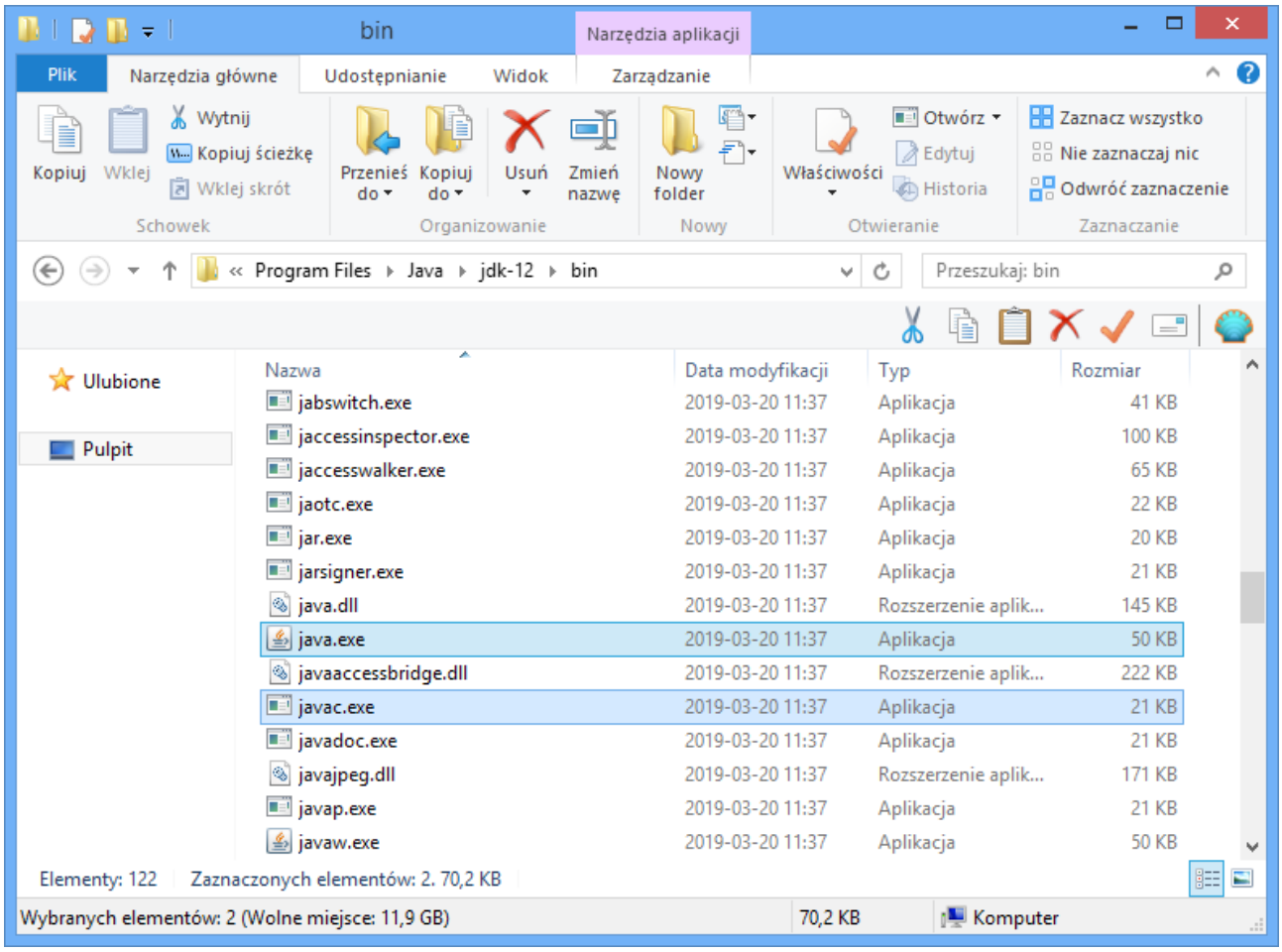


Docelowo, w dalszej części kursu, będziemy korzystać z IntelliJ IDEA, darmowego, wygodnego i profesjonalnego narzędzia do tworzenia aplikacji w języku Java. Jednakże,na początku, Notepad++ w zupełności nam wystarczy.

1.7.3 Instalacja Java

Aby móc tworzyć programy w Javie, musimy zainstalować JDK – *Java Development Kit*. Jest to po prostu zestaw programów dla programistów, które umożliwiają kompilowanie kodu źródłowego Java, jak i uruchamianie go.

Instalacja JDK zapewni nam dostęp do, zarówno, programu *javac* (kompilator Java), jak i do programu *java* (Maszyna Wirtualna Java). Na poniższym obrazku widzimy zainstalowany Java Development Kit – w podkatalogu `bin` widnieją zaznaczone: kompilator Java (program `javac.exe`) oraz Maszyna Wirtualna Java (program `java.exe`):



Nasze programy będziemy początkowo kompilować i uruchamiać z *linii poleceń* (zwanej także *linią komend*) systemu Windows. Linia poleceń to narzędzie, które pozwala na wywoływanie komend i programów. Aby móc korzystać w linii komend z kompilatora Java oraz Maszyny Wirtualnej Java, musimy wskazać systemowi Windows ich lokalizację. Aby to osiągnąć, po instalacji JDK, ustawimy w specjalnej zmiennej systemowej o nazwie `PATH` lokalizację Javy.

W moim artykule Podstawy linii poleceń dla użytkowników systemu Windows znajdziesz informacje o korzystaniu z linii poleceń, w tym: uruchamianie linii poleceń i wywoływanie komend, przydatne komendy, skróty, oraz ustawienia. Dowiesz się także czym są standardowe wejście i wyjście, przekierowanie komend, i wiele więcej.

Programiści korzystają w pracy z lini poleceń na porządku dziennym.

1.7.3.1 Instalacja Java Development Kit

JDK – *Java Development Kit* – możemy pobrać z następującej strony (należy zaznaczyć akceptację regulaminu firmy Oracle, a następnie wybrać plik `.exe` dla Windows):

https://www.oracle.com/technetwork/java/javase/downloads/jdk12-downloads-5295953.html

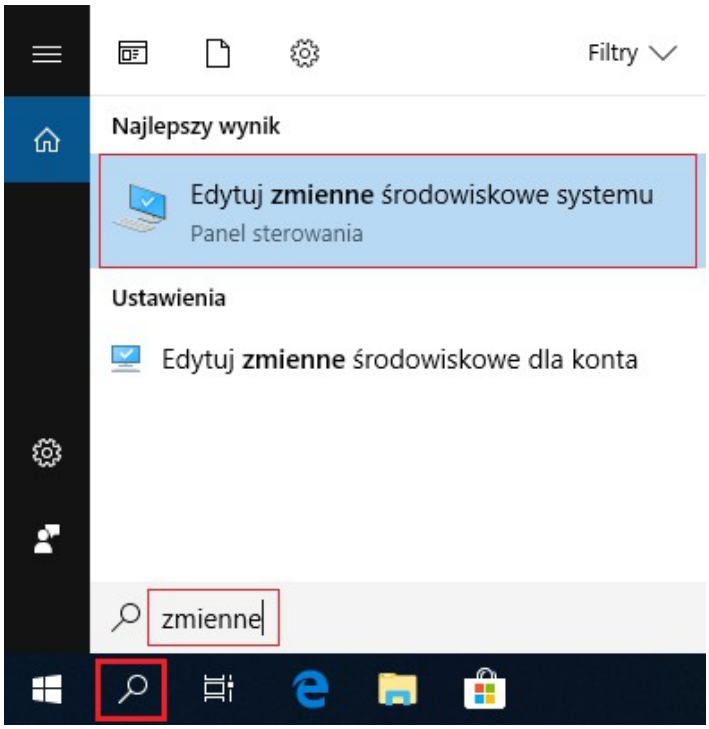
Instalacja ta zapewni nam dostęp do, zarówno, `javac` (kompilatora Java) oraz do `java` (Maszyny Wirtualnej Java). Podczas instalacji, możemy wybrać domyślne ustawienia.

1.7.3.2 Ustawienie lokalizacji Java w zmiennej PATH

Po instalacji, powinniśmy do zmiennej systemowej `PATH` dodać ścieżkę do katalogu `bin`, znajdującego się w katalogu, w którym zainstalowaliśmy Javę, abyśmy mogli korzystać z Javy z linii poleceń.

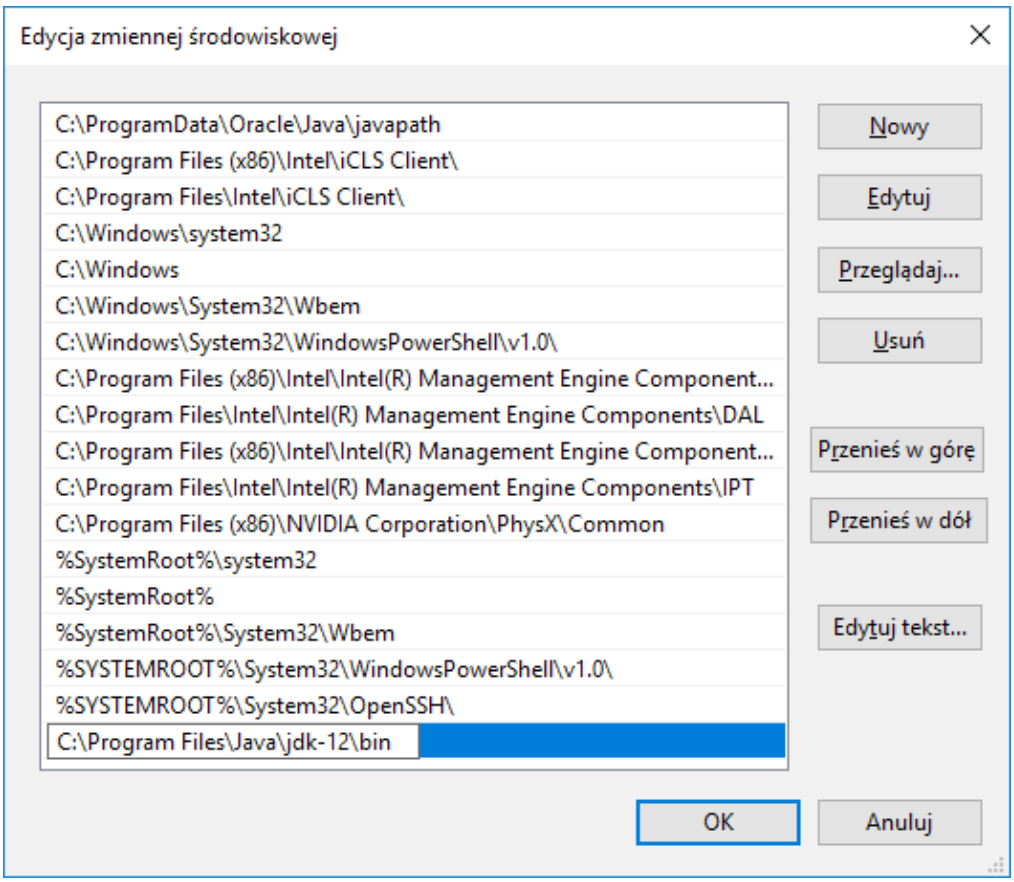
Dostęp do modyfikacji zmiennych systemowych różni się nieco w różnych wersjach systemu Windows. Aby zmodyfikować zmienne systemowe w systemie Windows 10:

1. Kliknij na ikonę lupy na dolnym pasku na ekranie.
2. Wpisz wyraz *zmienne* w polu tekstowym.
3. Kliknij na wyszukaną pozycję, jak pokazano na poniższym obrazku:



4. Następnie, w oknie, które się pojawi, kliknij na przycisk *Zmienne środowiskowe...*
5. W sekcji *Zmienne systemowe* znajdź zmienną o nazwie `PATH` (wielkość liter może się różnić). Wybierz ją, klikając na nią, a następnie kliknij przycisk *Edytuj...*
6. W kolejnym oknie, które się pojawi, kliknij na przycisk *Nowy* i wpisz w polu, które się podświetli, ścieżkę do katalogu `bin` zainstalowanej wcześniej Javy. Dla przykładu, u mnie Java zainstalowała się w katalogu `C:\Program Files\Java\jdk-12`, więc wpisałem w pole następującą wartość:

   C:\Program Files\Java\jdk-12\bin



7. Na koniec kliknij na przyciski *OK*, aby pozamykać wszystkie otwarte okna.

Poniższa informacja przeznaczona jest dla użytkowników systemów Windows wcześniejszych niz Windows 10.

Edycja zmiennych systemowych różni się nieco w poprzednich wersjach systemu Windows. Najpierw należy otworzyć Panel Sterowania – kliknij na Start (lub ikonę Windows w menu na dole ekranu), następnie Panel Sterowania (lub Control Panel), a potem System (zależnie od wersji systemu Windows, dostep do Panelu Sterowania może się różnić).

Następnie, kliknij Zaawansowane ustawienia systemu i Zmienne środowiskowe. Znajdź zmienną `PATH` *i kliknij przycisk Edytuj.*

W polu "Wartość zmiennej" dodaj na początku ścieżkę do katalogu `bin` *zainstalowanej wcześniej Javy* ***z dodatkiem średnika na końcu****. Dla przykładu, u mnie Java zainstalowała się w katalogu* `C:\Program Files\Java\jdk-12`*, więc dodałem na początek wartości zmiennej* `PATH` *taki oto wpis:*

C:\Program Files\Java\jdk-12\bin;

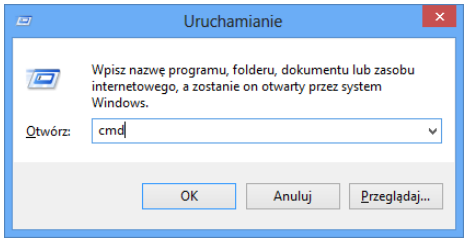
Zwróć uwagę, że do ścieżki dodałem (jak wspomniano wyżej), `\bin` *oraz* ***średnik****. Początek wartości zmiennej* `PATH` *z mojego systemu wygląda teraz następująco:*

C:\Program Files\Java\jdk-12\bin;*C:\apps\groovy-2.4.5\bin;C:\Program Files\Git*

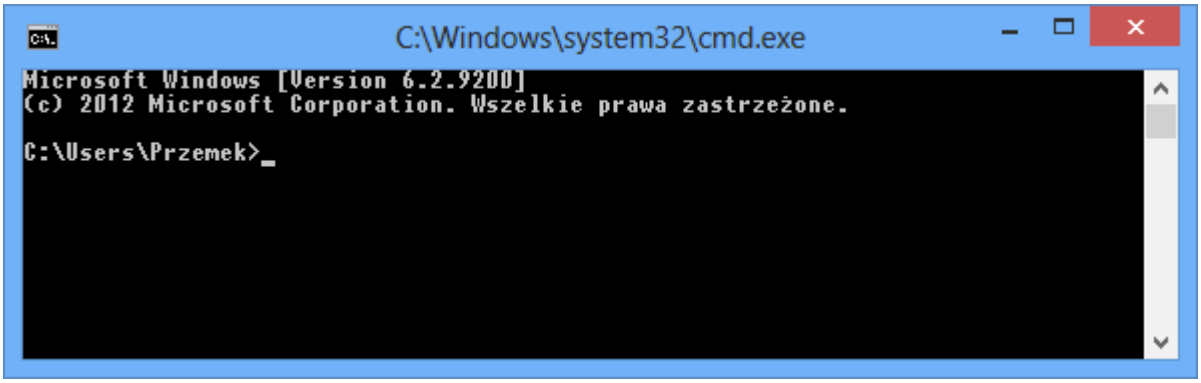
1.7.3.3 Sprawdzenie poprawności instalacji

Sprawdźmy teraz, czy kompilator Java oraz Maszyna Wirtualna Java są dostępne z linii poleceń:

1. Aby uruchomić linię poleceń systemu Windows, użyj skrótu klawiaturowego – wciśnij na klawiaturze klawisz **`Windows`** + **`r`** (klawisz **`Windows`** to ten z logo systemu Windows). Następnie wpisz `cmd` i wciśnij **`Enter`**:



2. Okno linii poleceń powinno pojawić się na ekranie – można w nim teraz wpisywać komendy – migający symbol _ (znak podkreślenia) to "symbol zachęty", oznaczający, że linia poleceń oczekuje na komendy od użytkownika:



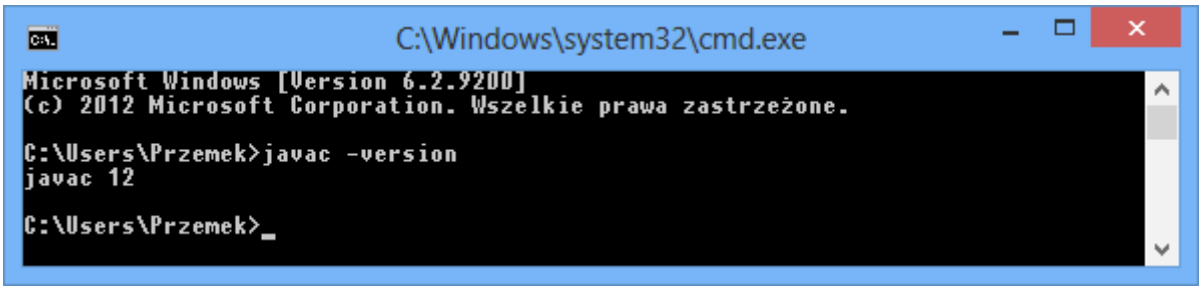
3. Wpisz z klawiatury poniższą komendą do linii poleceń i naciśnij **`Enter`**:

```
javac -version
```

Komenda ta to po prostu zlecenie linii poleceń wywołania programu o nazwie "javac", czyli kompilatora języka Java, z jednym argumentem – ciągiem znaków `-version` (*myślnik version*). Argument ten zostanie przekazany do programu javac. W przypadku tego konkretnego argumentu, kompilator Java w wyniku działania wypisze na ekran swoją wersję i zakończy działanie, ponownie umożliwiając nam na wpisywanie komend do linii poleceń.

> *Program* javac *jest dostępny z linii poleceń, ponieważ, po instalacji Java Development Kit, ustawiliśmy w zmiennej* `PATH` *lokalizację programu* javac *(znajduje się on w katalogu* `C:\Program Files\Java\jdk-12\bin`*, jak już wiemy z poprzedniego rozdziału).*

4. Efekt powinien być podobny, jak pokazany poniżej (wersja kompilatora może się różnić):



Jeżeli zobaczymy na ekranie komunikat:

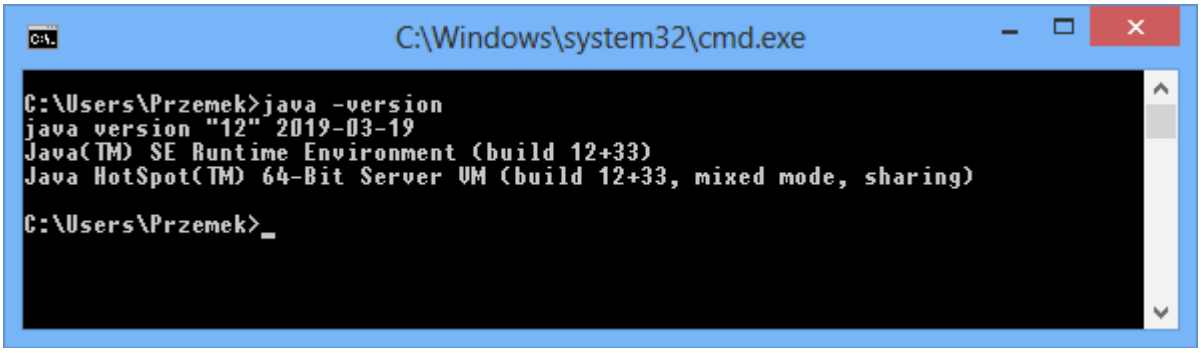
```
'javac' is not recognized as an internal or external command,
operable program or batch file.
```

będzie to oznaczać, iż mamy niepoprawnie ustawioną zmienną środowiskową. Zamknij okno linii poleceń, ustaw zmienną systemową i ponownie uruchom linię poleceń (w poprzednio otwartym oknie linii poleceń zmiany zmiennej `PATH` nie będą widoczne!).

5. Sprawdźmy jeszcze, czy mamy dostęp do drugiego programu – Maszyny Wirtualnej Java. W tym samym oknie linii poleceń, wykonaj kolejną komendę:

```
java -version
```

Podobnie, jak w przypadku programu javac, program java (czyli Maszyna Wirtualna Java), także wypisuje swoją wersję, jeżeli otrzyma argument `-version`. Wynik powienien być zbliżony do poniższego:



1.8 Pierwszy program

Najpierw napiszemy nasz kod źródłowy, skompilujemy i uruchomimy go, a następnie przeanalizujemy.

1.8.1 Krok pierwszy – kod źródłowy – Witaj Świecie!

W świecie programistów przyjęło się, że naukę nowego języka programowania zaczyna się od programu wypisującego na ekran tekst **"Witaj Świecie!"**. Poniższy kod Java realizuje właśnie to zadanie:

Plik: HelloWorld.java

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

Kod źródłowy programów napisanych w Javie zapisujemy w plikach z rozszerzeniem `.java`. Uruchamiamy zainstalowany wcześniej program Notepad++ i wpisujemy powyższy kod źródłowy, a następnie zapisujemy go w utworzonym wcześniej katalogu *programowanie* (lub innym, jeżeli użyłaś/użyłeś innej nazwy), jako plik o nazwie i rozszerzeniu `HelloWorld.java`. Skrót klawiaturowy do zapisywania plików to **`Ctrl`** + **`s`**.

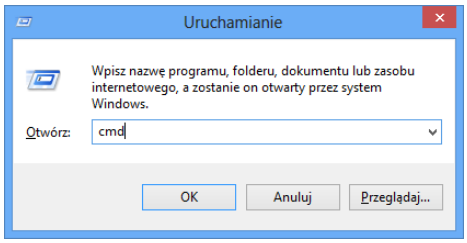
> Plik musi nazywać się dokładnie **HelloWorld.java**.
>
> Należy także zwrócić uwagę na wielkość liter użytych w kodzie źródłowym – jeżeli napiszemy np. `Public` zamiast `public`, kod nie zadziała!

Czas na skompilowanie naszego kodu.

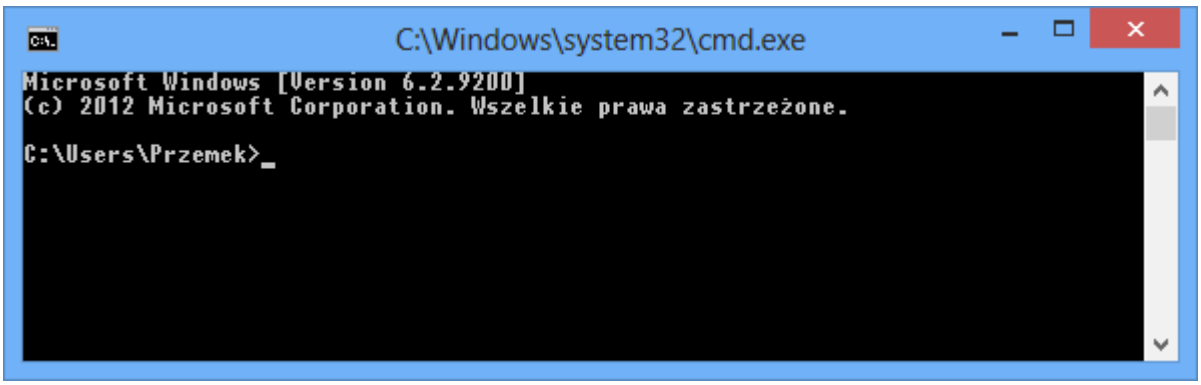
1.8.2 Krok drugi – kompilacja kodu Java

Mając zainstalowany Java Development Kit, możemy skorzystać z kompilatora *javac*, by skompilować nasz pierwszy program. Kompilację wykonamy z linii poleceń systemu Windows:

1. Jeżeli zamknęliśmy wcześniej linię poleceń, to ponownie, za pomocą skrótu na klawiaturze **`Windows`** + **`r`**, uruchamiamy ją, wpisując w oknie, które się pojawi, `cmd`, oraz wciskając **`Enter`**:

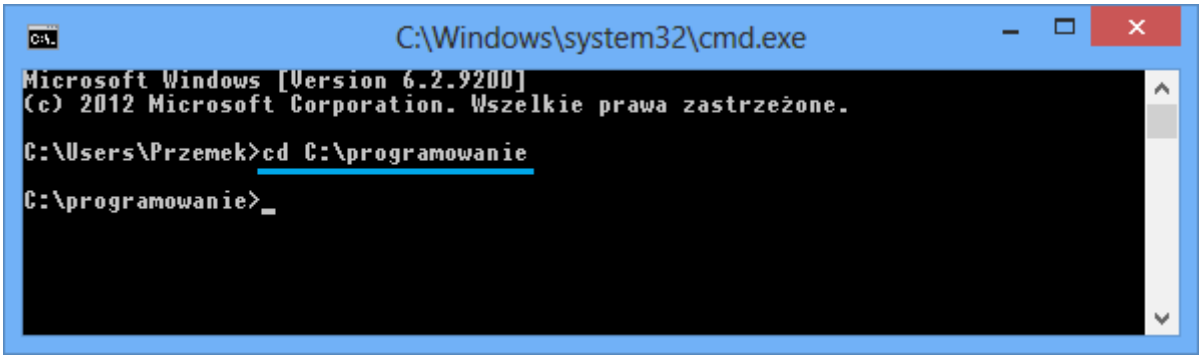


2. W oknie linii poleceń będziemy chcieli uruchomić kompilator języka Java, podając mu jako argument nazwę pliku z kodem źródłowym, który chcemy skompilować. Pliki naszych programów zapisujemy w katalogu `C:\programowanie`, a linia poleceń zazwyczaj ma inny aktualny *katalog roboczy* – widzimy to po uruchomieniu linii poleceń:

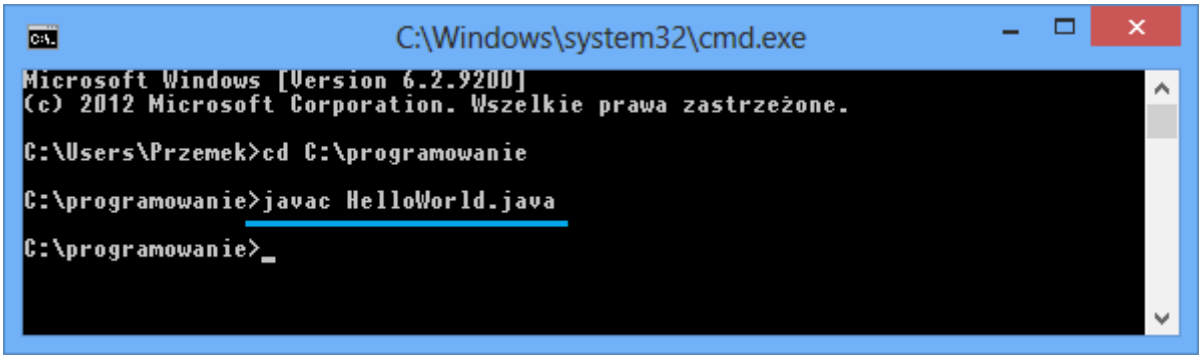


Katalog roboczy zaraz po uruchomieniu linii poleceń, w przypadku mojego systemu Windows, to `C:\Users\Przemek`. Potrzebujemy przejść do katalogu `C:\programowanie`, ponieważ to tam znajduje się nasz program `HelloWorld.java`. Do przechodzenia pomiędzy katalogami w linii poleceń używamy komendy `cd` (**c**hange **d**irectory), której podajemy jako argument lokalizację na dysku, do której chcemy przejść – w naszym przypadku będzie to `C:\programowanie`.

3. Używając komendy `cd`, przechodzimy do katalogu, w którym zapisaliśmy nasz plik z kodem źródłowym `HelloWorld.java` – wpisujemy komendę `cd`, po której powinna nastąpić ścieżka na dysku, do której chcemy przejść, i zatwierdzamy klawiszem **`Enter`**. Jak widać poniżej, w kolejnej linii okna linii poleceń katalog roboczy zmienił się z `C:\Users\Przemek` na `C:\programowanie`:



4. Możemy teraz skompilować nasz program za pomocą komendy `javac HelloWorld.java`



Na razie nie zamykajmy okna linii poleceń.

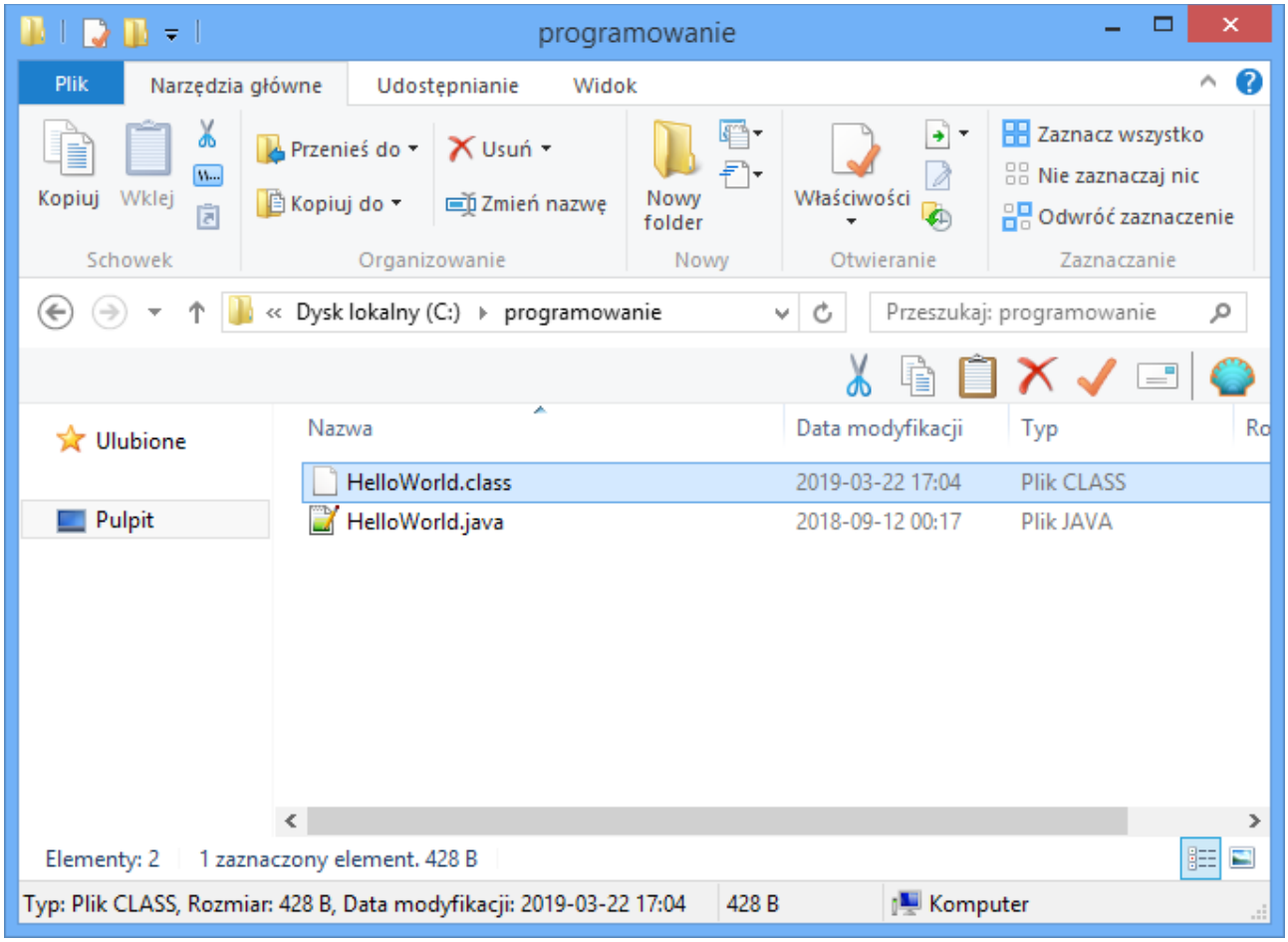
Jeżeli wszystko przebiegło bez problemów, nic nie zostanie wypisane na ekran. Co w zasadzie stało się po wykonaniu powyższej komendy?

Używając linii poleceń systemu Windows, uruchomiliśmy program – kompilator języka Java – o nazwie *javac*. Był on dla nas dostępny z linii poleceń, ponieważ po zainstalowaniu Java

Development Kit (w skład którego wchodzi kompilator *javac*), dodaliśmy do zmiennej systemowej `PATH` lokalizację, gdzie Java Development Kit zostało zainstalowane. Tym samym wskazaliśmy systemowi Windows, gdzie ma szukać m. in. programu o nazwie "javac".

Windows, wiedząc już, gdzie znaleźć program *javac*, uruchomił go, przekazując mu jako argument nazwę naszego pliku z kodem źródłowym Java – `HelloWorld.java`.

Kompilator następnie odczytał ten plik i skompilował go do bytecodu. Jeżeli spojrzymy do katalogu, gdzie zapisaliśmy wcześniej nasz plik z kodem źródłowym, to zobaczymy tam nowy plik – `HelloWorld.class` – jest to plik utworzony przez kompilator *javac* – plik ten zawiera nasz kod źródłowy Java skompilowany do bytecode'u:



Plik z bytecodem sam w sobie nie może zostać uruchomiony – system Windows nie wiedziałby, co z nim zrobić – system operacyjny nie rozumie bytecode'u. Jest jednak coś, co jest w stanie ten bytecode zinterpretować i wykonać – Maszyna Wirtualna Javy.

> Dzięki temu, że kompilujemy kod źródłowy Java do bytecode'u, może on zostać uruchomiony na różnych platformach, urządzeniach i systemach operacyjnych, o ile jest tam dostepna Maszyna Wirtualna Java – jest to jedna z idei i zalet języka Java.

1.8.3 Krok trzeci – uruchamiamy nasz program

Uruchamiamy nasz program za pomocą komendy `java HelloWorld`:

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\Przemek>cd C:\programowanie

C:\programowanie>javac HelloWorld.java

C:\programowanie>java HelloWorld
Witaj Swiecie!

C:\programowanie>_
```

Ponownie uruchomiliśmy z lini poleceń program – tym razem, Maszynę Wirtualną Java (JVM) – która nazywa się po prostu "java".

Maszyna Wirtualna Java otrzymała jako argument informację, co ma zinterpretować i wykonać – w tym przypadku był to nasz skompilowany kod `HelloWorld`. Wynikiem działania naszego programu jest wypisanie na ekran komunikatu "Witaj Swiecie!". Po wypisaniu tekstu, program zakończył działanie – nic więcej nie miał do zrobienia.

> Zauważ, że tym razem nie podaliśmy rozszerzenia pliku `.class` w argumencie. O powodzie porozmawiamy w dalszej części kursu. Na razie zapamiętaj, że **kompilator javac** oczekuje pliku z rozszerzeniem **.java**, natomiast **Maszyna Wirtualna Java** (**program java**, uruchomiony powyżej) **oczekuje nazwy bez rozszerzenia**.

Gratulacje! Napisaliśmy, skompilowaliśmy, i uruchomiliśmy, nasz pierwszy program!

1.9 Analiza programu Witaj Świecie!

Omówimy teraz nasz pierwszy program.

Nie będziemy zagłębiać się we wszystkie szczegóły – zajmiemy się nimi w dalszej części kursu. Pomimo tego, że nasz pierwszy program wydaje się prosty, jest w nim wiele istotnych elementów. Nie należy się przejmować, jeżeli nie wszystko będzie od razu zrozumiałe – dopiero co zaczynamy naszą przygodę z programowaniem!

Spójrzmy jeszcze raz na nasz kod źródłowy:

Plik: HelloWorld.java

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

1.9.1 Pierwsza linia – definicja klasy Hello World

Nasz program rozpoczyna się od zdefiniowania *klasy* `HelloWorld`.

Kod źródłowy Java zawsze znajduje się wewnątrz jednej badź wielu klas. *Klasa* opakowuje pewną funkcjonalność. Ciało klasy zawarte jest pomiędzy dwoma nawiasami klamrowymi { }.

Klasy służą m. in. do opisu obiektów ze świata za pomocą zawartych w nich pól oraz metod, które reprezentują cechy oraz funkcjonalności różnych zjawisk i obiektów. Przykładem mogłaby być np. klasa *Osoba*, która miałaby takie pola jak *imie*, *nazwisko*, *wiek*, oraz metody takie jak *idzDoPracy*, *zjedzSniadanie*, *czytajKsiazke*. O klasach, metodach i polach klas będziemy uczyć się w jednym z kolejnych rozdziałów i przeznaczymy na ich naukę dużo czasu. Zwróćmy jednak jeszcze uwagę na dwie sprawy, które będą dla nas teraz istotne:

1. Plik z kodem źródłowym musi nazywać się tak samo, jak nazwa publicznej klasy, która jest w nim zawarta – dlatego plik z naszym kodem źródłowym musieliśmy zapisać jako `HelloWorld.java`.
2. Nazwę klasy zawsze zaczynamy z wielkiej litery – nie jest to wymóg, lecz ogólnie przyjęta konwencja.

1.9.2 Druga linia – definicja metody main

W drugiej linii znajduje się początek *metody* `main`. Metoda to otoczka kodu odpowiedzialnego za wykonanie pewnego zadania.

Wszystkie programy pisane w języku Java mają specjalną metodę o nazwie `main`, od której rozpoczynają one swoje działanie.

Metoda `main` jest szczególna, ponieważ zawarty w niej kod definiuje, co robi nasz program. Czyli, w ramach uruchomienia naszego programu za pomocą komendy:

```
java HelloWorld
```

zadzieje się to, co umieścimy w ciele metody `main`. Ciało metody zdefiniowane jest jako instrukcje zawarte pomiędzy nawiasami klamrowymi { }.

1.9.3 Trzecia linia – wypisanie tekstu na ekran

Metoda `main` naszego pierwszego programu składa się z jednej *instrukcji*:

```
System.out.println("Witaj Swiecie!");
```

Instrukcja to wykonanie pewnej akcji – powyższa linijka odpowiedzialna jest za wypisanie na ekran napisu `Witaj Swiecie!`.

Odbywa się to poprzez użycie metody `println` jednej z klas z biblioteki standardowej, którą udostępnia nam Java. Po nazwie metody, w nawiasach, otoczony cudzysłowami, zawarty jest komunikat do wypisania, a na końcu instrukcji widnieje średnik. Kod nie zadziałałby, gdybyśmy zapomnieli o cudzysłowach, nawiasach, bądź średniku.

> W kodzie źródłowym ujęcie tekstu w cudzysłowy oznacza, iż dany fragment kodu jest ***łańcuchem tekstowym***, zwanym także ***stringiem*** (string of characters) bądź ***literałem tekstowym***.

Zwróćmy jeszcze uwagę na średnik kończący linię:

> Instrukcje w języku Java oddzielane są od siebie średnikami.

Spójrzmy na poniższy przykład wypisujący dwie linie w oknie konsoli:

Plik: DoubleMessage.java

```
public class DoubleMessage {
  public static void main(String[] args) {
    System.out.println("Pierwsza linia.")
    System.out.println("Druga linia.");
  }
}
```

Na końcu trzeciej linii brakuje średnika – sprawdźmy, co stanie się, gdy spróbujemy skompilować powyższy kod – próba skompilowania tego kodu zakończy się błędem, a kompilator wypisze na ekran przydatny komunikat błędu:

```
> javac DoubleMessage.java
DoubleMessage.java:3: error: ';' expected
    System.out.println("Pierwsza linia.")
                                         ^
1 error
```

Kompilator poinformował nas, że na końcu wskazanej linii spodziewał się średnika, którego nie znalazł. Co ważne, plik `DoubleMessage.class` (czyli skompilowana wersja naszego kodu) nie został wygenerowany, ponieważ w trakcie kompilacji kompilator napotkał powyższy błąd.

Jeżeli nasz kod będzie niepoprawnie zapisany (np. zabraknie średnika na końcu instrukcji), to proces kompilacji zakończy się niepowodzeniem, o czym powiadomi nas kompilator, wskazując problematyczne miejsca w naszym kodzie. Dzięki temu łatwo można zlokalizować i naprawić usterki w kodzie.

> *Warto czytać komunikaty błędów kompilatora – pomaga to w znajdywaniu i rozwiązywaniu błędów w naszym kodzie.*

1.10 Biblioteka standardowa Java

Zanim przejdziemy do podsumowania, opowiemy sobie jeszcze o bibliotece standardowej Java i dokumentacji Java.

Gdy wypisujemy na ekran tekst za pomocą:

```
System.out.println("Witaj Swiecie!");
```

to korzystamy z funkcjonalności klasy `System`, która wchodzi w skład *biblioteki standardowej Java*.

Biblioteka standardowa Java to zbiór tysięcy klas użytecznych, które dostepne są dla programistów – zostały one napisane przez twórców języka Java. Pisząc programy w Javie będziesz bardzo często korzystał z klas biblioteki standardowej.

W związku z ogromną liczbą klas, które są dla nas dostępne, nie sposób spamiętać ich wszystkich. Na szczeście, są one dokładnie udokumentowane, a dokumentacja ta, nazywana *JavaDoc*, dostępna jest online pod adresem:

https://docs.oracle.com/javase/8/docs/api/

> Podczas programowania w Javie będziesz często zaglądać zarówno do dokumentacji biblioteki standardowej Java, jak i do innych dokumentacji. Jest to nieodłączna część pracy programisty – codziennie wiele czasu przeznacza się na czytanie dokumentacji i szukaniu w internecie rozwiązań na różne problemy, które napotyka się w pracy.

W kolejnych rozdziałach tego kursu poznamy część z dostępnych klas biblioteki standardowej.

Przykład dokumentacji klasy `System` z dokumentacji JavaDoc:

System (Java Platform SE 8)

https://docs.oracle.com/ja

Szukaj

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.lang

Class System

java.lang.Object
    java.lang.System

```
public final class System
extends Object
```

The System class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the System class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

Since:

JDK1.0

Dokumentacja JavaDoc została wygenerowana na podstawie komentarzy dokumentujących z kodu biblioteki standardowej. Komentarze są tematem kolejnego rozdziału.

1.11 Podsumowanie

1.11.1 Podstawy

- Aktualna wersja Java to 12 – my skupiamy się na wersji 1.8, ponieważ wersja ta w zupełności wystarcza podczas nauki podstaw.
- Kompilacja to proces zamiany kodu źródłowego, zrozumiałego dla ludzi, na kod zrozumiały przez komputer.

1.11.2 Kompilacja i uruchamianie kodu Java

- Aby móc kompilować i uruchamiać programy napisane w Javie, potrzebujemy JDK – Java Development Kit.
- Lokalizację JDK należy dodać do zmiennej systemowej `PATH`, by kompilator i Maszyna Wirtualna Java (programy *javac* i *java*) były dostępne z linii poleceń.
- Linia poleceń to narzędzie, które pozwala na wywoływanie komend i aplikacji, na przykład pozwala na uruchomienie kompilatora języka Java. Aby uruchomić linię komend, możemy skorzystać ze skrótu klawiaturowego **Windows** + **r**, wpisać `cmd` i nacisnąć klawisz **Enter**.
- *Katalog roboczy* w linii poleceń to katalog, w którym wykonujemy komendy. Aby go zmienić, używamy komendy `cd` (**c**hange **d**irectory) – argumentem wywołania tej komendy powinien być katalog, do którego chcemy przejść, np. `cd C:\programowanie`
- JDK zawiera programy: kompilator Java o nazwie `javac` oraz Maszynę Wirtualną Java (JVM) o nazwie `java`.
- Kompilator `javac` służy do kompilacji kodu Java do kodu pośredniego, nazywanego *bytecodem* – w wyniku kompilacji pliku o nazwie `MojaNazwaKlasy.java` powstaje plik z bytecodem o nazwie `MojaNazwaKlasy.class`
- Bytecode rozumiany jest przez Maszynę Wirtualną Java, która go interpretuje i wykonuje.
- W przypadku napotkania błędów w składni w kodzie źródłowym, kompilator wypisuje komunikat o błędzie ze wskazaniem problematycznego miejsca. Problem trzeba rozwiązać, poprawiając kod, po czym ponownie spróbować skompilować kod.

1.11.3 Przykład utworzenia, kompilacji, i uruchomienia programu Java

1. Tworzymy plik o nazwie i rozszerzeniu `HelloWorld.java` i wpisujemy do niego:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

2. Uruchamiamy linię poleceń (skrót **Windows** + **r**, wpisujemy `cmd`, wciskamy **Enter**) i używając komendy `cd` (**c**hange **d**irectory) przechodzimy do katalogu, w którym zapisaliśmy plik z punktu 1.
3. Kompilujemy nasz kod źródłowy komendą:

```
javac HelloWorld.java
```

4. Uruchamiamy nasz program komendą:

```
java HelloWorld
```

5. Jeżeli wszystko przebiegło bez problemów, powinniśmy zobaczyć na ekranie napis `Witaj Swiecie!`

1.11.4 Język Java

- Kod źródłowy Java zapisujemy w plikach z rozszerzeniem `.java`, na przykład `HelloWorld.java`
- Kod Java piszemy w *klasach*, które są opakowaniem pewnej funkcjonalności. Kod w klasie zawarty jest między nawiasami klamrowymi { }.
- Plik z kodem źródłowym musi nazywać się tak, jak zawarta w nim klasa publiczna z dopisanym rozszerzeniem `.java`
- Metoda `main` to punkt wejścia do naszego programu i definiuje, co nasz program robi.
- Metody mają za zadanie wykonanie pewnego określonego zadania. Ciałem metody nazywamy kod zawarty pomiędzy nawiasami klamrowymi { }.
- Za pomocą wywołania metody `println` z biblioteki standardowej możemy wypisywać na ekran tekst.
- Tekst w kodzie źródłowym zaznacza się poprzez ujęcie go w cudzysłowy " ". Nazywa się on **stringiem** (bądź łańcuchem znaków/literałem tekstowym).
- Wszystkie instrukcje w Javie muszą kończyć się średnikiem.

Poniższy przykład z dopiskami obrazuje powyższe zagadnienia:

Nazwa pliku: HelloWorld.java

```
public class HelloWorld { ← początek klasy, nazwa pliku taka jak nazwa klasy
  public static void main(String[] args) { ← początek metody main
    System.out.println("Witaj Swiecie!"); ← wypisanie na ekran tekstu
                                          ^ średnik kończący instrukcję
  } ← koniec metody main
} ← koniec klasy
```

- Java udostępnia nam, programistom, bibliotekę standardową, która zawiera wiele przydatnych klas, w tym klasę `System`, której używamy m. in. do wypisywania tekstu na ekran.
- Biblioteka standardowa Java jest dobrze udokumentowana – dokumentacja ta nazywa się JavaDoc i znajdziemy ją pod adresem:

  https://docs.oracle.com/javase/8/docs/api/

1.12 Pytania

1. Czym jest proces kompilacji?
2. Jak nazywa się kompilator Java i jak się go używa?
3. Co powstaje w wyniku kompilacji kodu Java?
4. Jak uruchomić kod Java?
5. Czym różni się kompilator języka Java od Maszyny Wirtualnej Java?
6. Skąd wziąć kompilator Java i Maszynę Wirtualną Java?
7. Jak powinien nazywać się plik z poniższym kodem Java?

```
public class Zagadka {
  public static void main(String[] args) {
    System.out.println("Cegla wazy kilo i pol cegly - ile wazy cegla?");
  }
}
```

8. Mając w katalogu jeden plik, o nazwie `TestowyProgram.java`, z kodem źródłowym Java, czy poniższa komenda jest poprawna?

```
javac TestowyProgram
```

9. Mając plik o nazwie `TestowyProgram.class` ze skompilowanym kodem źródłowym Java, czy poniższa komenda jest poprawna?

```
java TestowyProgram.class
```

10. Jakie jest specjalne znaczenie metody `main`?
11. Jak wypisać na ekran tekst `Witajcie!` ?
12. Jaki będzie efekt próby kompilacji każdego z poniższych programów?

```
public class PrzykladPierwszy {
  public static void main(String[] args)
    System.out.println("Pierwsza zagadka.");
  }
}
```

```
public class PrzykladDrugi {
  public static void main(String[] args) {
    System.out.println("Druga zagadka.")
  }
}
```

```
public class PrzykladTrzeci {
  public static void main(String[] args) {
  }
}
```

```
public class PrzykladCzwarty {
  public static void main(String[] args) {
    System.out.println('Czwarta zagadka.');
  }
}
```

```
public class PrzykladPiaty {
  public static void main(String[] args) {
    System.out.println(”Piata zagadka.”);
  }
}
```

13. Które z poniższych stwierdzeń jest prawdziwe?
    1. System Windows rozumie bytecode.
    2. Program `javac` jest potrzebny do uruchomienia skompilowanego kodu Java.
    3. Stringi (łańcuchy tekstowe) powinny być zawarte w apostrofach.
    4. Ciało metody zawarte jest pomiędzy nawiasami: ( )
    5. Jeżeli kompilator napotka problemy w naszym kodzie, to nie wygeneruje pliku z rozszerzeniem `.class` z naszym skompilowanym kodem.

1.13 Zadania

1.13.1 Wypisz imię

Napisz program, który wypisze na ekran tekst "Czesc", po ktorym nastąpi Twoje imię (nie używaj polskich znaków).

1.13.2 Brak końcowego znaku }

Napisz program, który wypisze na ekran powitanie "Witajcie!". Następnie:

1. Skompiluj i uruchom program.
2. Po sprawdzeniu, że program działa, usuń końcowy znak } z kodu źródłowego.
3. Ponownie spróbuj skompilować kod źródłowy. Jaki będzie efekt?
4. Spróbuj uruchomić swój program. Jaki będzie efekt?

1.13.3 Zakładka do JavaDoc

Stwórz w swojej przeglądarce internetowej zakładkę do dokumentacji biblioteki standardowej Java – będziesz z niej często korzystał/korzystała:

https://docs.oracle.com/javase/8/docs/api/

2 Rodział II – Komentarze i formatowanie kodu

W tym rozdziale:

- dowiesz się czym są komentarze w kodzie źródłowym i kiedy je stosować,
- poznasz dostępne rodzaje komentarzy,
- nauczysz się, jak należy formatować kod, by był czytelny,
- dowiesz się, czym są konwencje w programowaniu oraz poznasz kilka z nich.

2.1 Komentarze

Komentarze są fragmentami kodu źródłowego, które są całkowicie pomijane przez kompilator. Możemy w nich umieścić dowolny tekst. Komentarze piszemy zarówno dla innych programistów, jak i dla nas samych.

Komentarze:

- są dobrym sposobem na wyjaśnianie, dlaczego coś zostało zrobione tak, a nie inaczej,
- są wykorzystywane do wytłumaczenia bardziej skomplikowanych fragmentów kodu bądź opisu zastosowanych tricków,
- są także używane do dokumentowania metod.

Komentarze mogą zostać użyte w każdej częsci programu.

2.1.1 Rodzaje komentarzy

W Javie istnieją trzy rodzaje komentarzy:

- jednolinijkowe – zaczynają się od dwóch znaków slash `//` i kończą się wraz z końcem linii,
- wielolinijkowe – zaczynają się od znaków `/*` oraz kończą znakami `*/` – nie mogą być w sobie zagnieżdżane, tzn. `/* komentarz /* zagniezdzony */ */` jest niepoprawny i spodowuje zakończenie kompilacji programu błędem,
- wielolinijkowe-dokumentacyjne – zaczynają się od znaków `/**` oraz kończą znakami `*/` Również nie mogą być w sobie zagnieżdżane. Istnieją narzędzia, które z odpowiednich komentarzy dokumentacyjnych w kodzie są w stanie wygenerować dokumentację. Jest to popularne rozwiązanie w języku Java – w ten właśnie sposób generowana jest dokumentacja JavaDoc, o której opowiedzieliśmy sobie w jednym z poprzednich rozdziałów.

Spójrzmy na nasz pierwszy program z dodanymi komentarzami każego z trzech powyższych rodzajów:

Nazwa pliku: HelloWorldZKomentarzami.java

```
/*
 To jest komentarz wielolinijkowy.
 Ponizej zaczyna sie klasa HelloWorldZKomentarzami
*/
public class HelloWorldZKomentarzami {
  /**
   * To jest komentarz dokumentujacy, jak dziala metoda main.
   */
  public static void main(String[] args) {
    // to jest komentarz jednolinijkowy - ponizej wypisujemy tekst
    System.out.println("Witaj Swiecie!"); // na koncu linii moze byc komentarz

    /*
       Ponizsza linia kodu nie zostanie wykonana, poniewaz
       zostala zakomentowana.
    */
    // System.out.println("Witajcie!");

    System.out.println(/* kolejny komentarz */ "Witajcie ponownie!");
  }
}
```

Ten program spowoduje wypisanie na ekran następujących komunikatów:

```
Witaj Swiecie!
Witajcie ponownie!
```

Powyższy przykład zawiera wiele komentarzy – po kolei:

- trzy komentarze jednolinijkowe:

```
// to jest komentarz jednolinijkowy - ponizej wypisujemy tekst
System.out.println("Witaj Swiecie!"); // na koncu linii moze byc komentarz

// System.out.println("Witajcie!");
```

Komentarze jednolinijkowe kończą się wraz z końcem linii – nic nie stoi na przeszkodzie, by umieścić je po instrukcji, tak jak w przypadku drugiej linii powyżej.

Zauważmy także, że w ostatniej linii komentarz rozpoczyna się już na początku – mimo, iż treścią komentarza jest instrukcja wypisania na ekran tekstu `Witajcie!`, instrukcja ta zostanie zignorowana przez kompilator, ponieważ z punktu widzenia kompilatora jest to komentarz, a kompilatora komentarze w ogóle nie interesują.

- trzy komentarze wielolinijkowe:

```
/*
 To jest komentarz wielolinijkowy.
 Ponizej zaczyna sie klasa HelloWorldZKomentarzami
*/
public class HelloWorldZKomentarzami {
```

```
/*
   Ponizsza linia kodu nie zostanie wykonana, poniewaz
   zostala zakomentowana.
*/
// System.out.println("Witajcie!");
```

```
System.out.println(/* kolejny komentarz */ "Witajcie ponownie!");
```

Pierwszy komentarz wielolinijkowy jest przed początkiem klasy, a drugi znajduje się w metodzie `main`. Najciekawszy jest trzeci komentarz wielolinijkowy – znajduje się on bowiem wewnątrz wywołania wypisywania na ekran tekstu – nie stanowi to dla kompilatora problemu, ponieważ całkowicie omija on komentarze.

- jeden komentarz dokumentacyjny:

```
/**
 * To jest komentarz dokumentujacy, jak dziala metoda main.
 */
public static void main(String[] args) {
```

O tym, że jest to komentarz dokumentacyjny świadczy to, iż rozpoczyna się od znaków `/**` a nie `/*`. Na razie nie będziemy stosować komentarzy tego typu – wystarczą nam komentarze jednolinijkowe (zaczynające się od znaków `//`) i wielolinijkowe (zaczynające się od znaków `/*` a kończące `*/`).

W rozdziale o metodach powiemy sobie więcej o komentarzach dokumentacyjnych, gdzie nauczymy się, jak można je wykorzystywać do dokumentowania metod.

2.1.2 Zagnieżdżanie komentarzy

Komentarze jednolinijkowe mogą być w sobie zagnieżdżane, w przeciwieństwie do komentarzy wielolinijkowych (oraz komentarzy dokumentacyjnych) – spójrzmy na poniższy przykład:

Nazwa pliku: ZagniezdzoneKomentarzeJednolinijkowe.java

```
public class ZagniezdzoneKomentarzeJednolinijkowe {
  public static void main(String[] args) {
    // wypisz komunikat // wypisz tekst
    System.out.println("Testujemy komentarze jednolinijkowe");
  }
}
```

Powyższy program skompiluje się bez problemów, a w wyniku uruchomienia go, na ekranie zobaczymy komunikat `"Testujemy komentarze jednolinijkowe"`.

Z kolei, próba kompilacji poniższego programu zakończy się błędem – kompilator zaprotestuje:

Nazwa pliku: ZagniezdzoneKomentarzeWielolinijkowe.java

```
public class ZagniezdzoneKomentarzeWielolinijkowe {
  public static void main(String[] args) {
    /*
      to jest komentarz wielolinijkowy
      /* a to jest kolejny, zagniezdzony komentarz */
    */
    System.out.println("Testujemy komentarze wielolinijkowe");
  }
}
```

Kompilator zasygnalizuje następujące problemy:

```
ZagniezdzoneKomentarzeWielolinijkowe.java:6: error: illegal start of
expression
    */
    ^
ZagniezdzoneKomentarzeWielolinijkowe.java:6: error: illegal start of
expression
    */
     ^
2 errors
```

Problem wynika z próby zagnieżdżenia w sobie komentarzy wielolinijkowych.

Warto jeszcze wspomnieć, że komentarze jednolinijkowe mogą być zawarte w komentarzach wielolinijkowych, oraz na odwrót – poniższy kod jest poprawny:

Nazwa pliku: ZagniezdzoneKomentarze.java

```
public class ZagniezdzoneKomentarze {
  public static void main(String[] args) {
    // komentarze jednolinijkowy /* inny komentarz */

    /*
      komentarz wielolinijkowy
      // druga linia
      trzecia linia
    */
  }
}
```

2.1.3 Komentarze w stringach

Komentarze w kodzie źródłowym można umieścić prawie wszędzie – wyjątkiem są stringi (czyli literały tekstowe zawarte między cudzysłowami " "), ponieważ umieszczenie komentarza (jedno- bądź wielolinijkowego) w stringu spowoduje, że będzie on jego częścią:

Nazwa pliku: KomentarzWStringu.java

```
public class KomentarzWStringu {
  public static void main(String[] args) {
    System.out.println("/* wypisujemy powitanie */ Witaj!");
  }
}
```

Powyższy program kompiluje się poprawnie, chociaż po uruchomieniu na ekranie zobaczymy:

```
/* wypisujemy powitanie */ Witaj!
```

To, co umieszczamy w literałach tekstowych, "traci" swoje specjalne znaczenie – kompilator nie traktuje w powyższym przypadku znaków `/*` i `*/` jako początku i końca komentarza wielolinijkowego, lecz, po prostu, jako zwykły tekst, fragment stringa, który chcemy wypisać na ekran.

2.1.4 Kiedy używać komentarzy?

Komentarzy nie należy nadużywać – powinny być unikane zawsze, kiedy to możliwe – **kod w pierwszej kolejności powinien być czytelny.** Wiele osób traktuje komentarze w kodzie jako ostateczność. Należy wypracować sobie pewien balans stanowiący kiedy warto, a kiedy nie powinno się, użyć komentarza.

Gdy piszemy kod, warto zastanowić się:

- czy za miesiąc będę wiedział/a, dlaczego użyta została taka konstrukcja w kodzie?
- czy za miesiąc będę wiedział/a, jak działa dany fragment kodu?
- czy za miesiąc będę wiedział/a, dlaczego w kodzie znajduje się coś, co na pierwszy rzut oka wydaje się zbędne bądź nieintuicyjne?
- czy napisanie komentarza pomoże w zrozumieniu kodu innemu programiście?
- czy napisanie komentarza zapobiegnie wprowdzenia błędów do kodu przez innego programistę, który może nie zdawać sobie sprawy, dlaczego kod jest taki a nie inny?

2.2 Formatowanie kodu i najlepsze praktyki

Pisząc kod musimy zwrócić uwagę na to, czy jest on czytelny.

W Javie, podobnie jak w wielu innych językach programowania, białe znaki nie robią dla kompilatora różnicy. Dzięki temu możemy w dowolny sposób operować nimi w celu poprawy czytelności, a co za tym idzie, także jakości kodu.

Spójrzmy ponownie na nasz pierwszy program, zapisany w inny sposób:

```
public class HelloWorld {public static void main(String[] args)
{System.out.println("Witaj Swiecie!");}}
```

Powyższa wersja programu działa tak samo, jak poniższa:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

Kompilator nie ma problemu z kodem źródłowym z pierwszego z powyższych przykładów pomimo, że dla programisty jest on nieczytelny.

Czy w praktyce ma to aż takie duże znaczenie? Wyobraźmy sobie, że pracujemy nad programem, który ma pół miliona linii kodu. Jeżeli jego autorzy nie dbają o kod źródłowy, to jego utrzymanie, zrozumienie, oraz rozwój, będą bardzo trudne i, co najgorsze, nieprzyjemne.

Powinniśmy zawsze stosować przywilej używania białych znaków (spacji, tabulacji i nowych linii) w celu zapisania kodu naszego programu w taki sposób, by był on czytelny i zrozumiały – zarówno dla nas, jak i innych osób, które potencjalnie będą nasz kod czytać i utrzymywać.

> Często zdarza się, że programista, analizując swój kod w celu poprawy błędu lub wprowadzeniu modyfikacji, nie pamięta, dlaczego napisał taki, a nie inny kod, zaledwie tydzień wcześniej.
>
> Utrzymywanie czystego kodu służy nie tylko ułatwienia współpracy z innymi programistami, ale także ułatwieniu pracy *sobie*.

Po napisaniu kodu warto spojrzeć na niego i zastanowić się, czy:

- po przeczytaniu go tydzień bądź miesiąc później, będziemy rozumieli, co i dlaczego robi?
- czy inny programista patrząc na nasz kod będzie rozumiał, dlaczego napisaliśmy go w taki, a nie inny sposób?

2.2.1 Najlepsze praktyki i konwencje

W związku z tworzeniem programów, często możemy usłyszeć o tzw. *najlepszych praktykach* (best practices), bądź też o różnych *konwencjach programistycznych*.

Zwracają one uwagę na przeróżne aspekty związane z pisaniem kodu źródłowego, jak i ogólnie z procesem wytwarzania oprogramowania. Niektóre z nich są mniej, inne bardziej popularne, część się nawzajem wyklucza, inne natomiast znajdują zastosowanie tylko w konkretnych językach programowania bądź rodzajach pisanych programów.

Najlepsze praktyki i *konwencje programistyczne* mają na celu m. in.:

- podwyższanie jakości wytwarzanego oprogramowania,
- zachowywanie spójności w kodzie źródłowym,
- zwiększanie czytelności kodu źródłowego,
- dbanie o to, by kod spełniał przyjęte wytyczne i standardy.

Ilu programistów, tyle najlepszych praktyk. Są różne konwencje, które są mniej lub bardziej popularne. Ważne jest, aby pracując na nowym projekcie stosować się do standardów (oraz formatowania kodu), które jest na nim używane, o ile nie narusza ono pewnych ogólnoprzyjętych standardów.

Z czasem (także w kolejnych rozdziałach tego kursu), poznasz bardzo wiele standardów, konwencji i najlepszych praktyk. Dobrze jest sobie wyrobić własne zdanie na ich temat, a także znaleźć swój własny styl, który będzie dla Ciebie wygodny. Pamiętaj jednak, by mieć na względzie inne osoby, z którymi będzie pracować.

Pamiętajmy – to, jak i jaki kod piszemy, świadczy o nas! Jeżeli będziemy pisali dobry i czytelny kod, to będziemy uznawani za lepszych programistów.

W kolejnych podrozdziałach opisanych zostało kilka najlepszych praktyk związanych z formatowaniem kodu, które warto stosować od samego początku nauki programowania, aby weszły w nawyk.

Na temat najlepszych praktyk związanych z tworzeniem kodu można znaleźć wiele książek, jedną z których jest "Clean Code", *autorem której jest* Robert C. Martin.

2.2.1.1 Jedna instrukcja na linię

Umieszczanie więcej niż jednej instrukcji w jednej linii powoduje, że kod jest nieczytelny:

```
public class HelloWorldFormatowanie {
  public static void main(String[] args) {
    System.out.println("Witaj"); System.out.println("Swiecie!");
  }
}
```

Zamiast tego, powinniśmy umieszczać po jednej instrukcji na linię:

```
public class HelloWorldFormatowanie {
  public static void main(String[] args) {
    System.out.println("Witaj");
    System.out.println("Swiecie!");
  }
}
```

W jednej linii umieszczamy jedną instrukcję.

2.2.1.2 Stosowanie wcięć

Wszystkie bloki kodu powinny mieć wcięcia w zależności od poziomu, na którym się znajdują.

Są różne konwencje odnośnie rodzaju i ilości białych znaków używanych do robienia wcięć w

kodzie na każdy poziom bloku kodu, np.:

- stosowany jest znak tabulacji (`\t`),
- stosowane są cztery znaki spacji,
- stosowane są dwa znaki spacji.

Wcięcie w kodzie wstawiamy za pomocą przycisku **Tab** *na klawiaturze.*

Przykłady w tym kursie będą zazwyczaj miały wcięcia złożone z dwóch znaków spacji.

W poniższym przykładzie pierwsza linia definiuje klasę `HelloWorld` – na początku nie ma wcięcia, ponieważ klasa ta jest na najwyższym poziomie, tzn. nic jej nie otacza.

Metoda `main` ma pojedyncze wcięcie (złożone z dwóch spacji), ponieważ zawarta jest w klasie.

Instrukcja wypisująca tekst na ekran ma dwa wcięcia, ponieważ jest wewnątrz metody, która jest z kolei wewnątrz klasy – stąd dwa poziomy wcięć. Wkrótce poznamy kolejne "bloki" kodu, które będziemy poprzedzać jeszcze większa liczbą wcięć.

Poniższy przykład ilustruje poziomy wcięć w naszym pierwszym programie:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj");
  }
}
```

Każdą linię w kodzie poprzedzamy odpowiednim wcięciem, którego poziom zależny jest od tego, iloma blokami jest poprzedzony. Każdy blok kodu ma o jeden poziom wcięć więcej, niż blok, który go otacza.

2.2.1.3 Nazwy-wielbłądy (czyli Camel Case)

Camel Case to nazwa konwencji pisania ciągów tekstowych, w których kolejne wyrazy rozpoczynamy od wielkiej litery:

```
toJestPrzykladNazwyStosujacejCamelCase
```

Inną, często stosowaną konwencją, jest rozdzielanie kolejnych słów za pomocą podkreślenia, np.:

```
to_jest_przyklad_nazwy_stosujacej_pokreslenia
```

W Javie stosuje się zazwyczaj Camel Case.

Nazwy klasy także stosują Camel Case*, z tym wyjątkiem, że pierwszy wyraz w nazwie także zapisany jest wielką literą.*

Jeżeli nie będziemy stosować Camel Case, nasze nazwy będą wyglądać jak poniżej i będą nieczytelne:

```
dluganazwawktorejniewiadomoocochodzi
```

Stosujmy *Camel Case* w naszych kodach źródłowych – każdy wyraz, poza pierwszym (wyjątek – nazwa klasy), powinien zaczynać się z wielkiej litery.

2.2.1.4 Nazwy klas zaczynamy wielką literą

Nazwy klas powinny zawsze zaczynać się wielką literą – dla przykładu, klasa w naszym pierwszym programie nazywała się `HelloWorld`.

> Nazwy klas zawsze zaczynamy wielką literą, a pozostałą część nazwy piszemy zgodnie z konwencją Camel Case.

2.2.1.5 Nie trzymamy zakomentowanego kodu

Często pisząc kod będziemy chcieli na chwilę "wyłączyć" fragment kodu, otaczając go komentarzem wielolinijkowym `/*` ... `*/`. O ile jest to przydatne na chwilę, to w finalnej wersji kodu nie powinniśmy przetrzymywać zakomentowanych fragmentów, gdyż zmniejszają one czytelność kodu.

Inni programiści, napotykając taki kod, nie będą pewni, czy kod ten jest w jakiś sposób istotny, czy nie, więc prawdopodobnie pozostawią go tam. Zdarza się, że zakomentowany kod potrafi zalegać w kodzie źródłowym przez lata. Lepszym stwierdzeniem byłoby wręcz, iż zakomentowany kod zaśmieca kod źródłowy.

```
public class HelloWorldZakomentowanyKod {
    public static void main(String[] args) {
        System.out.println("Witaj Swiecie!");

        /* moze to bedzie potrzebne jeszcze kiedys? zostawilem na wszelki wypadek
        System.out.println("Witajcie!");
        */
    }
}
```

> Nie zostawiajmy w napisanym przez nas kodzie źródłowym zakomentowanych bloków kodu – powodują one zmniejszenie czytelności kodu.

2.2.1.6 Nazwy obiektów oraz komentarze po angielsku

Pisząc nasz kod źródłowy na komercyjnym projekcie, zawsze powinniśmy stosować język angielski – zarówno w nazwach zmiennych, metod, klas itd., jak i w komentarzach – taka jest ogólnoprzyjęta zasada – wiele osób, z którymi będziemy pracować, nie będzie znała języka polskiego.

> Tą zasadę naruszamy w tym kursie ze względu na to, że dopiero uczymy się programować.
>
> W pracy zawsze powinniśmy stosować w kodzie źródłowym język angielski.

2.3 Podsumowanie

2.3.1 Komentarze

- Komentarze to tekst w kodzie źródłowym, który jest ignorowany przez kompilator.
- Komentarze służą programistom do wyjaśniania fragmentów kodu oraz w celu udokumentowania jego działania.
- Java posiada trzy rodzaje komentarzy:
  - `//` – dwa slashe ropoczynają komentarz jednolinijkowy, który kończy się wraz z końcem linii,
  - `/* */` – ten zestaw znaków służy do wstawiania komentarzy wielolinijkowych,
  - `/** */` – w ten sposób wstawiamy do kodu komentarze dokumentacyjne, z których może zostać wygenerowana dokumentacja naszego kodu.
- Komentarzy nie należy nadużywać – powinniśmy, w pierwszej kolejności, pisać kod w taki sposób, by był czytelny.
- Komentarze warto stosować w celu wyjaśnienia nietrywialnych części kodu.
- Komentarzy wielolinijkowych nie wolno w sobie zagnieżdzać – poniższy fragment kodu jest nieprawidłowy – zawiera on zagnieżdżony komentarz wielolinijkowy:

```
/*
  to jest komentarz wielolinijkowy
  /* a to jest kolejny, zagniezdzony komentarz */
*/
```

- Komentarze jednolinijkowe mogą być zawarte w komentarzach wielolin. i na odwrót.
- Komentarzy nie powinno się umieszczać w stringach (literałach tekstowych) – zostaną one po prostu uznane za fragment literału tekstowego, a nie za komentarze:

```
System.out.println("/* to nie jest komentarz! */ Witaj!");
```

2.3.2 Formatowanie kodu i najlepsze praktyki

- Kod źródłowy powinniśmy zapisywać w taki sposób, by był czytelny.
- Możemy w tym celu używac białych znaków – tabulacji, spacji, oraz nowych linii.
- Po napisaniu kodu warto zastanowić się, czy, zarówno my, jak i inni programiści, będą go rozumieli za tydzień/miesiąc?
- Najlepsze praktyki to zestaw konwencji, których przestrzeganie ma na celu pisanie kodu o wysokiej jakości. Oto kilka zasad, które warto stosować od początku nauki programowania:
  - piszemy jedną instrukcję na linię,
  - stosujemy wcięcia w blokach kodu,
  - stosujemy *Camel Case*, czyli `nazwyWygladajaWTenSposob`,
  - nazwy klasy zaczynamy wielką literą,
  - nie trzymamy zakomentowanego kodu w kodzie źródłowym,
  - używamy w kodzie języka angielskiego do nazywania obiektów, oraz w komentarzach.

2.4 Pytania

1. Jak zapisujemy każdy rodzaj komentarza w Javie?
2. Które komentarze mogą być zagnieżdżane?
3. Co można by poprawić w poniższym kodzie?

```
public class zadaniezleformatowanie {
  public static void main(String[] args) {
  // System.out.println("Witaj");
  // System.out.println("Swiecie!");
  System.out.println("Witaj Swiecie!");
  }
}
```

4. Która z poniższych nazw zapisana jest Camel-Casem?
    1. nazwa
    2. MojaNazwa
    3. mojanazwa
    4. Wiadomosc
5. Która z nazw klas jest poprawna?
    1. HelloWorld
    2. helloworld
    3. helloWorld
    4. Helloworld

2.5 Zadania

2.5.1 Dopisz komentarze

Dopisz do naszego pierwszego programu, wypisującego tekst `"Witaj Swiecie!"`, kilka komentarzy różnych typów.

2.5.2 Brak main

W programie z poprzedniego zadania zakomentuj całą metodę `main`.

1. Czy program się skompiluje?
2. Czy program da się uruchomić, i jeżeli tak, to co zobaczymy na ekranie?

3 Rozdział III – Zmienne

W tym rozdziale:

- dowiemy się, czym są zmienne i jak ich używać,
- poznamy reguły dotyczące nazw w języku Java,
- opowiemy sobie o typach podstawowych,
- zaczniemy korzystać z podstawowych operatorów,
- poznamy pierwszy typ złożony `String`,
- zobaczymy, w jaki sposób pobrać od użytkownika dane.

3.1 Czym są zmienne?

Pisząc programy, będziemy korzystali z różnych danych, które musimy gdzieś przechowywać. Dla przykładu, program, który liczy pole i obwód koła, będzie potrzebował przechować następujące dane:

- promień koła,
- wartość PI,
- wynikowe pole koła,
- wynikowy obwód koła.

Zmienne w programowaniu są *kontenerami na wartości*. Każda zmienna ma swoją nazwę, typ, oraz przypisaną wartość.

3.1.1 Definiowanie zmiennych

Zanim użyjemy zmiennej, musimy ją *zdefiniować*, czyli wyspecyfikować, czym jest dana zmienna.

Definicja zmiennej odbywa się poprzez podanie typu zmiennej, po którym następuje jej nazwa i średnik. Spójrzmy na poniższy przykład:

Nazwa pliku: DefinicjaZmiennej.java

```
public class DefinicjaZmiennej {
  public static void main(String[] args) {
    int liczbaOsob;
  }
}
```

Jedyne, co robi powyższy program, to zdefiniowanie w metodzie `main`, że `liczbaOsob` to zmienna *typu całkowitego* – od tej pory będziemy mogli z niej korzystać w kodzie naszego programu.

Zmienne nazywamy zmiennymi nie bez powodu – ich wartość może się zmieniać w trakcie działania programu. Dla kontrastu, istnieją także *stałe*, którym przypisujemy wartość jednorazowo, jak zobaczymy wkrótce.

Aby zmienić wartość zmiennej, piszemy jej nazwę, znak równości, oraz nową wartość:

NadanieZmiennejWartosci.java

```
public class NadanieZmiennejWartosci {
  public static void main(String[] args) {
    int liczbaOsob;

    liczbaOsob = 5;
    liczbaOsob = 10;
    liczbaOsob = 15;
  }
}
```

W kolejnych liniach powyższego programu, wartość zmiennej `liczbaOsob` to `5`, następnie `10`, a na końcu `15`.

Definiując zmienną, możemy od razu nadać jej wstępną wartość – nazywamy to *inicjalizacją* zmiennej. W takim przypadku korzystamy z *operatora przypisania* = (znak

równości), po którym należy umieścić wartość bądź wyrażenie, którego wartość, po wyliczeniu, będzie przypisana do zmiennej. Znak równości i wartość umieszczamy po nazwie zmiennej, a przed średnikiem:

Nazwa pliku: InicjalizacjaZmiennej.java

```
public class InicjalizacjaZmiennej {
  public static void main(String[] args) {
    double pi = 3.14;
  }
}
```

Zdefiniowana powyżej zmienna `pi` ma od razu przypisaną wartość `3.14`.

W jednej linii możemy zdefiniować więcej, niż jedną zmienna, poprzez oddzielenie nazw zmiennych od siebie przecinkiem:

Nazwa pliku: DwieZmienneNaRaz.java

```
public class DwieZmienneNaRaz {
  public static void main(String[] args) {
    int x = 5, y = -20;
  }
}
```

W powyższym przykładzie zdefiniowaliśmy dwie zmienne: `x` oraz `y`, a także zainicjalizowaliśmy je od razu wstępnymi wartościami, odpowiednio, `5` i `-20`.

3.1.2 Wypisywanie na ekran wartości zmiennych

Wartości zmiennych możemy wypisać na ekran za pomocą instrukcji `System.out.println` – wystarczy jako argument podać zmienną, a jej wartość zostanie wypisana na ekran.

Co więcej, podając komunikat do wypisania do instrukcji `System.out.println`, możemy także "sklejać" ze sobą wiele wartości za pomocą plusa +, dzięki czemu możemy wypisywać na ekran ciekawsze komunikaty – spójrzmy na poniższy przykład:

Nazwa pliku: WypisywanieZmiennychNaEkran.java

```
public class WypisywanieZmiennychNaEkran {
  public static void main(String[] args) {
    double pi = 3.14;
    double kwadratPi;
    kwadratPi = pi * pi;

    System.out.println(pi);
    System.out.println("Liczba pi to w przyblizeniu: " + pi);
    System.out.println("Liczba pi to w przyblizeniu: " + pi +
        ", a kwadrat liczby pi to " + kwadratPi );
  }
}
```

W powyższym przykładzie najpierw wypisujemy samą wartość zmiennej `pi`. Następnie, sklejamy ciąg znaków `"Liczba pi to w przyblizeniu: "` z wartością zmiennej `pi`, a na końcu wyświetlamy jeszcze jeden, bardziej złożony komunikat.

Zauważ, że w przypadku ostatniego wywołania instrukcji `System.out.println`, dla czytelności mogliśmy przenieść fragment komunikatu do wypisania do linii poniżej.

Ten program spowoduje wypisanie na ekran:

```
3.14
Liczba pi to w przyblizeniu: 3.14
Liczba pi to w przyblizeniu: 3.14, a kwadrat liczby pi to 9.8596
```

3.1.3 Przykład – liczenie pola i obwodu koła

Spójrzmy na przykład z kilkoma zmiennymi – jest to program liczący pole i obwód koła:

Nazwa pliku: ObwodPoleKola.java

```
public class ObwodPoleKola {
  public static void main(String[] args) {
    int promienKola = 8;
    double pi = 3.14;

    double poleKola = pi * promienKola * promienKola;
    double obwodKola = 2 * pi * promienKola;

    System.out.println("Pole kola wynosi: " + poleKola);
    System.out.println("Obwod kola wynosi: " + obwodKola);
  }
}
```

W wyniku działania tego programu, na ekranie zobaczymy:

```
Pole kola wynosi: 200.96
Obwod kola wynosi: 50.24
```

W powyższym programie zdefiniowaliśmy cztery zmienne:

- `promienKola`
- `pi`
- `poleKola`
- `obwodKola`

Każdej z nich przypisaliśmy wartość w momencie ich definicji, chociaż wartości te różnią się od siebie:

- zmienne `promienKola` oraz `pi` zostały zainicjalizowane konkretnymi liczbami: `8` i `3.14`,
- zmiennym `poleKola` oraz `obwodKola` zostały przypisane nie konkretne liczby, lecz wynik obliczenia wyrażeń na pole i obwód koła. Zmiennej `poleKola` zostanie przypisana wartość będąca iloczynem wartości zmiennej `pi` dwukrotnie pomnożonej przez wartość zmiennej `promienKola`. Podobnie zostanie wyliczona wartość zmiennej `obwodKola`.

Każda zmienna ma swoją nazwę oraz typ, który definiuje, jakie wartości może przechowywać. Poznamy teraz zasady nazewnictwa obiektów w Javie, a potem dowiemy się, czym są typy podstawowe.

3.2 Reguły nazw w języku Java

Nazwy w języku Java, w tym nazwy: zmiennych, metod, klas, i innych obiektów, muszą spełniać trzy następujące wymagania:

1. Nazwa musi zaczynać się od litery, podkreślenia _ bądź znaku dolara $
2. Kolejnymi znakami, poza tym wymienionymi w punkcie 1., mogą być także cyfry.
3. Nazwy nie mogą być takie same, jak nazwy zastrzeżone w języku Java (np. **class**, **public**, **void** itd.).

Słowa Kluczowe *to zastrzeżone słowa, które mają specjalne znaczenie w językach programowania – nie można używać ich jako nazw obiektów. Jeżeli spróbujemy, to kompilacja zakończy się porażką, a kompilator poinformuje nas o zaistniałym błędzie.*

Przykłady poprawnych nazw (przed nazwą zmiennych jest ich typ – zaraz sobie o nich opowiemy):

```
char jedenZnak;
double poleKola;
int liczba_pracownikow;
double _pi;
int $wynagrodzenieMiesieczne;
double wynikWRozliczeniu360;
```

Przykłady niepoprawnych nazw:

```
// kazda z ponizszych nazw jest niepoprawna i spowoduje blad kompilacji
double 314ToLiczbaPi; // nazwa nie moze zaczynac sie od liczby
double promien kola; // spacja w nazwie jest niedozwolona
int liczba#pracownikow; // znak # nie może wystapic w nazwie
int public; // public to slowo kluczowe w jezyku Java
```

Spójrzmy na program z powyższymi, błędnymi nazwami zmiennych:

Nazwa pliku: NiepoprawneNazwyZmiennych.java

```
public class NiepoprawneNazwyZmiennych {
  public static void main(String[] args) {
    double 314ToLiczbaPi;
    double promien kola;
    int liczba#pracownikow;
    int public;
  }
}
```

Próba kompilacji powyższego programu zakończy się wypisaniem przez kompilator wielu błędów:

```
NiepoprawneNazwyZmiennych.java:3: error: not a statement
    double 314ToLiczbaPi;
    ^
NiepoprawneNazwyZmiennych.java:3: error: ';' expected
    double 314ToLiczbaPi;
           ^
NiepoprawneNazwyZmiennych.java:3: error: not a statement
    double 314ToLiczbaPi;
               ^
```

```
NiepoprawneNazwyZmiennych.java:4: error: ';' expected
    double promien kola;
                   ^
NiepoprawneNazwyZmiennych.java:4: error: not a statement
    double promien kola;
                    ^
NiepoprawneNazwyZmiennych.java:5: error: illegal character: '#'
    int liczba#pracownikow;
              ^
NiepoprawneNazwyZmiennych.java:5: error: not a statement
    int liczba#pracownikow;
               ^
NiepoprawneNazwyZmiennych.java:6: error: not a statement
    int public;
    ^
NiepoprawneNazwyZmiennych.java:6: error: ';' expected
    int public;
       ^
NiepoprawneNazwyZmiennych.java:6: error: illegal start of expression
    int public;
        ^
NiepoprawneNazwyZmiennych.java:6: error: illegal start of type
    int public;
              ^
NiepoprawneNazwyZmiennych.java:8: error: class, interface, or enum expected
}
^
12 errors
```

Poza wymienionymi wcześniej regułami, trzeba jeszcze zwrócić uwagę na dwie rzeczy:

- nazwy mogą mięc dowolną długość,
- małe i wielkie litery są rozróżniane, więc zmienne `pi` oraz `PI` to dwie różne nazwy.

Przykład zmiennej o długiej nazwie:

```
// dluga, opisowa nazwa zmiennej
double wynikowyObwodKolaKtoryStanowiWartosc2PiR;
```

Poniżej zdefiniowane zostały dwie zmienne o podobnej nazwie, jednak dla kompilatora są to dwie zupełnie różne od siebie zmienne, ponieważ w ich nazwach zostały użyte litery różnej wielkości:

```
// dwie rozne zmienne! wielkosc liter jest rozrozniana
double pi = 3.14;
double PI = 3.14;
```

Jeżeli spróbowalibyśmy użyć zmiennej popełniając błąd w wielkości znaków, to kompilator zaprotestuje informując nas, że nie wie czym jest dany identyfikator, który napotkał w kodzie:

Nazwa pliku: NieznanaNazwaZmiennej.java

```
public class NieznanaNazwaZmiennej {
  public static void main(String[] args) {
    int promienKola = 8;

    // blad! zmienna PROMIENKOLA nie istnieje!
    System.out.println("Promien kola wynosi: " + PROMIENKOLA);
  }
}
```

Kompilator, dzięki naszej definicji w trzeciej linii, wie, czym jest `promienKola` – jest to zmienna typu całkowitego. Kompilator, nie wiedząc jednakże, czym jest `PROMIENKOLA`, przerwie kompilację i wypisze na ekran:

```
NieznanaNazwaZmiennej.java:6: error: cannot find symbol
    System.out.println("Promien kola wynosi: " + PROMIENKOLA);
                                                 ^
  symbol:   variable PROMIENKOLA
  location: class NieznanaNazwaZmiennej
1 error
```

Gdybyśmy zamiast `PROMIENKOLA` użyli nazwy `promienKola`, program skompilowałby się bez problemu.

3.2.1 Nadawanie nazw

Nazwy w kodach źródłowych niosą bardzo dużo informacji, o ile są one odpowiednio dobrane do obiektów, które opisują. Spójrz na program liczący pole i obwód koła ze zmienionymi nazwami:

```
public class MojProgram {
  public static void main(String[] args) {
    int a = 8;
    double b = 3.14;

    double x = b * a * a;
    double y = 2 * b * a;

    System.out.println("Pole kola wynosi: " + x);
    System.out.println("Obwod kola wynosi: " + y);
  }
}
```

Patrząc na nazwy zmiennych, nie jesteśmy w stanie odpowiedzieć na pytania:

- Do czego służą te zmienne?
- Jakie wartości będą przechowywać?
- Co jest obliczane w liniach, w których definiujemy zmienne `x` oraz `y`?

Powyższy przykład jest trywialny, ale jest to faktyczny problem podczas programowania. Nazywanie obiektów nie jest proste, a wymyślenie dobrej nazwy nierzadko sprawia programistom problemy – czasem nawet wtedy, gdy zapytają kogoś o pomoc w jej wymyśleniu.

Przy nazywaniu zmiennych i innych obiektów należy:

- Używać dobrze dobranych, opisowych nazw, tak, aby zwiększały one czytelność kodu. **Zawsze warto zastanowić się nad nazwą.**

- Stosować jedną z dwóch konwencji w nazwach, które składają się z więcej niż jednego słowa (**w Javie stosujemy Camel Case**):
  - *Camel Case* – wielka litera powinna rozpoczynać każde słowo w nazwie, poczynając od drugiego słowa, np. `poleKola` bądź `glownyTelefonKontaktowy`.

  lub

  - rozdzielanie słów znakiem podkreślenia (_), np. `pole_kola` bądź `glowny_telefon_kontaktowy`.

Warto nadawać naszym zmiennym (i innym obiektom) opisowe nazwy. Zwiększa to czytelność naszego kodu. **Pamiętaj – zawsze warto zastanowić się nad nazwą.**

3.3 Typy podstawowe

Znając już zasady dotyczące nadawania nazw w Javie, wróćmy do programu liczącego pole i obwód koła, który używa czterech zmiennych:

Nazwa pliku: ObwodPoleKola.java

```
public class ObwodPoleKola {
  public static void main(String[] args) {
    int promienKola = 8;
    double pi = 3.14;

    double poleKola = pi * promienKola * promienKola;
    double obwodKola = 2 * pi * promienKola;

    System.out.println("Pole kola wynosi: " + poleKola);
    System.out.println("Obwod kola wynosi: " + obwodKola);
  }
}
```

Program ten korzysta z jednej zmiennej liczbowej **typu całkowitego** `int`, oraz trzech zmiennych liczbowych **typu zmiennoprzecinkowego** `double`.

Typ zmiennej to informacja, jakiego rodzaju wartości zmienna będzie mogła przechowywać. Zmienna `promienKola` będzie służyła do przechowywania liczb całkowitych, a zmienne `pi`, `poleKola`, oraz `obwodKola`, będą mogły mieć wartości liczb rzeczywistych (czyli będą mogły mieć część ułamkową).

Typy zmiennych dzielimy na dwa rodzaje:

- typy prymitywne,
- typy złożone.

Język Java posiada 8 następujących typów prymitywnych, które są typami predefiniowanymi przez dany język programowania (w przeciwieństwie do typów złożonych, które definiują programiści – wkrótce dowiemy się więcej na ten temat):

Nazwa	Zakres Wartości	Opis	Przykład
`boolean`	`true` lub `false`	Typ logiczny – przyjmuje wartości prawda (`true`) bądź fałsz (`false`).	`true`
`byte`	od -128 do 127	8-bitowa liczba całkowita.	`10`
`short`	od -32768 do 32767	16-bitowa liczba całkowita.	`1500`
`int`	od -2147483648 do 2147483647	32-bitowa liczba całkowita.	`1000000`
`long`	od -2^63 do 2^63 - 1	64-bitowa liczba całkowita.	`10000000000L`
`float`	Liczby rzeczywiste	32-bitowa liczba zmiennoprzecinkowa (*floating-point*).	`3.14f`
`double`	Liczby rzeczywiste – większy zakres	64-bitowa liczba zmiennoprzecinkowa (*double-precision*).	`3.14`
`char`	0 do 2^{16}-1	Typ znakowy – jeden 16-bitowy znak Unicode.	`'z'`

Definiując zmienną, podajemy jej typ zgodnie z tym, jakie wartości chcemy w danej zmiennej przechowywać. Język Java jest *językiem typowanym*, co oznacza, że zmienne mają typ określony przy ich definiowaniu przez programistę i nie może się on zmienić w trakcie wykonywania programu, tzn. zmienna typu całkowitego może przechowywać jedynie wartości liczbowe typu całkowitego.

Spójrzmy na poniższy przykład – zmiennej `liczbaOkien` podczas definicji został nadany typ **`int`**. W kolejnej linii, próbujemy przypisać do tej zmiennej wartość rzeczywistą:

Nazwa pliku: UstalonyTyp.java

```
public class UstalonyTyp {
  public static void main(String[] args) {
    int liczbaOkien;

    liczbaOkien = 3.5; // blad!
  }
}
```

Kompilacja tego programu nie powiedzie się, ponieważ do zmiennej, która miała przechowywać liczby całkowite, próbowaliśmy przypisać liczbę rzeczywistą:

```
UstalonyTyp.java:5: error: incompatible types: possible lossy conversion
from double to int
    liczbaOkien = 3.5;
                  ^
1 error
```

Kompilator wie, że zmienna `liczbaOkien` nie może przechowywać części ułamkowej przypisywanej do niej wartości (bo typ **`int`** określa jedynie liczby całkowite), więc ewentualne przypisanie wartości `3.5` do zmiennej `liczbaOkien` spowodowałoby stratę części wartości – kompilator na to nie pozwoli i nie skompiluje naszego programu.

3.3.1 Literały

Aby nadać wartości zmiennym typów podstawowych, musimy je zapisać w kodzie programu.

Liczby takie jak **`10`**, **`3.14`**, znaki, jak np. `'A'`, `'N'`, tekst ujęty w cudzysłowy np. `"Witaj Swiecie!"` oraz wartości logiczne **`true`** i **`false`**, to tzw. ***literały***. Umieszczone w kodzie źródłowym reprezentują po prostu konkretne wartości danego typu:

Nazwa pliku: LiteralyPrzyklad.java

```
public class LiteralyPrzyklad {
  public static void main(String[] args) {
    int liczbaCalkowita = 10;
    double liczbaRzeczywista = 2.5;
    char znak = 'A';
    boolean wartoscLogiczna = true;
  }
}
```

Zwróć uwagę, że literał znakowy dla zmiennej `znak` zapisany jest w apostrofach, a nie cudzysłowach – za chwilę opowiemy sobie, dlaczego.

3.3.2 Typy całkowite i zmiennoprzecinkowe

Zgodnie z tabelą typów prymitywnych, język Java posiada cztery typy liczb całkowitych (**`byte`**,

short, **int**, **long**), a także dwa typy liczb zmiennoprzecinkowych **float** i **double**.

Po co nam kilka typów o taki samych rodzajach wartości, ale inny zakresach?

Wynika to z faktu, iż większy zakres oznacza, że zmienna będzie potrzebować więcej miejsca na swoją wartość. W dawnych czasach miało to duże znaczenie, gdy pamięć w komputerach była dużo bardziej ograniczona.

Zmienne typu **byte** mogą przechowywać wartości nie większe, niż 127, a zmienne typu **int** – nie większe, niż 2147483647 – zobaczmy, co stanie się, gdy spróbujemy przypisać do nich większe wartości:

Nazwa pliku: ZaDuzaWartoscByte.java

```
public class ZaDuzaWartoscByte {
  public static void main(String[] args) {
    byte malaLiczba = 123;

    // blad - wartosc przekracza zakres typu byte
    malaLiczba = 10000;
  }
}
```

Nazwa pliku: ZaDuzaWartoscInt.java

```
public class ZaDuzaWartoscInt {
  public static void main(String[] args) {
    int duzaLiczba = 12345;

    // blad - wartosc przekracza zakres typu int
    duzaLiczba = 9999999999;
  }
}
```

Kompilacja zakończy się błędem w przypadku obu powyższych programów:

```
ZaDuzaWartoscByte.java:6: error: incompatible types: possible lossy
conversion from int to byte
    malaLiczba = 10000;
                 ^
1 error
```

```
ZaDuzaWartoscInt.java:6: error: integer number too large
    duzaLiczba = 9999999999;
                 ^
1 error
```

Typ prymitywny określa nie tylko, jakiego rodzaju wartości zmienna może przechowywać, ale także jaki jest zakres tych wartości.

W trakcie nauki, będziemy stosować typ **int** – jest to domyślny typ liczb w kodzie Java – jeżeli kompilator widzi w kodzie źródłowym *literał* liczby całkowitej (np. 10), to traktuje go jako wartość typu **int**. Domyślnym typem wartości rzeczywistych (zmiennoprzecinkowych) jest typ **double** i taki też typ będziemy stosować w kursie.

Wróćmy jeszcze na chwilę do tabeli typów podstawowych – przykład wartości dla typu **long** ma na końcu liczby literę L: 10000000000L. Nie jest to błąd w tabeli – jak już wspomnieliśmy, kompilator domyślnie traktuje wszystkie literały liczb całkowitych, które napotka w kodzie, jako

liczby typu **int**. Gdybyśmy zapisali w kodzie Java liczbę 10000000000, to kompilator zgłosiłby błąd – ta wartość jest zbyt duża dla typu **int**. Aby jednak móc w kodzie zapisywać większe liczby, które będziemy mogli umieścić w zmiennych typu **long** (który to typ ma dużo większy zakres, niż typ **int**), należy na końcu liczby dopisać literę L (mała lub wielką, ale wielka jest bardziej czytelna):

Nazwa pliku: LiczbaLong.java

```
public class LiczbaLong {
  public static void main(String[] args) {
    long wielkaLiczba;

    // linia zakomentowana, poniewaz spowodowalaby blad kompilacji!
    // wielkaLiczba = 10000000000;

    // dopisalismy L na koniec - kompilator wie, ze będzie to duza liczba
    wielkaLiczba = 10000000000L;
  }
}
```

Powyższy kod kompiluje się bez problemów, dzięki zastosowaniu suffixu L przy dużej liczbie.

Podobnie rzecz ma się z liczbami typu **double** *i* **float** *– jeżeli chcemy przypisać wartość rzeczywistą do zmiennej typu* **float**, *musimy użyć suffixu* f.

3.3.3 Typ boolean

Typ **boolean** służy do przechowywania wartości typu prawda / fałsz – zaczniemy z niego korzystać w rozdziale o instrukcjach warunkowych.

3.3.4 Typ char

Typ **char** przechowuje pojedyncze znaki, które ujmujemy w apostrofy ' ', w przeciwieństwie do stringów (łańcuchów tekstowych), które zawieramy między cudzysłowami. W apostrofach może być zawarty co najwyżej jeden znak.

W poniższym przykładzie nieprawidłowo przypisujemy wartość do zmiennej `znak` w drugiej i trzeciej linii:

```
char znak;
znak = 'ABC';     // blad - maksymalnie jeden znak
znak = "Witaj!"; // blad - "Witaj!" to nie jest znak, lecz lancuch znakow
znak = 'X';       // poprawne przypisanie - jeden znak
```

3.4 Używanie zmiennych

Jeżeli zdefiniujemy zmienną w metodzie, to zanim jej użyjemy, musimy przypisać jej wartość.

Zmienne zdefiniowane w metodach nazywamy *zmiennymi lokalnymi*. O zmiennych innych niż zmienne lokalne porozmawiamy wkrótce.

Spójrzmy na poniższy przykład:

Nazwa pliku: UzycieNiezainicjalizowanejZmiennej.java

```
public class UzycieNiezainicjalizowanejZmiennej {
  public static void main(String[] args) {
    int x;

    // blad! nie nadalismy zmiennej x jeszcze zadnej wartosci
    System.out.println("Wartosc x wynosi: " + x);
  }
}
```

Jeżeli spróbujemy użyć zmiennej, której nie nadaliśmy jeszcze wartości, to próba kompilacji naszego kodu zakończy się błędem. Jest to spowodowane tym, że, co prawda, powiedzieliśmy kompilatorowi czym jest "*x*" – jest to *zmienna mogąca przechowywać liczby typu całkowitego* – nie powiedzieliśmy mu jednak, jaką zmienna `x` ma wartość, więc kompilator protestuje:

```
UzycieNiezainicjalizowanejZmiennej.java:6: error: variable x might not have
been initialized
        System.out.println("Wartosc x wynosi: " + x);
                                                  ^
1 error
```

Spójrzmy na kolejny przykład, w którym używamy kilku zmiennych różnych typów:

Nazwa pliku: UzycieZmiennych.java

```
public class UzycieZmiennych {
  public static void main(String[] args) {
    boolean padaDeszcz = false;

    byte liczbaDni = 127;
    int liczbaLatSwietlnych = 1000000;
    double pi = 3.14;

    char jedenZnak = 'a';

    System.out.println("Czy pada deszcze? " + padaDeszcz);

    System.out.println("Liczba dni: " + liczbaDni);
    System.out.println("Liczba lat swietlnych: " + liczbaLatSwietlnych);

    System.out.println("Liczba pi: " + pi);

    System.out.println("Pierwsza litera alfabetu to " + jedenZnak);
  }
}
```

W wyniku działania programu, na ekranie zobaczymy:

```
Czy pada deszcze? false
Liczba dni: 127
Liczba lat swietlnych: 1000000
Liczba pi: 3.14
Pierwsza litera alfabetu to a
```

3.5 Stałe

Czasem chcemy zapisać w kodzie wartości, które symbolizują jakieś niezmienne dane. Pisząc program, który będzie wykonywał różne operacje matematyczne, moglibyśmy chcieć zapisać wartość liczby Pi jako stała – jej wartość nie powinna się zmienić w trakcie wykonania programu.

Z pomocą przychodzą nam *stałe*, które definiuje się podobnie, jak zmienne, z tym, że:

- przed typem stałej dodajemy słowo kluczowe **`final`**, aby powiadomić kompilator, iż nadana jej wartość będzie niezmienna,
- zgodnie z konwencją, której jeszcze nie znamy, nazwę stałej zapisujemy wielkimi literami, a kolejne słowa w jej nazwie rozdzielamy znakami podkreślenia.

Spójrzmy na przykład definicji i użycia stałych:

Nazwa pliku: Stale.java

```
public class Stale {
  public static void main(String[] args) {
    final double PI = 3.14;
    final int LICZBA_DNI_W_TYGODNIU = 7;

    int promien = 20;
    double poleKola = PI * promien * promien;
  }
}
```

Jak widać w ostatniej linii, stałych możemy używać do wyliczania różnych wartości – w tym przypadku, skorzystaliśmy z stałej `PI` do wyliczenia pola koła.

Jeżeli w dalszej części programu spróbujemy przypisać stałej inną wartość:

Nazwa pliku: Stale.java

```
public class Stale {
  public static void main(String[] args) {
    final double PI = 3.14;
    final int LICZBA_DNI_W_TYGODNIU = 7;

    int promien = 20;
    double poleKola = PI * promien * promien;

    // blad!
    PI = 5;
  }
}
```

to podczas próby kompilacji naszego kodu zobaczymy następujący błąd:

```
Stale.java:10: error: cannot assign a value to final variable PI
        PI = 5;
        ^
1 error
```

3.6 Podstawy zmiennych – zadania

Poniższe zadania mają na celu przećwiczenie dotychczasowego materiału o zmiennych.

3.6.1 Dodawanie liczb

Napisz program, w którym zdefiniujesz trzy zmienne typu `int`. Do dwóch pierwszych przypisz dowolne liczby, a do trzeciej – wynik dodawania dwóch pierwszych liczb. Aby dodać do siebie wartości dwóch zmiennych, skorzystaj ze znaku + (plus). Wypisz wynik na ekran.

3.6.2 Obwód trójkąta

Napisz program, który skorzysta z czterech zmiennych w celu policzenia obwodu trójkąta. W trzech zmiennych zapisz długość każdego z boków, a do ostatniej zmiennej przypisz wynik – obwód trójkąta. Wypisz wynik na ekran.

3.6.3 Aktualna data

Napisz program, w którym do trzech różnych zmiennych przypiszesz aktualny dzień, miesiąc, i rok. Pamiętaj o odpowiednim nazewnictwie zmiennych. Wypisz na ekran wszystkie wartości.

3.6.4 Liczba miesięcy w roku

Napisz program, w którym zdefiniujesz stałą, do której przypiszesz liczbę miesięcy w roku. Pamiętaj o odpowiednim nazewnictwie stałej. Wypisz wartość zdefiniowanej stałej na ekran.

3.6.5 Inicjały

Napisz program, w którym przypiszesz swoje inicjały do dwóch zmiennych typu `char` – do każdej ze zmiennych po jednym znaku. Wypisz swoje inicjały na ekran – po każdej literze powinna następować kropka, np. P. K.

3.7 Operatory w programowaniu

W poprzednich programach użyliśmy * (gwiazdki) w celu wyliczenia pola i obwodu koła, plusa + do *konkatenacji* (łączenia) łańcuchów znaków, oraz znaku równości = w celu przypisania zmiennym wartości – są to przykłady *operatorów*.

Operatory wykonują pewną operację na ich argumentach, które nazywamy *operanadami*, i zwracają pewną wartość. Dla przykładu, operator * (mnożenia):

- oczekuje dwóch argumentów (operandów),
- wykonuje operację na argumentach – mnożenie,
- zwraca wartość, która jest wynikiem mnożenia argumentów.

Spójrzmy na przykład:

Nazwa pliku: PrzykladOperatoraMnozenia.java

```
public class PrzykladOperatoraMnozenia {
  public static void main(String[] args) {
    final int SEKUNDY_W_MINUCIE = 60;
    final int MINUTY_W_GODZINIE = 60;

    final int SEKUNDY_W_GODZINIE = SEKUNDY_W_MINUCIE * MINUTY_W_GODZINIE;

    System.out.println("Liczba sekund w godzinie: " + SEKUNDY_W_GODZINIE);
  }
}
```

W powyższym przykładzie, operator mnożenia * otrzymał dwa argumenty: stałą `SEKUNDY_W_MINUCIE` oraz `MINUTY_W_GODZINIE`. Mnożenie zostało wykonane, a jego wynik przypisany do stałej `SEKUNDY_W_GODZINIE`.

Istnieją różne rodzaje operatorów. Podzielić je możemy ze względu na:

- rodzaj wykonywanej operacji – wyróżniamy m. in. operatory:
  - arytmetyczne,
  - logiczne,
  - bitowe i inne.
- liczbę przyjmowanych argumentów – java posiada operatory:
  - jednoargumentowe,
  - dwuargumentowe,
  - jeden operator trójargumentowy.

Każdy język programowania posiada wiele różnych operatorów, które mają różne priorytety – tak jak w matematyce: mnożenie wykonujemy przed dodawaniem, o ile nie użyliśmy nawiasów do zmiany kolejności wykonania działań.

W pierwszej kolejności zaznajomimy się z operatorami arytmetycznymi.

3.7.1 Operatory arytmetyczne

Java posiada następujące operatory arytmetyczne:

- + dodawanie (służy także do łączenia ciągów znaków i innych wartości)
- - odejmowanie
- * mnożenie
- / dzielenie
- % modulo – reszta z dzielenia

Operatory te działają na dwóch argumentach i dlatego nazywamy je *dwuargumentowymi*, bądź *binarnymi*.

Operatory: mnożenia, dzielenia, oraz reszty z dzielenia, mają taki sam priorytet, który jest wyższy, niż priorytet operatorów dodawania i odejmowania – tak jak w matematyce. Możemy jednak skorzystać z nawiasów, aby zmienić priorytet wykonywania działań.

3.7.1.1 Operatory dodawania, odejmowania i mnożenia

Spójrzmy na kilka przykładów operatorów dodawania, odejmowania i mnożenia:

Nazwa pliku: OperatoryArytmetycznePodstawy.java

```
public class OperatoryArytmetycznePodstawy {
  public static void main(String[] args) {
    System.out.println(2 + 2);
    System.out.println(100 - 5);
    System.out.println(10 * 5);

    System.out.println(10 * 0 + 1);
    System.out.println(10 * (0 + 1));
  }
}
```

W wyniku działania tego programu, na ekranie zobaczymy:

```
4
95
50
1
10
```

Dwa ostatnie wyniki różnią się od siebie, ponieważ zmieniliśmy kolejność wykonywania zadań za pomocą nawiasów – w poniższej linii:

```
System.out.println(10 * 0 + 1);
```

najpierw mnożmy `10` razy `0`, a potem dodajemy `1`, a w kolejnej linii:

```
System.out.println(10 * (0 + 1));
```

najpierw dodajemy `1` do `0`, a potem mnożymy przez `10`.

3.7.1.2 Operatory dzielenia i reszty z dzielenia

Spójrzmy na przykład użycia operatorów dzielenia i reszty z dzielenia:

Nazwa pliku: OperatoryArytmetyczneDzielenie.java

```
public class OperatoryArytmetyczneDzielenie {
  public static void main(String[] args) {
    System.out.println(10 / 5);
    System.out.println(10 / 4);
    System.out.println(10 % 3);
  }
}
```

W wyniku działania tego programu, na ekranie zobaczymy:

```
2
2
1
```

W trzeciej linii widzimy wartość `1`, ponieważ reszta z dzielenia całkowitego `10` przez `3` wynosi `1`.

Dlaczego jednak `10` podzielone przez `4` dało `2`, a nie `2.5`? Otóż, operator dzielenia / zwraca liczbę całkowitą (zaokrągloną w dół), jeżeli oba jego argumenty są typu całkowitego. Jak zatem wykonać dzielenie, by wynikiem była liczba rzeczywista?

- Możemy albo zapisać jeden z argumentów (o ile jest to literał liczbowy) w taki sposób, aby był on traktowany jako liczba rzeczywista,

  lub

- możemy kazać traktować Javie daną liczbę (literał bądź zmienną) jako liczba rzeczywista wykorzystując tzw. *rzutowanie*.

3.7.1.3 Rzutowanie

Rzutowanie to traktowanie wartości pewnego typu jako wartości innego typu. Rzutowanie jest często stosowane w programowaniu, szczególnie w językach obiektowych, takich jak Java – więcej o rzutowaniu powiemy sobie, gdy zaczniemy naukę klas w Javie.

Gdy co najmniej jeden z operandów będzie liczbą rzeczywistą, całe wyrażenie będzie liczbą rzeczywistą – spójrzmy na przykład poniżej:

Nazwa pliku: OperatoryArytmetyczneRzutowanie.java

```
public class OperatoryArytmetyczneRzutowanie {
  public static void main(String[] args) {
    // dodanie wartosci po przecinku powoduje,
    // ze liczba 10.0 jest traktowana jako rzeczywista
    // w takim przypadku, operatora dzielenia zwroci liczbe rzeczywista
    System.out.println(10.0 / 4);

    // rzutowanie literalu liczby calkowitej na rzeczywista
    // wynik jak powyzej - liczba rzeczywista
    System.out.println((double)10 / 4);

    int liczbaCalkowita = 10;
    int innaLiczbaCalkowita = 4;
    // rzutowanie zmiennej calkowitej na rzeczywista
    // ponownie wynikiem bedzie liczba rzeczywista
    System.out.println(liczbaCalkowita / (double)innaLiczbaCalkowita);
  }
}
```

W pierwszym przykładzie, liczba `10.0` jest traktowana jako liczba rzeczywista, ponieważ ma wartość po przecinku. Co najmniej jeden z argumentów operatora dzielenia jest liczbą rzeczywistą, więc wynikiem będzie także liczba rzeczywista, czyli `2.5`.

W drugim i trzecim przypadku używamy niepoznanej jeszcze składni – w nawiasach wpisany jest typ **`double`** przed wartością `10` (drugi przykład) i przed zmienną `innaLiczbaCalkowita` (trzeci przykład). Ma to na celu wskazanie kompilatorowi, iż, zarówno, liczba `10`, jak i zmienna `innaLiczbaCalkowita`, mają być traktowane jako wartości typu **`double`** (liczba rzeczywista). W ten sposób, na ekranie zobaczymy:

```
2.5
2.5
2.5
```

3.7.1.4 Zmiana priorytetów operatorów za pomocą nawiasów

Spójrzmy na jeszcze jeden przykład – wykorzystujemy nawiasy, by zmienić kolejność wykonywania działań:

Nazwa pliku: OperatoryArytmetyczneNawiasy.java

```
public class OperatoryArytmetyczneNawiasy {
  public static void main(String[] args) {
    System.out.println(2 + 2 * 2);
    System.out.println((2 + 2) * 2);
    System.out.println(2 * 2 / 2 * 2);
    System.out.println(2 * 2 / (2 * 2));
  }
}
```

Powyższy kod spowoduje wypisanie na ekran następujących wartości:

```
6
8
4
1
```

Zwróćmy uwagę, iż w przypadku, gdy operatory mają taki sam priorytet, to operacje są wykonywane od lewej do prawej.

3.7.1.5 Plus jako operator konkatenacji

Operatora + możemy używać nie tylko do dodawania liczb, ale także do łączenia łańcuchów znaków z innymi wartościami. Już kilka razy w ten sposób formowaliśmy komunikaty do wypisania za pomocą instrukcji `System.out.println`:

Nazwa pliku: KonkatenacjaStringow.java

```
public class KonkatenacjaStringow {
  public static void main(String[] args) {
    int x = 25;
    int y = -10;

    System.out.println("Wspolrzednia x ma wartosc " + x +
        ", a wspolrzedna y ma wartosc " + y);
    System.out.println("Druga" + " " + "linia");
  }
}
```

W wyniku działania tego programu, na ekranie zobaczymy:

```
Wspolrzednia x ma wartosc 25, a wspolrzedna y ma wartosc -10
Druga linia
```

Jak widać, możemy łączyć ze sobą zarówno stringi i wartości innego rodzaju – w tym przypadku, zmienne przechowujące liczby typu całkowitego, jak i łańcuchy tekstowe z innymi łańcuchami tekstowymi.

3.7.2 Operatory przypisania

Innym operatorem dwuargumentowym, o którym już wspominaliśmy, jest operator przypisania = . Ma on za zadanie przypisać zmiennej po jego lewej stronie wartość *wyrażenia* po jego prawej stronie:

Nazwa pliku: OperatorPrzypisania.java

```
public class OperatorPrzypisania {
  public static void main(String[] args) {
    int x = 5;
    int y = x * 10;

    double z = (double)y / (x + 3);

    System.out.println(z); // wyswietli 6.25
  }
}
```

Wynik:

```
6.25
```

Wyrażenie jest ważnym pojęciem w programowaniu – jest to dowolna wartość liczbowa, znakowa, logiczna itp., bądź złożone zestawienie wartości i operatorów. Wynikiem wyrażenia jest zawsze finalna wyliczona wartość. Przykładem wyrażeń w powyższym programie są:

```
x * 10

(double)y / (x + 3)
```

Gdy przypisujemy zmiennej wartość, możemy użyć tej samej zmiennej do wyliczenia nowej wartości:

Nazwa pliku: OperatorPrzypisaniaTaSamaZmienna.java

```
public class OperatorPrzypisaniaTaSamaZmienna {
  public static void main(String[] args) {
    int a = 5;
    int b = 10;
    int c = 15;

    c = a + b + c;
  }
}
```

W powyższym przykładzie, przypisujemy zmiennej `c` wartość, która jest sumą trzech zmiennych – jedną z nich jest właśnie zmienna `c`. Istnieje jednak wyjątek – nie możemy użyć tej samej zmiennej

do nadania jej wartości, jeżeli ta zmienna nie została jeszcze zainicjalizowana – próba kompilacji poniższego programu zakończy się błędem:

Nazwa pliku: OperatorPrzypisaniaBrakInicjalizacji.java

```
public class OperatorPrzypisaniaBrakInicjalizacji {
  public static void main(String[] args) {
    int x = 5;

    // blad - zmienna y nie ma jeszcze wartosci
    // wiec nie moze byc czescia wyrazenia
    int y = y * x;
  }
}
```

Kompilator zgłosi następujący błąd:

```
OperatorPrzypisaniaBrakInicjalizacji.java:7: error: variable y might not
have be
en initialized
    int y = y * x;
            ^
1 error
```

Należy tutaj jeszcze zwrócić uwagę, że po lewej stronie operatora przypisania zawsze musi być zmienna – w przeciwnym razie, kompilator zgłosi błąd – nie ma sensu, dla przykładu, przypisywać wartości `10` do wartości `5`:

Nazwa pliku: OperatorPrzypisaniaBezZmiennej.java

```
public class OperatorPrzypisaniaBezZmiennej {
  public static void main(String[] args) {
    // blad! ponizsza linia nie ma sensu
    // do liczby 5 nie mozemy przypisac liczby 10
    5 = 10;
  }
}
```

Próba kompilacji powyższego programu kończy się następującym błędem:

```
OperatorPrzypisaniaBezZmiennej.java:5: error: unexpected type
    5 = 10;
    ^
  required: variable
  found:    value
1 error
```

3.7.2.1 Pomocnicze operatory przypisania

Istnieją także pomocnicze operatory, które można stosować jako "skróty":

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

Pierwszy operator dodaje do zmiennej po jego lewej stronie wartość wyliczoną po jego prawej stronie, czyli zapis:

```
zmienna += 1 + 2;
```

jest równoznaczny z zapisem:

```
zmienna = zmienna + (1 + 2);
```

Analogicznie z każdym z pozostałych operatorów. Przykład:

Nazwa pliku: OperatorPrzypisaniaPomocnicze.java

```
public class OperatorPrzypisaniaPomocnicze {
  public static void main(String[] args) {
    int a = 10;

    a += 100; // a = 10 + 100, wiec a bedzie rowne 110
    a -= 10; // a = a - 10, wiec a bedzie rowne 100
    a *= 5; // a = a * 5, wiec a bedzie rowne 500
    a /= 25 * 10; // a = a / (25 * 10), wiec a bedzie rowne 2
    a %= 2; // a = a % 2, wiec a bedzie rowne 0

    System.out.println(a); // wypisze 0
  }
}
```

Wynik działania:

```
0
```

Tak samo, jak w przypadku operatora przypisania =, po lewej stronie każdego z powyższych operatorów musi znajdować się zmienna.

3.7.3 Operatory jednoargumentowe

Istnieją także operatory jednoargumentowe:

- + oznacza, że liczba jest dodatnia (nieużywany – liczby są domyślnie traktowane jako dodatnie),
- - zamienia wartość wyrażenia na liczbę przeciwną,
- ++ (prefix) – pre-inkrementacja – zwiększanie wartości zmiennej o 1,
- ++ (postfix) – post-inkrementacja – zwiększanie wartości zmiennej o 1,
- -- (prefix) – pre-dekrementacja – zmniejszanie wartości zmiennej o 1,
- -- (postfix) – post-dekrementacja – zmniejszanie wartości zmiennej o 1.

Operatory inkrementacji i dekrementacji to skrótowe zapisy zwiększania bądź zmniejszania wartości zmiennej o `1`. Przykład użycia opeartorów jednoargumentowych:

Nazwa pliku: OperatoryJednoargumentowe.java

```
public class OperatoryJednoargumentowe {
  public static void main(String[] args) {
    int x = 0;
    int y = 0;
    int z = 100;

    x++;
    ++y;
    z = -z; // zmieniamy wartosc na przeciwna

    System.out.println("x ma wartosc " + x);
    System.out.println("y ma wartosc " + y);
    System.out.println("z ma wartosc " + z);
  }
}
```

Wynikiem działania tego programu jest:

```
x ma wartosc 1
y ma wartosc 1
z ma wartosc -100
```

Operator ++ spowodował zmianę wartości zmiennych `x` oraz `y` – obie z nich zwiększył o jeden. Dlaczego istnieją dwa operatory, jeden prefixowy, a drugi postfixowy?

Wersje prefixowe najpierw zmieniają wartość zmiennej, a potem zwracają tę wartość, podczas gdy wersje postfixowe najpierw zwracają wartość zmiennej, a dopiero potem zmieniają zmienną.

Spójrzmy na poniższy przykład:

Nazwa pliku: OperatoryInkrementacji.java

```
public class OperatoryInkrementacji {
  public static void main(String[] args) {
    int x = 5;
    int y = 5;
    int a = x++;
    int b = ++y;

    System.out.println(x);
    System.out.println(a);

    System.out.println(y);
    System.out.println(b);
  }
}
```

Wynik działania tego programu:

```
6
5
6
6
```

Operator postfixowy ++ najpierw zwrócił wartość zmiennej `x` (która wynosiła `5`) i to ta wartość została wpisana do zmiennej `a`, a dopiero później zwiększył wartość przechowywaną w zmiennej `x` o `1` – stąd w pierwszej i drugiej linii wartości to, odpowiednio, `6` oraz `5`.

W przypadku zmiennej `y` użyliśmy operatora prefixowego – **najpierw zmieniona została wartość `y`**, a później, ta już zmieniona wartość zmiennej `y`, została przypisana do zmiennej `b` – dlatego dwie ostatnie linie na ekranie mają wartości, odpowiednio, `6` oraz `6`.

Niektóre operatory operują wyłącznie na zmiennych – zapis `++5`, zakończyłby się błędem kompilacji, ponieważ operatory ++ i -- wymagają, by ich argumentem była zmienna – podobnie, jak w przypadku operatorów przypisania.

3.8 Typ String i wczytywanie danych od użytkownika

Na koniec rozdziału o zmiennych poznamy pierwszy *złożony typ danych* – `String`, oraz nauczymy się jak wczytywać dane od użytkownika.

3.8.1 Typ String

W tym rozdziale poznaliśmy wszystkie 8 prymitywnych typów Javy. Istnieją także typy złożone – przykładem jest typ `String`, który służy do przechowywania ciągu znaków, i udostępnia programistom wiele przydatnych operacji, które można wykonać na łańcuchu tekstowym, jak na przykład:

- zamiana liter na małe/wielkie,
- wycinanie fragmentu tekstu,
- zamiana fragmentu tekstu na inny,
- wiele innych, o których można przeczytać w dokumentacji JavaDoc:

  https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Użycie typu `String` jest proste – definiujemy zmienną o typie `String`, a następnie przypisujemy jej łańcuch tekstowy, otoczony cudzysłowami. Możemy też dodawać do stringu kolejne elementy, niekoniecznie tekstowe. Spójrzmy na przykład:

Nazwa pliku: UzycieTypuString.java

```
public class UzycieTypuString {
  public static void main(String[] args) {
    String imie = "Jan";
    String nazwisko = "Nowak";

    String osoba = imie + " " + nazwisko;
    int wiek = 25;

    String komunikat = osoba + " ma " + wiek + " lat.";

    System.out.println(komunikat);

    // korzystamy z metody toUpperCase, ktora zwraca
    //  string z malymi literami zamienionymi na wielkie
    System.out.println(komunikat.toUpperCase());
  }
}
```

Wynik działania programu:

```
Jan Nowak ma 25 lat.
JAN NOWAK MA 25 LAT.
```

Użyliśmy metody `toUpperCase` klasy (typu) `String`, która zwróciła nam **nowy** łańcuch tekstowy z wszystkimi literami zamienionymi na wielkie litery.

Wartość zmiennej `komunikat` w powyższym przykładzie nie uległa zmianie! Użycie `toUpperCase` spowodowało zwrócenie nowego tekstu z wielkimi literami – oryginał (tzn. zmienna `komunikat`) się nie zmienił.

3.8.2 Wczytywanie danych od użytkownika

Nasze programy będą ciekawsze, jeżeli będą pozwalały użytkownikowi na podanie danych wejściowych, na których będziemy mogli wykonywać różne operacje.

Będziemy w tym celu korzystali z poniższego kodu – należy skopiować zaznaczone fragmenty do własnego programu, by móc wczytywać dane od użytkownika:

Nazwa pliku: PobieranieDanychOdUzytkownika.java

```
import java.util.Scanner;

public class PobieranieDanychOdUzytkownika {
  public static void main(String[] args) {
    // informujemy uzytkownika, co ma zrobic
    System.out.println("Prosze podac imie:");

    // wczytujemy od uzytkownika pojedyncze slowo do zmiennej imie
    String imie = getString();

    // wypisujemy komunikat, uzywajac wczytane od uzytkownika imie
    System.out.println("Witaj, " + imie + "!");

    // kolejny przyklad - tym razem prosimy i wczytujemy liczbe calkowita
    System.out.println("Prosze podac promien kola:");

    int r = getInt();

    double poleKola = 3.14 * r * r;
    System.out.println("Pole kola o promieniu " + r + " wynosi: " + poleKola);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

Nie będziemy na razie zagłębiać się w szczegóły tego kodu – w skrócie:

- w pierwszej linii, za pomocą słowa kluczowego `import`, informujemy kompilator, że chcemy korzystać z funkcjonalności klasy `Scanner`, importując tą klasę – o importach porozmawiamy w jednym z kolejnych rozdziałów,
- definiujemy dwie nowe metody, które korzystają z klasy `Scanner` w celu pobrania od użytkownika wartości liczbowej bądź ciągu znaków, i zwracają je – te metody to `getInt` oraz `getString`. O metodach także porozmawiamy wkrótce.

Jak działa powyższy program?

1. Najpierw wypisany zostaje tekst "*Prosze podac imie:*".
2. Następnie, wykonanie programu dochodzi do linii, w której korzystamy z metody `getString`, która z kolei korzysta z klasy `Scanner` – w tym miejscu, program się zatrzymuje do czasu, aż użytkownik nie wpisze za pomocą klawiatury tekstu i wciśnie przycisk **Enter**.
3. Użytkownik wpisuje tekst, wciska klawisz **Enter**. Wartość wpisana przez użytkownika

zostaje zapisana w zmiennej `imie`. Analogicznie dzieje się podczas pobierania wartości dla promienia koła, który zostaje umieszczony w zmiennej `r`.

Spójrzmy na wynikowe działanie programu (na białym tle zaznaczono dane podane z klawiatury przez użytkownika w trakcie działania programu):

```
Prosze podac imie:
Przemek
Witaj, Przemek!
Prosze podac promien kola:
10
Pole kola o promieniu 10 wynosi: 314.0
```

Dodaj powyższą funkcjonalność wczytywania danych od użytkownika do jednego ze swoich programów i użyj jej, wzorując się na powyższym przykładzie, by pobrać od użytkownika dane.

Aby pobrać od użytkownika wartość, zdefiniuj zmienną, a następnie skorzystaj ze składni:

- jeżeli chcesz pobrać liczbę: `zmiennaLiczbowa = getInt();`
- jeżeli chcesz pobrać słowo: `zmiennaString = getString();`

W kolejnych rozdziałach będziemy często korzystali z powyższych metod.

Zdefiniowana powyżej metoda `getString` pobiera od użytkownika pierwsze wpisane przez niego słowo – jeżeli wpiszemy w linii poleceń tekst np. Jan Nowak, to do naszego programu zwrócone zostanie jedynie pierwsze słowo, czyli "Jan".

3.9 Podsumowanie

3.9.1 Zmienne

- Zmienne w programowaniu służą do przechowywania wartości.
- Zanim będziemy mogli użyć zmiennej, musimy ją *zdefiniować*.
- *Definicja* zmiennej to poinformowanie kompilatora, jaką zmienna będzie miałą nazwę oraz typ.
- Definicja zmiennej odbywa się poprzez podanie typu zmiennej, po którym następuje jej nazwa i średnik – możemy też zdefiniować więcej, niż jedną zmienną na raz:

```
int promienKola;
int x, y;
```

- Zmienna może zostać *zainicjalizowana* wstępną wartością – pomiędzy nazwą zmiennej a średnikiem należy wtedy wpisać znak równości oraz wartość, którą chcemy nadać zmiennej:

```
int promienKola = 8;
```

- Aby zmienić wartość zmiennej, piszemy jej nazwę, znak równości, oraz nową wartość:

```
double poleKola;
poleKola = 3.14 * promienKola * promienKola;
```

- Wartości zmiennych możemy wypisać na ekran za pomocą instrukcji `System.out.println`:

```
System.out.println(poleKola);
```

- Wypisując komunikaty na ekran, możemy łączyć ze sobą łańcuchy tekstowe i inne wartości:

```
System.out.println("Pole kola wynosi: " + poleKola);
```

- Zanim użyjemy zmiennej, musimy jej nadać wartość – w przeciwny razie, próba kompilacji naszego programu zakończy się błędem:

```
public class UzycieNiezainicjalizowanejZmiennej {
  public static void main(String[] args) {
    int x;

    // blad! nie nadalismy zmiennej x jeszcze zadnej wartosci
    System.out.println("Wartosc x wynosi: " + x);
  }
}
```

3.9.2 Nazwy

- Nazwy w języku Java muszą spełniać następujący wymagania:
  - muszą zaczynać się od litery, podkreślenia _ bądź znaku dolara $
  - kolejnymi znakami, poza tym wymienionymi w punkcie 1., mogą być także cyfry,
  - nie mogą być takie same, jak nazwy zastrzeżone w języku Java (np. **class**, **public**, **void** itd.).
- Przykłady poprawnych nazw zmiennych:

```
char jedenZnak;
double poleKola;
int liczba_pracownikow;
double _pi;
int $wynagrodzenieMiesieczne;
double wynikWRozliczeniu360;
```

- Przykłady niepoprawnych nazw zmiennych:

```
// kazda z ponizszych nazw jest niepoprawna i spowoduje blad kompilacji
double 314ToLiczbaPi; // nazwa nie moze zaczynac sie od liczby
double promien kola; // spacja w nazwie jest niedozwolona
int liczba#pracownikow; // znak # nie może wystapic w nazwie
int public; // public to slowo kluczowe w jezyku Java
```

- Małe i wielkie litery są rozróżniane w nazwach zmiennych – poniżej zdefiniowane zostały dwie **różne** zmienne:

```
double pi = 3.14;
double PI = 3.14;
```

- Długość zmiennych może być dowolna.
- Nazwy w kodach źródłowych niosą bardzo dużo informacji, o ile są one odpowiednio dobrane do obiektów, które opisują.
- Przy nazywaniu zmiennych i innych obiektów należy:
  - Używać dobrze dobranych, opisowych nazw, tak, aby zwiększały one czytelność kodu. Zawsze warto zastanowić się nad nazwą.
  - Stosować konwencję Camel-Case – wielka litera powinna rozpoczynać każde słowo w nazwie, poczynając od drugiego słowa, np. `poleKola` bądź `glownyTelefonKontaktowy`.

3.9.3 Typy

- Typ zmiennej to informacja, jakiego rodzaju wartości zmienna będzie mogła przechowywać.
- Typy prymitywne to typy udostępniane przez dany język programowania. Java posiada 8 typów prymitywnych:
  - **boolean** – typ logiczny, przyjmujący wartość **true** bądź **false**,
  - **byte**, **short**, **int**, **long** – typy przechowujące liczby całkowite,
  - **float**, **double** – typy przechowujące liczby rzeczywiste,
  - **char** – typ przechowujący znaki.
- Różne typy przechowujące dane tego samego rodzaju mają różne zakresy wartości, jakie mogą przyjmować kosztem większego miejsca potrzebowanego na ich przechowywanie w pamięci komputera.
- Typ prymitywny określa nie tylko, jakiego rodzaju wartości zmienna może przechowywać, ale także jaki jest zakres tych wartości – dla przykładu, typ **byte** oznacza, że zmienna będzie mogła przechowywać wartości od `-128` do `127`.
- Język Java jest *językiem typowanym*, co oznacza, że zmienne mają typ określony przy ich definiowaniu przez programistę i nie może się on zmienić w trakcie wykonywania programu:

```
int liczbaOkien;

// blad! zmienna liczbaOkien moze przechowywac tylko liczby calkowite!
liczbaOkien = 3.5;
```

- Liczby takie jak `10`, `3.14`, znaki, jak np. `'A'`, `'N'`, tekst ujęty w cudzysłowy np. `"Witaj Swiecie!"` oraz wartości logiczne **true** i **false**, to tzw. ***literały***. Zapisane w kodzie źródłowym oznaczają konkretne wartości, zazwyczaj, typów prymitywnych.
- Literały liczbowe zapisane w kodzie są domyślnie traktowane jako liczby typu **int**. Domyślnym typem wartości rzeczywistych (zmiennoprzecinkowych) jest typ **double**.
- Jeżeli chcemy zapisać w kodzie wartość większą, niż maksymalna liczba z zakresu typu **int** (czyli `2147483647`), to musimy na końcu liczby dodać literę L (od typu **long**), na przykład `10000000000L` – w przeciwnym razie kompilator zaprotestuje.

3.9.4 Stałe

- Stałe mają za zadanie przechowywać raz im nadaną, niezmienną wartość.
- Stałe definiujemy jak zmienne, z dodatkiem słowa kluczowego **final** przed typem stałej.
- Zgodnie z konwencją, nazwy stałych powinny być zapisane wielkimi literami, a kolejne słowa, z których ich nazwa się składa, powinny być oddzielone od siebie znakiem podkreślenia (_), np. `LICZBA_DNI_W_TYGODNIU`.
- Próba zmiany wartości stałej powoduje błąd na etapie kompilacji:

```
final double PI = 3.14;

// blad kompilacji!
```

```
PI = 5;
```

3.9.5 Operatory

- Operatory służą do wykonywania pewnych operacji. Do dyspozycji mamy m. in. operatory arytmetyczne i przypisania.
- Argumenty operatorów, czyli wartości, na których operują, nazywamy *operandami*.
- Operatory mają różne priorytety – nawiasy mogą zostać użyte do zmiany kolejności działania operatorów.
- Podstawowe operatory arytmetyczne to:
  - + dodawanie
  - - odejmowanie
  - * mnożenie
  - / dzielenie całkowite,
  - % modulo – reszta z dzielenia.
- Operator dzielenia wykonuje dzielenie całkowite, jeżeli oba jego operandy są typu całkowitego, tzn. zwraca zaokrągloną w dół wartość całkowitą. Jeżeli któryś z operandów jest typu rzeczywistego, wynik także będzie wartością rzeczywistą.
- **Rzutowanie typów**, czyli konwersję wartości jednego typu na inny, wykonuje się za pomocą składni `(typ)wartosc`, np. `(double)zmiennaTypuCalkowitego`:

```
System.out.println(10 / 4); // wypisze 2 - dzielenie calkowite
System.out.println((double)10 / 4); // wypisze 2.5
```

- Operator przypisania nadaje zmiennym wartość:

```
int bokKwadratu = 5;
int poleKwadratu = bokKwadratu *  bokKwadratu;
```

- Dostępne są też skrótowe operatory += *= -= i inne, które wykonują operację na zmiennej i przypisują jej wartość:

```
// rownoznaczne z bokKwadratu = bokKwadratu + 10
bokKwadratu += 10; // zwieksza bokKwadratu o 10
```

- Do dyspozycji mamy także jednoargumentowe operatory ikrementacji i dekrementacji:
  - ++ prefixowy i postfixowy – zwiększa wartość zmiennej o 1,
  - -- prefixowy i postfixowy – zmniejsza wartość zmiennej o 1,
- Operatory prefixowe różnią się tym od postfixowych, że wartość zmiennej jest zwracana przed lub po zmianie wartości:

```
x = 0;
y = x++;
System.out.println("x = " + x + ", y = " + y); // wypisze x = 1, y = 0

x = 0;
y = ++x;
```

```
System.out.println("x = " + x + ", y = " + y); // wypisze x = 1, y = 1
```

- Operatory przypisania oraz operatory inkrementacji i dekrementacji zawsze wymagają zmiennej. Próba zapisu `5 = 10` lub `20++` spowoduje błąd kompilacji.

3.9.6 Typ String i wczytywanie danych od użytkownika

- `String` to typ złożony, który służy do przechowywania ciągu znaków.
- Stringi można łączyć za pomocą operatora +

```
String imie = "Jan";
String nazwisko = "Nowak";

String osoba = imie + " " + nazwisko;

int wiek = 25;
String komunikat = osoba + " ma " + wiek + " lat.";
```

- `String` udostępnia wiele przydatnych metod, które operują na łańcuchach znaków, na przykład `toUpperCase`, która zamienia małe litery na wielkie:

```
String powitanie = "Witajcie!";
powitanie = powitanie.toUpperCase();

System.out.println(powitanie); // wypisze WITAJCIE!
```

- Poznaliśmy sposób na wczytywanie od użytkownika liczb oraz słów:

```
import java.util.Scanner;

public class PobieranieDanychOdUzytkownika {
  public static void main(String[] args) {
    // przyklad wczytywania boku kwadratu
    System.out.println("Prosze podac bok kwadratu:");

    int bokKwadratu = getInt();
    System.out.println(
      "Pole tego kwadratu wynosi " + (bokKwadratu * bokKwadratu));
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

- Aby wykorzystać wczytywanie danych od użytkownika, należy skopiować do swojego programu kod podświetlony na niebiesko powyżej.
- Aby pobrać od użytkownika wartość, zdefiniuj zmienną, a następnie skorzystaj ze składni:
  - jeżeli chcesz pobrać liczbę: `zmiennaLiczbowa = getInt();`
  - jeżeli chcesz pobrać słowo: `zmiennaString = getString();`
- Zdefiniowana powyżej metoda `getString` pobiera **pierwsze** słowo wpisane przez

użytkownika.

3.10 Pytania

1. Jak definiujemy zmienne?
2. Czy zmiennej trzeba nadać wartość początkową w momencie definicji?
3. Jakie są zasady nazewnictwa zmiennych (i innych obiektów) w Javie?
4. Czy wielkie litery są rozróżniane w nazwach w Javie?
5. Jakie typy podstawowe są dostępne w Javie? Czym się różnią?
6. Jak nazywamy zwykłe wartości zapisane w kodzie, takie jak `3.14`, `25`, `'a'`?
7. Do czego służą stałe, jak się je definiuje i nazywa?
8. Czy wszystkie operatory mają takie same priorytety?
9. Czy można zmieniać priorytet operatorów?
10. Jaki wynik daje użycie operatora dzielenia / gdy jego argumenty (operandy) są liczbami całkowitymi?
11. Co to jest rzutowanie typów?
12. Jak otrzymać w wyniku dzielenia liczbę rzeczywistą?
13. Do czego służą operatory +=, -= itp.?
14. Do czego służą operatory ++ i -- oraz czym się różnią ich post- i pre-fixowe wersje?
15. Jaką wartość będzie miała zmienna `x`, a jaką zmienna `y`, w poniższym przykładzie?

```
int x = 5++;
int y = ++5;
```

16. Do czego służy typ `String`?
17. Jak połączyć ze sobą dwa łańcuchy tekstowe?
18. Co zostanie wypisane w wyniku wykonania poniższego programu?

```
String powitanie = "Witajcie!";
powitanie.toUpperCase();

System.out.println(powitanie);
```

19. Co zostanie wypisane w wyniku działania poniższego programu?

```
public class WypiszX {
  public static void main(String[] args) {
    int x;
    System.out.println("x ma wartosc " + x);
  }
}
```

20. Jaką wartość będzie miała zmienna `liczba`?

```
int liczba = 2.5 * 20;
```

21. Które z poniższych nazw zmiennych są *niepoprawne* i dlaczego?

```
char ZNAK;
int class;
double $saldo;
int liczbaPrzedmiotow#;
int 60godzin;
```

22. Co zostanie wypisane w wyniku działania poniższego programu?

```
public class WypiszXY {
  public static void main(String[] args) {
    int x = 10;
    int y = -5;
    System.out.println("Wspolrzedne X i Y to: " + X + ", " + Y);
  }
}
```

23. Czy poniższy kod skompiluje się poprawnie?

```
char z = "Z";

System.out.println(z);
```

24. Jaka wartość zostanie wypisana?

```
double liczba = (double)15 / 10;

System.out.println(liczba);
```

25. Jaką wartość będzie miała zmienna `y`?

```
public class JakaWartosc {
  public static void main(String[] args) {
    int x = 5;
    int y = x + y;
  }
}
```

3.11 Zadania

3.11.1 Obwód trójkąta z pobranych danych

Napisz program, który pobierze od użytkownika trzy boki trójkąta, policzy jego obwód i wypisze wynik na ekran.

3.11.2 Pobrane słowa w odwrotnej kolejności

Napisz program, który wczyta od użytkownika trzy słowa i wypisze je w odwrotnej kolejności, niż podał je użytkownik, oddzielone przecinkami. Dla przykładu, gdy użytkownik poda:

1. Ala
2. ma
3. kota

To program powinien wypisać *kota, ma, Ala*

3.11.3 Liczba znaków w słowie

Napisz program, który wczyta od użytkownika jeden wyraz i wypisz liczbę znaków, z których się składa. Dla przykładu, dla podanego słowa *nauka* wypisze 5.

Sprawdź w dokumentacji JavaDoc dla typu `String` jak dowiedzieć się z ile znaków składa się tekst przetrzymywany w zmiennej typu `String`:

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

3.11.4 Wynik rzeczywisty

Zmień poniższy kod, by wynik wypisany na ekran nie był liczbą zaokrągloną do całkowitej wartości, lecz zmienną rzeczywistą (z częścią ułamkową):

```
public class ZadaniaWynikRzeczywisty {
  public static void main(String[] args) {
    int x = 5;
    int y = 2;

    double wynik = x / y;

    System.out.println(wynik);
  }
}
```

3.11.5 Wielkie litery

Napisz program, który pobierze od użytkownika słowo i wypisze je z małymi literami zamienionymi na wielkie. Skorzystaj z metody `toUpperCase` typu `String`.

3.11.6 Pole koła o podanym promieniu

Napisz program, który policzy pole koła o promieniu podanym przez użytkownika i wypisze wynik na ekran. Promień koła powienien być liczbą całkowitą – do jego przechowywania użyj zmiennej typu `int`.

4 Rozdział IV – Instrukcje warunkowe

W tym rozdziale:

- poznamy instrukcje warunkowe, dzięki którym będziemy mogli pisać bardziej skomplikowane programy,
- dowiemy się, czym są operatory relacyjne i warunkowe,
- opowiemy sobie o blokach kodu i zakresie zmiennych,
- poznamy instrukcję `switch`,
- zobaczymy, jak używać trój-argumentowego operatora logicznego.

4.1 Podstawy instrukcji warunkowych

Instrukcje warunkowe w programowaniu służą do warunkowego wykonywania instrukcji, w zależności od tego, czy pewien warunek (bądź warunki) jest spełniony, czy nie.

Instrukcje warunkowe sprawdzają *prawdziwość* pewnego warunku bądź warunków, tzn. sprawdzają, czy warunek jest *prawdą* (**true**) bądź *fałszem* (**false**). Warunki, które dają w wyniku wartości **true** bądź **false** zapisujemy za pomocą *operatorów relacyjnych*, o których wkrótce sobie opowiemy.

Spójrzmy na przykład użycia instrukcji warunkowej:

Nazwa pliku: SprawdzMianownik.java

```
public class SprawdzMianownik {
  public static void main(String[] args) {
    double licznik, mianownik;

    licznik = 5;
    mianownik = 0;

    if (mianownik != 0) {
      System.out.println("Wynik: " + licznik / mianownik);
    } else {
      System.out.println("Mianownik nie moze byc = 0!");
    }
  }
}
```

W powyższym przykładzie skorzystaliśmy z instrukcji warunkowej do sprawdzenia, czy mianownik jest różny od `0`. Sprawdzamy tutaj prawdziwość wyrażenia *mianownik jest różny od* `0` przy pomocy nowego operatora *różne od* (!=):

- jeżeli `mianownik` jest różny od `0`, to wykonana zostania instrukcja, która wypisuje na ekran wynik dzielenia zmiennej `licznik` przez `mianownik`,
- w przeciwnym razie, zostanie wykonana instrukcja, która wypisze na ekran komunikat o błędnej wartości mianownika.

Tylko jedna z instrukcji `System.out.println` zostanie wykonana. Ten przykład korzysta z instrukcji warunkowej **if**, a także sekcji **else**.

4.1.1 Składnia instrukcji warunkowych

Spójrzmy na składnię instrukcji warunkowych w języku Java:

```
if (warunek1) {
  instrukcja1;
} else if (warunek2) {
  instrukcja2;
} else if (warunek3) {
  instrukcja3;
} else {
  instrukcja_else;
}
```

Aby skorzystać z instrukcji warunkowej:

- najpierw używamy słowa kluczowego **`if`**,
- następnie, w nawiasach zapisujemy jeden bądź więcej warunków, których wartości mogą przyjmować jedną z dwóch możliwości: **`true`** bądź **`false`**,
- po warunku, w nawiasach klamrowych, umieszczamy instrukcje, które mają zostać wykonane w przypadku, gdy warunek będzie spełniony (będzie miał wartość **`true`**).

Zauważmy, że instrukcje przypisane do danej sekcji warunkowej mają wcięcia (są wysunięte o dwie spacje w prawo).

Spójrzmy na przykład prostej instrukcji warunkowej:

Nazwa pliku: ProstaInstrukcjaIf.java

```
public class ProstaInstrukcjaIf {
  public static void main(String[] args) {
    int x = 5;

    if (x > 0) {
      System.out.println("x jest wieksze od 0.");
    }
  }
}
```

W powyższm przykładzie sprawdzamy, czy zmienna `x` zawiera wartość większą od `0` za pomocą nowego operatora *większe od* (>).

Po instrukcji **`if`** możemy napisać kolejny człon instrukcji **`if`**, poprzedzając go słowem kluczowym **`else`**. Jeżeli pierwszy warunek nie będzie spełniony, to sprawdzony zostanie kolejny warunek itd.:

Nazwa pliku: ProstaInstrukcjaIf.java

```
public class ProstaInstrukcjaIf {
  public static void main(String[] args) {
    int x = 5;

    if (x > 0) {
      System.out.println("x jest wieksze od 0.");
    } else if (x < 0) {
      System.out.println("x jest mniejsze od 0.");
    }
  }
}
```

Zmodyfikowaliśmy przykład – dodaliśmy do niego kolejny warunek, poprzedzony słowem kluczowym **`else`**. Jeżeli warunek `x > 0` okazałby się fałszem, to sprawdzony zostałby warunek `x < 0`.

Na końcu instrukcji warunkowej możemy umieścić sekcję **`else`**, która zostanie wykonana, gdy żaden z warunków nie zostanie spełniony:

Nazwa pliku: ProstaInstrukcjaIf.java

```
public class ProstaInstrukcjaIf {
  public static void main(String[] args) {
    int x = 5;

    if (x > 0) {
      System.out.println("x jest wieksze od 0.");
    } else if (x < 0) {
      System.out.println("x jest mniejsze od 0.");
    } else {
      System.out.println("x jest rowne 0.");
    }
  }
}
```

W finalnej wersji przykładowego programu dodaliśmy do instrukcji warunkowej sekcję **else**, która nie ma żadnego warunku – instrukcje w niej zawarte zostaną wykonane, gdy żaden z poprzednich warunków nie będzie spełniony.

Liczba warunków w instrukcjach warunkowych jest nieograniczona. Dodatkowo, instrukcja warunkowa nie musi zawierać ani sekcji **else if**, ani **else**, ale jeżeli zawiera sekcję **else**, to musi ona być na końcu. Warunki sprawdzane są zawsze w kolejności od pierwszego do ostatniego.

4.1.2 Instrukcje w instrukcjach warunkowych

Liczba instrukcji wykonywanych w ramach danej sekcji warunkowej jest nieograniczona – nie musi to być tylko jedna instrukcja:

Nazwa pliku: DzieleniePrzezZero.java

```
import java.util.Scanner;

public class DzieleniePrzezZero {
  public static void main(String[] args) {
    double licznik, mianownik;

    // uzycie metody getInt omowionej w poprzednim rozdziale
    System.out.println("Podaj licznik:");
    licznik = getInt();

    System.out.println("Podaj mianownik:");
    mianownik = getInt();

    if (mianownik != 0) {
      double wynik = licznik / mianownik;
      System.out.println("Wynik dzielenia: " + wynik);
    } else {
      System.out.println("Mianownik nie moze byc = 0!");
    }
  }
  public static int getInt() {
      return new Scanner(System.in).nextInt();
  }
}
```

W tym przykładzie korzystamy z funkcjonalności do pobierania liczb od użytkownika, poznanej pod koniec rozdziału o zmiennych. Pierwsza sekcja instrukcja warunkowej ma przypisane dwie instrukcje do wykonania – definicję zmiennej `wynik` wraz z jej inicjalizacją oraz wypisanie jej

wartości na ekran.

W opisie składni instrukcji warunkowych zobaczyliśmy, że instrukcje do wykonania w ramach danej sekcji warunkowej muszą być otoczone nawiasami klamrowymi.

Jeżeli jednak do sekcji warunkowej przypisana jest tylko jedna instrukcja, to nie musimy otaczać jej nawiasami klamrowymi – poniższy przykład jest poprawny:

```
public class ProstaInstrukcjaIf {
  public static void main(String[] args) {
    int x = 5;

    if (x > 0)
      System.out.println("x jest wieksze od 0.");
    else if (x < 0)
      System.out.println("x jest mniejsze od 0.");
    else
      System.out.println("x jest rowne 0.");
  }
}
```

W tym przykładzie, instrukcje podlegające pod kolejne sekcje instrukcji `if` nie są otoczone nawiasami klamrowymi.

Mimo, iż w przypadku pojedynczych instrukcji klamry nie są wymagane, to i tak powinniśmy je stosować – taka konwencja panuje wśród programistów języka Java.

Możemy stosować wiele instrukcji warunkowych w naszym kodzie – każdy zestaw powiązanych ze sobą instrukcji warunkowych będzie sprawdzany osobno:

Nazwa pliku: KilkaInstrukcjiWarunkowych.java

```
public class KilkaInstrukcjiWarunkowych {
  public static void main(String[] args) {
    int temperatura = 25;
    int dzienTygodnia = 6;

    if (temperatura >= 20) {
      System.out.println("Cieplo!");
    } else {
      System.out.println("Zimno.");
    }

    if (dzienTygodnia >= 6) {
      System.out.println("Weekend!");
    }
  }
}
```

W tym przykładzie mamy dwie niezależne instrukcje warunkowe – pierwsza sprawdza wartość zmiennej `temperatura`, a druga – wartość zmiennej `dzienTygodnia`. Uruchomienie tego programu spowoduje wypisanie na ekran:

```
Cieplo!
Weekend!
```

4.1.3 Formatowanie instrukcji warunkowych

Instrukcje warunkowe do tej pory zapisywaliśmy w następujący sposób:

```
if (x > 0) {
  System.out.println("x jest wieksze od 0.");
} else if (x < 0) {
  System.out.println("x jest mniejsze od 0.");
} else {
  System.out.println("x jest rowne 0.");
}
```

Formatowanie, jakie użyliśmy:

- klamra otwierająca instrukcje danej sekcji jest w tej samej linii, co warunek:

```
if (x > 0) {
```

- klamry zamykające instrukcje warunkowe są w osobnych liniach,
- jeżeli mam kolejne sekcje warunkowe **else if** bądź sekcję **else**, to umieszczamy słowa kluczowe po klamrach zamykających poprzednie sekcje:

```
} else if (x < 0) {
```

```
} else {
```

- po nawiasie zamykającym warunek, a przed klamrą otwierającą sekcję instrukcji do wykonania, znajduje się jedna spacja:

```
(x > 0) {
```

- instrukcje do wykonania w ramach danej sekcji mają pojedyczne wcięcia (złożone z dwóch spacji):

```
if (x > 0) {
  System.out.println("x jest wieksze od 0.");
} else if (x < 0) {
```

Formatowanie instrukcji warunkowych w taki sposób nie jest wymogiem, ale ogólnie przyjętą konwencją. Moglibyśmy zapisać powyższy kod w następujący sposób i nadal byłoby on poprawny:

```
if (x > 0){
  System.out.println("x jest wieksze od 0."); }
else if (x < 0)
{
System.out.println("x jest mniejsze od 0.");
}
else { System.out.println("x jest rowne 0."); }
```

W ramach spójności i czytelności kodu warto jednak stosować się do ogólnie przyjętego sposobu zapisywania instrukcji warunkowych.

4.2 Operatory relacyjne

W rozdziale o zmiennych poznaliśmy kilkanaście operatorów: arytmetycznych, jednoargumentów i przypisania.

Aby sprawdzić prawdziwość warunków w instrukcjach warunkowych, korzystamy z kolejnego rodzaju operatorów: *operatorów relacyjnych*. Są to operatory dwuargumentowe, które porównują wartości swoich argumentów, i zwracają jedną z wartości: **true** bądź **false**.

Do dyspozycji mamy następujace operatory relacyjne:

- < mniejsze niż
- > większe niż
- <= mniejsze bądź równe
- >= większe bądź równe
- == równe (dwa znaki równa się – nie mylić z operatorem przypisania =)
- != nierówne

Korzystając z powyższych operatorów nie wolno oddzielić od siebie znaków, z których się składają, znakiem spacji. Zapis < = jest nieprawidłowy – poprawny zapis to <=.

Spójrzmy na przykład wykorzystania powyższych operatorów:

Nazwa pliku: OperatoryRelacyjne.java

```
import java.util.Scanner;

public class OperatoryRelacyjne {
  public static void main(String[] args) {
    int liczba;

    System.out.println("Podaj liczbe:");
    liczba = getInt();

    if (liczba < 0) {
      System.out.println("Podales liczbe ujemna.");
    } else if (liczba == 0) {
      System.out.println("Podales zero.");
    } else {
      System.out.println("Podales liczbe dodatnia");
    }

    int promien;

    System.out.println("Podaj promien kola:");
    promien = getInt();

    if (promien <= 0) {
      System.out.println("Nieprawidlowy promien: " + promien);
    } else {
      double obwodKola = 2 * 3.14 * promien;
      System.out.println("Obwod tego kola wynosi: " + obwodKola);
    }
  }
```

```
  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

W zaznaczonych liniach skorzystaliśmy z kilku operatorów relacyjnych w celu sprawdzenia wartości pobranych od użytkownika.

Dzięki wykorzystaniu instrukcji warunkowych i operatorów relacyjnych mogliśmy zwalidować dane pobrane od użytkownika. Sprawdziliśmy, czy promień koła jest poprawny, to znaczy nie jest mniejszy bądź równy `0`. Tylko w takim przypadku chcemy liczyć obwód koła – dla ujemnej wartości promienia nie ma to sensu.

Przykładowe uruchomienie powyższego programu:

```
Podaj liczbe:
10
Podales liczbe dodatnia
Podaj promien kola:
3
Obwod tego kola wynosi: 18.84
```

4.3 Typ boolean

Jednym z typów podstawowych języka Java, które poznaliśmy w rozdziale o zmiennych, był typ **boolean**.

Zmienne typu **boolean** mogą przechowywać tylko jedną z dwóch możliwych wartości – **true** bądź **false**:

Nazwa pliku: ZlaWartoscBoolean.java

```
public class ZlaWartoscBoolean {
  public static void main(String[] args) {
    boolean zmienna;

    // blad kompilacji
    // zmienna moze przechowywac tylko wartosc true/false!
    zmienna = 5;
  }
}
```

Próba kompilacji powyższego programu zakończy się błędem, ponieważ zmienne typu **boolean** nie mogą przechowywać liczb:

```
ZlaWartoscBoolean.java:7: error: incompatible types: int cannot be
converted to boolean
    zmienna = 5;
              ^
1 error
```

Zmienne typu **boolean** przydają się do przechowywania informacji typu prawda/fałsz, np. `czyUzytkownikZalogowany`, `czyLiczbaParzysta`, `czyZakonczycProgram`.

Spójrzmy na przykład programu, który zapisuje w zmiennej typu **boolean** informację, czy podana przez użytkownika liczba jest parzysta, czy nie:

Nazwa pliku: CzyLiczbaParzysta.java

```
import java.util.Scanner;

public class CzyLiczbaParzysta {
  public static void main(String[] args) {
    int liczba;
    boolean czyParzysta;

    System.out.println("Prosze podac liczbe:");
    liczba = getInt();

    if (liczba % 2 == 0) {
      czyParzysta = true;
    } else {
      czyParzysta = false;
    }

    System.out.println("Czy liczba parzysta: " + czyParzysta);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Po pobraniu od użytkownika liczby, korzystamy z operatora % (modulo, reszta z dzielenia) i wynik przyrównujemy do liczby `0`:

```
if (liczba % 2 == 0) {
```

Jeżeli `liczba` jest parzysta, to operator % zwróci `0`, które zostanie przyrównane do `0` i warunek będzie spełniony – w tym przypadku, zmiennej `czyParzysta` nadajemy wartość **`true`**. W przeciwnym razie, gdy `liczba` będzie nieparzysta, operator % zwróci `1`, jako reszta z dzielenia, więc wykonana zostania instrukcja związana z sekcją **`else`** – zmiennej `czyParzysta` przypiszemy **`false`**.

Na końcu programu wypisujemy na ekran wartość zmiennej `czyParzysta`. Przykładowe wykonanie programu:

```
Prosze podac liczbe:
5
Czy liczba parzysta: false
```

Powyższy program jest poprawny, ale moglibyśmy skorzystać z pewnej właściwości operatorów relacyjnych. Jak już wspomnieliśmy, operator relacyjny zwraca jedną z dwóch możliwych wartośći: **`true`** bądź **`false`**.

Są to dokładnie takie same wartości, jakie może przechowywać typ **`boolean`**! Czy nie moglibyśmy w takim razie przypisać wyniku działania operatora relacyjnego do zmiennej typu **`boolean`**? Możemy – spójrzmy na przykład powyższego programu korzystającego z tej właściwości:

Nazwa pliku: CzyLiczbaParzystaWersja2.java

```
import java.util.Scanner;

public class CzyLiczbaParzystaWersja2 {
  public static void main(String[] args) {
    int liczba;
    boolean czyParzysta;

    System.out.println("Prosze podac liczbe:");
    liczba = getInt();

    czyParzysta = liczba % 2 == 0;

    System.out.println("Czy liczba parzysta: " + czyParzysta);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

W tej wersji programu, zamiast skorzystać z instrukcji warunkowej, przypisaliśmy do zmiennej `czyParzysta` wynik wyrażenia:

```
liczba % 2 == 0;
```

Wyrażenie to ma wartość **`false`**, jeżeli `liczba` jest nieparzysta, a wartość **`true`**, gdy `liczba` jest parzysta.

Możemy w ten sposób przypisywać do zmiennych typu **`boolean`** wyniki różnego rodzaju porównań korzystających z operatorów relacyjnych – spójrzmy na przykład:

Nazwa pliku: TypBooleanOperatoryRelacyjne.java

```
public class TypBooleanOperatoryRelacyjne {
  public static void main(String[] args) {
    int pewnaLiczba = 100;
    int temperatura = 10;
    int innaLiczba = -1;
    int dzienTygodnia = 3;

    boolean czyUjemna;
    boolean czyJestCieplo;
    boolean czyRozneOdZero;
    boolean czySobota;

    czyUjemna = pewnaLiczba < 0;
    czyJestCieplo = temperatura >= 20;
    czyRozneOdZero = innaLiczba != 0;
    czySobota = dzienTygodnia == 6;
  }
}
```

Do czterech zmiennych typu **boolean** przypisaliśmy wyniki porównywania kilku zmiennych do różnych wartości. Zwróćmy jeszcze raz uwagę, że do przyrównania wartości do innej wartości używamy dwóch znaków ==. Pojedynczy znak = to operator przypisania.

4.4 Warunki instrukcji if

Na początku rozdziału o instrukcjach warunkowych wspomnieliśmy, że warunek instrukcji warunkowej musi mieć wartość *prawda* (**true**) lub *fałsz* (**false**).

Oznacza to, że w warunku instrukcji warunkowej musimy umieścić wyrażenie, którego wartość będzie miała wartość **true** bądź **false**.

Możemy, tak jak już widzieliśmy, zapisać wyrażenie z użyciem operatora relacyjnego:

Nazwa pliku: SprawdzMianownik.java

```
public class SprawdzMianownik {
  public static void main(String[] args) {
    double licznik, mianownik;

    licznik = 5;
    mianownik = 0;

    if (mianownik != 0) {
      System.out.println("Wynik: " + licznik / mianownik);
    } else {
      System.out.println("Mianownik nie moze byc = 0!");
    }
  }
}
```

Możemy też skorzystać ze zmiennej typu **boolean** i przyrównać jej wartość do wartości **true** bądź **false**:

Nazwa pliku: BooleanWIf.java

```
public class BooleanWIf {
  public static void main(String[] args) {
    boolean czyPadaDeszcz = false;

    if (czyPadaDeszcz == true) {
      System.out.println("Wez parasol!");
    } else {
      System.out.println("Zostaw parasol w domu.");
    }
  }
}
```

Powyższy warunek `czyPadaDeszcz == true` jest co prawda poprawny, ale jest nadmiarowy – w końcu sama zmienna `czyPadaDeszcz` niesie już informację typu **true** / **false** – możemy więc zapisać powyższy warunek w następujący, skrócony sposób:

Nazwa pliku: BooleanWIf.java

```
public class BooleanWIf {
  public static void main(String[] args) {
    boolean czyPadaDeszcz = false;

    if (czyPadaDeszcz) {
      System.out.println("Wez parasol!");
    } else {
      System.out.println("Zostaw parasol w domu.");
    }
  }
}
```

Usunęliśmy operator relacyjny == przyrównujący wartość zmiennej `czyPadaDeszcz` do wartości **`true`**. Kod jest poprawny, ponieważ warunek instrukcji warunkowej to nadal jedna z dwóch możliwych wartości – **`true`** bądź **`false`**, ponieważ takie właśnie wartości mogą przechowywać zmienne typu **`boolean`**. Kod działa tak samo, jak jego poprzednia wersja, ale jest krótszy.

W ten właśnie sposób zapisujemy zawsze warunki w instrukach **`if`**, jeżeli sprawdzamy wartość zmiennej typu **`boolean`** – nie powinniśmy nigdy zapisywać warunków ze zmiennymi typu **`boolean`** w postaci `zmiennaTypuBoolean == true` (bądź **`false`**).

4.5 Operatory warunkowe i operator logiczny !

Instrukcje warunkowe byłyby bardziej przydatne, gdybyśmy mogli zapisywać bardziej złożone warunki – możemy to osiągnąć dzięki operatorom warunkowym `&&` oraz `||`. Do dyspozycji mamy także operator logiczny `!`, który zmienia wartość wyrażenia na przeciwne:

- `||` (or) – logiczne "lub" – zwraca prawdę (**`true`**), jeżeli chociaż jeden z jego argumentów jest prawdą, a w przeciwnym razie fałsz (**`false`**).
- `&&` (and) – logiczne "i" – zwraca prawdę (**`true`**), jeżeli każdy z jego argumentów jest prawdą, a w przeciwnym razie fałsz (**`false`**).
- `!` (not) – zwraca zaprzeczenie wartości: jeżeli jest nieprawdziwa (**`false`**), zwraca prawdę, a gdy jest prawdziwa (**`true`**), zwraca fałsz.

Aby zapisać bardziej skomplikowany warunek w instrukcji **`if`**, łączymy ze sobą kolejne warunki przy użyciu operatora `&&` bądź `||`. Warunki mogą być dowolnie złożone i używać każdego z operatorów dowolną liczbę razy.

Spójrzmy na poniższy przykład – pobieramy od użytkownika dwa boki prostokąta i liczymy jego pole, ale tylko w przypadku, gdy obie podane liczby są poprawnymi bokami prostokąta:

Nazwa pliku: PoleProstokata.java

```
import java.util.Scanner;

public class PoleProstokata {
  public static void main(String[] args) {
    int a, b;

    System.out.println("Podaj pierwszy bok prostokata:");
    a = getInt();
    System.out.println("Podaj drugi bok prostokata:");
    b = getInt();

    // uzywamy operatora && - warunek bedzie spelniony,
    // gdy zarowno zmienna a, jak i b, beda wieksze od 0
    if (a > 0 && b > 0) {
      System.out.println("Pole wynosi " + a * b);
    } else {
      System.out.println("Nieprawidlowe dane.");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Instrukcja warunkowa posiada złożony warunek – sprawdzamy zarówno wartość zmiennej `a`, jak i `b`. Warunki są ze sobą złączone za pomocą operatora logicznego `&&` (and), więc całe wyrażenie będzie miało wartość **`true`** tylko w przypadku, gdy zarówno warunek `a > 0`, jak i `b > 0`, będą spełnione.

Spójrzmy na dwa przykładowe uruchomienia powyższego programu:

```
Podaj pierwszy bok prostokata:
5
Podaj drugi bok prostokata:
4
Pole wynosi 20
```

Obie liczby są poprawnymi bokami prostokąta, więc na ekranie zobaczyliśmy policzone pole.

Drugim razem użytkownik podał nieprawidłową wartość dla pierwszego boku – w tym przypadku, na ekranie zobaczyliśmy inny komunikat:

```
Podaj pierwszy bok prostokata:
-2
Podaj drugi bok prostokata:
10
Nieprawidlowe dane.
```

4.5.1 Operator logiczny !

Operator logiczny `!` to jednoargumentowy operator, logiczne zaprzecznie (not), który oczekuje jako argumentu wartość **`true`** bądź **`false`**, i zwraca wartość przeciwną, tzn. dla argumentu **`true`** zwraca wartość **`false`**, a dla **`false`** – zwraca **`true`**.

Spójrzmy na przykład:

Nazwa pliku: OperatorNot.java

```java
public class OperatorNot {
  public static void main(String[] args) {
    boolean czyPadaDeszcz = true;

    if (!czyPadaDeszcz) {
      System.out.println("Sloneczna pogoda.");
    } else {
      System.out.println("Zostaje w domu!");
    }
  }
}
```

W tym przykładzie, w warunku instrukcji **`if`**, sprawdzamy, *czy nie jest prawdą, że pada deszcz*, tzn. czy zmienna `czyPadaDeszcz` jest fałszem? Jeżeli tak, to całe wyrażenie będzie miało wartość **`true`** (zaprzeczeniem **`false`** jest **`true`**) i zobaczymy na ekranie komunikat `"Sloneczna pogoda"`.

Po uruchomieniu programu, na ekranie widzimy jednak:

```
Zostaje w domu!
```

Ponieważ zmienna `czyPadaDeszcz` ma wartość **`true`**. Zaprzeczeniem tej wartości jest **`false`** – warunek **`if`** nie jest spełniony (to znaczy nie ma wartości **`true`**) i przechodzimy do sekcji **`else`**.

Operatora logicznego `!` możemy używać także do zaprzeczania bardziej skomplikowanych wyrażeń:

Nazwa pliku: OperatorNotZaprzeczenieWyrazeniu.java

```
public class OperatorNotZaprzeczenieWyrazeniu {
  public static void main(String[] args) {
    int a, b;

    a = 5;
    b = -10;

    if (!(a > 0 && b > 0)) {
      System.out.println("Niepoprawne dane.");
    } else {
      System.out.println("Pola prostokata = " + a * b);
    }
  }
}
```

W tym przykładzie zaprzeczyliśmy złożonemu wyrażeniu sprawdzającemu, czy boki prostokąta są poprawne. Jeżeli któryś z boków będzie mniejszy bądź równy `0`, czyli warunek:

```
a > 0 && b > 0
```

nie zostanie spełniony i będzie miał wartość `false`, to operator `!` spowoduje, że wartość całego wyrażenia zostanie odwrócona i będzie miała wartość `true` – w takim przypadku, na ekran zostanie wypisany komunikat `"Niepoprawne dane."` – taki właśnie komunikat zobaczymy, gdy uruchomimy powyższy program:

```
Niepoprawne dane.
```

Dodatkowo, operatora logicznego `!` możemy także użyć, by zamienić wartość zmiennej typu `boolean` na przeciwną wartość:

Nazwa pliku: OperatorNotZaprzeczenieBoolean.java

```
public class OperatorNotZaprzeczenieBoolean {
  public static void main(String[] args) {
    boolean czyPadaDeszcz = false;

    czyPadaDeszcz = !czyPadaDeszcz;

    System.out.println("Czy pada? " + czyPadaDeszcz);
  }
}
```

W podświetlonej linii przypisaliśmy do zmiennej `czyPadaDeszcz` wartość przeciwną tej zmiennej – w ten sposób, zmieniliśmy wartość zmiennej `czyPadaDeszcz` na przeciwną – w wyniku uruchomienia tego programu, na ekranie zobaczymy:

```
Czy pada? true
```

4.6 Tablica prawdy operatorów warunkowych

Tablica prawdy to tabela informująca, jaką wartość ma wyrażenie, w którym wykorzystywany jest pewien operator z podanymi mu argumentami.

Poniższa tabela pokazuje możliwe wartości każdego z operatorów poznanych w tym rozdziale w zależności od tego, jakie wartości mają jego argumenty:

x	y	x || y	x && y	!x
false	false	false	false	true
true	false	true	false	false
false	true	true	false	true
true	true	true	true	false

Jak widać, operator `&&` zwraca `true` tylko wtedy, gdy *oba* jego argumenty mają wartość `true`. Z kolei operator `||` zwraca `true`, gdy *chociaż jeden* z argumentów ma wartość `true`.

Przykład dla wartości `a = 0` i `b = 100`:

a > 0	b >= 100	a > 0 || b >= 100	a > 0 && b >= 100	!(a > 0)	!(b >= 100)
false	true	true	false	true	false

Spójrzmy na przykład korzystający z operatorów warunkowych:

Nazwa pliku: DzienTygodniaOperatoryWarunkowe.java

```
import java.util.Scanner;

public class DzienTygodniaOperatoryWarunkowe {
  public static void main(String[] args) {
    int dzienTygodnia;

    System.out.println("Podaj numer dnia tygodnia:");
    dzienTygodnia = getInt();

    if (dzienTygodnia < 1 || dzienTygodnia > 7) {
      System.out.println("Nieprawidlowy dzien tygodnia.");
    } else if (dzienTygodnia == 5) {
      System.out.println("Jutro weeekeend! :)");
    } else if (!(dzienTygodnia == 6 || dzienTygodnia == 7)) {
      System.out.println("Praca :(");
    } else {
      System.out.println("Weekend! :)");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

W tym przykładzie korzystamy zarówno z operatora `||` (logiczne "lub"), jak i operatora zaprzeczenia `!`.

W pierwszej cześci instrukcji `if`:

```
if (dzienTygodnia < 1 || dzienTygodnia > 7) {
```

sprawdzamy, czy dzień tygodnia jest mniejszy od `1` **lub** większy od `7` – w takim przypadku informujemy użytkownika, że podany dzień jest nieprawidłowy.

W jednym z kolejnych warunków:

```
else if (!(dzienTygodnia == 6 || dzienTygodnia == 7)) {
```

sprawdzamy:

1. Czy dzisiejszy dzień to sobota **lub** niedziela?

a następnie:

2. Czy zaprzeczenie powyższego (z pomocą użycia operatora `!`) jest prawdą? Innymi słowy, sprawdzamy, czy dniem tygodnia nie jest ani sobota, ani niedziela.

W wyniku tak zapisanych warunków, instrukcja związana z tą sekcją **`else if`** będzie wykonana wtedy, gdy zmienna `dzienTygodnia` będzie miała wartość z inną niż `6` i `7`.

Dwa przykładowe wykonanie programu:

```
Podaj numer dnia tygodnia:
5
Jutro weeekeend! :)
```

```
Podaj numer dnia tygodnia:
6
Weekend! :)
```

4.7 Nawiasy i priorytety operatorów warunkowych

Gdy będziemy pisać bardziej złożone programy, będzie się zdarzało, że warunki w instrukcjach `if` będą skomplikowane i będą składały się z wielu mniejszych warunków, złączonych ze sobą za pomocą operatorów warunkowych `&&` oraz `||`.

Podczas omawiania operatorów arytmetycznych dowiedzieliśmy się, że różne operatory mają różne priorytety, które wpływają na kolejność ich wykonywania. Kolejność ta może zostać zmieniona przy użyciu nawiasów – wyrażenie `2 + 2 * 2` ma inną wartość, niż wyrażenie `(2 + 2) * 2`.

Spójrzmy na poniższy przykład instrukcji `if`, w której korzystamy z obu operatorów warunkowych: `&&` oraz `||`:

Nazwa pliku: OperatoryWarunkowePriorytety.java

```
public class OperatoryWarunkowePriorytety {
  public static void main(String[] args) {
    int x = -5;
    int y = -10;
    int z = 20;

    if (x > 0 && y > 0 || z > 0) {
      System.out.println("Warunek spelniony.");
    } else {
      System.out.println("Warunek NIE jest spelniony.");
    }
  }
}
```

Pytanie: czy warunek:

```
x > 0 && y > 0 || z > 0
```

jest równoznaczny z:

```
(x > 0 && y > 0) || z > 0
```

czy może z:

```
x > 0 && (y > 0 || z > 0)
```

Czy ma to w ogóle znaczenie?

Tak – powyższe warunki z dodanymi nawiasami to **dwa zupełnie rożne wyrażenia** – spójrzmy, jaką wartość będzie miało każde z tych wyrażeń dla podanych wyżej wartości zmiennych `x`, `y`, `z`:

```
x = -5;
y = -10;
z = 20;
```

Zastąpmy w pierwszym warunku zmienne liczbami, a nastepnie obliczmy wartość każdego z porównań i zastąpmy je wartościami `true`/`false`:

```
( x > 0 &&   y > 0) ||  z > 0
(-5 > 0 && -10 > 0) || 20 > 0
( false && false  ) || true
       false        || true
                    true
```

Po obliczeniu wszystkich wyrażeń, całe wyrażenie ma wartość `true`, ponieważ na końcu zostaliśmy z wyrażeniem `false || true` – co najmniej jeden z argumentów operatora `||` ma wartość `true`, więc całe wyrażenie będzie miało wartość `true`.

Sprwadźmy teraz tym samym sposobem drugie wyrażenie:

```
 x > 0 && (  y > 0 ||  z > 0)
-5 > 0 && (-10 > 0 || 20 > 0)
 false && ( false  || true  )
 false &&          true
     false
```

Objęcie nawiasami dwóch ostatanich członów wyrażenia sprawiło, że wynik całego wyrażenia jest inny, niż pierwszym przypadku.

Jak w takim razie zachowa się nasz program?

Nazwa pliku: OperatoryWarunkowePriorytety.java

```
public class OperatoryWarunkowePriorytety {
  public static void main(String[] args) {
    int x = -5;
    int y = -10;
    int z = 20;

    if (x > 0 && y > 0 || z > 0) {
      System.out.println("Warunek spelniony.");
    } else {
      System.out.println("Warunek NIE jest spelniony.");
    }
  }
}
```

Po uruchomieniu tego programu, na ekranie zobaczymy:

```
Warunek spelniony.
```

Wynika więc z tego, że warunek:

```
x > 0 && y > 0 || z > 0
```

Jest równoznaczny z warunkiem:

```
(x > 0 && y > 0) || z > 0
```

Wynika to z faktu, że operator `&&` ma wyższy priorytet, niż operator `||` – należy o tym pamiętać pisząc instrukcje warunkowe, a najlepiej – stosować nawiasy w celu poprawy czytelności naszego kodu, by nie było niejasności – nigdy nie powinniśmy pisać warunków w stylu:

```
// brakuje nawiasow!
x > 0 && y > 0 || z > 0
```

zamiast tego, powinniśmy ująć w nawiasy odpowiednie wyrażenia, by było jasne, jakie są nasze intencje:

```
(x > 0 && y > 0) || z > 0
// lub
x > 0 && (y > 0 || z > 0)
```

Z nawiasów będziemy także korzystać przy skomplikowanych wyrażeniach w instrukcjach warunkowych:

```
if ((czyUzytkownikZalogowany &&
    (uzytkownikMaRoleDoZapisu || uzytkownikMaRoleAdministratora)) ||
    czySrodowiskoDeveloperskie) {
  // ...
}
```

Zauważmy jeszcze, że przy mniej złożonych warunkach, nawiasy nie są potrzebne:

```
if (x > 0 && y > 0) {
  System.out.println("Poprawne boki prostokata!");
}
```

4.8 Short-circuit evaluation

Pisząc złożone instrukcje warunkowe, często jesteśmy w stanie odpowiedzieć na pytanie, czy całe wyrażenie ma szansę być prawdą bądź nie jeszcze zanim obliczymy każde z wyrażeń, które na ten warunek się składa – spójrzmy na poniższy fragment kodu:

```
if (a <= 0 || b <= 0) {
  System.out.println("Nieprawidlowe dane.");
} else {
  System.out.println("Pole wynosi " + a * b);
}
```

W powyższej instrukcji warunkowej sprawdzamy poprawność wartości boków prostokąta, przechowywanych w zmiennych `a` oraz `b`. Zauważmy, że wystarczy, że pierwsze wyrażenie, `a <= 0`, będzie prawdą, aby całe wyrażenie:

```
a <= 0 || b <= 0
```

miało wartość **`true`**. W takim przypadku, obliczanie wartości wyrażenia `b <= 0` nie ma sensu, bo i tak wiemy już, że niezależnie od wartości wyrażenia `b <= 0`, całe wyrażenie, `a <= 0 || b <= 0` będzie miało wartość **`true`** (ponieważ operatorowi `||` wystarczy jeden argument o wartości **`true`**, aby całe wyrażenie miało wartość **`true`**).

Wartość wyrażenia `b <= 0` musiałaby być wyznaczona tylko w przypadku, gdy wartość wyrażenia `a <= 0` miałaby wartość **`false`** – ponieważ jest jeszcze szansa, że całe wyrażenie a `<= || b <= 0` będzie miało wartość **`true`**, jeżeli `b <= 0` będzie miało wartość **`true`**.

Podobnie sprawa ma się z operatorem `&&` – zapiszmy powyższy fragment kodu z użyciem tego operatora:

```
if (a > 0 && b > 0) {
  System.out.println("Pole wynosi " + a * b);
} else {
  System.out.println("Nieprawidlowe dane.");
}
```

Jeżeli okaże się, że wartość `a > 0` to **`false`**, to nie ma sensu sprawdzać wartości wyrażenia `b > 0` – całe wyrażenie `a > 0 && b > 0` nie ma szansy mieć wartości **`true`**, jeżeli chociaż jeden z jego członów ma wartość **`false`** (operator `&&` zwróci **`true`** tylko, gdy oba operatory będą miały wartość **`true`**).

Jeżeli wartością `a > 0` byłoby **`true`**, to musimy jeszcze sprawdzić drugą wartość: `b > 0` – ponieważ, po obliczeniu pierwszego wyrażenia `a > 0` nadal nie jesteśmy pewni, czy całe wyrażenie `a > 0 && b > 0` będzie miało wartość **`true`** czy **`false`** – aby odpowiedzieć na to pytanie, potrzebujemy wartości wyrażenia `b > 0`.

Wyobraźmy sobie teraz, że mamy złożoną instrukcję warunkową, składającą się z wielu wyrażeń, których obliczenie jest kosztowane z punktu widzenia działania komputera, bądź kolejne warunki w wyrażeniu są zależne od poprzednich.

Dzieki funkcjonalności zwanej *short-circuit evaluation*, nasz program nie będzie zawsze sprawdzał wszystkich warunków, jeżeli jest w stanie wcześniej jednoznacznie wyznaczyć końcową wartość danego wyrażenia. Nie musimy nic robić, aby z tej funkcjonalności korzystać – jest ona po prostu automatycznie stosowana przez Maszynę

Wirtualną Java, która wykonuje kod naszego programu.

Na razie *short-circuit evaluation* nie będzie miało dla nas dużego znaczenia, ale zobaczymy, że jest to istotna funkcjonalność, gdy zaczniemy korzystać z metod oraz klas.

4.9 Zagnieżdżanie instrukcji warunkowych

Instrukcje warunkowe mogą być zagnieżdżone – w bloku powiązanym z daną sekcją instrukcji `if`, możemy umieścić dowolny kod – w tym także kolejne instrukcji warunkowe.

Poniższy program pobiera od użytkownika dwie liczby oraz operację do wykonania na tych liczbach: dodawanie, odejmowanie, dzielenie, bądź mnożenie.

W przypadku, gdy użytkownik zechce wykonać dzielenie, sprawdzamy w zagnieżdżonej instrukcji `if`, czy `liczba2` jest różna od `0` – tylko w takim przypadku będziemy mogli wykonać dzielenie:

Nazwa pliku: WykonajDzialanie.java

```
import java.util.Scanner;

public class WykonajDzialanie {
  public static void main(String[] args) {
    int liczba1, liczba2;
    String operacja;

    System.out.println("Podaj pierwsza liczbe:");
    liczba1 = getInt();

    System.out.println("Podaj druga liczbe:");
    liczba2 = getInt();

    System.out.println("Podaj operacje do wykonania (+ - * /):");
    operacja = getString();

    int wynik = 0;
    boolean nieprawidlowaOperacja = false;

    if (operacja.equals("+")) {
      wynik = liczba1 + liczba2;
    } else if (operacja.equals("-")) {
      wynik = liczba1 - liczba2;
    } else if (operacja.equals("*")) {
      wynik = liczba1 * liczba2;
    } else if (operacja.equals("/")) {
      if (liczba2 != 0) {
        wynik = liczba1 / liczba2;
      } else {
        nieprawidlowaOperacja = true;
        System.out.println("Mianownik nie moze byc zerem!");
      }
    } else {
      nieprawidlowaOperacja = true;
      System.out.println("Nieprawidlowa operacja!");
    }

    if (!nieprawidlowaOperacja) {
      System.out.println(
          liczba1 + " " + operacja + " " + liczba2 + " = " + wynik
      );
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
```

```
  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

Za pomocą głównej instrukcji warunkowej sprawdzamy działanie do wykonania – w zależności od niego, albo dodajemy do siebie pobrane od użytkownika liczby, albo je odejmujemy, mnożymy, bądź dzielimy.

Jeżeli użytkownik poda dzielenie jako operację do wykonania, to korzystamy z kolejnej, zagnieżdżonej instrukcji warunkowej, w której sprawdzamy, czy mianownik (czyli wartość zmiennej `liczba2`) jest równy `0`.

Jeżeli mamy podzielić liczby, a `liczba2` jest równa `0`, lub w przypadku, gdy użytkownik poda nieznane przez nas działanie, ustawiamy wartość zmiennej `nieprawidlowaOperacja` na **`true`**. W ostatniej instrukcji warunkowej sprawdzamy wartość *zaprzeczenia* wartości zmiennej `nieprawidlowaOperacja`, więc wynik działania wypiszemy tylko wtedy, gdy podane działanie było poprawne i mianownik nie był zerem (w przypadku dzielenia).

Zauważmy, że w powyższym przykładzie mamy dwie sekcje **`else`**. Każda instrukcja warunkowa może mieć maksymalnie jedną sekcję **`else`**. Powyższy program nie łamie tej zasady, ponieważ każda z sekcji **`else`** przyporządkowana jest do innej instrukcji warunkowej – jedna do "głównej", a druga do zagnieżdżonej.

W przypadku zagnieżdżonych instrukji warunkowych, klauzule **`else`** oraz **`else if`** odnoszą się do poprzedniej instrukcji **`if`**. Należy używać wcięć (za pomocą tabulatorów bądź spacji) w celu zwiększenie czytelności kodu, jak w powyższym przykładzie – zagnieżdżona instrukcja **`if`** oraz jej klauzula **`else`** mają większe wcięcia, niż zewnętrzna instrukcja **`if`**.

Zwróć uwagę, że powyższym przykładzie używamy metody `equals` na zmiennej `operacja` do sprawdzenia, jaką operację matematyczną podał użytkownik – nie korzystamy tutaj z operatora relacyjnego ==.

Nie powinniśmy stosować operatora == do porównywania wartości zmiennych typu `String` – zamiast tego, zawsze stosujmy metodę `equals`. O powodzie opowiemy sobie dokładnie w rozdziale o klasach.

Metoda `equals` zwraca **`true`**, jeżeli przekazany do niej jako argument ciąg znaków jest taki sam, jak zmienna typu `String`, na rzecz której tą metodę wywołaliśmy. W przeciwnym razie zwracana jest wartość **`false`**. **Uwaga: małe i wielkie litery są rozróżniane**!

Spójrzmy na przykładowe wyniki działania programu – pierwsze uruchomienie:

```
Podaj pierwsza liczbe:
20
Podaj druga liczbe:
4
Podaj operacje do wykonania (+ - * /):
a
Nieprawidlowa operacja!
```

Drugie uruchomienie:

```
Podaj pierwsza liczbe:
10
```

```
Podaj druga liczbe:
0
Podaj operacje do wykonania (+ - * /):
/
Mianownik nie moze byc zerem!
```

Trzecie uruchomienie:

```
Podaj pierwsza liczbe:
5
Podaj druga liczbe:
10
Podaj operacje do wykonania (+ - * /):
-
5 - 10 = -5
```

Zagnieżdżone instrukcje warunkowe mogą także posiadać zagnieżdżone instrukcje warunkowe itd., jednak zbyt wiele zagnieżdżonych instrukcji warunkowych sprawia, że kod jest trudny w zrozumieniu.

4.10 Bloki kodu i zakres zmiennych

Ważnym konceptem w programowaniu jest zakres widoczności i "życia" zmiennych i innych obiektów.

Spójrzmy na poniższy przykład – jaki będzie wynik działania tego programu?

Nazwa pliku: UzyciePrzedDefinicja.java

```
public class UzyciePrzedDefinicja {
  public static void main(String[] args) {
    System.out.println("Liczba = " + liczba);

    int liczba = 5;
  }
}
```

Powyższy program w ogóle się nie skompiluje – kompilator zgłosi błąd, ponieważ próbowaliśmy użyć zmiennej `liczba` zanim ją zdefiniowaliśmy w metodzie `main`:

```
UzyciePrzedDefinicja.java:3: error: cannot find symbol
      System.out.println("Liczba = " + liczba);
                                      ^
  symbol:   variable liczba
  location: class UzyciePrzedDefinicja
1 error
```

Zmienna `liczba` istnieje dopiero od linii, w której zostanie zdefiniowana.

Spójrzmy na kolejny przykład – jaki będzie wynik Twoim zdaniem?

Nazwa pliku: ZmiennaWBlokuIf.java

```
public class ZmiennaWBlokuIf {
  public static void main(String[] args) {
    int licznik = 9;
    int mianownik = 3;

    if (mianownik != 0) {
      double wynik = licznik / mianownik;
    }

    System.out.println("Wynik = " + wynik);
  }
}
```

Program ponownie się nie skompiluje:

```
ZmiennaWBlokuIf.java:10: error: cannot find symbol
      System.out.println("Wynik = " + wynik);
                                      ^
  symbol:   variable wynik
  location: class ZmiennaWBlokuIf
1 error
```

Kompilator protestuje, ponieważ nie wie czym jest "wynik".

Powodem jest fakt, że zmienną `wynik` utworzyliśmy w bloku instrukcji **`if`**. Zakres "życia" zmiennej `wynik` to jedynie blok instrukcji **`if`** – poza nim, zmienna ta przestaje istnieć i jest niedostępna w dalszej części kodu.

Spójrzmy na jeszcze jeden przykład:

Nazwa pliku: ZakresZmiennych.java

```
public static void main(String[] args) {
  int liczba = 1;

  if (liczba > 0) {
    int drugaLiczba = 10;
    // tutaj mamy dostep do zmiennych:
    //  liczba, drugaLiczba

    if (drugaLiczba > 5) {
      int trzeciaLiczba = 25;
      // tutaj mamy dostep do zmiennych:
      //  liczba, drugaLiczba, trzeciaLiczba
    }

    // tutaj mamy dostep do zmiennych:
    //  liczba, drugaLiczba
  } else {
      // tutaj mamy dostep do zmiennych:
      //  liczba
  }

  // tutaj mamy dostep do zmiennej:
  //  liczba
}
```

Kolorami zaznaczone są fragmenty kodu, w których dostępne są poszczególne zmienne:

- Zmienna `trzeciaLiczba` dostępna jest jedynie w zagnieżdżonym bloku instrukcji **`if`**, ponieważ została w nim zdefiniowana – poza tym blokiem, zmienna `trzeciaLiczba` nie istnieje.
- Z kolei zmienna `drugaLiczba` dostępna jest zarówno w "głównym" bloku **`if`**, jak i w drugim, zagnieżdzonym w nim, bloku **`if`**. Zagnieżdżone bloki kodu mają dostęp do zmiennych zdefiniowanych *wcześniej* w zewnętrznych blokach, ale nie na odwrót.
- Zmienna `liczba` dostępna jest w całej metodzie `main`, od jej początku, aż do końca metody, ponieważ została zdefiniowana na samym jej początku.

W sekcji **`else`** głównej instrukcji warunkowej mamy dostęp jedynie do zmiennej `liczba`, ponieważ została zdefiniowana wcześniej w zewnętrznym bloku (którym jest ciało metody `main`). W sekcji **`else`** ani zmienna `drugaLiczba`, ani `trzeciaLiczba`, nie są znane, bo jeżeli doszło do wykonania kodu w sekcji **`else`**, to zmienne te nie zostały nigdy utworzone.

Wrócimy jeszcze do zakresu widoczności oraz czasu "życia" zmiennych w kolejnych rozdziałach.

4.10.1 Bloki kodu w instrukcjach warunkowych

Na początku rozdziału o instrukcjach warunkowych dowiedzieliśmy się, że klamry przed i po instrukcjach przyporządkowanych do danej sekcji instrukcji warunkowej nie są wymagane, jeżeli do wykonania jest tylko jedna instrukcja – poniższy kod:

```
int x = 5;

if (x > 0) {
  System.out.println("x jest wieksze od 0.");
} else if (x < 0) {
  System.out.println("x jest mniejsze od 0.");
} else {
  System.out.println("x jest rowne 0.");
}
```

moglibyśmy zapisać także jako:

```
int x = 5;

if (x > 0)
  System.out.println("x jest wieksze od 0.");
else if (x < 0)
  System.out.println("x jest mniejsze od 0.");
else
  System.out.println("x jest rowne 0.");
```

Wynika to z faktu, że w powyższym przykładzie sekcje warunkowe mają po jednej instrukcji do wykonania. Powinniśmy jednak zawsze stosować klamry { }, nawet wtedy, gdy instrukcja warunkowa ma przypisaną tylko jedną instrukcję do wykonania.

Zobaczmy w poniższym przykładzie-zagadce co może się stać, jeżeli zapomnimy otoczyć więcej niż jedną instrukcję w klamry:

Nazwa pliku: IfBezBloku.java

```
import java.util.Scanner;

public class IfBezBloku {
  public static void main(String[] args) {
    int licznik, mianownik;

    System.out.println("Podaj licznik:");
    licznik = getInt();

    System.out.println("Podaj mianownik:");
    mianownik = getInt();

    double wynik;

    if (mianownik != 0)
      wynik = licznik / mianownik;
      System.out.println("Wynik: " + wynik);
}
  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Jaki będzie wynik działania powyższego programu?

Program nawet się nie skompiluje! Zobaczymy komunikat, który widzieliśmy już w jednym z przykładów w poprzednim rozdziale, gdy poznawaliśmy zmienne:

```
IfBezBloku.java:18: error: variable wynik might not have been initialized
            System.out.println("Wynik: " + wynik);
                                           ^
1 error
```

Dlaczego zobaczyliśmy powyższy błąd? Nie otoczyliśmy obu linijek pod instrukcją `if` klamrami, więc do instrukcji `if` przyporządkowana jest tylko jedna instrukcja – nadanie wartości zmiennej `wynik`.

Jak wiemy z poprzedniego rozdziału, każdej zmiennej, zanim zostanie użyta, musi zostać nadana jakaś wartość.

W powyższym kodzie nadajemy zmiennej `wynik` wartość tylko w przypadku, gdy zmienna `mianownik` ma wartość różną od zera, więc istnieje taka ścieżka wykonania programu, w której zmiennej `wynik` nie zostałaby przypisana żadna wartość przed jej wypisaniem w linijce:

```
System.out.println("Wynik: " + wynik);
```

Kompilator Java jest na tyle "mądry", że jest w stanie wychwycić takie sytuacje i poinformować nas o nich jeszcze na etapie kompilacji.

Gdybyśmy nie zapomnieli o objęciu obu instrukcji (nadania wartości i jej wypisania) w blok za pomocą nawiasów klamrowych { }, to próba wypisania wartości odbyła by się tylko po uprzednim nadaniu jej wartości, dzięki czemu kod byłby poprawny, a kompilator by nie protestował.

4.11 Instrukcja switch

Poza instrukcją **if**, mamy do dyspozycji jeszcze inny rodzaj instrukcji warunkowej – jest to instrukcja **switch**. Jej składnia jest następująca:

```
switch (zmienna) {
  case staleWyrazenie:
    instrukcja;
    break;
  case staleWyrazenie2:
    instrukcja2;
    break;
  default:
    instrukcja3;
}
```

Instrukcja **switch** ma za zadanie przyrównanie wartość zmiennej `zmienna` do podanych *stałych* wartości – jeżeli wartość będzie pasować do którejś z porównywanych wartości, to wykonane zostaną instrukcje z nią powiązane. Zwróćmy uwagę, że instrukcje powiązane z daną sekcją **case** nie są objęte w klamry { }.

Jeżeli natomiast żadna z wartości nie będzie pasować, wykonane zostaną instrukcje z bloku **default**, który nie jest konieczny w instrukcji **switch** – można go pominąć.

*W przykładzie powyżej pojawiło się nowe słowo kluczowe – **break**, o którym zaraz sobie opowiemy.*

Spójrzmy na przykład użycia instrukcji **switch**:

Nazwa pliku: InstrukcjaSwitchDzienTygodnia.java

```
import java.util.Scanner;

public class InstrukcjaSwitchDzienTygodnia {
  public static void main(String[] args) {
    int dzienTygodnia;

    System.out.println("Podaj dzien tygoddnia:");
    dzienTygodnia = getInt();

    switch (dzienTygodnia) {
      case 1:
        System.out.println("Poniedzialek.");
        break;
      case 2:
        System.out.println("Wtorek.");
        break;
      case 3:
        System.out.println("Sroda.");
        break;
      case 4:
        System.out.println("Czwartek.");
        break;
      case 5:
        System.out.println("Piatek.");
        break;
      case 6:
        System.out.println("Sobota.");
        break;
```

```
        case 7:
          System.out.println("Niedziela.");
          break;
        default:
          System.out.println("Nieznany dzien tygodnia!");
      }
    }

    public static int getInt() {
      return new Scanner(System.in).nextInt();
    }
  }
```

W powyższym przykładzie pobieramy od użytkownika numer dnia tygodnia i wypisujemy jego nazwę przy użyciu instrukcji **switch** – porównujemy wartość zmiennej `dzienTygodnia` do liczb w kolejnych sekcjach **case**. Jeżeli żadna z wyszczególnionych liczb nie będzie zgadzała się z wartością zmiennej `dzienTygodnia`, wypisany zostanie komunikat z sekcji **default**.

Spójrzmy na dwa przykładowe wykonania powyższego programu:

```
Podaj dzien tygoddnia:
2
Wtorek.
```

```
Podaj dzien tygoddnia:
-5
Nieznany dzien tygodnia!
```

Instrukcji switch można używać tylko z kilkoma typami, którymi są:

- **byte** i `Byte`
- **short** i `Short`
- **char** i `Character`
- **int** i `Integer`
- `String`
- **enum**

O typach `Byte`, `Short`, **enum** i pozostałych porozmawiamy w jednym z kolejnych rozdziałów.

4.11.1 Użycie break w instrukcji switch

Zadaniem słowa kluczowego **break** jest przerwanie wykonania kolejnych, nastąpujących po nim operacji.

Wracając do przykładu programu z dniem tygodnia – jeżeli usunęlibyśmy słowa kluczowe **break**, a podalibyśmy z klawiatury np. dzień o numerze `3`, to po wypisaniu na ekran informacji `"Sroda"` wypisane zostałyby kolejne linie z komunikatami `"Czwartek"`, `"Piatek"` itd., łącznie z komunikatem wypisywanym pod sekcją **default**. Spójrzmy na poprzedni przykład bez instrukcji **break**:

Nazwa pliku: InstrukcjaSwitchDzienTygodniaBezBreak.java

```
import java.util.Scanner;

public class InstrukcjaSwitchDzienTygodniaBezBreak {
  public static void main(String[] args) {
    int dzienTygodnia;

    System.out.println("Podaj dzien tygoddnia:");
    dzienTygodnia = getInt();

    switch (dzienTygodnia) {
      case 1:
        System.out.println("Poniedzialek.");
      case 2:
        System.out.println("Wtorek.");
      case 3:
        System.out.println("Sroda.");
      case 4:
        System.out.println("Czwartek.");
      case 5:
        System.out.println("Piatek.");
      case 6:
        System.out.println("Sobota.");
      case 7:
        System.out.println("Niedziela.");
      default:
        System.out.println("Nieznany dzien tygodnia!");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Spójrzmy jak zachowa się ta wersja programu na kilku przykładach:

```
Podaj dzien tygoddnia:
1
Poniedzialek.
Wtorek.
Sroda.
Czwartek.
Piatek.
Sobota.
Niedziela.
Nieznany dzien tygodnia!
```

```
Podaj dzien tygoddnia:
3
Sroda.
Czwartek.
Piatek.
Sobota.
Niedziela.
Nieznany dzien tygodnia!
```

```
Podaj dzien tygoddnia:
-5
Nieznany dzien tygodnia!
```

Brak słowa kluczowego **break** powoduje, że po dopasowaniu wartości zmiennej `dzienTygodnia` do wartości z jednej z sekcji **case**, wykonywane są nie tylko powiązane z nią instrukcje, ale także instrukcje wszystkich sekcji **case** po niej następujące.

Użycie **break** nie jest wymagane, ale zazwyczaj jego brak świadczy o tym, że programista zapomniał go po prostu dopisać. Czasami opisane działanie instrukcji **switch** jest przydatne, ale niekiedy powoduje też trudne do wychwycenia błędy.

4.12 Trój-argumentowy operator logiczny

Trój-argumentowy (ternary operator) operator logiczny jest jedynym trój-argumentowym operatorem w języku Java.

Operator ten może posłużyć jako skrócona forma prostych instrukcji warunkowych – jego składnia jest następująca:

```
wyrazenieLogiczne ? wyrazenieGdyPrawda : wyrazenieGdyFalsz
```

Działanie jest następujące: jeżeli wartość wyrażenia `wyrazenieLogiczne` jest prawdą (`true`), to wartością całego wyrażenia będzie `wyrazenieGdyPrawda`. W przeciwnym razie, gdy `wyrazenieLogiczne` jest fałszem (`false`), zwrócone zostanie `wyrazenieGdyFalsz`.

Spójrzmy na poniższy przykład instrukcji `if` oraz użycia trój-argumentowego operatora logicznego w celu zastąpienia instrukcji `if`:

Nazwa pliku: TrojargumentowyOperatorLogiczny.java

```
public class TrojargumentowyOperatorLogiczny {
  public static void main(String[] args) {
    int x = -5;
    int wartoscAbsolutna;

    if (x >= 0) {
      wartoscAbsolutna = x;
    } else {
      wartoscAbsolutna = -x;
    }

    System.out.println(
        "Wartosc absolutna liczby " + x + " wynosi " + wartoscAbsolutna
    );
  }
}
```

Ten prosty program wyznacza wartość absolutną liczby `x`. Spójrzmy na ten sam program, w którym zamiast z instrukcji warunkowej, korzystamy z trój-argumentowego operatora logicznego:

Nazwa pliku: TrojargumentowyOperatorLogiczny.java

```
public class TrojargumentowyOperatorLogiczny {
  public static void main(String[] args) {
    int x = -5;
    int wartoscAbsolutna;

    wartoscAbsolutna = x >= 0 ? x : -x;

    System.out.println(
        "Wartosc absolutna liczby " + x + " wynosi " + wartoscAbsolutna
    );
  }
}
```

Tym razem wartość absolutną wyznaczamy za pomocą trój-argumentowego operatora logicznego. Jeżeli `x >= 0`, to wartością wyrażenia, która zostanie przypisana do zmiennej `wartoscAbsolutna`, będzie po prostu wartość zmiennej `x`. Jeżli jednak `x` jest mniejsze od zera, to do zmiennej `wartoscAbsolutna` zostanie przypisana liczba przeciwna do `x`, czyli `-x`.

Użycie operatora logicznego może być zagnieżdżone:

Nazwa pliku: ZagniezdzonyOperatorTrojargumentowy.java

```
public class ZagniezdzonyOperatorTrojargumentowy {
  public static void main(String[] args) {
    int y = -2;
    String komunikat;

    if (y < 0) {
      komunikat = "mniejsze od zero";
    } else if (y > 0) {
      komunikat = "wieksze od zero";
    } else {
      komunikat = "rowne zero";
    }

    // powyższe, zapisane przy uzyciu zagniezdzonych
    // troj-argumentowych operatorow logicznych
    komunikat = y < 0 ? "mniejsze od zero" :
        (y > 0 ? "wieksze od zero" : "rowne zero");
  }
}
```

W tym przykładzie, za pomocą trój-argumentowego operatora logicznego przypiszemy zmiennej `komunikat` wartość `"mniejsze od zero"`, gdy `y < 0`. Jeżeli jednak wartość zmiennej `y` jest większa bądź równa zero, to ponownie wykorzystany zostanie trój-argumentowy operator logiczny:

```
y > 0 ? "wieksze od zero" : "rowne zero"
```

Jeżeli `y > 0`, to wartością całego wyrażenia będzie `"wieksze od zero"` i taką wartość otrzyma zmienna `komunikat`. W przeciwnym razie, wartością będzie `"rowne zero"`.

Trój-argumentowego operatora logicznego nie należy nadużywać – jeżeli mamy kilka warunków lub jeden skomplikowany warunek, lepiej zastosować instrukcję warunkową **`if`**, by nasz kod był bardziej czytelny.

4.13 Podsumowanie

4.13.1 Instrukcje warunkowe if

- Instrukcje warunkowe służą do warunkowego wykonywania fragmentów kodu:

```
if (mianownik != 0) {
    System.out.println("Wynik: " + licznik / mianownik);
} else {
    System.out.println("Mianownik nie moze byc = 0!");
}
```

- Instrukcje warunkowe sprawdzają *prawdziwość* pewnego warunku bądź warunków, tzn. sprawdzają, czy warunek jest prawdą (**true**) bądź fałszem (**false**) – w powyższym przykładzie:
  - jeżeli `mianownik` jest różny od `0`, to wykonana zostania instrukcja, która wypisuje na ekran wynik dzielenia zmiennej `licznik` przez `mianownik`.
  - w przeciwnym razie, zostanie wykonana instrukcja, która wypisze na ekran komunikat o błędnej wartości zmiennej `mianownik`.
- Instrukcje warunkowe mogą mieć wiele sekcji – kolejne zapisujemy za pomocą **else if**.
- Instrukcje warunkowe mogą mieć jedną opcjonalną sekcję **else**, która wykonywana jest, gdy żaden z warunków nie będzie prawdziwy. Sekcja ta musi być zawsze na końcu instrukcji warunkowej.
- Instrukcje warunkowe mogą obejmować wiele instrukcji – należy je wtedy ująć w nawiasy klamrowe { }.
- Jeżeli do sekcji instrukcji warunkowej przypisana jest tylko jedna instrukcja, to nie musimy używać nawiasów klamrowych. Mimo tego, i tak powinniśmy stosować nawiasy klamrowe w takich przypadkach, by nasz kod był zgodny z ogólnie przyjętą w języku Java konwencją.
- Instrukcje warunkowe mogą być zagnieżdżane – wtedy klauzule **else** oraz **else if** odnoszą się do poprzedniego bloku instrukcji warunkowych.

```
if (operacja.equals("+")) {
    wynik = liczba1 + liczba2;
} else if (operacja.equals("-")) {
    wynik = liczba1 - liczba2;
} else if (operacja.equals("*")) {
    wynik = liczba1 * liczba2;
} else if (operacja.equals("/")) {
    if (liczba2 != 0) {
        wynik = liczba1 / liczba2;
    } else {
        System.out.println("Mianownik nie moze byc zerem!");
    }
} else {
    System.out.println("Nieprawidlowa operacja!");
}
```

- Instrukcje przypisane do danej sekcji warunkowej powinny mieć wcięcia (powinny być wysunięte o np. dwie spacje), jak w przykładzie powyżej, dotyczącym zagnieżdżonych

instrukcji `if`. Instrukcje w zewnętrznej instrukcji `if` mają pojedyczne wcięcie. W zagnieżdżonej instrukcji `if`, przyporządkowane do niej instrukcje wcięte są o kolejny poziom.

- Warunki sprawdzane w instrukcji `if` muszą zawsze dawać w wyniku jedną z dwóch wartości: **prawda** (`true`) bądź **fałsz** (`false`).
- Warunki sprawdzane są zawsze w kolejności od pierwszego do ostatniego. Jeżeli pierwszy warunek nie zostanie spełniony, to sprawdzony będzie kolejny (o ile istnieje kolejna sekcja `else if`) itd., aż znaleziony zostanie warunek, który będzie miał wartość `true`, lub żaden z warunków nie będzie miał wartości `true`. W takim przypadku, wykonane zostaną instrukcje pod sekcją `else`, o ile jest obecna.

4.13.2 Operatory relacyjne i typ boolean

- Operatory relacyjne to operatory dwuargumentowe, które porównują wartości swoich argumentów, i zwracają jedną z wartości: `true` bądź `false`.
- Do dyspozycji mamy następujące operatory relacyjne:
  - `<` mniejsze niż
  - `>` większe niż
  - `<=` mniejsze bądź równe
  - `>=` większe bądź równe
  - `==` równe (nie mylić z operatorem przypisania `=`)
  - `!=` nierówne
- Należy uważać, by nie pomylić operatora porównującego wartości (`==`) z operatorem przypisania (`=`).
- Zmienne typu `boolean` mogą przechowywać tylko jedną z dwóch możliwych wartości – `true` bądź `false`.
- Zmienne typu `boolean` przydają się do przechowywania informacji typu prawda/fałsz, np. `czyUzytkownikZalogowany`, `czyLiczbaParzysta`, `czyZakonczycProgram`.
- Wartość wyrażenia, które wykorzystuje operator relacyjny, ma zawsze wartość `true` lub `false` – dokładnie takie same wartości, jakie mogą przechowywać zmienne typu `boolean` – możemy więc przypisać do zmiennej typu `boolean` wynik takiego wyrażenia:

```
int liczba = 5;

// przypisujemy do zmiennej wynik ponizszego porownania
boolean czyParzysta = liczba % 2 == 0;
```

- Zmienna typu `boolean` może być użyta jako warunek w instrukcji warunkowej `if`:

```
boolean czyPadaDeszcz = false;

if (czyPadaDeszcz) {
  System.out.println("Wez parasol!");
} else {
  System.out.println("Zostaw parasol w domu.");
}
```

- Jeżeli chcemy porównać ze sobą zmienne typu `String` lub zmienną typu `String` z łańcuchem tekstowym (zapisanym w cudzysłowach " "), należy skorzystać z metody `equals` klasy `String` – nie powinniśmy porównywać stringów za pomocą operatora ==, ponieważ wynik będzie inny, niż się spodziewamy!
- Metoda `equals` zwraca **`true`**, jeżeli przekazany do niej jako argument ciąg znaków jest taki sam, jak zmienna typu `String`, na rzecz której tą metodę wywołaliśmy. W przeciwnym razie zwracana jest wartość **`false`**. Uwaga: małe i wielkie litery są rozróżniane! Poniższy kod spowoduje wypisanie na ekran komunikatu `"Rozne"`, ponieważ wielkość znaków jest różna w zmiennej typu `String` oraz w porównywanej do niej wartości `"WITAJ"`:

```
String tekst = "Witaj";

if (tekst.equals("WITAJ")) {
  System.out.println("Takie same");
} else {
  System.out.println("Rozne");
}
```

4.13.3 Operatory warunkowe

- Aby zapisać bardziej skomplikowany warunek w instrukcji **`if`**, łączymy ze sobą kolejne warunki przy użyciu operatora `&&` (and) bądź `||` (lub). Warunki mogą być dowolnie złożone i używać każdego z operatorów dowolną liczbę razy.
- Tablica prawdy to tabela informująca, jaką wartość ma wyrażenie, w którym wykorzystywany jest pewny operator z podanymi mu argumentami.
- Tabela prawdy możliwych wyników zastosowania operatorów `&&`, `||`, oraz `!`

x	y	x || y	x && y	!x
false	false	false	false	true
true	false	true	false	false
false	true	true	false	true
true	true	true	true	false

- Przykład dla wartości **`a = 0`** i **`b = 100`**:

a > 0	b >= 100	(a > 0) || (b >= 100)	(a > 0) && (b >= 100)	!(a > 0)	!(b >= 100)
false	true	true	false	true	false

- Operator `&&` ma wyższy priorytet, niż operator `||` – poniższy kod:

```
x > 0 && y > 0 || z > 0
```

jest równoznaczny z poniższym zapisem, który wykorzystuje nawiasy w celu zwiększenia czytelności:

```
(x > 0 && y > 0) || z > 0
```

- Moglibyśmy zmienić kolejność wykonywania porównań w powyższym wyrażeniu stosując nawiasy:

```
x > 0 && (y > 0 || z > 0)
```

- Jeżeli mamy kilka wyrażeń w warunku i korzystamy z różnych operatorów (`&&` i `||`), to warto stosować nawiasy w celu zwiększenia czytelności kodu.
- Operatopr logiczny `!` to jednoargumentowy operator, logiczne zaprzecznie (not), który oczekuje jako argumentu wartości **`true`** bądź **`false`**, i zwraca wartość przeciwną, tzn. dla argumentu **`true`** zwraca wartość **`false`**, a dla **`false`** – zwraca **`true`**.
- Operatora logicznego `!` możemy użyć w celu odwrócenia wartości w instrukcji **`if`** – w poniższym przykładzie, komunikat `"Sloneczna pogoda."` zostanie wyświetlony, gdy wartość `czyPadaDeszcz` będzie równa **`false`**, bo zaprzeczenie `czyPadaDeszcz` (`!czyPadaDeszcz`) będzie miało wartość **`true`**:

```
boolean czyPadaDeszcz = true;

if (!czyPadaDeszcz) {
  System.out.println("Sloneczna pogoda.");
} else {
  System.out.println("Zostaje w domu!");
}
```

- Operatora logicznego `!` możemy używać także do zaprzeczania bardziej skomplikowanych wyrażeń:

```
int a, b;

a = 5;
b = -10;

if (!(a > 0 && b > 0)) {
  System.out.println("Niepoprawne dane.");
} else {
  System.out.println("Pola prostokata = " + a * b);
}
```

- Dodatkowo, operatora logicznego `!` możemy także użyć, by zamienić wartość zmiennej typu **`boolean`** na przeciwną wartość:

```
boolean czyPadaDeszcz = false;

// zmiennej czyPadaDeszcz zostanie przypisana wartosc true
czyPadaDeszcz = !czyPadaDeszcz;
```

- Pisząc złożone instrukcje warunkowe, często jesteśmy w stanie odpowiedzieć na pytanie, czy całe wyrażenie ma szansę być prawdą bądź nie jeszcze zanim obliczymy każde z wyrażeń, które na ten warunek się składa – w poniższym przykładzie:

```
if (a <= 0 || b <= 0) {
  System.out.println("Nieprawidlowe dane.");
} else {
  System.out.println("Pole wynosi " + a * b);
}
```

W powyższej instrukcji warunkowej wystarczy, że pierwsze wyrażenie, `a <= 0` , będzie prawdą, aby całe wyrażenie `a <= 0 || b <= 0` miało wartość **`true`**. W takim przypadku, obliczanie wartości wyrażenia `b <= 0` nie ma sensu, bo i tak wiemy już, że niezależnie od wartości wyrażenia

`b <= 0`, całe wyrażenie, `a <= 0 || b <= 0` będzie miało wartość **`true`** (ponieważ operatorowi `||` wystarczy jeden argument o wartości **`true`**, aby całe wyrażenie miało wartość **`true`**).

- Dzieki funkcjonalności zwanej *short-circuit evaluation*, nasz program nie będzie zawsze sprawdzał wszystkich warunków, jeżeli jest w stanie wcześniej jednoznacznie wyznaczyć końcową wartość danego wyrażenia. Nie musimy nic robić, aby z tego funkcjonalności korzystać – jest ona po prostu automatycznie stosowana przez Maszynę Wirtualną Java, która wykonuje kod naszego programu. Funkcjonalność ta działa zarówno dla operatora `||`, jak i operatora `&&`.

4.13.4 Bloki kodu i zakres zmiennych

- Blok kodu to fragment zawarty w nawiasach klamrowych.
- Zmienna zdefiniowana np. w bloku instrukcji **`if`** będzie niedostępna, gdy wyjdziemy poza zakres bloku tej instrukcji **`if`** – poniższy kod się nie skompiluje, ponieważ zmienna `wynik` nie istnieje poza blokiem **`if`**, w którym została zdefiniowana – kompilator zgłosi błąd w linijce `System.out.println("Wynik = " + wynik);` ponieważ w tym miejscu nie istnieje już żadna zmienna o nazwie `wynik`:

```
public static void main(String[] args) {
  int licznik = 9;
  int mianownik = 3;

  if (mianownik != 0) {
    double wynik = licznik / mianownik;
  }

  // ponizsza linia spowoduje blad kompilacji!
  // nie ma tu juz zadnej zmiennej o nazwie wynik
  System.out.println("Wynik = " + wynik);
}
```

4.13.5 Instrukcja switch

- Instrukcja **`switch`** służy do porównania wartości zmiennej z wylistowanymi stałymi wartościami – wykonane zostaną te instrukcje, które są skojarzone z dopasowaną wartością:

```
int liczba = 10;

switch (liczba) {
  case 10:
    System.out.println("liczba == 10");
    break;
  case 100:
    System.out.println("liczba == 100");
    break;
  default:
    System.out.println("liczba ma inna wartosc");
}
```

- Instrukcja **`switch`** może mieć klauzulę **`default`**, która zostania wykonana, jeżeli żaden z warunków nie będzie spełniony.
- Słowa kluczowe **`break`** powinny zostać użyte w każdym bloku **`case`**, jeżeli nie chcemy, aby wykonane zostały instrukcje z kolejnych bloków **`case`**, gdy porównywana wartość zostanie dopasowana do któregoś z nich. W poniższym przykładzie nie ma instrukcji **`break`**, przez

co, po dopasowanie wartości `10` do zmiennej `liczba`, wykonywane są wszystkie operacje pod spodem:

```
int liczba = 10;

switch (liczba) {
  case 10:
    System.out.println("liczba == 10");
  case 100:
    System.out.println("liczba == 100");
  default:
    System.out.println("liczba ma inna wartosc");
}
```

```
liczba == 10
liczba == 100
liczba ma inna wartosc
```

- Instrukcji **`switch`** można używać jedynie z następującymi typami:
  - **`byte`** i `Byte`, **`short`** i `Short`, **`char`** i `Character`, **`int`** i `Integer`,
  - `String`
  - **`enum`**

4.13.6 Trój-argumentowy operator logiczny

- Operator ten to skrócona forma prostych instrukcji warunkowych – jego składnia jest następująca:

```
wyrazenieLogiczne ? wyrazenieGdyPrawda : wyrazenieGdyFalsz
```

- W poniższym przykładzie przypiszemy do zmiennej `wartoscAbsolutna` wartość `x`, jeżeli `x` jest liczbą dodatnią. W przeciwnym razie, przypiszemy do zmiennej `wartoscAbsolutna` przeciwieństwo wartości zmiennej `x`:

```
int x = 5;
int wartoscAbsolutna;

wartoscAbsolutna = x > 0 ? x : -x;
```

4.14 Pytania

1. Spójrz na poniższe fragmenty kodu i odpowiedz na pytanie, czy są one poprawnie zapisanym kodem źródłowym Java?

```
int x = 5;

if x == 5 {
  System.out.println("x = 5");
}
```

```
int z = 5;

if (z == 5) {
  System.out.println("z = 5");
}
else System.out.println("z != 5");
else if (z < 5) {
  System.out.println("z < 5");
}
```

```
int a = 5;

if (a) {
  System.out.println("a = 5");
}
```

```
boolean b = true;

if (b) {
  System.out.println("Warunek spelniony.");
}
```

```
int y = 5;

if (y = 5) {
  System.out.println("y = 5");
}
```

```
int c = 5;
boolean czyDodatnia = c > 0;

if (czyDodatnia) {
  System.out.println("Warunek spelniony.");
}
```

2. Jaki będzie wynik uruchomienia poniższego programu, gdy zmienna `x` będzie równa:
    1. `10`
    2. `20`
    3. `30`
    4. Będzie miała inną wartość.

```
switch (x) {
  case 10:
    System.out.println("10");
  case 20:
    System.out.println("20");
    break;
  case 30:
    System.out.println("30");
  default:
    System.out.println("Inna wartosc.");
}
```

3. Jaki będzie wynik działania poniższego programu?

```
double liczba = 50;

switch (liczba) {
  case 0:
    System.out.println("0");
    break;
  case 50:
    System.out.println("50");
    break;
  case 100:
    System.out.println("100");
    break;
  default:
    System.out.println("Inna wartosc.");
}
```

4. Czy poniższa instrukcja `if` oraz użycie trój-argumentowego operatora logicznego są sobie równoważne?

```
int z = 5;
int wartoscAbsolutna;

if (z > 0) {
  wartoscAbsolutna = z;
} else {
  wartoscAbsolutna = -z;
}

int wartoscAbsolutna2 = z > 0 ? -z : z;
```

5. Jak porównać ze sobą dwa łańcuchy tekstowe (`String`)?

6. W jaki sposób możnaby skrócić poniższy kod?

```
int x = 5;
boolean czyDodatnia;

if (x > 0) {
  czyDodatnia = true;
} else {
  czyDodatnia = false;
}
```

7. Jaka będzie wartość zmiennej `pewnaZmienna`?

```
boolean pewnaZmienna ;

pewnaZmienna = !pewnaZmienna;
```

8. Czy operatora = można używać do porównywania wartości?
9. Jaka będzie wartość poniższego wyrażenia, jeżeli `x = -5`, `y = -10`, `z = 20` ?

```
x > 0 && y > 0 || z > 0
```

10. Jakie wartości mogą mieć zmienne typu **`boolean`**?
11. Jeżeli mamy warunek `x > 0 && y > 0`, to czy wartość wyrażenia `y > 0` będzie zawsze obliczana? Jeśli nie, to dlaczego i w jakim przypadku?
12. Jaki będzie wynik działania poniższego programu (załóż, że `getInt` zwraca liczbę pobraną od użytkownika)?

```
int x;
x = getInt();

if (x >= 0) {
  int poleKwadratu = x * x;
}

System.out.println("Pole kwadratu wynosi: " + poleKwadratu);
```

13. Dla jakich wartości argumentów operatory && oraz || zwracają **`true`**? A dla jakiego argumentu zwraca wartość **`true`** operator ! (zaprzeczenie logiczne)?
14. Który z operatorów ma wyższy priorytet: || czy && ?
15. Co zostanie wypisane w wyniku uruchomienia poniższego programu (załóż, że `getInt` zwraca liczbę pobraną od użytkownika)?

```
int x;
x = getInt();

if (x < 0)
  System.out.println("x jest mniejsze od 0");
  x = -x;

System.out.println("Wartosc absolutna pobranej liczby to: " + x);
```

16. Co zostanie wypisane i dlaczego?

```
String komunikat = "Bedzie padac";

if (komunikat.equals("bedzie padac")) {
  System.out.println("Wez parasol!");
} else {
  System.out.println("Sloneczna pogoda.");
}
```

17. Jaki będzie wynik działania poniższego fragmentu kodu (załóż, że `getInt` zwraca liczbę pobraną od użytkownika)?

```
int x;
int poleKwadratu;

x = getInt();

if (x > 0) {
  System.out.println("x jest wieksze od zero.");
  poleKwadratu = x * x;
}

System.out.println("Pole kwadratu wynosi: " + poleKwadratu);
```

4.15 Zadania

4.15.1 Czy liczba podzielna przez trzy

Napisz program, który wczyta od użytkownika liczbę i wypisze, czy jest podzielna bez reszty przez `3`. Skorzystaj z operatora reszty z dzielenia – jeżeli reszta z dzielenia jest równa `0`, to liczba jest podzielna przez `3`.

4.15.2 Czy można zbudować trójkąt

Napisz program, który wczyta od użytkownika trzy liczby i odpowie na pytanie, czy można z nich zbudować trójkąt (suma każdych dwóch boków powinna być większa od trzeciego boku).

4.15.3 Wypisz największą z dwóch liczb

Napisz program, który pobierze od użytkownika dwie liczby i wypisze największą z nich.

4.15.4 Wypisz największą z trzech liczb

Napisz program, który pobierze od użytkownika trzy liczby i wypisze największą z nich.

4.15.5 Zamień liczbę na nazwę miesiąca

Napisz program, który pobierze od użytkownika numer miesiąca i wypisze jego nazwę, lub komunikat `"Nieprawidlowy numer miesiaca"`, jeżeli podany numer będzie spoza zakresu `1..12`. Skorzystaj z instrukcji **`switch`**.

4.15.6 Sprawdź imię

Napisz program, który pobierze od użytkownika jego imię i odpowie na pytanie, czy jego imię jest takie samo, jak Twoje (załóżmy, że użytkownik podaje swoje imię bez polskich znaków).

Uwaga! Pamiętaj, aby skorzystać z metody `equals` typu `String` zamiast porównywać stringi za pomocą operatora == !

4.15.7 Czy pełnoletni

Napisz program, który pobiera wiek od użytkownika. Zapisz w zmiennej typu **`boolean`** informację, czy użytkownik jest pełnoletni, czy nie. Skorzystaj z trój-argumentowego operatora warunkowego. Wypisz wynik zdefiniowanej zmiennej typu **`boolean`** na ekran.

4.15.8 Czy rok przestępny

Napisz program, który pobierze od użytkownika rok i odpowie na pytanie, czy podany rok jest rokiem przestępnym, czy nie. Wskazówka: rok jest rokiem przestępnym, jeżeli:

- dzieli się przez `4` i nie dzieli się przez `100`

  lub

- dzieli się przez `400`.

5 Rozdział V – Pętle

W tym rozdziale:

- dowiemy się, czym są pętle w programowaniu,
- poznamy trzy z czterech dostępnych w języku Java rodzajów pętli: `for`, `while`, oraz `do`..`while`,
- dowiemy się, do czego służą słowa kluczowe `break` oraz `continue`,
- zobaczymy, jak możemy wykorzystywać pętle do pracy z łańcuchami tekstowymi.

5.1 Czym są pętle?

Pętle w programowaniu służą do wykonywania zestawu instrukcji tak długo, jak określony warunek jest prawdziwy, to znaczy, jego wartość to `true`. Każdy obieg pętli nazywamy *iteracją*.

Warunek pętli zapisujemy tak samo, jak warunki w instrukcjach warunkowych:

- korzystamy w nich z operatorów relacyjnych,
- bardziej złożone warunki możemy zapisać używając operatorów warunkowych && oraz ||
- możemy też korzystać ze zmiennych typu `boolean`,
- końcową wartością wyrażenia będącego warunkiem pętli musi być jedną z dwóch wartości: `true` bądź `false`.

Język Java posiada cztery rodzaje pętli, które różnią się od siebie składnią, zastosowaniem i sposobem działania – najpierw poznamy trzy z nich, a z ostatnią zaznajomimy się w rozdziale o tablicach.

5.2 Pętla while

Pierwszym rodzajem pętli, który poznamy, jest pętla **while**.

Pętla ta wykonuje przyporządkowane jej instrukcje, nazywane *ciałem pętli*, zawarte w nawiasach klamrowych { }, dopóki warunek pętli jest spełniony, to znaczy ma wartość **true**. Jej składnia jest następująca:

```
while (warunek) {
  instrukcja1;
  instrukcja2;
  // ...
  instrukcjaN;
}
```

Zauważmy, że instrukcje, które pętla ma wykonywać (ciało pętli), mają wcięcie.

Pętla rozpoczyna się od słowa kluczowego **while**, po którym, w nawiasach, należy zawrzeć warunek wykonywania pętli, np. `x <= 5`. W nawiasach klamrowych należy umieścić instrukcje (bądź instrukcję), które mają się wykonać w każdym obiegu pętli (w każdej *iteracji* pętli). Dopóki warunek pętli będzie spełniony (będzie prawdą), instrukcje objęte w klamry będą wykonywane.

Pętla **while** działa w następujący sposób:

1. Sprawdzany jest warunek pętli – jeżeli jest spełniony, tzn. jego wartość to **true**, to przechodzimy do punktu 2. Jeżeli warunek nie jest spełniony, tzn. jego wartość to **false**, to przechodzimy do punktu 3.
2. Wykonywane są instrukcje przyporządkowane do pętli (ciało pętli), czyli instrukcje zawarte w nawiasach klamrowych { }. Po wykonaniu wszystkich instrukcji, wracamy do punktu 1.
3. Pętla kończy działanie.

Bardzo ważnym aspektem działania pętli **while** jest to, że pętla **while** może nigdy nie wykonać swoich instrukcji, jeżeli już przy pierwszym sprawdzeniu jej warunek będzię miał wartość **false** – z drugiej jednak strony, jeżeli warunek będzie zawsze spełniony, to program się zawiesi – za chwilę zobaczymy przykłady z takimi przypadkami.

Tak samo, jak w przypadku instrukcji warunkowych, jeżeli pętla ma wykonywać tylko jedną instrukcję, to nie musi ona być otoczona nawiasami klamrowymi { }. Jednak, podobnie, jak w przypadku instrukcji warunkowych, zawsze powinniśmy stosować klamry { } nawet w przypadku jednej instrukcji – taka konwencja jest zazwyczaj stosowana wśród programistów.

5.2.1 Przykład: wypisywanie ciągu liczb

Po teorii, czas na praktykę. Spójrzmy na przykład programu, który korzysta z pętli **while** do wypisania na ekran liczb od 1 do 5:

Nazwa pliku: WypiszLiczbyOd1Do5.java

```
public class WypiszLiczbyOd1Do5 {
  public static void main(String[] args) {
    int x = 1;

    while (x <= 5) {
      System.out.println(x);
      x++;
    }
  }
}
```

W wyniku działania tego programu, na ekranie zobaczymy:

```
1
2
3
4
5
```

Na początku programu definiujemy zmienną `x` – będziemy sprawdzać jej wartość w warunku pętli. Inicjalizujemy ją wartością 1.

Warunek pętli **while** jest następujący:

```
x <= 5
```

Naszą pętla będzie wykonywać się dopóki wartość zmiennej `x` będzie mniejsza bądź równa 5. Zmienna `x` ma na początku wartość 1, więc powyższe wyrażenie ma wartość **true**.

Pętla ma za zadanie wykonywać dwie następujące instrukcje:

```
System.out.println(x);
x++;
```

W każdej iteracji pętli (w każdym jej obiegu), najpierw wypisana zostanie na ekran wartość zmiennej `x`, a następnie wartość `x` zostanie zwiększona o 1 za pomocą jednoargumentowego operatora inkrementacji ++ (operator ten, jak dowiedzieliśmy się w rozdziale o zmiennych, zwiększa wartość swojego argumentu o 1).

Po każdej iteracji pętli, wartość zmiennej `x` będzie o 1 większa niż podczas poprzedniej iteracji. W końcu, gdy wartość zmiennej `x` będzie wynosić 6, wyrażenie w warunku pętli:

```
x <= 5
```

będzie miało wartość **false**, więc nastąpi koniec pętli. Program zakończy działanie, ponieważ po pętli nie ma już żadnych instrukcji do wykonania.

Prześledźmy jeszcze raz proces wykonywania się pętli w powyższym programie:

1. Sprawdzany jest warunek `x <= 5` dla `x` mającego wartość `1`. Warunek jest spełniony.
2. Wykonywane jest ciało pętli:
   1. Na ekran wypisywana jest wartość zmiennej `x`, czyli `1`.
   2. Wartość zmiennej `x` zwiększana jest o `1` – będzie teraz miała wartość `2`.
3. Ponownie sprawdzany jest warunek `x <= 5`, tym razem dla `x` mającego wartość `2`. Warunek jest spełniony.
4. Wykonywane jest ciało pętli:
   1. Na ekran wypisywana jest wartość zmiennej `x`, czyli `2`.
   2. Wartość zmiennej `x` zwiększana jest o `1` – będzie teraz miała wartość `3`.
5. Ponownie sprawdzany jest warunek `x <= 5`, tym razem dla `x` mającego wartość `3`. Warunek jest spełniony.
6. Wykonywane jest ciało pętli:
   1. Na ekran wypisywana jest wartość zmiennej `x`, czyli `3`.
   2. Wartość zmiennej `x` zwiększana jest o `1` – będzie teraz miała wartość `4`.
7. Ponownie sprawdzany jest warunek `x <= 5`, tym razem dla `x` mającego wartość `4`. Warunek jest spełniony.
8. Wykonywane jest ciało pętli:
   1. Na ekran wypisywana jest wartość zmiennej `x`, czyli `4`.
   2. Wartość zmiennej `x` zwiększana jest o `1` – będzie teraz miała wartość `5`.
9. Ponownie sprawdzany jest warunek `x <= 5`, tym razem dla `x` mającego wartość `5`. Warunek jest spełniony.
10. Wykonywane jest ciało pętli:
    1. Na ekran wypisywana jest wartość zmiennej `x`, czyli `5`.
    2. Wartość zmiennej `x` zwiększana jest o `1` – będzie teraz miała wartość `6`.
11. Ponownie sprawdzany jest warunek `x <= 5`, tym razem dla `x` mającego wartość `6`. **Tym razem jednak warunek nie jest spełniony – ma wartość `false`. Pętla kończy działanie.**

W naszym przykładowym programie, chociaż nie jest on skomplikowany, moglibyśmy wykorzystać funkcjonalność operatora ++, aby skrócić go jedną linię. Zauważmy, że operator *postfixowy* ++ najpierw zwraca wartość zmiennej, którą ma zmodyfikować, a dopiero potem zwiększa wartość tej zmiennej o `1`. Oznacza to, że wartością wyrażenia:

```
x++;
```

jest aktualna wartość zmiennej `x`. Moglibyśmy w takim razie skorzystać z tego operatora wewnątrz instrukcji `System.out.println`:

```
System.out.println(x++);
```

W tej jednej linii kodu wypisujemy wartość wyrażenia `x++` – ma ono wartość zmiennej `x` sprzed inkrementacji o `1` (z powodu użycia operatora postfixowego ++, a nie prefixowego). Ponadto,

operator ++ zwiększa wartość zmiennej `x` o `1`. Finalnie, linia:

```
System.out.println(x++);
```

jest równoważna instrukcjom:

```
System.out.println(x);
x++;
```

Cały program mógłby zatem wyglądać następująco:

Nazwa pliku: WypiszLiczbyOd1Do5Skrocony.java

```
public class WypiszLiczbyOd1Do5Skrocony {
  public static void main(String[] args) {
    int x = 1;

    while (x <= 5) {
      System.out.println(x++);
    }
  }
}
```

Ten program działa tak samo, jak jego pierwsza wersja – w wyniku jego uruchomienia zobaczymy na ekranie wypisane liczby od `1` do `5`.

5.2.2 Przykład: wypisywanie gwiazdek

W kolejnym przykładzie użycia pętli `while` wyświetlamy na ekranie tyle gwiazdek (znaków *), ile zażyczy sobie użytkownik.

Od użytkownika pobierzemy liczbę za pomocą poznanej w trzeciem rozdziale metody `getInt`:

Nazwa pliku: WypiszGwiazdki.java

```
import java.util.Scanner;

public class WypiszGwiazdki {
  public static void main(String[] args) {
    System.out.println("Ile gwiazdek wypisac?");
    int liczbaGwiazdek = getInt();

    // dopoki liczba gwiazdek jest wieksza od zera...
    while (liczbaGwiazdek > 0) {
      // ...wypisz na ekran gwiazdke
      System.out.print("*");
      liczbaGwiazdek--;
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

W powyższym przykładzie wypisaliśmy na ekran tekst używając nie metody `println`, lecz `print` – różnią się one tym, że `print` nie dodaje znaku końca linii na końcu tekstu – dzięki temu, każda z gwiazdek będzie wypisana w tej samej linii na ekranie.

W tym programie, warunkiem wykonania pętli jest prawdziwość wyrażenia `liczbaGwiazdek > 0`. W każdej iteracji pętli wypisujemy na ekran jedną gwiazdkę (znak *), po czym zmniejszamy wartość zmiennej `liczbaGwiazdek` o `1` za pomocą operatora dekrementacji `--`. Tak zapisany warunek pętli spowoduje, że wypiszemy na ekran dokładnie tyle gwiazdek, ile podał użytkownik.

Przykładowe pierwsze uruchomienia tego programu:

```
Ile gwiazdek wypisac?
5
*****
```

Wynik drugiego uruchomienia:

```
Ile gwiazdek wypisac?
0
```

W pierwszym uruchomieniu, *ciało pętli* wykonało się pięć razy. Drugie uruchomienie programu z podaniem liczby `0` pokazuje, że jeżeli warunek na wstępie pętli **`while`** nie jest prawdziwy, to ciało pętli nie wykona się ani razu.

Pętle mogą działać w nieskończoność, jeżeli ich warunek nigdy nie osiągnie wartości **`false`**. Jedyne, co można wtedy zrobić, to wymusić zakończenie programu za pomocą skrótu **`Ctrl + c`**. W jednym z kolejnych podrozdziałów zobaczymy, jakie zastosowanie mają nieskończone pętle.

5.3 Pętla do...while

Pętla **do**...**while** jest bardzo podobna do pętli **while**, z jedną zasadniczą różnicą: pętla **do**...**while** zawsze wykona swoje instrukcje, niezależnie od tego, czy jej warunek jest na wstępie spełniony, czy nie. Składnia pętli **do**...**while** jest następująca:

```
do {
  instrukcja1;
  instrukcja2;
  // ...
  instrukcjaN;
} while (warunek);
```

Zauważmy, że warunek sprawdzający, czy pętla ma działać, jest na końcu pętli, więc **pętla do..while zawsze wykona powiązane z nią instrukcje co najmniej raz**.

W pierwszym przykładzie w rozdziale o pętli **while** analizowaliśmy program, który wypisywał na ekran liczby od 1 do 5 – spójrzmy, jak wyglądałby ten sam program korzystający z pętli **do**..**while**:

Nazwa pliku: WypiszLiczbyOd1Do5DoWhile.java

```
public class WypiszLiczbyOd1Do5DoWhile {
  public static void main(String[] args) {
    int x = 1;

    do {
      System.out.println(x++);
    } while (x <= 5);
  }
}
```

Ta wersja programu niewiele różni się od wersji z pętlą **while**. Spójrzmy jednak na drugi program – wypisywanie gwiazdek – zmieniony tak, by korzystał z pętli **do**..**while**:

Nazwa pliku: WypiszGwiazdkiDoWhile.java

```
import java.util.Scanner;

public class WypiszGwiazdkiDoWhile {
  public static void main(String[] args) {
    System.out.println("Ile gwiazdek wypisac?");
    int liczbaGwiazdek = getInt();

    do {
      System.out.print("*");
      liczbaGwiazdek--;
    } while (liczbaGwiazdek > 0);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Czy ten program jest równoważny jego wersji z użyciem pętli while?

Spróbujmy uruchomić program i podać `3` jako liczbę gwiazdek do wypisania:

```
Ile gwiazdek wypisac?
3
***
```

Druga próba – tym razem podajemy `0`:

```
Ile gwiazdek wypisac?
0
*
```

Tym razem program nie zadziałał poprawnie – wypisał jedną gwiazdkę, chociaż poprosiliśmy o `0`. Dlaczego tak się stało? Wynika to po prostu ze sposobu działania pętli **`do..while`** – ciało pętli tego rodzaju jest zawsze wykonywane co najmniej raz, niezależnie od tego, czy warunek tej pętli jest na początku spełniony, czy nie.

Aby poprawić powyższy program, moglibyśmy opakować pętlę **`do..while`** w instrukcję warunkową **`if`**, w której sprawdzilibyśmy, czy podana przez użytkownika liczba gwiazdek jest większa od `0`:

Nazwa pliku: WypiszGwiazdkiDoWhile.java

```
import java.util.Scanner;

public class WypiszGwiazdkiDoWhile {
  public static void main(String[] args) {
    System.out.println("Ile gwiazdek wypisac?");
    int liczbaGwiazdek = getInt();

    if (liczbaGwiazdek > 0) {
      do {
        System.out.print("*");
        liczbaGwiazdek--;
      } while (liczbaGwiazdek > 0);
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Ta wersja programu działa już poprawnie – dzięki zastosowaniu poznanej w poprzednim rozdziale instrukcji warunkowej **`if`**, pętla **`do..while`** zostanie wykonana tylko wtedy, gdy liczba gwiazdek podana przez użytkownika będzie większa od `0`.

> Jak już widać, a co będzie jeszcze lepiej widoczne, gdy poznamy wszystkie cztery rodzaje pętli, różne rodzaje pętli mają różne cechy, a co za tym idzie, różne zastosowania.

5.3.1 Przykład: dodawanie kolejnych liczb

Spójrzmy na jeszcze jeden, ciekawy przykład. Poniższy program to prosty kalkulator sumujący dwie wczytane od użytkownika liczby. Pisaliśmy już podobny program w rozdziale trzecim o zmiennych – to, co sprawia, że ten poniższy jest ciekawszy od tamtego to to, że ten program oferuje dodawanie wielu par liczb dopóki użytkownik nie zażyczy sobie zakończyć działania programu:

Nazwa pliku: DodawanieLiczbDoWhile.java

```
import java.util.Scanner;

public class DodawanieLiczbDoWhile {
  public static void main(String[] args) {
    int x, y;
    String czyKoniec;

    do {
      System.out.print("Podaj pierwsza liczbe: ");
      x = getInt();

      System.out.print("Podaj druga liczbe: ");
      y = getInt();

      System.out.println("Ich suma wynosi: " + (x + y));

      System.out.print("Czy chcesz zakonczyc program? [t/n] ");
      czyKoniec = getString();

      // wykonuj petle dopoki uzytkownik nie wpisze litery "t"
    } while (!czyKoniec.equals("t"));
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

W tym programie:

1. W pętli `do..while`, pobieramy w każdym obiegu pętli (w każdej iteracji pętli) dwie liczby i wypisujemy ich sumę.
2. Następnie, prosimy użytkownika o podanie informacji, czy chce zakończyć działanie programu poprzez podanie litery `t` bądź innej.
3. Na końcu, sprawdzany jest warunek pętli – jeżeli użytkownik podał inną literę niż `t`, to warunek `!czyKoniec.equals("t")` będzie miał wartość `true` i pętla rozpocznie kolejny obieg, czyli wrócimy do punktu 1. W przeciwnym, razie pętla się zakończy.

Ważne jest tutaj to, że ciało pętli `do..while` wykonało się jeszcze przed sprawdzeniem, czy pętla powinna się zakończyć, dzięki czemu pobierzemy od użytkownika liczby i wypiszemy ich sumę co najmniej raz.

Przykładowe uruchomienie programu:

```
Podaj pierwsza liczbe: 10
Podaj druga liczbe: 7
Ich suma wynosi: 17
Czy chcesz zakonczyc program? [t/n] n

Podaj pierwsza liczbe: 100
Podaj druga liczbe: -20
Ich suma wynosi: 80
Czy chcesz zakonczyc program? [t/n] t
```

Prześledźmy powyższe wykonanie programu:

1. W ciele pętli:
    1. Pobieramy od użytkownika dwie liczby: `10` oraz `7`.
    2. Wypisujemy sumę liczb tych: `17`.
    3. Pytamy użytkownika, czy chce zakończyć program – aby to zrobić, powinien wpisać literę `t` z klawiatury.
    4. Pobieramy znak od użytkownika – jest to litera `n`.
2. Sprawdzamy warunek działania pętli: `!czyKoniec.equals("t")`. Wartość zapisana w zmiennej `czyKoniec` to `"n"` – taki znak podał użytkownika. Metoda `equals` zwróci **`false`**, a użycie operatora przeczenia logicznego ! (not) spowoduje, że finalna wartość wyrażenia w warunku pętli będzie miała wartość **`true`**, czyli pętla powinna działać dalej.
3. W drugiej iteracji, w ciele pętli:
    1. Pobieramy od użytkownika dwie liczby: `100` oraz `-20`.
    2. Wypisujemy sumę liczb tych: `80`.
    3. Pytamy użytkownika, czy chce zakończyć program.
    4. Pobieramy znak od użytkownika – **tym razem jest to litera** `y`.
4. Ponownie sprawdzamy warunek działania pętli: `!czyKoniec.equals("t")`. Tym razem, wartość zapisana w zmiennej `czyKoniec` to `"t"`. Metoda `equals` zwróci **`true`**, a użycie operatora przeczenia logicznego ! (not) spowoduje, że finalna wartość wyrażenia w warunku pętli będzie miała wartość **`false`**. Warunek pętli nie jest spełniony – pętla kończy działanie.

*Pętle **`do..while`** są rzadko wykorzystywane. Zdecydowanie częściej korzysta się z pozostałych trzech rodzajów pętli: **`while`**, **`for`**, oraz **`for`**-each.*

5.4 Pętle nieskończone

Pętle mogą działać w nieskończoność, jeżeli ich warunek nigdy nie osiągnie wartości **false**. Program będzie w kółko wykonywał instrukcje z ciała nieskończonej pętli.

Jedyne, co można wtedy zrobić, to wymusić zakończenie takiego programu. Programy konsolowe możemy zatrzymać skrótem **Ctrl + c**

Spójrzmy na najprostszy przykład nieskończonej pętli:

```
while (true) {
  // instrukcje
}
```

Tak zapisana pętla nigdy się nie zakończy – wyrażenie, które jest warunkiem pętli, to po prostu wartość **true**. Spójrzmy na przykład poniższego, natrętnego programu, który uparcie wypisuje na ekran komunikat "Witaj!":

Nazwa pliku: NieskonczonaPetla.java

```
public class NieskonczonaPetla {
  public static void main(String[] args) {
    while (true) {
      System.out.println("Witaj!");
    }
  }
}
```

Program po uruchomieniu z ogromną prędkością produkuje na ekranie komunikaty "Witaj!". Użycie skrótu klawiaturowego **Ctrl + c** powoduje wymuszenie jego zatrzymania.

Czy pętle nieskończone są do czegoś przydatne? Tak – często chcemy, aby nasz program działał cały czas i wykonywał określone operacje. Programy oferują też często użytkownikowi możliwość zakończenia jego działania, jeżeli wykona on odpowiednią akcję – podobnie, jak widzieliśmy w przykładzie pętli **do..while** do sumowania par liczb. Program ten działał tak długo, aż użytkownik nie wyraził chęci to jego zatrzymania poprzez podanie znaku "t", który warunkował zakończenie pętli.

Z drugiej jednak strony, nieskończone pętle często są wynikiem źle działającego kodu, w którym znajduje się jakiś błąd. Po napisaniu pętli zawsze warto zastanowić się, czy w każdym przypadku działania naszego programu warunek pętli w pewnym momencie będzie miał wartość **false**, a tym samym, pętla się zakończy.

5.5 Pętla for

Pętla **for** jest często używaną pętlą. Jej składnia jest trochę bardziej skomplikowana, niż poznane do tej pory pętle **while** oraz **do**..**while**:

```
for (instrukcja_inicjalizująca; warunek; instrukcja_kroku) {
  instrukcja1;
  instrukcja2;
}
```

Spójrzmy najpierw na przykład pętli **for** w programie, który wypisuje liczby od 1 do 5:

Nazwa pliku: WypiszLiczbyOd1Do5For.java

```
public class WypiszLiczbyOd1Do5For {
  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
    }
  }
}
```

W wyniku działania tego programu zobaczymy na ekranie liczby od 1 do 5.

Przeanalizujmy teraz poszczególne elementy pętli **for**:

- **instrukcja_inicjalizująca** – jest to instrukcja (bądź instrukcje rozdzielone przecinkami) wykonywana jednorazowo, jeszcze przed rozpoczęciem pętli – w naszym przykładzie jest to utworzenie zmiennej o nazwie `i` typu **int**, oraz zainicjalizowanie jej wartością 1.
- **warunek** – jest to wyrażenie obliczane przed każdą iteracją pętli – jeżeli jest spełnione (jego wartość to **true**), to wykonane zostanie ciało pętli. W naszym przykładzie porównujemy zmienną `i` z liczbą 5 – jeżeli zmienna `i` jest mniejsza bądź równa 5, to warunek będzie spełniony, a co za tym idzie, wykona się ciało pętli.
- **instrukcja_kroku** – jest to instrukcja (bądź instrukcje rozdzielone przecinkami) wykonywana zaraz po zakończeniu każdego obiegu pętli. W naszym przykładzie zwiększamy (używając postfixowego operatora inkrementacji) wartość zmiennej `i` o 1.

Prześledźmy działanie pętli **for** w powyższym programie:

1. Wykonywana jest *instrukcja inicjalizująca* pętli **for** – definiowana jest zmienna `i` o początkowej wartości 1.
2. Sprawdzany jest warunek pętli `i <= 5` dla `i` mającego wartość 1. Wyrażenie to ma wartość **true**, więc przechodzimy dalej.
3. Wykonywane jest ciało pętli – wypisujemy wartość zmiennej `i`, czyli 1.
4. Na końcu iteracji pętli, wykonywana jest *instrukcja kroku* – w tym przypadku, zwiększamy wartość zmiennej `i` o 1 – zmienna `i` będzie teraz miała wartość 2.
5. Ponownie sprawdzany jest warunek, ale dla `i` równego 2. Ponownie wartością jest **true**.
6. Wykonywane jest ciało pętli – wypisujemy wartość zmiennej `i`, czyli 2.
7. Ponownie wykonujemy *instrukcję kroku* – `i` będzie teraz miało wartość 3.

8. Ponownie sprawdzany jest warunek, ale dla `i` równego `3`. Ponownie wartością jest **`true`**.
9. Wykonywane jest ciało pętli – wypisujemy wartość zmiennej `i`, czyli `3`.
10. Ponownie wykonujemy *instrukcję kroku* – `i` będzie teraz miało wartość `4`.
11. Ponownie sprawdzany jest warunek, ale dla `i` równego `4`. Ponownie wartością jest **`true`**.
12. Wykonywane jest ciało pętli – wypisujemy wartość zmiennej `i`, czyli `4`.
13. Ponownie wykonujemy *instrukcję kroku* – `i` będzie teraz miało wartość `5`.
14. Ponownie sprawdzany jest warunek, ale dla `i` równego `5`. Ponownie wartością jest **`true`**.
15. Wykonywane jest ciało pętli – wypisujemy wartość zmiennej `i`, czyli `5`.
16. Ponownie wykonujemy *instrukcję kroku* – `i` będzie teraz miało wartość `6`.
17. Ponownie sprawdzany jest warunek, ale dla `i` równego `6`. **Tym razem, wartość wyrażenia `i <= 5` ma wartość `false`. Następuje koniec pętli.**

Zasadę działania pętli **`for`** można uogólnić w następujący sposób:

1. Wykonaj wszystkie *instrukcje inicjalizujące*.
2. Sprawdź warunek pętli:
   a) Jeżeli jest prawdziwy, wykonaj ciało pętli.
   b) Jeżeli jest nieprawdziwy, przerwij działanie pętli, pomijając kolejne kroki.
3. Po wykonaniu ciała pętli, wykonaj *intrukcje kroku* i przejdź do kroku 2.

Pętla **`while`** i pętla **`for`** mają wspólną cechę – jeżeli ich warunek nie będzie spełniony już na początku wykonywania pętli, to ciało pętli nie wykona się ani razu. Spójrzmy na przykład z wypisywaniem gwiazdek:

Nazwa pliku: WypiszGwiazdkiFor.java

```
import java.util.Scanner;

public class WypiszGwiazdkiFor {
  public static void main(String[] args) {
    System.out.println("Ile gwiazdek wypisac?");
    int liczbaGwiazdek = getInt();

    for (int i = 0; i < liczbaGwiazdek; i++) {
      System.out.print("*");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Spróbujmy uruchomić program dwa razy – raz podamy `3` gwiazdki do wypisania, a drugi raz `0`:

```
Ile gwiazdek wypisac?
3
***
```

```
Ile gwiazdek wypisac?
0
```

Program działa poprawnie, gdy podamy do wypisania `0` gwiazdek, ponieważ, w takim przypadku, warunek pętli `i < liczbaGwiazdek` nie będzie spełniony już na samym początku działania pętli i ciało pętli nie wykona się ani razu.

5.5.1 Instrukcje inicjalizujące i kroku

Instrukcje inicjalizujące i instrukcje kroku nie muszą być pojedyncze w pętli **`for`** – spójrzmy na poniższy przykład:

Nazwa pliku: InstrukcjeInicjalizujacePetlaFor.java

```
public class InstrukcjeInicjalizujacePetlaFor {
  public static void main(String[] args) {
    for (int j = 1, k = 1; j < 100 && k < 100; j = j * 2, k = k * 3) {
      System.out.println("j = " + j + ", k = " + k);
    }
  }
}
```

W tej pętli mamy:

- dwie **instrukcje inicjalizujące**, rozdzielone przecinkami – utworzenie dwóch zmiennych: `j` oraz `k`,
- dwie **instrukcje kroku** – pomnożenie wartości zmiennej `j` razy `2` oraz zmiennej `k` razy `3`.

Ten program wypisuje na ekran kolejne potęgi liczb `2` oraz `3` do czasu, aż któraś z nich nie przekroczy liczby `100`. Zauważmy, że w warunku pętli mamy dwuczłonowe wyrażenie, w którym wykorzystujemy operator warunkowy && (and). Wynik działania tego programu to:

```
j = 1, k = 1
j = 2, k = 3
j = 4, k = 9
j = 8, k = 27
j = 16, k = 81
```

Należy jeszcze zwrócić uwagę, iż każdy z elementów pętli **`for`** jest opcjonalny – poniższa pętla będzie wykonywać się w nieskończoność:

```
for (;;) {

}
```

Pętla ta nie zawiera ani instrukcji inicjalizującej, ani warunku pętli, ani instrukcji kroku – widzimy jedynie średniki, które oddzielają od siebie poszczególne elementy pętli **`for`**, ponieważ średniki te są one wymagane.

W ramach inicjalizacji powyższej pętli nic się nie zadzieje. Podobnie po każdym obiegu pętli – nie będzie wykonanej żadnej instrukcji kroku. Warunek pętli nie został zdefiniowany, więc pętla będzie działała w nieskończoność.

*Instrukcje **`for`**, w której nie zdefiniowano któregoś z członów (bądź wszystkich) nie są zbyt często używane – po prostu nie mają zbyt wielu zastosowań.*

5.6 Zakres (scope) zmiennych w pętlach

W rozdziale o instrukcjach warunkowych zaczęliśmy mówić o *zakresie zmiennych*.

W poniższym przykładzie, w instrukcji inicjalizującej pętli `for`, definiujemy zmienną o nazwie `i`. Czy ten program skompiluje się bez błędów?

Nazwa pliku: WypiszLiczbyOd1Do5ForZakresZmiennej.java

```
public class WypiszLiczbyOd1Do5ForZakresZmiennej {
  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
    }

    System.out.println("Po ukonczeniu petli i = " + i);
  }
}
```

Powyższy program jest niepoprawny, ponieważ zmienna `i` jest niedostępna po zakończeniu działania pętli – została ona zdefiniowana w ramach pętli `for` i tylko w niej jest dostępna. *Zakres* zmiennej `i` to jedynie ciało pętli `for`. Po wyjściu z pętli, zmienna o nazwie `i` przestaje istnieć.

Kompilator wypisze następujący błąd podczas próby kompilacji:

```
WypiszLiczbyOd1Do5ForZakresZmiennej.java:7: error: cannot find symbol
    System.out.println("Po ukonczeniu petli i = " + i);
                                                    ^
  symbol:   variable i
  location: class WypiszLiczbyOd1Do5ForZakresZmiennej
1 error
```

Aby kod zadziałał, musielibyśmy zdefiniować zmienną `i` przed wykonaniem pętli.

Ta sama zasada odnosi się do zmiennych definiowanych w ciałach pętli `while`, `do...while`, `for`, w instrukcjach warunkowych `if`, oraz ogólnie w blokach kodu.

5.7 Instrukcje break oraz continue

Czasem potrzebujemy przerwać działanie pętli – wyobraźmy sobie sytuację, gdy w ciągu liczb szukamy konkretnej liczby. Gdy ją znajdziemy, nie ma już potrzeby do kontynuowania działania pętli. W tym celu możemy skorzystać z instrukcji **break**, która powoduje natychmiastowe przerwanie pętli – żadne instrukcje zawarte w ciele pętli (oraz instrukcji kroku w pętli **for**) nie będą już wykonywane.

Słowo kluczowe **break** poznaliśmy już przy okazji omawiania instrukcji **switch** w rozdziale czwartym. Możemy je stosować w każdym z poznanych rodzajów pętli: **while**, **do..while**, oraz **for**, a także w pętli **for**-each, którą poznamy w rozdziale o tablicach.

Może się też zdarzyć, że będziemy potrzebowali przerwać aktualną iterację pętli i przejść do kolejnej – do tego celu służy instrukcja **continue**.

5.7.1 Instrukcja break

Spójrzmy najpierw na przykład działania instrukcji **break**:

Nazwa pliku: PrzykladBreakFor.java

```
public class PrzykladBreakFor {
  public static void main(String[] args) {
    for (int i = 0; i < 1000; i++) {
      System.out.println(i);

      if (i >= 3) {
        break;
      }

      System.out.println("Nadal dzialam.");
    }
  }
}
```

Pętla ma wypisać liczby od `0` do `999`, ale w instrukcji **if** wewnątrz pętli sprawdzamy wartość zmiennej `i` – jeżeli jest większa bądź równa `3`, wykonana zostanie instrukcja **break**. Spójrzmy, co zostanie wypisane w ramach działania tego programu:

```
0
Nadal dzialam.
1
Nadal dzialam.
2
Nadal dzialam.
3
```

Zauważmy, że gdy zmienna `i` ma wartość `3` i zostaje wykonana instrukcja **break**, to komunikat `"Nadal dzialam."` nie zostaje wypisany – działanie pętli jest natychmiast przerywane i wszelkie instrukcje, które znajdują się za instrukcją **break** w ciele pętli, zostają pominięte.

W kolejnym przykładzie, który korzysta z nieskończonej pętli **while**, sumujemy liczby podawane przez użytkownika, dopóki użytkownik nie poda liczby `0`:

Nazwa pliku: SumujLiczbyWhileBreak.java

```
import java.util.Scanner;

public class SumujLiczbyWhileBreak {
  public static void main(String[] args) {
    System.out.println(
        "Podaj liczby do zsumowania lub 0, aby zakonczyc."
    );

    int suma = 0;
    int liczba;

    while (true) {
      System.out.print("Podaj liczbe: ");
      liczba = getInt();

      if (liczba == 0) {
        break;
      }

      suma += liczba;
    }

    System.out.println("Suma podanych przez Ciebie liczb to " + suma);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Powyższy program najpierw informuje użytkownika, że zsumuje podane przez niego liczby, a zakończy swoją pracę, gdy użytkownik poda liczbę `0`.

Następnie, w nieskończonej pętli w każdej jej iteracji pobieramy od użytkownika liczbę, po czym:

- jeżeli liczba jest równa `0`, to przerywamy działanie pętli za pomocą instrukcji **`break`**,
- w przeciwnym razie, dodajemy do zmiennej `suma` wczytaną liczbę.

Gdy użytkownik poda `0`, pętla się zakończy, a na końcu programu wypisana zostanie suma wszystkich wczytanych liczb.

Spójrzmy na trzy uruchomienia powyższego programu:

```
Podaj liczby do zsumowania lub 0, aby zakonczyc.
Podaj liczbe: 10
Podaj liczbe: 999
Podaj liczbe: -20
Podaj liczbe: 0
Suma podanych przez Ciebie liczb to 989
```

```
Podaj liczby do zsumowania lub 0, aby zakonczyc.
Podaj liczbe: -1
Podaj liczbe: 1
Podaj liczbe: 0
Suma podanych przez Ciebie liczb to 0
```

```
Podaj liczby do zsumowania lub 0, aby zakonczyc.
Podaj liczbe: 0
Suma podanych przez Ciebie liczb to 0
```

Program poprawnie sumuje podawane liczby. Gdy podana zostanie liczba `0`, pętla, w której pobieramy i sumuje liczby kończy działanie i przechodzimy wykonywania kodu programu za pętlą – w naszym przypadku, jest to tylko jedna instrukcja – wypisanie policznej sumy na ekran.

5.7.2 Instrukcja continue

Instrukcja **`continue`** jest podobna do instrukcji **`break`** z tym, że nie przerywa działania całej pętli, lecz tylko jej aktualną iterację. Wykonanie programu przechodzi wtedy albo do sprawdzenia warunku pętli (pętle **`while`**, **`do`**..**`while`**, oraz **`for`**-each), lub do instrukcji kroku i sprawdzenia warunku pętli (pętla **`for`**).

Spójrzmy na prosty przykład – poniższy program używa słowa kluczowego **`continue`**, gdy wartość zmiennej używanej w pętli jest parzysta:

Nazwa pliku: WypiszLiczbyContinue.java

```
public class WypiszLiczbyContinue {
  public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
      if (i % 2 == 0) {
        continue;
      }

      System.out.println(i);
    }
  }
}
```

Spójrzmy na wynik uruchomienia tego programu:

```
1
3
5
7
9
```

Program wypisuje tylko liczby nieparzyste. Jeżeli liczba jest parzysta, to spełniony jest warunek `i % 2 == 0` (reszta z dzielenia dla liczb parzystych wynosi `0`), więc zostaje wykonana instrukcja **`continue`** – aktualna iteracja pętli natychmiast się kończy i wszelkie dalsze operacje w ciele pętli zostają pominięte. Wykonana zostaje instrukcja kroku (zwiększenie `i` o `1` za pomocą `i++`), po czym sprawdzony zostaje warunek pętli i jeżeli nadal jest spełniony, rozpoczyna się kolejna iteracja pętli.

Spójrzmy na inny przykład – poniższy program próbuje policzyć pole koła o promieniu podanym przez użytkownika – jeżeli jednak podana przez użytkownika liczba będzie nieprawidłową wartością promienia koła (będzie mniejsza bądź równa `0`), program ponownie spróbuje pobrać liczbę – i tak do skutku:

Nazwa pliku: PoliczPoleKolaContinue.java

```
import java.util.Scanner;

public class PoliczPoleKolaContinue {
  public static void main(String[] args) {
    int promien;

    do {
      System.out.print("Podaj promien kola: ");
      promien = getInt();

      if (promien <= 0) {
        System.out.println(
            "Nieprawidlowy promien. Podaj liczbe wieksza od 0"
        );
        continue;
      }

      double poleKola = 3.14 * promien * promien;
      System.out.println("Pole kola o tym promieniu: " + poleKola);
    } while (promien <= 0);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

W każdej iteracji pętli pobieramy od użytkownika liczbę i sprawdzamy, czy jest ona prawidłowym promieniem koła – jeżeli nie, to wypisujemy stosowny komunikat i korzystamy z instrukcji **`continue`** – wykonanie programu przechodzi natychmiast do sprawdzenia warunku działania pętli.

Pętla będzie działać do czasu, aż użytkownik nie poda poprawnego promienia – wtedy warunek pętli przestanie mieć wartość **`true`** i pętla się zakończy.

*Instrukcje **`break`** oraz **`continue`** mogą wystąpić w dowolnym miejscu w ciele pętli, ale nie mogą wystąpić poza pętlą.*

5.8 Ten sam kod z użyciem różnych pętli

Spójrzmy na dwa przykłady kodu zapisane za pomocą każdego z rodzajów pętli.

5.8.1 Wyświetlanie kwadratu podanej liczby

Poniższe fragmenty programów (które są treścią metod `main`) wykonują to samo zadanie używając różnych rodzajów pętli:

- pobierają od użytkownika jedną liczbę,
- wypisują jej kwadrat,
- pytają użytkownika, czy chce zakończyć działanie programu.

Pętla **`while`**:

Nazwa pliku: KwadratLiczbyWhile.java

```
int x;
String czyKoniec = "n";

while (!czyKoniec.equals("t")) {
  System.out.print("Podaj liczbe: ");
  x = getInt();

  System.out.println("Kwadrat tej liczby wynosi: " + (x * x));

  System.out.print("Czy chcesz zakonczyc program? [t/n] ");
  czyKoniec = getString();
}
```

W powyższym przykładzie musieliśmy przypisać zmiennej `czyKoniec` wstępną wartość inną niż `"t"`, ponieważ w przeciwnym razem pętla nie wykonałaby się ani razu.

Pętla **`do..while`**:

Nazwa pliku: KwadratLiczbyDoWhile.java

```
int x;
String czyKoniec;

do {
  System.out.print("Podaj liczbe: ");
  x = getInt();

  System.out.println("Kwadrat tej liczby wynosi: " + (x * x));

  System.out.print("Czy chcesz zakonczyc program? [t/n] ");
  czyKoniec = getString();
} while (!czyKoniec.equals("t"));
```

W tym przykładzie pętla wykona się co najmniej raz, więc wartość zmiennej `czyKoniec` możemy nadać pod koniec każdej iteracji pętli.

Pętla **for**:

Nazwa pliku: KwadratLiczbyFor.java

```
String czyKoniec = "n";

for (int x; !czyKoniec.equals("t"); czyKoniec = getString()) {
  System.out.print("Podaj liczbe: ");
  x = getInt();

  System.out.println("Kwadrat tej liczby wynosi: " + (x * x));
  System.out.print("Czy chcesz zakonczyc program? [t/n] ");
}
```

W tej wersji programu, w instrukcji inicjalizującej pętlę, definiujemy zmienną `x`, do której będziemy przypisywać liczbę pobraną od użytkownika. W instrukcji kroku wstawiliśmy przypisanie do zmiennej `czyKoniec` informacji podanej przez użytkownika, czy chce on zakończyć działanie programu.

5.8.2 Wypisanie kolejnych liczb parzystych

Poniższe fragmenty kodów wypisują kolejne liczby parzyste mniejsze bądź równe `20`.

Pętla **while**:

Nazwa pliku: WypiszParzysteWhile.java

```
public class WypiszParzysteWhile {
  public static void main(String[] args) {
    int x = 0;

    while (x <= 20) {
      System.out.print(x + " ");
      x += 2;
    }

    System.out.println(); // wypisz pusta linie
    System.out.println("x na koncu jest rowne " + x);
  }
}
```

Przykładowy wynik działania powyższego programu:

```
0 2 4 6 8 10 12 14 16 18 20
x na koncu jest rowne 22
```

Pętla **do..while**:

Nazwa pliku: WypiszParzysteDoWhile.java

```
public class WypiszParzysteDoWhile {
  public static void main(String[] args) {
    int x = 0;

    do {
      System.out.print(x + " ");
      x += 2;
    } while (x <= 20);

    System.out.println(); // wypisz pusta linie
    System.out.println("x na koncu jest rowne " + x);
  }
}
```

Powyższy program wypisuje na ekran dokładnie takie same dane, jak przykład z pętlą **while**.

Petla **for**:

Nazwa pliku: WypiszParzysteFor.java

```
public class WypiszParzysteFor {
  public static void main(String[] args) {
    for (int x = 0; x <= 20; x += 2) {
      System.out.print(x + " ");
    }

    // spowoduje blad kompilacji - zmienna x juz tutaj nie istnieje!
    //System.out.println("x na koncu jest rowne " + x);
  }
}
```

Każda z pętli ma takie samo zadania – wypisanie liczb parzystych od `0` do `20` (włącznie). Na końcu wypisujemy też wartość zmiennej `x` po zakończeniu każdej pętli – poza ostatnim programem, w którym korzystamy z pętli **for**.

Należy zwrócić uwagę, że w przypadku pętli **for**, użyta zmienna nie istnieje poza pętlą i dlatego ostatnia linijka musi być zakomentowana – inaczej program by się w ogóle nie skompilował.

5.9 Zagnieżdżanie pętli

Pętle można zagnieżdżać – w ciele jednej pętli możemy skorzystać z kolejnej pętli. Poniższy przykład wypisuje na ekran tabliczę mnożenia:

Nazwa pliku: TabliczkaMnozenia.java

```
public class TabliczkaMnozenia {
  public static void main(String[] args) {
    for (int i = 1; i <= 6; i++) {
      for (int j = 1; j <= 6; j++) {
        int wynik = i * j;

        if (wynik >= 10) {
          System.out.print(wynik + " ");
        } else {
          // dla jednocyfrowych wynikow dodajemy spacje
          // na poczatku, by wynik byl ladnie sformatowany
          System.out.print(" " + wynik + " ");
        }
      }

      System.out.println(); // pusta linia
    }
  }
}
```

Wynik jest następujący:

```
 1  2  3  4  5  6
 2  4  6  8 10 12
 3  6  9 12 15 18
 4  8 12 16 20 24
 5 10 15 20 25 30
 6 12 18 24 30 36
```

W każdym pojedynczym obiegu zewnętrznej pętli `for` wykonywana jest cała wewnętrzna pętla `for` – łącznie w tym programie odbędzie się `6` iteracji pętli zewnętrznej i `36` iteracji pętli wewnętrznej (ponieważ cała pętla wewnętrzna wykona się sześć razy). Zmienna `j` w pętli wewnętrznej sześć razy przejdzie przez wszystkie liczby od `1` do `6`.

W każdej iteracji pętli wewnętrznej obliczamy wynik mnożenia obu zmiennych pętli: `i` oraz `j`. Następnie, wypisujemy `wynik` w jednej linii na ekranie (korzystamy z `print` zamiast `println`, dzięki czemu kolejne komunikaty będą wypisywane w tej samej linii). Jeżeli aktualnie policzony `wynik` mnożenia jest liczbą jednocyfrową, to przed wypisywanym wynikiem dodajemy jeszcze znak spacji, aby wyniki były równo wypisane.

Po tym, jak wykona się pętla wewnętrzna, wykonujemy jeszcze jedną instrukcję w obiegu pętli zewnętrznej:

```
System.out.println(); // pusta linia
```

Instrukcja ta powoduje wypisanie na ekran znaku nowej linii, dzięki czemu, gdy w kolejnym obiegu zewnętrznej pętli rozpocznie się wykonywanie pętli wewnętrznej, kolejne wyniki będą wypisywane w linii poniżej.

Spójrzmy, jak będą zmieniać się wartości zmiennych w powyższym programie dla dwóch pierwszych obiegów pętli zewnętrznej:

1. Inicjalizowana jest pętla zewnętrzna – zmiennej `i` nadawana jest wartość `1`.
2. Sprawdzany jest warunek pętli zewnętrznej `i <= 6`, który jest spełniony.
3. Wykonywane jest ciało pętli zewnętrznej:
    1. Inicjalizowana jest pętla wewnętrzna – zmiennej `j` nadawa jest wartość `1`.
    2. Sprawdzany jest warunek pętli wewnętrznej – jest spełniony.
    3. Wykonywane jest ciało pętli wewnętrznej – obliczany jest wynik `i * j`, czyli `1 * 1`, a następnie wynik `1` jest wypisywany na ekran.
    4. Pętla wewnętrzna kończy iterację – wykonywana jest instrukcja kroku `j++`.
    5. Ponownie sprawdzany jest warunek pętli wewn. – `j` wynosi `2`, warunek jest spełniony.
    6. W ciele pętli wewnętrznej obliczany i wypisywany jest wynik `i * j`, czyli `1 * 2`.
    7. Pętla wewnętrzna kończy iterację – wykonywana jest instrukcja kroku `j++`.
    8. Ponownie sprawdzany jest warunek pętli wewn. – `j` wynosi `3`, warunek jest spełniony.
    9. W ciele pętli wewnętrznej obliczany i wypisywany jest wynik `i * j`, czyli `1 * 3`.
    10. Pętla wewnętrzna kończy iterację – wykonywana jest instrukcja kroku `j++`.
    11. *Pętla wewnętrzna kontynuuje działanie do momentu, aż warunek `j <= 6` przestaje być spełniony.*
4. Gdy pętla wewnętrzna kończy swoje działanie, w pętli zewnętrznej zostaje jeszcze do wykonania instrukcja `System.out.println();`
5. Dopiero w tym miejscu kończy się **pierwsza** iteracja zewnętrznej pętli – wykonywana jest instrukcja kroku pętli zewnętrznej, czyli `i++` – `i` będzie miało teraz wartość `2`.
6. Sprawdzany jest warunek pętli zewnętrznej `i <= 6`. Jest spełniony, gdyż `i` ma wartość `2`.
7. Ponownie wykonywane jest ciało pętli zewnętrznej:
    1. Ponownie inicjalizowana jest pętla wewnętrzną – zmiennej `j` nadawana jest wartość `1`.
    2. Sprawdzany jest warunek pętli wewnętrznej.
    3. Wykonywane jest ciało pętli wewnętrznej – obliczany jest wynik `i * j`, czyli `2 * 1`,
    4. I tak dalej.

Dalsze działanie programu jest takie samo, jak opisano wcześniej – pętla wewnętrzna działa do czasu, aż jej warunek przestaje być spełniony. Nastepuje zakończenie pętli wewnętrznej, kończy się iteracja pętli zewnętrznej i następuje ponowna jej iteracja, do czasu, aż warunek pętli zewnętrznej przestaje być prawdziwy.

Pętlę zagnieżdżone powinniśmy stosować wybiórczo. Używanie wielu zagnieżdżonych pętli często powoduje powstawanie kodu, który jest trudny w zrozumieniu, utrzymaniu, oraz testowaniu.

Ponadto, zagnieżdżone pętle mogą też powodować wolne działanie programu – zauważmy, że na każdy obieg pętli zewnętrznej wykonywane są wszystkie iteracje pętli wewnętrznej – liczba iteracji rośnie bardzo szybko, im więcej mamy pętli zagnieżdżonych.

5.9.1 Użycie break i continue w pętlach zagnieżdżonych

Wiemy już, że instrukcje **break** oraz **continue** powodują, odpowiednio, przerwanie działania pętli oraz przerwanie aktualnej iteracji.

Instrukcje **break** oraz **continue** odnoszą się tylko do pętli, w których zostały użyte – ma to znaczenie w przypadku pętli zagnieżdżonych, ponieważ użycie **break** lub **continue** w pętli zagnieżdżonej nie spowoduje, że pętla zewnętrzna przerwie działanie.

Spójrzmy na poniższy przykład – jaki będzie wynik jego działania?

Nazwa pliku: ZagniezdzonaPetlaBreak.java

```
public class ZagniezdzonaPetlaBreak {
  public static void main(String[] args) {
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 5; j++) {
        System.out.print(j + ", ");

        if (j == 1) break;
      }

      System.out.println();
    }
  }
}
```

Na ekranie zobaczymy:

```
0, 1,
0, 1,
0, 1,
```

Dlaczego instrukcja **break** nie przerwała obu pętli? Jest to normalne działanie – zarówno instrukcje **break** jak i **continue**, odnoszą się zawsze do tej pętli, w której zostały użyte. Aby przerwać główną pętlę, musielibyśmy użyć **break** "o poziom wyżej":

Nazwa pliku: ZagniezdzonaPetlaBreak2.java

```
public class ZagniezdzonaPetlaBreak2 {
  public static void main(String[] args) {
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 5; j++) {
        System.out.print(j + ", ");

        if (j == 1) break;
      }

      System.out.println();
      break;
    }
  }
}
```

Wynik:

```
0, 1,
```

A co w przypadku, gdybyśmy chcieli spowodować w zagnieżdżonej pętli, by przerwana została główna pętla (bądź spowodować przejście do kolejnej iteracji za pomocą **continue**)? Możemy w tym celu skorzystać z opcjonalnych etykiet, które możemy przyporządkować pętlom – spójrzmy na przykład:

Nazwa pliku: ZagniezdzonaPetlaBreakEtykieta.java

```
public class ZagniezdzonaPetlaBreakEtykieta {
  public static void main(String[] args) {
    glowna_petla: for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 5; j++) {
        System.out.print(j + ", ");

        if (j == 1) break glowna_petla;
      }
    }
  }
}
```

W zaznaczonej linii, przed słowem kluczowym **for** umieściliśmy *etykietę*, po której następuje dwukropek. Następnie, w pętli zagnieżdżonej po instrukcji **break** podaliśmy nazwę etykiety. Wynik działania jest następujący:

```
0, 1,
```

Instrukcja **break** z użyciem etykiety spowodowała przerwanie głównej, a nie zagnieżdżonej, pętli.

Etykiety można stosować:

- z każdym rodzajem pętli,
- na dowolnym poziomie zagnieżdżenia,
- z instrukcjami **break** oraz **continue**.

Etykiety i przerywanie pętli "wyższego poziomu" z zagnieżdżonych pętli są bardzo rzadko stosowane – po prostu rzadko zdarzają się przypadki, które takiej funkcjonalności wymagają.

5.10 Typ String i metoda charAt oraz pętle

Typ `String`, przechowujący ciągi znaków, pozwala na pobranie z niego znaku o podanym indeksie.

Indeksy znaków zaczynają się od `0`, a nie od `1`, więc pierwszy znak przechowywany w zmiennej typu `String` znajduje się pod indeksem `0`, a indeks ostatniego znaku to liczba znaków pomniejszona o `1`.

Dla przykładu, mając następującą zmienną typu `String`:

```
String komunikat = "Ala ma kota";
```

Znaki składające się na wartość zmiennej `komunikat` mają następujące indeksy:

Indeks	0	1	2	3	4	5	6	7	8	9	10
Litera	A	l	a		m	a		k	o	t	a

Zauważmy, iż:

- pierwszy znak, czyli `'A'`, ma indeks `0`, a nie `1`.
- ostatni znak, czyli `'a'`, ma indeks równy długości całego stringu (tzn. liczby znaków, które zawiera) minus `1`, czyli `10` (ponieważ wszystkich znaków jest `11`). Wynika to z faktu, że indeksy rozpoczynają sie od `0`, a nie `1`.

Aby otrzymać znak znajdujący się na danej pozycji, stosujemy metodę `charAt`, której podajemy jako argument indeks znaku, który chcemy pobrać.

Spójrzmy na przykład, w którym wypisujemy na ekran pierwszy znak zmiennej typu `String`:

Nazwa pliku: WypiszPierwszyZnakCharAt.java

```
public class WypiszPierwszyZnakCharAt {
  public static void main(String[] args) {
    String komunikat = "Witaj!";

    System.out.println(komunikat.charAt(0));
  }
}
```

Ten program wypisuje na ekran jedną literę:

```
W
```

Skorzystaliśmy w nim z metody `charAt` do pobrania pierwszego znaku w stringu, czyli tego, który ma indeks `0`. Znak jest zwracany i służy jako argument dla instrukcji wypisującej tekst na ekran.

A co stanie się, jeżeli podamy nieprawidłowy indeks? Na przykład, zapytamy o znak o indeksie `100`, gdy zmienna typu `String` będzie zawierała krótszy tekst?

Nazwa pliku: WypiszZnakCharAtNieprawidlowyIndeks.java

```
public class WypiszZnakCharAtNieprawidlowyIndeks {
  public static void main(String[] args) {
    String komunikat = "Witaj!";

    System.out.println(komunikat.charAt(100));
  }
}
```

Ten program skompiluje się bez błędów, jednak po uruchomieniu go wystąpi błąd i działanie programu zakończy się. Na ekranie zobaczymy następujący błąd:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 100
   at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:47)
   at java.base/java.lang.String.charAt(String.java:702)
   at WypiszZnakCharAtNieprawidlowyIndeks.main(
                           WypiszZnakCharAtNieprawidlowyIndeks.java:5)
```

Błąd wystąpił, ponieważ podany indeks jest nieprawidłowy – w zmiennej `komunikat` nie ma znaku o indeksie `100`, ponieważ string ten zawiera jedynie `11` znaków. Komunikat błędu jest jasny: `String index out of range: 100`.

> Zwróćmy uwagę na bardzo ważny aspekt tego programu. Program *skompilował* się bez błędów, tzn. kompilator poprawnie nie stwierdził żadnych problemów w kodzie.
>
> Problem w programie nie wynika z błędnej składni programu, którą kompilator jest w stanie wychwycić – problem wynika z logicznego błędu, który uwidacznia się dopiero po uruchomieniu programu.

Skoro znaki w stringach mają indeksy, które są kolejnymi liczbami całkowitymi, to moglibyśmy skorzystać z funkcjonalności pętli **`for`**, która w prosty sposób umożliwi nam odwoływanie się do kolejnych znaków w stringu. Będziemy jednak potrzebować sposobu, by dowiedzieć się, z ilu znaków łańcuch tekstowy się składa.

W jednym z zadań do rozdziału trzeciego o zmiennych (*Liczba znaków w słowie*) należało skorzystać z metody, która zwraca liczbę znaków przechowywanych w zmiennej typu `String`. Ta metoda to `length`.

Spójrzmy na użycie obu metod, `charAt` oraz `length`, by "rozstrzelić" zapisany napis (tzn. wstawić po każdym znaku tekstu dodatkową spację):

Nazwa pliku: CharAtILengthPrzyklad.java

```java
public class CharAtILengthPrzyklad {
  public static void main(String[] args) {
    String tekst = "Ala ma kota";

    for (int i = 0; i < tekst.length(); i++) {
      System.out.print(tekst.charAt(i) + " ");
    }
  }
}
```

Użyliśmy metody `length`, by pobrać długość tekstu, a także metody `charAt`, by pobrać znak na danej pozycji (pamiętajmy, że pierwszy znak ma indeks `0`!). Wynik:

```
A l a   m a   k o t a
```

Zastosowana pętla **`for`** przechodzi po wszystkich liczbach, które są indeksami znaków w stringu `tekst`, zaczynając od `0`. Zauważmy, że warunek pętli korzysta z operatora < a nie <=. Gdybyśmy skorzystali z operatora <=, to zmienna `i` w ostatniej iteracji miałaby wartość `11` – tyle wynosi liczba znaków w zmiennej `tekst` i taką wartość zwraca `tekst.length()` – spowodowałoby to błąd, ponieważ indeks ostatniego znaku to `10`, a nie `11` – dlatego musieliśmy skorzystać z operatora <.

Spójrzmy na jeszcze jeden przykład – wypisujemy w nim tekst od końca:

Nazwa pliku: TekstOdKonca.java

```
public class TekstOdKonca {
  public static void main(String[] args) {
    String tekst = "Ala ma kota";

    for (int i = tekst.length() - 1; i >= 0; i--) {
      System.out.print(tekst.charAt(i));
    }
  }
}
```

Zwróćmy tutaj uwagę, iż zmienna `i`, zdefiniowana w pętli, zaczęła od indeksu ostatniego znaku w zmienej `tekst`, którego indeks równy jest liczbie znakow w tym stringu minus `1`, o czym już wspominaliśmy. W kroku pętli zmniejszamy ten indeks o `1`. W ciele pętli wypisujemy znak pod aktualnym indeksem, co skutkuje, że idziemy od końca tekstu do początku. Pętla kończy działanie, gdy zmienna `i` przekroczy indeks pierwszego znaku, czyli `0`.

Wynik:

```
atok am alA
```

5.10.1 Porównywanie znaków zwracanych przez charAt

Wykorzystywana przez nas metoda `charAt` zwraca pojedynczy znak, czyli wartość typu podstawowego **`char`**, a nie łańcuch tekstowy typu `String`.

Powoduje to, że możemy porównywać znaki zwracane przez metodę `charAt` z innymi znakami za pomocą operatora ==. Nie musimy, a nawet nie możemy, stosować w takim przypadku metody `equals`. Spójrzmy na przykład programu, który zlicza samogłoski w wyrazie podanym przez użytkownika:

Nazwa pliku: ZliczSamogloski.java

```
import java.util.Scanner;

public class ZliczSamogloski {
  public static void main(String[] args) {
    System.out.println("Podaj slowo:");
    String slowo = getString();

    int liczbaSamoglosek = 0;

    for (int i = 0; i < slowo.length(); i++) {
      char znak = slowo.charAt(i);

      if (znak == 'a' || znak == 'e' || znak == 'i' ||
          znak == 'u' || znak == 'o') {
        liczbaSamoglosek++;
      }
    }

    System.out.println(
        "Slowo " + slowo + " ma " + liczbaSamoglosek + " samoglosek."
    );
  }
```

```
  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

Program pobiera od użytkownika słowo i w pętli `for` przechodzi przez każdy znak, z którego się składa. Aktualny znak przypisujemy do zmiennej o nazwie `znak`, którą następnie przyrównujemy do samogłosek za pomocą operatora ==. Nie korzystamy z metody `equals`, ponieważ w tym przypadku nie działamy na stringach, lecz na znakach.

Warunek sprawdzający, czy `znak` to samogłoska, to złożone wyrażenie wykorzystujące operator logiczny || (lub) – jeżeli aktualny znak będzie którąkolwiek z samogłosek, to warunek będzie spełniony, więc zwiększymy liczbę znalezionych samogłosek o jeden za pomocą instrukcji `liczbaSamoglosek++;`.

Po przejściu przez całe słowo, wypisujemy na ekran informację o znalezionej liczbie samogłosek.

Przykładowe uruchomienie tego programu:

```
Podaj slowo:
programowanie
Slowo programowanie ma 6 samoglosek.
```

5.11 Podsumowanie

- Pętle służą do wykonywania instrukcji tak długo, jak określony warunek jest prawdziwy.
- Każdy obieg pętli nazywamy *iteracją*.
- Java posiada cztery rodzaje pętli: **while**, **do**...**while**, **for**, **for**-each (pętla **for**-each omówiona jest w rozdziale o tablicach).
- Pętla **while** wykonuje się, dopóki jej warunek jest spełniony:

```
while (warunek) {
  instrukcja1;
  instrukcja2;
  // ...
  instrukcjaN;
}
```

- Jeżeli warunek na wstępie pętli **while** nie będzie prawdziwy, to pętla nie wykona się ani razu.
- Pętla **do**...**while** zawsze wykona swoje instrukcje co najmniej raz, niezależnie od tego, czy jej warunek jest na wstępie spełniony, czy nie, ponieważ warunek jej zakończnia sprawdzany jest na końcu każdej iteracji. Składnia pętli **do**...**while** jest następująca:

```
do {
  instrukcja1;
  instrukcja2;
  // ...
  instrukcjaN;
} while (warunek);
```

- Pętla for posiada trzy elementy:

```
for (instrukcja_inicjalizująca; warunek; instrukcja_kroku) {
  instrukcja1;
  instrukcja2;
  // ...
  instrukcjaN;
}
```

  - ***instrukcję inicjalizującą*** – jest to instrukcja (bądź instrukcje rozdzielone przecinkami) wykonywana jednorazowo, jeszcze przed rozpoczęciem pętli,
  - ***warunek*** – jest to wyrażenie obliczane przed każdym obiegiem (iteracją) pętli – jeżeli jest prawdą (**true**), to wykonane zostaje ciało pętli,
  - ***instrukcja_kroku*** – jest to instrukcja wykonywana zaraz po zakończeniu każdego obiegu pętli.
- Podobnie jak w przypadku instrukcji warunkowych, zmienne zdefiniowane w pętlach nie są dostępne po zakończeniu pętli:

```
for (int z = 0; z <= 20; z += 2) {
  System.out.print(z + " ");
}

// spowoduje blad kompilacji - zmienna z juz tutaj nie istnieje!
System.out.println("z na koncu jest rowne " + z);
```

- Instrukcja **break** służy do przerwania aktualniej wykonującej się pętli. **Uwaga**: jeżeli pętla jest zagnieżdżona w innej pętli, to przerwana zostanie jedynie zagnieżdżona pętla.
- Instrukcja **continue** służy do przerwania aktualnie wykonującego się obiegu pętli i przejście do kolejnego obiegu (o ile warunek pętli jest spełniony). **Uwaga**: jeżeli pętla jest zagnieżdżona w innej pętli, to przerwany zostanie jedynie obieg zagnieżdżonej pętla.
- Można zastosować etykiety pętli, by wskazać, do której pętli odnosi się instrukcja **break** lub **continue**:

```
glowna_petla: for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 5; j++) {
    System.out.print(j + ", ");

    if (j == 1) break glowna_petla;
  }
  System.out.println();
}
```

- Instrukcji **break** oraz **continue** można używać w każdym rodzaju pętli: **for**, **while**, **do**..**while**, oraz **for**-each.
- Używając typu `String`, możemy sprawdzić, jaki znak jest na danej pozycji – służy do tego metoda `charAt`, której podajemy indeks znaku, który chcemy poznać. **Uwaga**: indeks pierwszego znaku to `0`, a nie `1`! Przykład:

```
String tekst = "Ala ma kota";

// na ekranie zobaczymy tekst: A l a  m a  k o t a
for (int i = 0; i < tekst.length(); i++) {
  System.out.print(tekst.charAt(i) + " ");
}
```

- Metoda `charAt` zwraca wartość typu podstawowego **char** – jeżeli chcemy sprawdzić taką wartość, to porównujemy ją za pomocą operatora ==. Dla przykładu, aby sprawdzić, czy `'A'` jest pierwszą literą poniższej zmiennej:

```
String tekst = "Ala ma kota";
```

  powinniśmy skorzystać z warunku `tekst.charAt(0) == 'a'`.
- Metoda `length` typu `String` zwraca liczbę znaków, z których składa się zmienna typu `String`. Dla przykładu, dla powyższej zmiennej `tekst`, użycie `tekst.length()` zwraca `11`.

5.12 Pytania

1. Do czego służą pętle?
2. Czym różnią się od siebie poznane dotąd pętle?
3. Jak nazywamy obieg pętli?
4. Czy ciało pętli zawsze wykona się chociaż raz?
5. Co się stanie, gdy warunek pętli będzie zawsze spełniony i nie zmieni się w trakcie działania programu?
6. Z jakich części składa się pętla `for`? Czy są one wymagane?
7. Jak sprawdzić znak na danej pozycji w zmiennej typu `String`? Jak sprawdzić pierwszy znak, a jak ostatni?
8. Jak sprawdzić z ilu znaków składa się zmienna typu `String`?
9. Czy poniższe fragmenty kodów źródłowych są sobie równoważne?

```
int i = 0;
while (i < 10) {
  System.out.print(i + " ");
  i++;
}
```

```
for (int i = 0; i < 10; i++) {
  System.out.print(i + " ");
}
```

10. Do czego służą instrukcje `break` oraz `continue`?
11. Jaki będzie wynik działania poniższego fragmentu kodu?

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 10; j++) {
    System.out.println("j = " + j);

      if (j == 1) {
        break;
      }
  }
}
```

12. Co zostanie wypisane w ramach działania poniższego fragmentu kodu?

```
String komunikat = "Witaj";

for (int i = 0; i <= komunikat.length(); i++) {
  System.out.print(komunikat.charAt(i) + " ");
}
```

13. Przepisz kod poniżej tak, by używał pętli `while`:

```
for (int i = 1, j = 1; i * j < 100; i++, j += 2) {
  System.out.print((i * j) + " ");
}
```

5.13 Zadania

5.13.1 While i liczby od 1 do 10

Napisz program z użyciem pętli `while`, który wypisuje wszystkie liczby od `1` do `10` (włącznie), oddzielone przecinkami, poza liczbą `10`, po której nie powinno być przecinka.

5.13.2 Policz silnię

Napisz program, który policzy i wypisze silnię liczby, którą poda użytkownik. Silnia to iloczyn kolejnych liczb całkowitych od `1` do danej liczby, np. *silnia* `5` to `1 * 2 * 3 * 4 * 5`, czyli `120`. Silnia liczby `0` to `1`.

5.13.3 Palindrom

Napisz program, który odpowie na pytanie, czy podane przez użytkownika słowo jest *palindromem*. Palindrom to słowo, które jest takie samo czytane od początku i od końca, np. kajak.

5.13.4 Wypisz największą liczbę z podanych

Napisz program, który z liczb podanych przez użytkownika wypisze największą. Program po pobraniu każdej liczby powinien pytać, czy użytkownik chce podać kolejną liczbę. Po podaniu liczb, program powinien wypisać największą z nich.

5.13.5 Zagnieżdżone pętle

Napisz program z dwoma pętlami (jedna zagnieżdżona w drugiej), każda z pętli powinna iterować od `1` do `10`.

1. Pętla główna powinna pomijać swoje iteracje za pomocą instrukcji `continue`, gdy jej zmienna jest nieparzysta.
2. Pętla zagnieżdżona powinna wypisywać wartość swojej zmiennej. Następnie, gdy zmienna pętli zagnieżdżonej jest większa od zmiennej pętli głównej, pętla zagnieżdżona powinna spowodować, że przejdziemy do kolejnej iteracji pętli głównej (w tym przypadku skorzystaj z etykiety i instrukcji `continue`).

5.13.6 Kalkulator

Napisz program, który będzie pobierał od użytkownika liczby i działania do wykonania na nich. Program powinien wypisywać wynik po każdym działaniu. Możliwe działania to:

- * mnożenie,
- / dzielenie,
- - odejmowanie,
- + dodawanie.

Jeżeli podane zostanie inne działanie, lub podana zostanie liczba `0` jako dzielnik podczas dzielenia, program powinien wypisać stosowny komunikat i ponownie pobrać od użytkownika dane.

Na początku, program powinien pobrać od użytkownika dwie liczby i działanie do wykonania na nich. Za każdy kolejnym razem, program powinien pobierać od użytkownika już tylko jedną liczbę

i działanie, po czym powinien wykonać podane działanie na poprzednim wyniku i podanej liczbie.

Dla przykładu:

1. Program pobiera najpierw dwie liczby od użytkownika: `10` i `15`, oraz działanie: dodawanie.
2. Program dodaje do siebie liczby i wypisuje wynik `25` na ekran.
3. Program pyta, czy użytkownik chce wykonać kolejne działanie.
   a) Jeżeli nie, program kończy działanie.
   b) Jeżeli tak, to program pobiera jedną liczbę i działanie, np. `2` i mnożenie. Program mnoży poprzedni wynik działania – czyli `25 * 2` i wypisuje wynik `50` na ekran. Wracamy do punktu 3. i ponownie pytamy o chęć dalszych kalkulacji.

5.13.7 Choinka

Napisz program, który pobierze od użytkownika jedną liczbę całkowitą. Następnie, program powinien wypisać na ekran choinką ze znaków *, gdzie w ostatniej linii będzie liczba gwiazdek podana przez użytkownika, a w każdej powyższej o dwie gwiazdki mniej, niż w poniższej.

Przykład pierwszy – użytkownika podał liczbę `5`, efekt wyświetlony na ekranie:

```
  *
 ***
*****
```

Przykład drugi – użytkownik podał liczbę `6`, efekt na ekranie:

```
  **
 ****
******
```

6 Rozdział VI – Tablice

W tym rozdziale:

- dowiemy się, czym są tablice,
- nauczymy się, jak definiować tablice oraz z nich korzystać,
- zobaczymy, jak wykorzystywać pętle w pracy z tablicami,
- opowiemy sobie o tablicach wielowymiarowych,
- poznamy czwarty rodzaj pętli: pętlę `for`-each.

6.1 Czym są tablice?

Tablice to ciągi obiektów tego samego typu. Obiekty te nazywamy *elementami* tablicy. Są one przechowywane jeden za drugim.

Tablice mają określony podczas ich tworzenia rozmiar, który nie może ulec zmianie. Raz utworzona tablica będzie miała ten sam rozmiar w trakcie działania naszego programu. Rozmiar ten możemy odczytać za pomocą pola `length` każdej tablicy.

Aby odnieść się do danego elementu tablicy, będziemy stosować *indeks* tego elementu oraz operator [ ] (nawiasy kwadratowe). Indeksy elementów tablicy zaczynają się od `0`, a nie `1`, podobnie, jak w przypadku znaków, z których składają się zmienne typu `String`, gdy chcemy pobrać je za pomocą metody `charAt`.

Tablice to pierwszy rodzaj *kolekcji*, jaki poznamy. *Kolekcje* to zbiór powiązanych ze sobą elementów. Kolekcje są bardzo często wykorzystywane w programowaniu, ponieważ często mamy potrzebę przechowywać w programach tablice (bądź listy, zbiory itp.) wielu elementów, a nie pojedyncze wartości.

Dla przykładu, wyobraźmy sobie, że chcielibyśmy wczytać od użytkownika ciąg liczb, a następnie go posortować. Zamiast umieszczać każdą liczbę w osobnej zmiennej, możemy zdefiniować tablicę liczb i zapisać w jej elementach dane wczytane od użytkownika.

Innym przykładem może być lista osób w programie "książka adresowa" czy też zbiór zamówień i produktów pobrany z bazy danych – lista zastosowań tablic i innych kolekcji nie ma końca.

> *Pisząc* "kolekcje" *nie mam tutaj na razie na myśli interfejsu* `Collection` *i implementujących go klas, które poznamy w jednym z późniejszych rozdziałów, lecz po prostu "zestaw powiązanych ze sobą elementów".*

6.2 Definiowanie i używanie tablic

Tablice definiujemy w następujący sposób:

```
typ[] nazwaZmiennej;
```

Przykładowo, tablica, która może przechowywać liczby typu `int`, to:

```
int[] mojaTablica;
```

Różnica względem tego, jak do tej pory definiowaliśmy zmienne, jest taka, że po nazwie typu umieściliśmy nawiasy kwadratowe – wskazuje to kompilatorowi, że zmienna `mojaTablica` jest tablicą. Zauważmy, że zmienna `mojaTablica` nie jest typu `int`, lecz jest *tablicą, która może przechowywać wiele elementów typu* `int`.

Zanim będziemy mogli używać tablicy, musimy ją *utworzyć*. Aby to zrobić, możemy:

- utworzyć tablicę za pomocą nowego słowa kluczowego `new`, podając ile maksymalnie elementów będzie ona mogła przechowywać,
- zainicjalizować tablicę w momencie jej definiowania wartościami zawartymi w nawiasach klamrowych,
- skorzystać z rozwiązania będącego połączeniem dwóch powyższych – utworzenia nowej tablicy z pomocą słowa kluczowego `new` wraz z podaniem początkowych elementów, z których tablica ma się składać.

Spójrzmy na przykład każdego z powyższych sposobów:

Nazwa pliku: TworzenieIUzywanieTablic.java

```
public class TworzenieIUzywanieTablic {
  public static void main(String[] args) {
    // tablica, ktora moze przechowywac maksymalnie 5 wartosci calkowitych
    int[] calkowite = new int[5];

    // tablica, ktora moze przechowywac maksymalnie 3 wartosci rzeczywiste
    //  wstepnie zainicjalizowane wartosciami 3.14, 5, -20.5
    double[] rzeczywiste = { 3.14, 5, -20.5 };

    // tablica, ktora bedzie mogla przechowywac ciagi znakow
    // na razie nie podalismy, ile wartosci typu String
    // ta tablica bedzie mogla przechowywac
    String[] slowa;

    // tworzymy tablice, ktora bedzie mogla miec maksymalnie
    // trzy elementy, i inicjalizujemy ja trzema elementami,
    // kolejno: Ala, ma, kota
    slowa = new String[] { "Ala", "ma", "kota" };
  }
}
```

W powyższym programie utworzyliśmy trzy tablice – spójrzmy, w jaki sposób to osiągnęliśmy:

1. Pierwsza tablica, `calkowite`, może przechowywać maksymalnie 5 wartości o typie `int`. Należy tutaj zwrócić uwagę, że użyliśmy nowego słowa kluczowego `new`, aby utworzyć tablicę – po nim następuje nazwa typu, jaki ma przechowywać tablica, a w nawiasach kwadratowych – z ilu elementów podanego typu tablica będzie mogła się składać.

2. Druga tablica, o nazwie `rzeczywiste`, może przechowywać maksymalnie 3 wartości o typie **`double`**. Ta druga tablica została zainicjalizowana trzema wartościami typu **`double`**, które zostały zawarte w nawiasach klamrowych { }: `3.14`, `5`, `-20.5` – w tej właśnie kolejności.
3. Ostatnia tablica, o nazwie `slowa`, to tablica wartości typu `String`. Zauważmy, że nie zainicjalizowaśmy tej tablicy żadną wartością – dopiero w linii poniżej korzystamy ze słowa kluczowego **`new`**, jak w przypadku tablicy `calkowite`, z tym, że nie podaliśmy rozmiaru tablicy – zamiast tego, po nawiasach kwadratowych [ ], podaliśmy w nawiasach klamrowych { } wartości, jakimi elementy tablicy mają zostać zainicjalizowane.

Przykładowa reprezentacja tablicy `rzeczywiste` mogłaby być następująca:

Indeks	**0**	**1**	**2**
Wartość	`3.14`	`5`	`-20.5`

Co ważne, rozmiar każdej z powyższych tablic został ustalony w momencie ich tworzenia – tablica `calkowite` ma ustalony rozmiar pięciu elementów, a tablice `rzeczywiste` i `slowa` – trzy elementy.

Drugi z powyższych sposobów można użyć tylko w momencie inicjalizacji tablicy. W poniższym kodzie, druga linia spowodowałaby błąd kompilacji:

```
double[] rzeczywiste = { 3.14, 5, -20.5 };
reczywiste = { 1, 2, 3 }; // blad kompilacji!
```

Kiedy stosować każdy z tych sposobów?

1. Jeżeli nie mamy konkretnych wartości początkowych, którymi chcemy zainicjalizować tablicę, korzystamy z pierwszego sposobu – podajemy liczbę elementów, które tablica będzie mogła przechowywać. Wartości konkretnych elementów przypiszemy w dalszej części programu.
2. Jeżeli znamy konkretne wartości, które tablica ma przechowywać, możemy użyć drugiego sposobu (podania wartości w nawiasach klamrowych) podczas definiowania tablicy.
3. Trzeciego sposobu możemy użyć, jeżeli chcemy w jednym kroku utworzyć tablicę i zainicjalizować ją pewnymi wartościami, ale nie w momencie definiowania zmiennej.

Mając już zmienne tablicowe, spójrzmy teraz, jak odnieść się do ich poszczególnych elementów.

Tablice można też definiować z nawiasami kwadratowymi zapisanymi po nazwie zmiennej, zamiast po typie:

```
int mojaTablica[];
```

My będziemy jednak stosować sposób definiowania tablic, w którym nawiasy kwadratowe stawiamy po nazwie typu.

6.2.1 Odnoszenie się do elementów tablicy

Aby odnieść się do poszczególnych elementów tablicy, używamy operatora [ ] (nawiasy kwadratowe), któremu podajemy indeks elementu, do którego chcemy się odnieść.

> Elementy tablicy w języku Java, a także w wielu innych językach programowania, zaczynają się od indeksu `0`, a nie `1`!

Przykład użycia tablic z programu o definiowaniu i tworzeniu tablic:

Nazwa pliku: TworzenieIUzywanieTablic.java

```
// ustawiamy wartosc pierwszego, drugiego, i piatego elementu tablicy
calkowite[0] = 10;  // pierwszy element ma indeks 0 (a nie 1)!
calkowite[1] = 15;
calkowite[4] = 200; // piaty element ma indeks 4 (a nie 5)!

System.out.println(
    "Suma dwoch pierwszych elementow to " + (calkowite[0] + calkowite[1])
);

// zmieniamy wartosc pierwszego elementu
rzeczywiste[0] = 100;
System.out.println(
    "Pierwszy element tablicy rzeczywiste = " + rzeczywiste[0]
);

// zmieniamy pierwszy i trzeci (czyli ostatni) element
slowa[0] = "Ania";
slowa[2] = "psa";
System.out.println(slowa[0] + " " + slowa[1] + " " + slowa[2]);
```

Wynik działania:

```
Suma dwoch pierwszych elementow to 25
Pierwszy element tablicy rzeczywiste = 100.0
Ania ma psa
```

Z pomocą użycia operatora [ ] możemy zarówno wskazać element, który chcemy zmienić, jak i element, który chcemy pobrać z tablicy. Zwróćmy jeszcze raz uwagę, że pierwszy element w każdej tablicy ma indeks `0`, a nie `1`, oraz ostatni element ma indeks o jeden mniejszy, niż liczba wszystkich elementów w tablicy.

Jeżeli spróbowalibyśmy odnieść się do elementu tablicy, podając indeks, który wykracza poza zakres tablicy, to wykonanie naszego programu zakończy się błędem:

Nazwa pliku: WyjsciePozaZakresTablicy.java

```
public class WyjsciePozaZakresTablicy {
  public static void main(String[] args) {
    int[] tablica = { 1, 2, 3 };

    // element o indeksie 3 nie istnieje!
    // ostatni (trzeci) element tablicy ma indeks 2
    // kod sie skompiluje, ale w trakcie dzialania programu pojawi sie blad
    System.out.println(tablica[3]);
  }
}
```

Po uruchomieniu programu zobaczymy następujący błąd:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at WyjsciePozaZakresTablicy.main(WyjsciePozaZakresTablicy.java:8)
```

Java poinformowała nas, że wyszliśmy poza zakres tablicy (a także wskazała numer problematycznej linii kodu). W wyniku wystąpienia tego błędu, nasz program zakończył działanie.

Nazwa tego konkretnego błędu, to `ArrayIndexOutOfBoundsException` – jest to dość często widywany błąd – jeszcze nie raz go zobaczymy!

Zwróćmy jeszcze uwagę, że kompilator nie miał obiekcji odnośnie powyższego programu. Z punktu widzenia kompilatora, nasz program jest poprawny – nie ma w nim żadnych błędów składniowych. Błąd wydarzył się dopiero w momencie działania programu – tego typu błędu są zdecydowanie gorsze, niż błędy wskazywane przez kompilator, ponieważ ujawniają się dopiero w momencie działania programu, często wtedy, gdy korzystają z niego nasi użytkownicy.

6.2.2 Domyślne wartości w tablicach

Pytanie: jeżeli utworzymy tablicę `5` elementów, nie podając, jakie wartości mają być w niej na wstępie przechowywane, to jakie wartości będą pod kolejnymi indeksami `0`, `1`, `2`, `3`, `4`?

Wartości tablic inicjalizowane są domyślną wartością danego typu. W przypadku typów liczbowych jest to `0`, dla typu **`boolean`** jest to **`false`**, a dla typów złożonych (takich jak `String`) jest to wartość **`null`**, o której dokładnie opowiemy sobie w rozdziale o klasach. Spójrzmy na poniższy przykład:

Nazwa pliku: DomyslneWartosciWTablicach.java

```
public class DomyslneWartosciWTablicach {
  public static void main(String[] args) {
    int[] calkowite = new int[5];

    calkowite[0] = 10;
    calkowite[1] = 15;

    // trzeci element nie byl ustawiony - bedzie mial
    //  domyslna wartosc typu int, czyli 0
    System.out.println("Trzeci element: " + calkowite[2]);
  }
}
```

W wyniku uruchomienia tego programu, na ekranie zobaczymy:

```
Trzeci element: 0
```

Jak widzimy, trzeci element ma wartość `0` – nie ustawiliśmy nigdzie tej wartości – została domyślnie nadana wszystkim elementom tablicy `calkowite` w momencie jej tworzenia za pomocą słowa kluczowego **`new`**. Potem zmienilismy dwa pierwsze elementy, nadając im wartości `10` i `15`. Pozostałe trzy elementy nadal mają wartość `0`.

Zmienne, które definiowaliśmy do tej pory, nigdy nie miały wartości domyślnych – zawsze trzeba było przypisać zmiennej pewną wartość, zanim tej zmiennej mogliśmy zacząć używać. Jak widzimy powyżej, elementy tablic zachowują się inaczej i mają nadawane wartości domyślne.

6.2.3 Sprawdzanie liczby elementów tablicy

Zmienną tablicową można "odpytać" o liczbę elementów, które przechowuje, korzystając z atrybutu tablicy o nazwie `length`:

Nazwa pliku: SprawdzRozmiarTablicy.java

```
public class SprawdzRozmiarTablicy {
  public static void main(String[] args) {
    int[] calkowite = new int[5];
    double[] rzeczywiste = { 3.14, 5, -20.5 };

    System.out.println(
        "Liczba elementow w tablicy calkowite: " + calkowite.length
    );

    System.out.println(
        "Liczba elementow w tablicy rzeczywiste: " + rzeczywiste.length
    );
  }
}
```

Wynik działania tego programu:

```
Liczba elementow w tablicy calkowite: 5
Liczba elementow w tablicy rzeczywiste: 3
```

Aby sprawdzić rozmiar tablicy, korzystamy z atrybutu o nazwie `length`, do którego odnosimy się poprzez napisanie kropki po nazwie tablicy.

W jednym z poprzednich rozdziałów zobaczyliśmy, że zmienną typu `String` także możemy odpytać o liczbę znaków, z których się składa, i także stosujemy w tym celu `length` – jest jednak znacząca różnica pomiędzy sprawdzaniem liczby znaków w stringach a liczbą elementów w tablicy.

W przypadku stringów, korzystamy z *metody* `length`, więc po jej nazwie musimy zawsze podać nawiasy ( ). W przypadku tablic, korzystamy z *atrybutu* `length`, więc z nawiasów nie korzystamy. O różnicach pomiędzy metodami a atrybutami porozmawiamy w rozdziale o klasach.

Spójrzmy na przykład użycia `length` na tablicy i na zmiennych typu `String`:

Nazwa pliku: RozmiarStringITablicy.java

```
public class RozmiarStringITablicy {
  public static void main(String[] args) {
    String tekst = "Witajcie!";

    String[] slowa = { "Ania", "ma", "kota" };

    System.out.println("Liczba slow w zmiennej tekst: " + tekst.length());

    System.out.println("Liczba elementow w tablicy: " + slowa.length);

    System.out.println(
        "Liczba znakow w pierwszym slowie z tablicy: " + slowa[0].length()
    );
  }
}
```

Wynik działania tego przykładu jest następujący:

```
Liczba slow w zmiennej tekst: 9
Liczba elementow w tablicy: 3
Liczba znakow w pierwszym slowie z tablicy: 4
```

Zwróćmy uwagę, że w pierwszej instrukcji `System.out.println` wypisujemy na ekran liczbę znaków w łańcuchu tekstowym – korzystamy więc z `length()` z nawiasami.

W kolejnej instrukcji `System.out.println`, wypisujemy liczbę elementów tablicy – korzystamy więc z `length` bez nawiasów.

W ostatniej instrukcji `System.out.println` odnosimy się do pierwszej wartości zawartej w tablicy – wartość ta jest typu `String`, bo tak została zdefiniowana tablica `slowa` – przechowuje ona wartości typu `String`. Sprawdzamy, ile znaków ma pierwszy element (pierwszy string w tej tablicy) za pomocą `length()` – z nawiasami, bo działamy na wartości typu `String`.

6.3 Użycie pętli z tablicami

Pętle świetnie nadają się do *iterowania* (czyli przechodzenia) przez elementy tablicy. Możemy w tym celu skorzystać z pętli `for` w następujący sposób:

1. Zmienną, używaną w pętli, zainicjalizujemy wartością `0`, która jest indeksem pierwszego elementu w każdej tablicy.
2. Warunek pętli napiszemy w taki sposób, by pętla zakończyła się, gdy wartość zmiennej przekroczy zakres indeksów tablicy – skorzystamy tutaj z atrybutu `length` tablicy.
3. W instrukcji kroku będziemy zwiększać wartość zmiennej pętli o `1`, dzięki czemu przejdziemy przez indeksy wszystkich elementów tablicy.

Spójrzmy na prosty przykład programu, który wypisujemy na ekran wszystkie elementy tablicy:

Nazwa pliku: WypiszElementyTablicy.java

```
public class WypiszElementyTablicy {
  public static void main(String[] args) {
    int[] liczby = { -5, 100, 1, 0, 20 };

    for (int i = 0; i < liczby.length; i++) {
      System.out.print(liczby[i] + " ");
    }
  }
}
```

Pętlę `for` zapisaliśmy zgodnie z powyższym opisem. Dzięki temu, w prosty sposób jesteśmy w stanie przejść po kolei przez wszystkie elementy tablicy `liczby`. Na ekranie zobaczymy:

```
-5 100 1 0 20
```

Zwróćmy uwagę, że w warunku pętli korzystamy z operatora < a nie <=. Jak już wiemy, `length` zwraca liczbę elementów, a ostatni element w tablicy ma zawsze indeks o jeden mniejszy od liczby elementów (ponieważ numeracja indeksów rozpoczyna się od `0`). Gdybyśmy użyli z operatora <=, to w ostatnim obiegu pętli zmienna `i` wynosiłaby `5`, a indeks `5` jest poza zakresem tablicy `liczby` – w takim przypadku, na ekranie zobaczylibyśmy błąd `ArrayIndexOutOfBoundsException`.

Moglibyśmy również przejść przez elementy tablicy w odwrotnej kolejności – wystarczy odwrócić zapis pętli `for`:

Nazwa pliku: WypiszElementyTablicy.java

```
for (int i = liczby.length - 1; i >= 0 ; i--) {
  System.out.print(liczby[i] + " ");
}
```

Tym razem, zaczynamy od ostatniego indeksu i kierujemy się do pierwszego. Na ekranie zobaczymy:

```
20 0 1 100 -5
```

Spójrzmy na bardziej złożony przykład – program, który odpowiada na pytanie, czy w tablicy liczb znajduje się dana liczba:

Nazwa pliku: CzyLiczbaJestWTablicy.java

```
public class CzyLiczbaJestWTablicy {
  public static void main(String[] args) {
    boolean znaleziona = false;
    int[] liczby = { -20, 105, 0, 26, -99, 7, 1026 };

    int szukanaLiczba = 7;

    for (int i = 0; i < liczby.length; i++) {
      if (liczby[i] == szukanaLiczba) {
        znaleziona = true;
        break; // znalezlismy liczbe - mozemy wiec przerwac petle
      }
    }

    if (znaleziona) {
      System.out.println("Liczba zostala znaleziona!");
    } else {
      System.out.println("Liczba nie zostala znaleziona.");
    }
  }
}
```

Na początku programu tworzymy tablicę `liczby` zainicjalizowaną kilkoma liczbami. Szukaną liczbę zapisujemy natomiast w zmiennej `szukanaLiczba`. Zmienna `znaleziona` służy jako wyznacznik tego, czy udało nam się znaleźć liczbę, czy nie. Początkowo nadajemy tej zmiennej wartość **`false`** – jeżeli odnajdziemy liczbę w pętli, to zmienimy wartość na **`true`**.

Następnie, przechodzimy w pętli przez elementy tablicy `liczby` i porównujemy każdy z nich z szukaną liczbą. Jeżeli liczby są sobie równe, to znaczy, że znaleźliśmy szukaną liczbę w tablicy. Ustawiamy zmienna `znaleziona` na **`true`** oraz przerywamy pętlę – dalsze procesowanie elementów tablicy nie ma sensu, ponieważ już wiemy, że tablica zawiera szukaną liczbę.

Na końcu programu wypisujemy na ekran komunikat, w zależności do tego, czy znaleźliśmy szukaną liczbę, czy nie.

Szukana liczba to `7`. W kodzie programu w tablicy `liczby` znajduje się taka liczba – dlatego na ekranie zobaczymy następujący komunikat:

```
Liczba zostala znaleziona!
```

Szukanie konkretnych elementów w tablicach (i ogólnie w kolekcjach) to częste wymaganie w programowaniu. W zależności od rodzaju kolekcji i rozłożeniu w niej danych, czasy potrzebne na znalezienie elementu bardzo się różnią. Opowiemy sobie więcej o tym problemie w rozdziale o kolekcjach języka Java.

6.4 Tablice wielowymiarowe

Tablice mogą mieć więcej, niż jeden wymiar. Jeżeli zdefiniujemy *tablicę tablic*, to wtedy będzie to tablica dwuwymiarowa – każdy element tablicy będzie kolejną tablicą, do której będziemy mogli się odnieść za pomocą jej indeksów.

Można zdefniować tablice o większej liczbie wymiarów, ale w praktyce tablice o więcej niż trzech wymiarach nie są wykorzystywane.

Aby zdefiniować tablicę wielowymiarową, dodajemy do definicji kolejny zestaw nawiasów kwadratowych:

```
int[][] tablica2d = new int[3][5];
```

Zdefiniowana powyżej tablica to *tablica trzech tablic*, z których każda może przechowywać pięć wartości typu `int`.

Aby skorzystać z tej tablicy i odnieść się do konkretnego elementu pod-tablicy, dodajemy kolejny zestaw nawiasów kwadratowych [ ]:

Nazwa pliku: TablicaDwuwymiarowa.java

```
public class TablicaDwuwymiarowa {
  public static void main(String[] args) {
    int[][] tablica2d = new int[3][5];

    // pierwszy element pierwszej pod-tablicy
    tablica2d[0][0] = 5;

    // drugi, trzeci, i czwarty element drugiej pod-tablicy
    tablica2d[1][1] = 10;
    tablica2d[1][2] = 100;
    tablica2d[1][3] = 1000;

    // drugi element trzeciej pod-tablicy
    tablica2d[2][1] = 50;

    System.out.println(tablica2d[0][0]);
    System.out.println(tablica2d[0][4]);
    System.out.println(tablica2d[1][3]);
    System.out.println(tablica2d[2][3]);
  }
}
```

W tym przykładzie odnosimy się do każdej z trzech pod-tablic – indeks podany w pierwszym zestawie nawiasów kwadratowych [ ] to indeks pod-tablicy, do której chcemy się odwołać, więc zapis:

```
tablica2d[0]
tablica2d[1]
tablica2d[2]
```

to odniesienie się do, kolejno, pierwszej, drugiej, i trzeciej pod-tablicy (bo indeksy zaczynamy od `0`). Skoro elementami `tablica2d` są inne tablice, to możemy teraz dopisać kolejny zestaw nawiasów kwadratowych z indeksem, aby odnieść się do konkretnego elementu pod-tablicy. Dla przykładu, zapis:

```
tablica2d[0][0]
```

to odniesienie się do pierwszego elementu pierwszej pod-tablicy. Z kolei:

```
tablica2d[2][3]
```

to odniesienie się do czwartego elementu trzeciej pod-tablicy.

W powyższym przykładowym programie ustawiamy wartości kilku elementów pod-tablic oraz kilka z nich wypisujemy – zauważmy, że wypisujemy kilka elementów, których wartości nie ustawiliśmy – jak już wiemy, domyślne wartości, jakie przyjmują niezainicjalizowane elementy tablic typu `int` to zera – stąd na ekranie zobaczymy:

```
5
0
1000
0
```

Możemy dowiedzieć się, ile ma każdy z wymiarów tablicy wielowymiarowej używając atrybutu `length`. Poniżej pokazano, jak odczytać rozmiar obu wymiarów:

Nazwa pliku: TablicaDwuwymiarowa.java

```
System.out.println("Pierwszy wymiar: " + tablica2d.length);
System.out.println("Drugi wymiar: " + tablica2d[0].length);
```

Wynik działania:

```
Pierwszy wymiar: 3
Drugi wymiar: 5
```

W poprzednim rozdziale zobaczyliśmy, że pętle w prosty sposób umożliwiają nam przechodzenie przez elementy tablicy. Aby przejść przez tablicę wielowymiarową, możemy użyć zagnieżdżonych pętli:

Nazwa pliku: TablicaDwuwymiarowa.java

```
for (int i = 0; i < tablica2d.length; i++) {
  for (int j = 0; j < tablica2d[i].length; j++) {
    System.out.print(tablica2d[i][j] + ", ");
  }

  System.out.println(); // nowa linia
}
```

W tym przykładzie skorzystaliśmy z zagnieżdżonej pętli `for` – w pętli zewnętrznej iterujemy po "głównym" wymiarze tablicy – w tym przypadku, ma on trzy elementy – trzy pod-tablice. W pętli wewnętrznej przechodzimy natomiast przez wszystkie elementy każdej pod-tablicy (gdzie każda z nich ma pięć elementów).

Zwróćmy uwagę na to, jak zapisany jest warunek pętli wewnętrznej:

```
j < tablica2d[i].length
```

Tak zapisany warunek będzie korzystał z rozmiaru każdej pod-tablicy podczas wykonywania się pętli wewnętrznej.

Na ekranie zobaczymy następujące wartości (część z widocznych wartości ustawiliśmy w jednym z fragmentów kodu powyżej – pozostałe to niezainicjalizowane wartości domyślne, czyli zera):

```
5, 0, 0, 0, 0,
0, 10, 100, 1000, 0,
0, 50, 0, 0, 0,
```

6.4.1 Inicjalizacja tablic wielowymiarowych

Tablicę jednowymiarową inicjalizowaliśmy w następujący sposób:

```
double[] rzeczywiste = { 3.14, 5, -20.5 };
```

By zainicjalizować tablicę więcej niż jednego wymiaru, należy użyć kolejnego zestawu nawiasów klamrowych:

Nazwa pliku: TablicaDwuwymiarowaInicjalizacja.java

```
public class TablicaDwuwymiarowaInicjalizacja {
  public static void main(String[] args) {
    char[][] kwadrat = {
        { 'X', 'X', 'X', 'X' },
        { 'X', 'O', 'O', 'X' },
        { 'X', 'O', 'O', 'X' },
        { 'X', 'X', 'X', 'X' }
    };

    for (int rzad = 0; rzad < kwadrat.length; rzad++) {
      for (int kolumna = 0; kolumna < kwadrat[rzad].length; kolumna++) {
        System.out.print(kwadrat[rzad][kolumna]);
      }

      System.out.println(); // nowa linia
    }
  }
}
```

Aby zainicjalizować tablicę dwuwymiarową `kwadrat`, w "głównych" nawiasach klamrowych { } zamieściliśmy kolejne zestawy nawiasów klamrowych, w których znajdują się wartości każdej z pod-tabel, jakie mają się w nich znajdować.

W zagnieżdżonej pętli przechodzimy przez wszystkie pod-tablice i wypisujemy je na ekran.

W wyniku działania powyższego programu, na ekranie zobaczymy:

```
XXXX
XOOX
XOOX
XXXX
```

6.5 Pętla for-each

W poprzednim rozdziale poznaliśmy trzy rodzaje pętli: **while**, **do..while**, oraz **for**. Języka Java posiada jeszcze jeden rodzaj pętli: **for**-each. Ten ostatni rodzaj pętli poznajemy dopiero teraz, ponieważ jest on dedykowany do pracy z tablicami i innymi kolekcjami.

Pętla **for**-each pozwala na łatwe iterowanie po tablicy bądź kolekcji. Jej składnia jest następująca:

```
for (typ nazwaZmiennejPetli : tablica_badz_kolekcja) {
  // instrukcje
}
```

W pętli tej definiujemy *zmienną pętli* oraz jej typ, a po dwukropku podajemy tablicę (bądź kolekcję), którą chcemy przeprocesować. Typ zmiennej pętli powinien być zgodny z typem, jakie tablica może przechowywać. Zmienna pętli w każdym obiegu będzie zawierała wartość kolejnego elementu tablicy. Pętla kończy się, gdy przejdzie przez wszystkie elementy tablicy.

Spójrzmy na przykład programu, który korzysta z pętli **for**-each, by wypisać elementy tablicy:

Nazwa pliku: ForEachWypiszElementy.java

```
public class ForEachWypiszElementy {
  public static void main(String[] args) {
    int[] liczby = { 1, 5, 20 };

    for (int x : liczby) {
      System.out.print(x + ", ");
    }
  }
}
```

W tym programie zmienna pętli nazywa się `x` i w każdym obiegu pętli przyjmuje wartość z tablicy `liczby`. Będą to, kolejno, liczby: `1`, `5`, oraz `20`, ponieważ takie liczby przechowuje tablica `liczby`.

Zauważmy, że typ **int** zmiennej `x` jest zgodny z typem wartości, jakie przechowuje tablica `liczby`.

Ten program wypisze na ekran:

```
1, 5, 20,
```

Pętla **for**-each ma dwie istotne cechy:

- zawsze procesuje ona elementy w kolejności od pierwszego do ostatniego,
- zawsze procesowane są wszystkie elementy tablicy (o ile nie skorzystamy z instrukcji **break**).

Pętla **for**-each jest wygodną alternatywą dla pętli **for**, ponieważ jest krótsza – nie musimy sami zadbać o nadanie zmiennej pętli odpowiedniej wartości, napisać warunku pętli, ani instrukcji kroku – wszystko dzieje się automatycznie. Wystarczy napisać ciało pętli.

Jeżeli potrzebujemy przeiterować po elementach tablicy, a indeks nie jest nam potrzebny, to powinniśmy stosować pętlę **for**-each, ponieważ jest prostsza w zapisie niż jej alternatywy. Nieraz zdarza się jednak, że potrzebujemy przeiterować tylko po części elementów, bądź w innej kolejności, lub jest nam potrzebny indeks – wtedy będziemy chcieli skorzystać np. z pętli **for**.

*W pętlach **for**-each możemy korzystać z instrukcji **break** oraz **continue** tak samo, jak w poprzednio poznanych rodzajach pętli.*

6.6 Porównywanie tablic i zmiana rozmiaru tablic

Prędzej czy później będziemy w naszych programach potrzebowali porownać ze sobą dwie tablice, by odpowiedzieć na pytanie, czy zawierają takie same elementy. Spróbujmy porównać dwie tablice za pomocą operatora ==:

Nazwa pliku: PorownywanieTablicOperatorRownosci.java

```
public class PorownywanieTablicOperatorRownosci {
  public static void main(String[] args) {
    int[] pierwszaTablica = { 10, 5, 20 };
    int[] drugaTablica = { 10, 5, 20 };

    if (pierwszaTablica == drugaTablica) {
      System.out.println("Tablice sa takie same.");
    } else {
      System.out.println("Tablice nie sa takie same.");
    }
  }
}
```

Ku potencjalnemu zdziwieniu, na ekranie zobaczymy komunikat:

```
Tablice nie sa takie same.
```

Tablice te mają co prawda takie same elementy, lecz każda z nich jest osobnym *obiektem*. Tablice to *typy referencyjne* (złożone), podobnie jak typ `String`, a w przeciwieństwie do ośmiu poznanych już typów podstawowych (`int`, `boolean`, `double` itd.). O typach referencyjnych będziemy bardzo dużo mówić zaczynając od rozdziału o klasach. Na razie zapamiętajmy, że nie powinniśmy porównywać tablic za pomocą operatora == bądź !=.

W przypadku typu `String`, wartości należy porównywać za pomocą metody `equals`, jednakże porównywanie dwóch tablic musimy wykonać w inny sposób. Aby sprawdzić, czy dwie tablice mają takie same wartości (i w takiej samej kolejności), musimy skorzystać z pętli i przejść przez wszystkie elementy tablic i porównać je ze sobą:

Nazwa pliku: PorownajTablice.java

```
public class PorownajTablice {
  public static void main(String[] args) {
    int[] pierwszaTablica = { 10, 5, 20 };
    int[] drugaTablica = { 10, 5, 20 };

    // jezeli tablice maja rozne rozmiary,
    // to na pewno nie bede takie same
    if (pierwszaTablica.length != drugaTablica.length) {
      System.out.println("Tablice nie sa takie same.");
    } else {
      boolean czyRoznicaZnaleziona = false;

      for (int i = 0; i < pierwszaTablica.length; i++) {
        // sprawdzamy, czy elementy o tych samych indeksach roznia
        // sie wartoscia - jezeli znajdziemy choc jedna roznice,
        // to bedziemy wiedziec, iz tablice nie sa takie same
        if (pierwszaTablica[i] != drugaTablica[i]) {
          czyRoznicaZnaleziona = true;
          break;
        }
      }
```

```
        if (czyRoznicaZnaleziona) {
          System.out.println("Tablice nie sa takie same.");
        } else {
          System.out.println("Tablice sa takie same.");
        }
      }
    }
  }
```

Jak widać, jest dużo więcej zachodu z porównywaniem dwóch tablicy, niż można się było spodziewać.

Najpierw sprawdzamy rozmiar obu tablic – jeżeli tablice mają różną liczbę elementów, to na pewno nie będą takie same. Jeżeli rozmiar się zgadza, w pętli przechodzimy przez wszystkie elementy tablic i porównujemy elementy o tych samych indeksach – jeżeli znajdziemy chociaż jedną parę elementów, które się różnią, będzie to oznaczało, że tablice nie są takie same – możemy w takim przypadku skorzystać z instrukcji `break`, by zakończyć pętlę. Na końcu programu, na podstawie wartości zmiennej `czyRoznicaZnaleziona`, wypisujemy informację, czy tablice są takie same, czy nie. Na ekranie w tym przypadku zobaczymy:

```
Tablice sa takie same.
```

W rozdziale o klasach dowiemy się dokładnie o różnicach pomiędzy typami referencyjnymi (złożonymi), a ośmioma typami podstawowymi (`int`, `boolean`, `char` itd.). Na razie zapamiętajmy, że aby porównać zawartość dwóch tablic, musimy skorzystać z pętli i porównać ze sobą elementy obu tablic na odpowiednich indeksach.

Dla dociekliwych

Zmienne *typu referencyjnego* wskazują na obiekty w pamięci – nie są konkretnie tymi obiektami, które zostały utworzone – **w przykładzie powyżej, zmienna `pierwszaTablica` to nie tablica trzech elementów, lecz wskazanie na obiekt w pamięci, który jest tablicą trzech elementów typu `int`**. Podobnie ze zmienną `drugaTablica`. Obie zmienne wskazują na dwa różne obiekty w pamięci – dlatego próba ich porównania za pomocą operatora == zwraca `false`, ponieważ obie tablice są różnymi obiektami w pamięci, tzn. nie są tym samym obiektem. W rozdziale o klasach dowiemy się dużo więcej o typach referencyjnych (złożonych), ich cechach, sposobie użycia, oraz tworzenia.

6.6.1 Zmiana rozmiaru tablicy

Wiemy już, że możemy utworzyć tablicę za pomocą słowa kluczowego `new` – albo podając, ile elementów tablica będzie mogła przechowywać, albo, bez podawania rozmiaru tablicy, podać elementy, z których tablica ma się składać:

```
int[] calkowite = new int[10];

double[] rzeczywiste;
rzeczywiste = new double[] { 3.14 , 2.44, 0.99 };
```

Wiemy już także, że tablice mają określony w trakcie ich tworzenia rozmiar, który możemy sprawdzić używając atrybutu `length`.

Czasem może się zdarzyć, że będziemy potrzebowali zwiększyć rozmiar tablicy, jednakże rozmiaru raz utworzonej tablicy w trakcie działanie programu zmienić się nie da.

Jest jednak możliwe rozwiązanie: możemy utworzyć nową tablicę o większym rozmiarze i przepisać do niej elementy z poprzedniej tablicy, a następnie przypisać nową tablicę do zmiennej, która poprzednio wskazywała na tablicę z mniejszą liczbą elementów:

Nazwa pliku: NowaTablicaZWiekszaLiczbaElementow.java

```
public class NowaTablicaZWiekszaLiczbaElementow{
  public static void main(String[] args) {
    int[] liczby = { 10, 100, -5 };

    System.out.println("Elementy tablicy liczby:");

    for (int x : liczby) {
      System.out.print(x + ", ");
    }

    System.out.println(); // przejdz do nowej linii

    // tworzymy druga tablice o wiekszym rozmiarze
    int[] tymczasowaTabela = new int[5];

    // przepisujemy elementy z pierwszej tablicy
    for (int i = 0; i < liczby.length; i++) {
      tymczasowaTabela[i] = liczby[i];
    }

    // ustawiamy dodatkowe elementy
    tymczasowaTabela[3] = 20;
    tymczasowaTabela[4] = 1;

    // przypisujemy druga tablice do pierwszej
    liczby = tymczasowaTabela;

    System.out.println("Elementy tablicy liczby:");

    for (int x : liczby) {
      System.out.print(x + ", ");
    }
  }
}
```

W tym programie utworzyliśmy tablicę o trzech elementach, dostępną za pomocą zmiennej o nazwie `liczby`, i wypisaliśmy jej zawartość na ekran.

Następnie, dla przykładu, w powyższym programie symulujemy potrzebę posiadania tablicy o większej liczbie elementów – tworzymy więc nową tablicę `tymczasowaTablica`, która może przechowywać `5` elementów. Następnie, w pętli przepisujemy wartości z mniejszej tablicy do większej.

Po wykonaniu pętli, ustawiamy dwa ostatnie elementy nowej tablicy przykładowymi wartościami. W końcu, w podświetlonej linii kodu, ustawiamy zmienną `liczby`, by wskazywała na nowoutworzoną tablicę. Na końcu programu ponownie wypisujemy elementy tej tablicy – tym razem widzimy pięć liczb:

```
Elementy tablicy liczby:
10, 100, -5,
Elementy tablicy liczby:
10, 100, -5, 20, 1,
```

W ten sposób, tablica `liczby` może teraz przechowywać pięć elementów, chociaż na początku programu mogła przechowywać tylko trzy elementy – **a przynajmniej tak by się mogło wydawać**. **Oryginalna tablica trój-elementowa się nie zmieniła** – to, co osiągnęliśmy w tym programie, to utworzenie **nowej** tablicy o większej liczbie elementów, oraz ustawiliśmy zmienną `liczby`, by na nią wskazywała. Wynika to z faktu, że tablice są przykładem typów referencyjnych (złożonych), o których dokładnie sobie opowiemy w rozdziale o klasach.

Dla dociekliwych

W poprzednim podrozdziale dowiedzieliśmy się, że porównywać tablice powinniśmy poprzez sprawdzenie ich rozmiarów i elementów, a nie operatora ==. Jednakże, gdybyśmy teraz porównali tablice `liczby` oraz `tymczasowaTabela` za pomocą operatora ==, to zwróciłby on wartość **`true`**! Powodem jest to, że *obie zmienne wskazują teraz na ten sam obiekt w pamięci*. W rozdziale o klasach wyjaśnimy sobie dokładnie to zachowanie.

6.7 Podsumowanie

- Tablice to obiekty, które mogą przechowywać określoną liczbę elementów danego typu.
- Tablicę definiuje się poprzez podanie typu, po którym następują nawiasy kwadratowe, a potem nazwa zmiennej. Dla przykładu, poniższa tablica to tablica elementów typu **int**, czyli może ona przechowywać liczby całkowite typu **int**:

```
int[] mojaTablica;
```

- Zanim będziemy mogli używać tablicy, musimy ją utworzyć. Aby to zrobić, możemy:
  - utworzyć tablicę, podając ile maksymalnie elementów może przechowywać, korzystając ze słowa kluczowego **new**,
  - zainicjalizować tablicę w momencie jej definiowania wartościami zawartymi w nawiasach klamrowych,
  - nie tworzyć tablicy od razu, lecz utworzyć ją później, korzystając ze słowa kluczowego **new** oraz podając w nawiasach klamrowych wartości, jakie na wstępie mają być zawarte w tablicy:

```
// tablica, ktora moze przechowywac maksymalnie 5 wartosci calkowitych
int[] calkowite = new int[5];

// tablica, ktora moze przechowywac maksymalnie 3 wartosci rzeczywiste
//  wstepnie zainicjalizowane wartosciami 3.14, 5, -20.5
double[] rzeczywiste = { 3.14, 5, -20.5 };

// tablica, ktora bedzie mogla przechowywac ciagi znakow
// na razie nie podalismy, ile wartosci typu String
// ta tablica bedzie mogla przechowywac
String[] slowa;

// tworzymy tablice, ktora bedzie mogla miec maksymalnie
// trzy elementy, i inicjalizujemy ja trzema elementami,
// kolejno: Ala, ma, kota
slowa = new String[] { "Ala", "ma", "kota" };
```

- Rozmiar raz utworzonej tablicy pozostaje niezmienny.
- Aby odnieść się do poszczególnych elementów tablicy, używamy operatora [ ] (nawiasy kwadratowe), któremu podajemy indeks elementu, do którego chcemy się odnieść.

```
calkowite[0] = 10;  // pierwszy element ma indeks 0 (a nie 1)!
calkowite[1] = 15;
calkowite[4] = 200; // piaty element ma indeks 4 (a nie 5)!
```

- Elementy tablic w języku Java zaczynają się od indeksu `0`, a nie `1`!
- Wartości tablic inicjalizowane są domyślną wartością danego typu. W przypadku typu **int** jest to `0`.

- Zmienną tablicową można "odpytać" o liczbę elementów, które przechowuje, korzystając z atrybutu tablicy o nazwie `length`:

```
double[] rzeczywiste = { 3.14, 5, -20.5 };

System.out.println(rzeczywiste.length); // wypisze 3
```

- Typ `String` można odpytać o liczbę znaków, z których się składa, za pomocą metody `length()` – należy zwrócić uwagę, że w przypadku typu `String`, korzystając z `length`, należy dodać na końcu nawiasy. W przypadku tablic, używamy atrybutu `length` – bez nawiasów.
- Jeżeli spróbujemy pobrać element spoza zakresu tablicy, to w trakcie działania programu zobaczymy błąd `ArrayIndexOutOfBoundsException`.
- Podczas iterowania po tablicy (przechodzenia przez jej elementy) należy zwrócić uwagę, by odpowiednio ustawić warunek zakończenie pętli. Pierwszy z poniższych przykładów jest poprawny, a drugi zakończy się błędem `ArrayIndexOutOfBoundsException` po uruchomieniu:

```
for (int i = 0; i < rzeczywiste.length; i++) {
  System.out.print(rzeczywiste[i] + ", ");
}

// blad! powinnismy uzyc operatora < zamiast <=
// poniewaz ostatni element w tablicy ma indeks rowny length - 1
// (bo numeracja elementow zaczyna sie od 0, a nie 1)
for (int i = 0; i <= rzeczywiste.length; i++) {
  System.out.print(rzeczywiste[i] + ", ");
}
```

- Tablice mogą mieć więcej niż jeden wymiar:

```
int[][] tablica2d = new int[3][5];
char[][] kwadrat = {
    { 'X', 'X', 'X', 'X' },
    { 'X', 'O', 'O', 'X' },
    { 'X', 'O', 'O', 'X' },
    { 'X', 'X', 'X', 'X' }
};

tablica2d[1][1] = 10;
tablica2d[1][2] = -2;
tablica2d[1][3] = 100;
tablica2d[1][4] = 42;
```

- Możemy także sprawdzić długość każdego wymiaru tablicy wielowymiarowej w następujący sposób:

```
System.out.println("Pierwszy wymiar: " + tablica2d.length);
System.out.println("Drugi wymiar: " + tablica2d[0].length);
```

```
Pierwszy wymiar: 3
Drugi wymiar: 5
```

- Możemy użyć zagnieżdżonych pętli, by przejść przez wszystkie elementy tablicy wielowymiarowej:

```
for (int rzad = 0; rzad < tablica2d.length; rzad++) {
  for (int kolumna = 0; kolumna < tablica2d[rzad].length; kolumna++) {
    System.out.print(tablica2d[rzad][kolumna]);
  }
  System.out.println();
}
```

- Pętla zwana **for**-each pozwala na łatwe iterowanie po tablicy bądź kolekcji:

```
for (typ nazwaZmiennejPetli : tablica_badz_kolekcja) {
  // instrukcje
}
```

Przykład:

```
int[] liczby = { 1, 5, 20 };

for (int x : liczby) {
  System.out.println(x);
}
```

- Pętla **for**-each iteruje po wszystkich elementach tablicy, od pierwszego do ostatniego elementu. Jest krótką i wygodną alternatywą dla np. pętli **for**.
- Aby sprawdzić, czy dwie tabele są takie same, nie powinniśmy korzystać z operatora ==, lecz:
  - sprawdzić, czy obie tablice mają taką samą liczbę elementów (korzystając z atrybutu `length` obu tablic),
  - porównać wszystkie elementy obu tablic o tych samych indeksach – jeżeli rozmiary się zgadzają i wszystkie elementy parami (pod tymi samymi indeksami) są takie same, to tablice są sobie równe.
- Rozmiaru tablic nie można co prawda zmienić, ale możemy utworzyć nową tablicę o większym rozmiarze, przepisać do niej elementy z pierwszej tablicy, a następnie przypisać nowoutworzoną tablicę do pierwszej zmiennej tablicy.

6.8 Pytania

1. Do czego służą tablice?
2. Jak definiuje się tablice?
3. Jak utworzyć tablicę?
4. Czy rozmiar tablicy można zmienić?
5. Jak odnieść się do danego elementu tablicy?
6. Jak odnieść się do pierwszego, a jak do ostatniego elementu tablicy?
7. Czy poniższy kod jest poprawny?

```
int[] tablica = { 1, 2, 3 };

System.out.println(tablica[3]);
```

8. Jak sprawdzić ile elementów znajduje się w tablicy?
9. Jaki będzie wynik działania poniższego programu, gdy wartość zmiennej `szukanaLiczba` będzie równa:
   a) 0
   b) 500

```
public static void main(String[] args) {
  boolean znaleziona = false;
  int[] tablica = { -20, 105, 0, 26, -99, 7, 1026 };

  int szukanaLiczba = ?; // pewna wartosc

  for (int i = 0; i <= tablica.length; i++) {
    if (tablica[i] == szukanaLiczba) {
      znaleziona = true;
      break; // znalezlismy liczbe - mozemy wiec przerwac petle
    }
  }

  if (znaleziona) {
    System.out.println("Liczba " + szukanaLiczba + " zostala znaleziona!");
  } else {
    System.out.println("Liczba " + szukanaLiczba + " nie zostala znaleziona.");
  }
}
```

10. Czy poniższy kod:
    a) Skompiluje się?
    b) Wykona się bez błędów?

```
int[][] tablica2d = new int[3][5];

tablica2d[3][1] = 1;
```

11. Czy poniższy kod jest poprawny?

```
String powitanie = { "Witaj", "Swiecie" };

for (int i = 0; i < powitanie.length(); i++) {
  System.out.println(powitanie[i] + " ");
}
```

12. Jaki będzie wynik działania poniższego fragmentu kodu?

```
double[] a = { 3.14, 2.44, 0.1 };
double[] b = { 3.14, 2.44, 0.1 };

if (a == b) {
  System.out.println("Tablice sa takie same.");
} else {
  System.out.println("Tablice nie sa takie same.");
}
```

13. Która z poniższych tablic jest zdefiniowana/utworzona niepoprawnie i dlaczego?

```
int liczby = { 1, 2, 3 };

String[] litery = { 'a', 'b', 'c' };

String[] slowa = new String[];
slowa = { "Ala", "ma", "kota" };

double[] rzeczywiste = new double[] { 3.14, 2.44, 20 };

double[] innaTablica = new int[3];

int[] tablica = new int[5] { 1, 10, 100 };

double[] kolejnaTablica = new double[3];
kolejnaTablica = { 5, 10, 15 };

String[] tab = { "Ala ma kota" };
```

14. Do czego służy pętla `for`-each i jak się z niej korzysta?

15. Co zostanie wypisane w ramach działania poniższego fragment kodu?

```
int[] liczby = { 0, 1, 2, -1 };

for (int i : liczby) {
  System.out.println(liczby[i] + ", ");
}
```

16. Co wypisze na ekran poniższy fragment kodu?

```
int[] liczby = new int[3];

System.out.println(liczby[1]);
```

17. Czy poniższy fragment kodu jest poprawny?

```
int[][] dwuwymiarowa =
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 };
```

6.9 Zadania

6.9.1 Co druga wartość tablicy

Napisz program, który wypisze co drugi element zdefiniowanych przez Ciebie tablic. Pierwsza tablica powinna mieć parzystą liczbę elementów, a druga nieparzystą.

6.9.2 Największa liczba w tablicy

Napisz program, który wypisze największą liczbę z tablicy. Tablicę zainicjalizuj przykładowymi wartościami.

6.9.3 Słowa z tablicy wielkimi literami

Napisz program, w którym zdefiniujesz tablicę wartości typu `String` i zainicjalizujesz ją przykładowymi wartościami. Skorzystaj z pętli `for`-each, aby wypisać wszystkie wartości tablicy ze wszystkimi literami zamienionymi na wielkie. Skorzystaj z funkcjonalności `toUpperCase` wartości typu `String`, którą poznaliśmy już w jednym z poprzednich rozdziałów.

6.9.4 Odwrotności słów w tablicy

Napisz program, który pobierze od użytkownika pięć słów i zapisze je w tablicy. Następnie, program powinien wypisać wszystkie słowa, od ostatniego do pierwszego, z literami zapisanymi od końca do początku. Dla przykładu, dla podanych słów `"Ala"`, `"ma"`, `"kota"`, `"i"`, `"psa"` program powinien wypisać: `"asp"`, `"i"`, `"atok"`, `"am"`, `"alA"`.

6.9.5 Sortowanie liczb

Napisz program, który pobierze od użytkownika osiem liczb, zapisze je w tablicy, a następnie posortuje tą tablicę rosnąco i wypisze wynik sortowania na ekran. Dla przykładu, dla liczb `10`, `-2`, `1`, `100`, `20`, `-15`, `0`, `10`, program wypisze `-15`, `-2`, `0`, `1`, `10`, `10`, `20`, `100`. Zastanów się, jak można posortować ciąg liczb i spróbuj zaimplementować swoje rozwiązanie. Przetestuj je na różnych zestawach danych. Możesz też skorzystać z jednego z popularnych algorytmów sortowania, np. *sortowania przez wstawianie*. Opis tego algorytmu znajdziesz w internecie.

6.9.6 Silnia liczb w tablicy

Napisz program, który pobierze od użytkownika pięć liczb, zapisze je w tablicy, a następnie policzy i wypisze silnię każdej z pobranych liczb.

6.9.7 Porównaj tablice stringów

Napisz program, w którym zdefiniujesz dwie tablice przechowujące wartości typu `String`. Zainicjalizuj obie tablice takimi samymi wartościami, w takiej samej kolejności. Napisz kod, który porówna obie tablice i odpowie na pytanie, czy są one takie same.

7 Rozdział VII – Metody

W tym rozdziale:

- dowiemy się, czym są metody i dlaczego są nam potrzebne,
- opowiemy sobie o: wywoływaniu metod, zwracaniu wartości, oraz argumentach metod,
- zobaczymy, czym jest przeładowywanie metod,
- dowiemy się, jak dokumentować metody,
- poznamy kilka z przydatnych metod typu `String`.

W tym rozdziale, wiele z podrozdziałów ma własne sekcje z podsumowaniem, pytaniami sprawdzającymi znajomość materiału, oraz zadania.

7.1 Czym są metody?

Metody w programowaniu to nazwane, wydzielone fragmenty kodu. Ich celem jest wykonanie określonego zadania, jak np.:

- policzenie pola koła,
- zapisanie w bazie danych informacji o złożeniu przez użytkownika nowego zamówienia,
- znalezienie w danym tekście pewnego ciągu znaków,
- posortowanie tablicy liczb,
- pobranie od użytkownika danych (w rozdziale trzecim zaczęliśmy używać dwóch metod do wczytywania danych wpisanych przez użytkownika w oknie konsoli):

```
public static int getInt() {
  return new Scanner(System.in).nextInt();
}

public static String getString() {
  return new Scanner(System.in).next();
}
```

- ...i wiele, wiele innych.

Ciało metody stanowi kod źródłowy zawarty między dwoma nawiasami klamrowymi { }. Nawiasy klamrowe są zawsze wymagane, nawet, gdy ciało metody stanowi tylko jedna instrukcja.

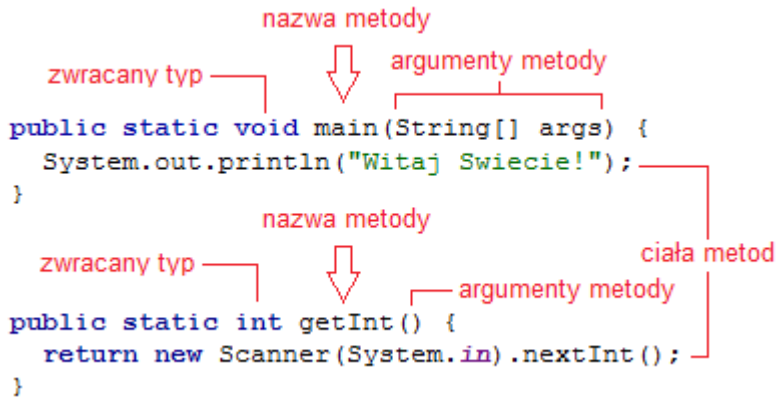
Metody:

- mają nazwę, zgodnę z nazewnictwem obiektów w Javie,
- mogą (ale nie muszą) przyjmować argumenty,
- mogą (ale nie muszą) zwracać wartość, za pomocą użycia słowa kluczowego **`return`**,
- mogą mieć modyfikatory – poznamy je w rozdziale o klasach – na razie nasze metody zawsze będziemy poprzedzać modyfikatorami **`public static`**.

W innych językach programowania możemy spotkać się z określeniami funkcja *oraz* procedura *w odniesieniu do metod. Procedury to zazwyczaj takie funkcje, które nie zwracają żadnego wyniku.*

Pierwszą metodą, której używaliśmy i którą sami napisaliśmy, była metoda `main`. Korzystaliśmy w niej z kilku innych metod, dla przykładu – `println("tekst")` z obiektu `System.out` oraz `length()` podczas używania zmiennych typu `String`.

Spójrzmy na opisane powyżej elementy metod na przykładzie metody `main` z naszego pierwszego programu oraz metody `getInt`, którą wprowadziliśmy w trzecim rozdziale:



Aby "uruchomić" metodę, tzn. kazać naszemu programowi ją wykonać, należy ją *wywołać*. Wywołanie metody odbywa się poprzez napisanie nazwy metody oraz dwóch nawiasów ( ), pomiędzy którymi powinny być zawarte argumenty metody (o ile jakieś przyjmuje). Nawiasy ( ) są wymagane nawet wtedy, gdy metoda nie przyjmuje żadnych argumentów.

Spójrzmy na przykład użycia metody, która wypisuje sumę dwóch przesłanych do niej liczb:

Nazwa pliku: WypiszSume.java

```
public class WypiszSume {
  public static void main(String[] args) {
    wypiszSume(100, 200);
    wypiszSume(-5, -20);
    wypiszSume(0, 0);
  }

  public static void wypiszSume(int a, int b) {
    System.out.println(a + b);
  }
}
```

Pod metodą `main` zdefiniowaliśmy nową metodę:

- jej nazwa to `wypiszSume`,
- nie zwraca ona żadnej wartości (użycie słowa kluczowego **`void`**),
- przyjmuje jako argumenty dwie liczby całkowite o typie **`int`** i nazwach `a` oraz `b`,
- jej ciało składa się z jednej instrukcji, która ma za zadanie wypisać na ekran sumę przesłanych do niej argumentów,
- ma dwa modyfikatory: **`public`** oraz **`static`** (poznamy je w rozdziale o klasach).

Następnie, w metodzie `main` wywołujemy naszą nową metodę trzykrotnie z różnymi argumentami. Odbywa się to poprzez napisanie nazwy metody oraz zawarcie w nawiasach ( ) argumentów, z jakimi ją wywołujemy. Argumenty te rozdzielone są przecinkami. Na końcu linii następuje średnik:

```
wypiszSume(100, 200);
wypiszSume(-5, -20);
wypiszSume(0, 0);
```

Rezultatem wykonania powyższego programu jest wypisanie na ekran wartości:

```
300
-25
0
```

7.1.1 Do czego potrzebne są metody?

Zauważmy w przykładzie z poprzedniego rozdziału, że definicję (ciało, nazwę, argumenty, zwracany typ) metody `wypiszSume` napisaliśmy tylko raz, a potem mogliśmy z niej wielokrotnie korzystać.

Wyobraźmy sobie teraz, że piszemy program, w którym musimy napisać wyszukiwanie w bazie danych zamówień klienta. Jest to złożona operacja i możemy:

- w każdym miejscu naszego programu napisać kod odpowiedzialny za szukanie zamówień klienta

lub

- napisać metodę, której zadaniem będzie szukanie i zwracanie zamówień klienta i korzystać z niej w różnych miejscach naszego programu.

Drugie podejście ma wiele zalet:

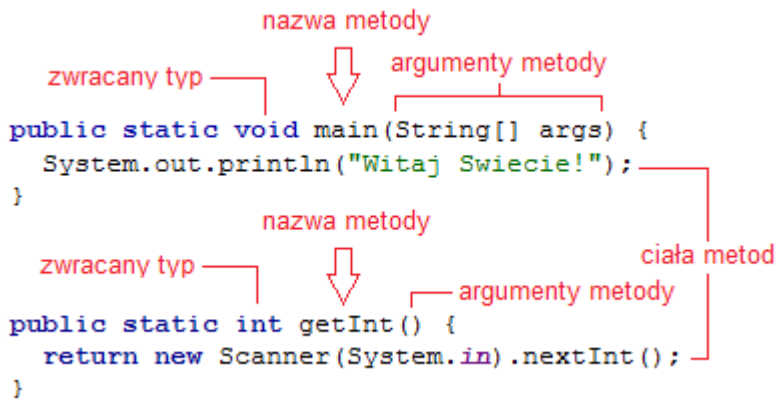
1. Nie duplikujemy kodu – logika szukania zamówień klienta będzie tylko w jednym miejscu.
2. Jeżeli będziemy musieli coś zmienić w sposobie szukania zamówień klienta, to będziemy to musieli zrobić tylko w jednej metodzie.
3. Opakowanie kodu w metodę, która ma nazwę, zwracany typ i argumenty, opisuje ten fragment kodu i zwiększa jego czytelność.
4. Nasz program będzie bardziej spójny ponieważ będzie korzystał z jednego sposobu na szukanie zamówień.
5. Łatwiej będzie przetestować nasz program, ponieważ testowanie szukania zamówień sprowadzi się do przetestowania jednej, dedykowanej do tego celu metody.

Dzięki pisaniu metod, możemy wielokrotnie korzystać z pewnej funkcjonalności. Metody dzielą także nasz program, opakowując logicznie powiązane ze sobą fragmenty kodu.

Duplikacja kodu jest złą praktyką. Gdy duplikujemy kod, musimy, w przypadku wymaganej zmiany, znaleźć i zmienić wszystkie jego wystąpienia. Do duplikowanego kodu także łatwo wkradają się błędy, dlatego warto pisać metody, które dostarczą wymaganą funkcjonalność, i korzystać z nich w innych miejscach programu, gdzie ta funkcjonalność jest potrzebna.

7.1.2 Podsumowanie podstaw metod

- Metody to nazwany fragmenty kodu, które wykonują pewne zadania, jak na przykład:
  - policzenie pola koła,
  - zapisanie w bazie danych informacji o zamówieniu,
  - posortowanie ciągu liczb.
- Dzięki metodom, możemy wielokrotnie korzystać z pewnej funkcjonalności (jak np. szukanie zamówień klienta), zamiast duplikować ten sam kod w wielu miejscach naszego programu.
- Ciało metody to zestaw instrukcji, które wykonywane są w ramach działania tej metody.
- Ciało metody zawarte jest pomiedzy nawiasami klamrowymi { }
- Metody muszą mieć nazwę zgodnę z nazewnictwem obiektów w Javie:
  - przykłady poprawnych nazw: `dodajLiczby`, `zapiszZamowienieKlienta`, `policzSilnie`, `zapiszdaneklienta` (*chociaż ta ostatnia nie spełnia reguły camelCase, tzn. drugi i trzeci wyraz w nazwie nie ma wielkiej pierwszej litery*),
  - przykład niepoprawnych nazw: `2doKwadratu` (zaczyna sie od liczby), `*mnozenieLiczb` (zaczyna się od znaku *).
- Metody mogą przyjmować argumenty i mogą zwracać wartość za pomocą słowa kluczowego **`return`**. Jeżeli metoda nic nie zwraca, to używamy słowa kluczowego **`void`**.
- Metody mogą mieć modyfikatory – na razie używamy modyfikatorów **`public static`** (*te i inne modyfikatory poznamy w kolejnym rozdziale*).
- Pierwszą metodą, jaką sami napisaliśmy, była metoda `main` w każdym z naszych programów. Inna używaną przez nas metodą była metoda `getInt`, która pobierała od użytkownika liczbę i zwracała ją.
- Opis składowych metod:



- Aby uruchomić metodę, należy ją wywołać poprzez napisanie nazwy metody oraz dwóch nawiasów ( ), pomiędzy którymi powinny być zawarte argumenty metody (o ile jakieś przyjmuje). Nawiasy ( ) są wymagane nawet wtedy, gdy metoda nie przyjmuje żadnych argumentów. Przykład użycia metody:

```
public class WypiszSume {
  public static void main(String[] args) {
    wypiszSume(100, 200);
    wypiszSume(-5, -20);
    wypiszSume(0, 0);
  }

  public static void wypiszSume(int a, int b) {
    System.out.println(a + b);
  }
}
```

7.1.3 Pytania do podstaw metod

1. Spójrz na poniższą metodę i odpowiedz na pytania:
   a) Co robi ta metoda?
   b) Jakie argumenty przyjmuje i co zwraca?
   c) Czy i w jaki sposób ta metoda mogłaby być lepiej napisana?

```
public static int wykonajDzialanie(int x, int y) {
  return x / y;
}
```

2. Co zrobić, aby użyć metody (wywołać ją)?
3. Które z poniższych są poprawnymi nazwami metod?
   a) _mojaMetoda
   b) zapiszUstawienia
   c) zapiszKosztW$
   d) 5NajlepszychOfert
   e) pobierz5OstatnichZamowien

7.1.4 Zadania do podstaw metod

7.1.4.1 Metoda wypisująca Witajcie!

Napisz metodę, która wypisuje na ekran tekst `Witajcie!` i użyj jej w metodzie `main`.

7.1.4.2 Metoda odejmująca dwie liczby

Napisz metodę, która wypisuje na ekran wynik odejmowania dwóch przesłanych do niej liczb i użyj jej w metodzie `main`.

7.2 Zakres i wywoływanie metod, zmienne lokalne

7.2.1 Zakres metod

W rozdziale 4.10. Bloki kodu i zakres zmiennych dowiedzieliśmy się, że zmienne tworzone w metodach muszą być zdefiniowane, zanim zostaną użyte – poniższy przykład nie skompiluje się, ponieważ kompilator w zaznaczonej linii nie wie jeszcze, czym jest "`liczba`":

Nazwa pliku: Rozdzial_03__Zmienne/UzyciePrzedDefinicja.java

```
public class UzyciePrzedDefinicja {
  public static void main(String[] args) {
    System.out.println("Liczba = " + liczba);

    int liczba = 5;
  }
}
```

Wynik próby kompilacji:

```
UzyciePrzedDefinicja.java:3: error: cannot find symbol
        System.out.println("Liczba = " + liczba);
                                          ^
  symbol:   variable liczba
  location: class UzyciePrzedDefinicja
1 error
```

Wróćmy do programu liczącego sumę dwóch liczb:

Nazwa pliku: WypiszSume.java

```
public class WypiszSume {
    public static void main(String[] args) {
        wypiszSume(100, 200);
        wypiszSume(-5, -20);
        wypiszSume(0, 0);
    }

    public static void wypiszSume(int a, int b) {
        System.out.println(a + b);
    }
}
```

Pytanie: dlaczego w tym przypadku kompilacja kończy się sukcesem, a program, po uruchomieniu, wypisuje na ekran sumę podanych liczb, pomimo, że metodę `wypiszSume` używamy zanim zostanie ona zapisana w kodzie tego programu?

Otóż, metody w klasach nie muszą być w zdefiniowane w kolejności użycia – kompilator analizuje cały plik źródłowy i wie, jakie metody utworzyliśmy. Stąd kompilator nie protestuje widząc linię:

```
wypiszSume(100, 200);
```

ponieważ, po analizie całego pliku, wie, że `wypiszSume` jest metodą z dwoma argumentami, która nie zwraca żadnej wartości.

7.2.2 Wywoływanie metod

Co w zasadzie dzieje się, gdy wywołujemy metodę?

Pytanie: jaki będzie wynik działania poniższego programu (co po kolei się zadzieje i jakie komunikaty zobaczymy na ekranie):

Nazwa pliku: KolejnoscInstrukcji.java

```
import java.util.Scanner;

public class KolejnoscInstrukcji {
  public static void main(String[] args) {
    System.out.println("Prosze podac liczbe:");

    int podanaLiczba = getInt();

    System.out.println("Teraz policzymy kwadrat tej liczby.");

    double liczbaDoKwadratu = policzKwadrat(podanaLiczba);

    System.out.println("Kwadrat tej liczby wynosi " + liczbaDoKwadratu);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  public static double policzKwadrat(int liczba) {
    return liczba * liczba;
  }
}
```

Wynikiem działania programu będzie:

```
Prosze podac liczbe:
10
Teraz policzymy kwadrat tej liczby.
Kwadrat tej liczby wynosi 100.0
```

Komunikaty zostały wyświetlone w tej kolejności, ponieważ gdy wywołujemy metodę, to skaczemy do niej z miejsca, gdzie do tej pory odbywało się wykonywanie programu. W komentarzach poniżej została zapisana kolejność wykonywania instrukcji w powyższym programie:

```java
import java.util.Scanner;

public class KolejnoscInstrukcji {
    public static void main(String[] args) {
        System.out.println("Prosze podac liczbe:");     // (1)

        int podanaLiczba = getInt();     // (2)

        System.out.println("Teraz policzymy kwadrat tej liczby."); // (4)

        double liczbaDoKwadratu = policzKwadrat(podanaLiczba); // (5)

        System.out.println("Kwadrat tej liczby wynosi " + liczbaDoKwadratu); // (7)
    }

    public static int getInt() {
        return new Scanner(System.in).nextInt(); // (3)
    }

    public static double policzKwadrat(int liczba) {
        return liczba * liczba; // (6)
    }
```

Strzałki obrazują przepływ działania programu:

- najpierw wypisujemy komunikat z prośbą o podanie liczby ***(1)***,
- następnie, gdy pobieramy od użytkownika liczbę ***(2)***, skaczemy z metody `main` do metody `getInt` ***(3)***,
- wracamy do metody `main` i wypisujemy tekst instrukcją ***(4)***,
- ponownie skaczemy z metody `main` ***(5)*** – tym razem do metody `policzKwadrat` ***(6)***,
- po powrocie, wypisujemy wynik obliczenia ***(7)***.

Wywołanie metody powoduje przeskok w kodzie do niej z metody, która ją wywołała.

7.2.3 Zmienne lokalne

W poprzednich rozdziałach korzystaliśmy ze zmiennych w metodzie `main`, na przykład w programie z trzeciego rozdziału, który liczył pole i obwód koła użyliśmy czterech zmiennych (`promienKola`, `pi`, `poleKola`, oraz `obwodKola`):

Nazwa pliku: Rozdzial_03__Zmienne/ObwodPoleKola.java

```
public class ObwodPoleKola {
  public static void main(String[] args) {
    int promienKola = 8;
    double pi = 3.14;

    double poleKola = pi * promienKola * promienKola;
    double obwodKola = 2 * pi * promienKola;

    System.out.println("Pole kola wynosi: " + poleKola);
    System.out.println("Obwod kola wynosi: " + obwodKola);
  }
}
```

Napiszmy metodę, która będzie liczyła dla nas pole koła na podstawie jego promienia:

Nazwa pliku: PoleKola.java

```
public class PoleKola {
  public static void main(String[] args) {
    System.out.println("Pole kola o promieniu 2 wynosi " + obliczPoleKola(2));
  }

  public static double obliczPoleKola(int promienKola) {
    double promienDoKwadratu = promienKola * promienKola;

    return 3.14 * promienDoKwadratu;
  }
}
```

Metoda `obliczPoleKola`:

- przyjmuje jeden argument – promień koła, którego pole chcemy policzyć,
- wylicza kwadrat promienia koła i zapisuje go w zmiennej `promienDoKwadratu`,
- zwraca wyliczone pole koła przez przemnożenie promienia do kwadratu przez wartość liczby PI.

Pytanie: czy w metodzie `main` możemy odnieść się do zmiennej `promienDoKwadratu`, którą zdefiniowaliśmy w metodzie `obliczPoleKola`?

Spróbujmy:

Nazwa pliku: PoleKolaZBledem.java

```
public class PoleKolaZBledem {
  public static void main(String[] args) {
    System.out.println("Pole kola o promieniu 2 wynosi " + obliczPoleKola(2));

    System.out.println("Promien kola do kwadratu wynosi: " + promienDoKwadratu);
  }

  public static double obliczPoleKola(int promienKola) {
    double promienDoKwadratu = promienKola * promienKola;

    return 3.14 * promienDoKwadratu;
  }
}
```

Próba kompilacji powyższego programu zakończy się następującym błędem:

```
PoleKolaZBledem.java:5: error: cannot find symbol
    System.out.println("Promien kola do kwadratu wynosi: " + promienDoKwadratu);
                                                            ^
  symbol:   variable promienDoKwadratu
  location: class PoleKolaZBledem
1 error
```

Kompilator poinformował nas, że nie wie, czym jest "`promienDoKwadratu`". Wynika to z faktu, że zmienna `promienDoKwadratu` jest lokalna dla metody `obliczPoleKola` i jest niedostępna nigdzie indziej w programie poza ciałem tej metody.

Jest to bardzo ważna informacja:

> Zakresem życia zmiennych utworzonych w metodzie jest ciało metody, w której je zdefiniowaliśmy.

Zmienna `promienDoKwadratu` dostępna jest jedynie w metodzie `obliczPoleKola`, od momentu jej (zmiennej) zdefiniowania, do końca metody `obliczPoleKola` – po zakończeniu działania tej metody, zmienna `promienDoKwadratu` przestaje istnieć:

Nazwa pliku: PoleKolaZBledem.java

```
public class PoleKolaZBledem {
  public static void main(String[] args) {
    System.out.println("Pole kola o promieniu 2 wynosi " + obliczPoleKola(2));

    // w metodzie main zmienna promienDoKwadratu nie istnieje,
    //  wiec kompilator zaprotestuje!
    System.out.println("Promien kola do kwadratu wynosi: " + promienDoKwadratu);
  }

  public static double obliczPoleKola(int promienKola) {
    // zmienna promienDoKwadratu dostepna jest od ponizszej linii
    double promienDoKwadratu = promienKola * promienKola;

    return 3.14 * promienDoKwadratu;
  } // metoda sie konczy, zmienna promienDoKwadratu przestaje istniec
}
```

Kolejne pytanie: czy możemy użyć w `main` argumentu `promienKola`? Nie – efekt będzie taki sam, jak przy próbie użycia zmiennej `promienDoKwadratu`. Obie te zmienne to zmienne lokalne.

> *Wkrótce poznamy rodzaj zmiennych inny niż lokalne.*

7.2.4 Czas życia zmiennych lokalnych

W poprzednim rozdziale dowiedzieliśmy się, że zmienne lokalne są dostępne jedynie w metodach, w których zostały utworzone.

Za każdym razem, gdy wywołujemy metodę, zmienne lokalne tworzone są od nowa.

Jaki, w takim razie, będzie wynik działania poniższego programu? Czy zobaczymy na ekranie liczby `10` i `20`, czy `10` i `30`?

Nazwa pliku: CzasZyciaZmiennychLokalnych.java

```
public class CzasZyciaZmiennychLokalnych {
  public static void main(String[] args) {
    testowaMetoda(10);
    testowaMetoda(20);
  }

  public static void testowaMetoda(int liczba) {
    int zmiennaLokalna = 0;

    zmiennaLokalna = zmiennaLokalna + liczba;

    System.out.println(zmiennaLokalna);
  }
}
```

Na ekranie zobaczymy liczby `10` i `20` – co prawda dodajemy do lokalnej zmiennej o nazwie `zmiennaLokalna` wartość przesłanego do metody argumentu, ale po zakończeniu metody, zmienna `zmiennaLokalna` przestaje istnieć i jej wartość przestaje być dostępna – po ponownym uruchomieniu metody `testowaMetoda`, zmienna `zmiennaLokalna` tworzona jest po raz kolejny i "nie pamięta", że przy okazji poprzedniego wywołania przypisaliśmy jej wartość `10`.

7.2.5 Podsumowanie zakresu i wywoływania metod, zmiennych lokalnych

- Metody w klasach nie muszą być zdefiniowane w kolejności użycia – poniższy kod działa poprawnie – metoda `wypiszSume` nie musi być przed metodą `main`:

```
public class WypiszSume {
    public static void main(String[] args) {
        wypiszSume(100, 200);
        wypiszSume(-5, -20);
        wypiszSume(0, 0);
    }

    public static void wypiszSume(int a, int b) {
        System.out.println(a + b);
    }
}
```

- Gdy wywołujemy metodę, to skaczemy do niej z miejsca, gdzie do tej pory odbywało się wykonywanie programu. Poniższy przykład obrazuje kolejność wykonywania kodu programu, w którym używanych jest kilka metod:

```
import java.util.Scanner;

public class KolejnoscInstrukcji {
    public static void main(String[] args) {
        System.out.println("Prosze podac liczbe:");     // (1)

        int podanaLiczba = getInt();     // (2)

        System.out.println("Teraz policzymy kwadrat tej liczby."); // (4)

        double liczbaDoKwadratu = policzKwadrat(podanaLiczba); // (5)

        System.out.println("Kwadrat tej liczby wynosi " + liczbaDoKwadratu); // (7)
    }

    public static int getInt() {
        return new Scanner(System.in).nextInt(); // (3)
    }

    public static double policzKwadrat(int liczba) {
        return liczba * liczba; // (6)
    }
```

- Zmienne lokalne to zmienne utworzone w ciele metody.
- Zmienne lokalne są lokalne dla metody, w której zostały zdefiniowane i są niedostępne nigdzie indziej w programie poza ciałem tej metody.

- Poniższy przykład obrazuje lokalność zmiennych – kompilacja programu zakończy się błędem, ponieważ w metodzie `main` próbujemy odnieść się do zmiennej `promienDoKwadratu`, która jest lokalna (istnieje tylko) w metodzie `obliczPoleKola`:

```
public class PoleKolaZBledem {
  public static void main(String[] args) {
    System.out.println("Pole kola o promieniu 2 wynosi " + obliczPoleKola(2));

    System.out.println("Promien kola do kwadratu wynosi: " + promienDoKwadratu);
  }

  public static double obliczPoleKola(int promienKola) {
    double promienDoKwadratu = promienKola * promienKola;

    return 3.14 * promienDoKwadratu;
  }
}
```

- Zmienne lokalne "żyją" w ramach metody, w której zostały utworzone. Za każdym razem, gdy wywołujemy metodę, zmienne lokalne tworzone są od nowa.

7.2.6 Pytania do zakresu i wywoływania metod, zmiennych lokalnych

1. Czym są zmienne lokalne?
2. Czy możemy z metody `abc` odwołać się do zmiennej lokalnej zdefiniowanej w metodzie `xyz`?
3. Jaki jest zakres życia zmiennych lokalnych?
4. Czy poniższy kod jest poprawny i jak go ewentualnie naprawić?

```
public class ZakresIWywolywanieMetodPytanie1 {
  public static void main(String[] args) {
    System.out.println("Wynik: " + kwadrat);
    int kwadrat = policzKwadrat(10);
  }

  public static int policzKwadrat(int liczba) {
    return liczba * liczba;
  }
}
```

5. Jaki będzie wynik działania poniższego programu?

```
public class ZakresIWywolywanieMetodPytanie2 {
  public static void main(String[] args) {
    policzKwadrat(10);
    System.out.println("Wynik: " + wynik);
  }

  public static void policzKwadrat(int liczba) {
    int wynik = liczba * liczba;
  }
}
```

6. Jaki będzie wynik działania poniższego programu – co po kolei zobaczymy na ekranie?

```
public class ZakresIWywolywanieMetodPytanie3 {
   public static void main(String[] args) {
    System.out.println("Witajcie! Kwadrat 10 to: " + policzKwadrat(10));
    System.out.println("Policzone!");
   }

   public static int policzKwadrat(int liczba) {
    System.out.println("Liczymy kwadrat liczby " + liczba);
    return liczba * liczba;
   }
}
```

7.3 Wartości zwracane przez metody

Metody mogą zwracać dowolną wartość, zarówno typu prymitywnego (jak `int` czy `boolean`), oraz złożonego (np. `String` lub tablica liczb rzeczywistych `double[]`). Metody mogą także nie zwracać żadnej wartości – wtedy, zamiast podawać zwracany typ, używamy słowa kluczowego `void`. Przykładem takiej metody jest każda napisana przez nas metoda `main`:

```
public static void main(String[] args) {
  System.out.println("Witaj Swiecie!");
}
```

Uwaga: zwracany typ (lub słowo kluczowe `void`) musi poprzedzać nazwę metody – poniższa definicja metody `main` jest niepoprawna i jej próba kompilacji zakończy się błędem:

```
// blad! void musi byc zaraz przed main!
void public static main(String[] args) { // blad kompilacji
  System.out.println("Witaj Swiecie!");
}
```

7.3.1 Słowo kluczowe return

Metody zwracają wartość poprzez użycie słowa kluczowego `return`. Użycie słowa kluczowego `return` natychmiast przerywa metodę i zwraca wartość. Metoda może zawierać więcej niż jedno użycie `return`, ponieważ możemy uzależnić zwracaną wartość od pewnego warunku:

Nazwa pliku: ZwrocWiekszaLiczbe.java

```
public class ZwrocWiekszaLiczbe {
  public static void main(String[] args) {
    int x = ktoraWieksza(100, 200);
    int y = ktoraWieksza(-5, -20);

    System.out.println("x wynosi " + x); // wypisze 200
    System.out.println("y wynosi " + y); // wypisze -5
  }

  public static int ktoraWieksza(int a, int b) {
    if (a > b) {
      return a;
    } else {
      return b;
    }
  }
}
```

Zdefiniowana powyżej metoda `ktoraWieksza` zwraca większą z dwóch liczb. Słowo kluczowe `return` użyte zostało dwa razy – w zależności od tego, która z liczb jest większa, metoda zwróci albo wartość zapisaną w argumencie `a` albo `b`.

Metody, które nie zwracają wartości (tzn. definiują zwracany typ jako `void`), także mogą zawierać użycie słowa kluczowego `return`, ale musi po nim od razu następować średnik – może być to przydatne, gdy chcemy zakończyć działanie metody przed wykonaniem wszystkich operacji (np. na podstawie jakiegoś warunku):

Nazwa pliku: WypiszWynikDzielenia.java

```
public class WypiszWynikDzielenia {
  public static void main(String[] args) {
    wypiszWynikDzielenia(10, 0);
    wypiszWynikDzielenia(25, 5);
  }

  public static void wypiszWynikDzielenia(int x, int y) {
    if (y == 0) {
      System.out.println("Nie mozna dzielic przez 0!");
      return;
    }

    System.out.println("Wynik dzielenia: " + (x / y));
  }
}
```

Powyższa metoda `wypiszWynikDzielenia` ma za zadanie wypisać wynik dzielenia dwóch podanych liczb i nic nie zwraca (**`void`**). Na początku metody sprawdzamy, czy dzielnik nie jest równy `0` – jeżeli tak, to wypisujemy na ekran informację, że przez zero dzielić nie można i kończymy działanie metody `wypiszWynikDzielenia` z pomocą użycia **`return`** – w takim przypadku nie dojdzie do dzielenia. Wynik działania tego programu:

```
Nie mozna dzielic przez 0!
Wynik dzielenia: 5
```

Pytanie: co stanie się, jeżeli zdefiniujemy, że metoda ma zwracać wartość, ale nie użyjemy **`return`** w tej metodzie?

Spójrzmy na poniższy przykład, w którym metoda `kwadratLiczby` powinna zwrócić wartość typu **`int`**, jednak nie zostało w niej użyte słowo kluczowe **`return`**:

Nazwa pliku: BrakReturn.java

```
public class BrakReturn {
  public static void main(String[] args) {
    int liczbaDoKwadratu = kwadratLiczby(5);
  }

  public static int kwadratLiczby(int x) {
    int wynik = x * x;
    // ups! zapomnielismy zwrocic wynik!
  }
}
```

Próba kompilacji tego programu kończy się następującym błędem:

```
BrakReturn.java:9: error: missing return statement
    }
    ^
1 error
```

Jeżeli zdefiniowaliśmy zwracany typ, to nasza metoda musi zwracać jakąś wartość – inaczej kod się nie skompiluje[1].

Podobnie, jeżeli metoda ma nic nie zwracać (**`void`**), a użyjemy w niej **`return`**

1 Chyba, że metoda rzuca wyjątek – zapoznamy się z takim przypadkiem w rozdziale o wyjątkach.

i podamy jakąś wartość, to próba kompilacji takiego programu także zakończy się błędem:

Nazwa pliku: VoidMetodaZwracaWartosc.java

```
public class VoidMetodaZwracaWartosc {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");

    return 5; // blad kompilacji
  }
}
```

Błąd kompilacji – kompilator informuje nas, że w tej metodzie nie spodziewał się zwracania jakiejkolwiek wartości:

```
VoidMetodaZwracaWartosc.java:5: error: incompatible types: unexpected
return value
        return 5; // blad kompilacji
               ^
1 error
```

7.3.2 Używanie wartości zwracanych przez metody

Wiemy już jak wywoływać metody oraz jak zwracać z nich wartości, ale gdzie w zasadzie możemy użyć wywołania metody i zwracanej przez nią wartości?

Wartość zwracana z metody może być:

- przypisana do zmiennej,
- przesłana jako parametr do innej metody,
- użyta w instrukcji `if` lub pętli itp.,
- w ogóle nie użyta.

Spójrzmy na kilka przykładów.

7.3.2.1 Przypisanie wyniku metody do zmiennej

Jak już wielokrotnie widzieliśmy w tym rozdziale, wynik działania metody możemy przypisać do zmiennej. Typ zmiennej musi być tego samego typu[2], co definiowany przez metodę zwracany typ:

Nazwa pliku: RezultatMetodyPrzypisanyDoZmiennej.java

```
public class RezultatMetodyPrzypisanyDoZmiennej {
  public static void main(String[] args) {
    int kwadrat = podniesDoKwadratu(16);

    /* zakomentowana linijka, poniewaz powoduje ona blad kompilacji
       nie mozna przypisac liczby do zmiennej typu String:
        error: incompatible types: int cannot be converted to String
    */
    // String tekst = podniesDoKwadratu(16);
  }

  public static int podniesDoKwadratu(int liczba) {
    return liczba * liczba;
 }
}
```

W powyższym przykładzie przypisujemy wynik wywołania metody `podniesDoKwadratu` do zmiennej typu `int` o nazwie `kwadrat`. Linia z przypisaniem wyniku tej metody do zmiennej typu `String` jest zakomentowana, ponieważ spowodowałaby ona błąd kompilacji – metoda `podniesDoKwadratu` definiuje, że będzie zwracać wartość liczbową typu `int`, a `String` przechowuje tekst – kompilator zaprotestuje.

2 Lub typu pochodnego, o czym dowiemy się w rozdziale o klasach. Przypisanie wyniku metody zwracającej liczbę typu `int` do zmiennej typu `double` także zadziała, ponieważ typ `double` ma "szerszy" zakres wartości niż typ `int` i może on przechowywać wszystkie wartości typu `int`.

7.3.2.2 Rezultat metody jako argument innej metody

Wynik metody możemy nie tylko przypisać do zmiennej, ale podać także jako argument innej metody:

Nazwa pliku: RezultatMetodyJakoArgumentInnejMetody.java

```
import java.util.Scanner;

public class RezultatMetodyJakoArgumentInnejMetody {
  public static void main(String[] args) {
    System.out.println("Podaj liczbe, a ja wypisze jej kwadrat:");
    System.out.println(podniesDoKwadratu(getInt()));
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  public static int podniesDoKwadratu(int liczba) {
    return liczba * liczba;
  }
}
```

W powyższym przykładzie argumentem metody `println` jest wynik działania metody `podniesDoKwadratu`. Jednakże, aby ta metoda zwróciła wartość, najpierw musi zostać wywołana metoda `getInt`, która pobierze od użytkownika liczbę, która zostanie zwrócona i stanie się argumentem metody `podniesDoKwadratu`. Gdy metoda `podniesDoKwadratu` wykona się, zwróci liczbę podniesioną do kwadratu, która stanie się z kolei argumentem metody `println`, służącej do wypisywania tekstu na ekran.

Kolejność wykonania metod będzie więc następująca:

1. Najpierw wykona się metoda `getInt`.
2. Następnie, metoda `podniesDoKwadratu`.
3. Na końcu wykona się `println`, które wypisze wynik podnoszenia liczby do kwadratu na ekran.

7.3.2.3 Metoda użyta w instrukcji warunkowej

Wyniki metod są często używane jako warunki instrukcji warunkowych:

Nazwa pliku: RezultatMetodyWInstrukcjiWarunkowej.java

```
import java.util.Scanner;

public class RezultatMetodyWInstrukcjiWarunkowej {
  public static void main(String[] args) {
    System.out.println("Podaj liczbe:");
    int podanaLiczba = getInt(); // (1)

    // wywolujemy metode, a nastepnie sprawdzamy wynik
    //  przy pomocy instrukcji warunkowej if
    if (czyParzysta(podanaLiczba)) { // (2)
      System.out.println("Ta liczba jest parzysta.");
    } else {
      System.out.println("Ta liczba nie jest parzysta.");
    }

    System.out.println("Podaj kolejna liczbe:"); // (3)

    if (getInt() >= 0) { // (4)
      System.out.println("Podales nieujemna liczbe.");
    } else {
      System.out.println("Podales liczbe ujemna.");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  public static boolean czyParzysta(int liczba) { // (5)
    return liczba % 2 == 0; // (6)
  }
}
```

W powyższym przykładzie dzieje się kilka rzeczy:

- Pobieramy `(1)` od użytkownika liczbę.
- Jak wiemy, instrukcja warunkowa `(2)` oczekuje wartość `true` bądź `false` – tak się składa, że nasza metoda `czyParzysta` `(5)` zwraca wartość typu `boolean`, a typ ten może mieć jedną z dwóch wartości – właśnie `true` bądź `false`. Metoda `czyParzysta` używa operatora % (reszta z dzielenia) do sprawdzenia, czy reszta z dzielenia przesłanej w argumencie liczby wynosi `0` i zwraca wynik tego porównania `(6)`.
- Wypisujemy komunikat `(3)`, aby użytkownik podał kolejną liczbę.
- Tym razem `(4)`, nie przypisujemy wyniku metody `getInt` do żadnej zmiennej, lecz porównujemy wynik działania tej metody (czyli liczbę, którą pobraliśmy od użytkownika) od razu do liczby `0` i wypisujemy komunikat, czy jest ona nieujemna.

Warto tutaj jeszcze dodać, że *wynik poniższego porównania*:

```
liczba % 2 == 0 // (6)
```

wynosi `true` bądź `false` i tą wartość możemy zwrócić z metody `czyParzysta`. Powyższy kod

moglibyśmy zapisać w mniej zwięzłej formie jako:

```
public static boolean czyParzysta(int liczba) {
  boolean czyLiczbaJestParzysta;

  if (liczba % 2 == 0) {
    czyLiczbaJestParzysta = true;
  } else {
    czyLiczbaJestParzysta = false;
  }

  return czyLiczbaJestParzysta;
}
```

czy też:

```
public static boolean czyParzysta(int liczba) {
  if (liczba % 2 == 0) {
    return true;
  } else {
    return false;
  }
}
```

7.3.2.4 Nieużywanie wyniku metody

Wartość zwracana z metody nie musi zostać nigdzie użyta – nie ma takiego wymogu. Czasem po prostu chcemy zignorować zwracaną wartość, bo nie jest nam ona do niczego potrzebna:

Nazwa pliku: RezultatMetodyNieUzyty.java

```
import java.util.Scanner;

public class RezultatMetodyNieUzyty {
  public static void main(String[] args) {
    int liczba = getInt();

    // pobieramy od uzytkownika druga liczbe, ale nigdzie jej nie uzywamy
    getInt();
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

7.3.3 Nieosiągalne ścieżki wykonania i ścieżki bez return

Jak wiemy z jednego z poprzednich rozdziałów, gdy metoda definiuje, że będzie zwracać wartość, a nie użyjemy słowa kluczowego **return**, by zwrócić z niej jakąś wartość, nasz program się nie skompiluje:

Nazwa pliku: BrakReturn.java

```
public class BrakReturn {
  public static void main(String[] args) {
    int liczbaDoKwadratu = kwadratLiczby(5);
  }

  public static int kwadratLiczby(int x) {
    int wynik = x * x;
    // ups! zapomnielismy zwrocic wynik!
  }
}
```

```
BrakReturn.java:9: error: missing return statement
  }
  ^
1 error
```

Spójrzmy na bardziej skomplikowany przykład, w którym uzależniamy zwracaną wartość od pewnego warunku – jaki będzie wynik działania tego programu?

Nazwa pliku: ReturnWIfie.java

```
public class ReturnWIfie {
  public static void main(String[] args) {
    double wynikDzielenia = podzielLiczby(100, 25);
  }

  public static double podzielLiczby(int x, int y) {
    if (y != 0) {
      return x / y;
    }
  }
}
```

Program w ogóle się nie skompiluje! Powodem jest to, że istnieje taka ścieżka wykonania metody `podzielLiczby`, w której metoda ta nie zwróci żadnej wartości – jeżeli wartość argumentu `y` będzie wynosić `0`, to metoda nie zwróci wartości.

Kompilator, analizując nasz kod źródłowy, jest w stanie wychwycić takie przypadki i zasygnalizować błąd jeszcze na etapie kompilacji:

```
ReturnWIfie.java:10: error: missing return statement
  }
  ^
1 error
```

Kompilator wskazuje błąd w linijce, która zawiera klamrę zamykającą metodę `podzielLiczby`. dając znam do zrozumienia, że w tym miejscu spodziewał się zwrotu wartości z metody za pomocą **return**.

Wartość z metody musi zostać zwrócona w każdej możliwej jej ścieżce wykonania[3] – inaczej program się nie skompiluje.

7.3.3.1 Nieosiągalny kod

Kompilator jest także w stanie wykryć instrukcje w metodzie, które nie mają szansy zostać wykonane, tak jak w poniższym przykładzie:

Nazwa pliku: InstrukcjaPoReturn.java

```
public class InstrukcjaPoReturn {
  public static void main(String[] args) {
    int kwadrat = liczbaDoKwadratu(10);
  }

  public static int liczbaDoKwadratu(int x) {
    return x * x;

    // linia po return nie ma szansy sie wykonac
    System.out.println("Instrukcja po return!");
  }
}
```

Kompilacja tego programu zakończy się poniższym błędem, ponieważ kompilator wykrył, że instrukcja `System.out.println("Instrukcja po return!");` nie ma szansy się wykonać – instrukcja `return`, znajdująca się nad nią, spowoduje zakończenie metody i zwrócenie wartości.

```
InstrukcjaPoReturn.java:10: error: unreachable statement
    System.out.println("Instrukcja po return!");
    ^
```

3 O ile dana ścieżka wykonania nie kończy się rzuceniem wyjątku – w takim przypadku zwrócenie wartości nie jest wymagane, ale o takim przypadku opowiemy sobie więcej, gdy będziemy uczyli się o wyjątkach.

7.3.4 Void, czyli niezwracanie wartości

Na koniec rozdziału o zwracanych przez metody wartościach spójrzmy na przypadek szczególny, w którym... metoda nic nie zwraca. Jak już wiemy, w takim przypadku, zamiast podawać zwracany typ przed nazwą metody, używamy słowa kluczowego `void`.

Pytanie: co się stanie, jeżeli spróbujemy przypisać wynik metody, która nic nie zwraca, do, na przykład, zmiennej?

Nazwa pliku: BrakZwracanejWartosciPrzypisanie.java

```
public class BrakZwracanejWartosciPrzypisanie {
  public static void main(String[] args) {
    int x = wypiszKomunikat("Wczoraj padal deszcz.");
  }

  public static void wypiszKomunikat(String komunikat) {
    System.out.println("Uwaga - komunikat!");
    System.out.println(komunikat);
  }
}
```

Powyższy program w ogóle się nie skompiluje. Skoro nasza metoda `wypiszKomunikat` nic nie zwraca, to nie powinniśmy próbować przypisać zwracanego przez nią wyniku (ponieważ takowego nie będzie) do zmiennej. Kompilator wyświetli komunikat o błędzie, w którym poinformuje nas, że nie może przypisać "niczego" do zmiennej typu `int`:

```
BrakZwracanejWartosciPrzypisanie.java:3: error:
     incompatible types: void cannot be converted to int
        int x = wypiszKomunikat("Wczoraj padal deszcz.");
                                ^
1 error
```

7.3.5 Podsumowanie do zwracania wartości

- Metody mogą zwracać wartość typu prymitywnego (np. `int`) oraz złożonego (np. `String`).
- Jeżeli metoda ma nic nie zwracać, zamiast podawać zwracany typ używamy słowa kluczowego `void`. Poniższa metoda `main` nie zwraca żadnej wartości:

```
public static void main(String[] args) {
  System.out.println("Witaj Swiecie!");
}
```

- Zwracany typ musi zawsze poprzedzać nazwę metody – poniższy kod jest niepoprawny:

```
// blad! void musi byc zaraz przed main
void public static main(String[] args) {
  System.out.println("Witaj Swiecie!");
}
```

- Aby zwrócić wartość z metody, używamy słowa kluczowego `return`, po którym następuje wartość, którą chcemy zwrócić.
- Użycie słowa kluczowego `return` natychmiast przerywa metodę i zwraca wartość.
- Metoda może zawierać więcej niż jedno użycie `return`, ponieważ możemy uzależnić zwracaną wartość od pewnego warunku:

```
public static int ktoraWieksza(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

- Metody, które nic nie zwracają (`void`) także mogą używać słowa kluczowego `return`, aby przerwać działanie metody:

```
public static void wypiszWynikDzielenia(int x, int y) {
  if (y == 0) {
    System.out.println("Nie mozna dzielic przez 0!");
    return;
  }

  System.out.println("Wynik dzielenia: " + (x / y));
}
```

- Jeżeli metoda definiuje, że będzie zwracała wartość, a nic z niej nie zwrócimy, to nasz program się nie skompiluje:

```
public static int kwadratLiczby(int x) {
  int wynik = x * x;
  // ups! zapomnielismy zwrocic wynik! blad kompilacji
}
```

- Wartość zwracana przez metodę może być:
  a) Przypisana do zmiennej:

```
int kwadrat = podniesDoKwadratu(16);
```

  b) Przesłana jako parametr do innej metody:

```
System.out.println(podniesDoKwadratu(getInt()));
```

  c) Użyta w instrukcji `if` lub pętli itp.:

```
if (czyParzysta(podanaLiczba)) {
  System.out.println("Ta liczba jest parzysta.");
} else {
  System.out.println("Ta liczba nie jest parzysta.");
}
```

  d) W ogóle nie użyta:

```
public class RezultatMetodyNieUzyty {
  public static void main(String[] args) {
    int liczba = getInt();

    // pobieramy od uzytkownika druga liczbe, ale nigdzie jej nie uzywamy
    getInt();
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

- Jeżeli metoda definiuje, że nie będzie nic zwracać (`void`), to nie możemy jej przypisać do zmiennej, ani użyć w instrukcji `if`, pętli itp. – **poniższy kod się nie skompiluje**:

```
public class BrakZwracanejWartosciPrzypisanie {
  public static void main(String[] args) {
    // blad - probujemy przypisac wynik metody do zmiennej,
    // a metoda ta nie zwraca zadnej wartosci (void)
    int x = wypiszKomunikat("Wczoraj padal deszcz.");
  }

  public static void wypiszKomunikat(String komunikat) {
    System.out.println("Uwaga - komunikat!");
    System.out.println(komunikat);
  }
}
```

- Jeżeli metoda ma wiele ścieżek wykonania (np. używamy w niej instrukcji **`if`**), to w każdej możliwej ścieżce wykonania tej metody musi znajdować się instrukcja **`return`** (*wyjątkiem od tej reguły jest użycie wyjątków, o których będziemy się wkrótce uczyć*).

  Poniższa metoda spowodowałaby, że program by się nie skompilował, ponieważ metoda nie zwróci żadnej wartości, gdy `y == 0`:

```
public static double podzielLiczby(int x, int y) {
  if (y != 0) {
    return x / y;
  } // brakuje return w przypadku, gdy y == 0! blad kompilacji
}
```

- Jeżeli po instrukcji **`return`** będą znajdować się jakieś instrukcje, to kod takiej metody się nie skompiluje, gdyż kompilator jest w stanie wychwycić takie przypadki i zaprotestować:

```
public static int liczbaDoKwadratu(int x) {
  return x * x;

  // linia po return nie ma szansy sie wykonac - blad kompilacji
  System.out.println("Instrukcja po return!");
}
```

7.3.6 Pytania do zwracania wartości

1. Jak zwrócić z metody wartość?
2. Czy metoda, która zwraca liczbę, może także zwrócić tekst (`String`)?
3. Czy metoda może nic nie zwracać, a jeśli tak, to jak to osiągnąć?
4. Czy możemy użyć słowa kluczowego **`return`** więcej niż raz w metodzie?
5. Czy możemy użyć słowa kluczowego **`return`** w metodzie, która nic nie zwraca?
6. Jeżeli metoda ma zwrócić wartość typu **`double`**, ale nie użyjemy w niej **`return`**, czy kod się skompiluje?
7. Gdzie możemy użyć wartości zwracanej przez metodę?
8. Czy wartość zwrócona przez metodę musi zostać zawsze użyta?
9. Czy poniższy kod jest poprawny (kod klasy opakowującej metody został pominięty)?

```
public static void main(String[] args) {
  String tekst = podniesDoKwadratu(16);
}

public static int podniesDoKwadratu(int liczba) {
  return liczba * liczba;
}
```

10. Które z poniższych metod są nieprawidłowe i dlaczego?

```
public static wypiszKomunikat() {
  System.out.println("Witajcie!");
}
```

```
public static void x() {}
```

```
public static void getInt() {
  return new Scanner(System.in).nextInt();
}
```

```
public static int doKwadratu(int c) {
  int wynik = c * c;
}
```

```
public static int podzielLiczby(int a, int b) {
  if (b == 0) {
    return;
  }
  return a / b;
}
```

```
public static String getInt() {
  return new Scanner(System.in).nextInt();
}
```

```
public static int ktoraNajwieksza(int a, int b, int c) {
  if (a > b) {
    if (a > c) {
      return a;
    }
  } else {
    if (c > b) {
      return c;
    } else {
      return b;
    }
  }
}
```

```
public static void wypiszKwadrat(int a) {
  System.out.println("Kwadrat wynosi: " + a * a);
  return;
  System.out.println("Policzone!");
}
```

```
public static void wypiszPowitanie {
  System.out.println("Witajcie!");
}
```

7.3.7 Zadania do zwracania wartości

7.3.7.1 Metoda podnosząca do sześcianu

Napisz metodę, która zwróci liczbę przesłaną jako argument podniesioną do sześcianu.

7.3.7.2 Metoda wypisująca gwiazdki

Napisz metodę, która wypisze podaną liczbę gwiazdek (znak *) na ekran.

7.4 Argumenty metod

Jak już widzieliśmy, metody mogą przyjmować zero, jeden lub więcej argumentów.

Argumenty metody:

- mają określony typ (prymitywny, jak na przykład `int`, bądź złożony, jak `String`),
- mają nazwę (zgodną z zasadami nazwnictwa obiektów w Javie),
- są rozdzielone przecinkami,
- nie muszą być tego samego typu,
- nie mogą mieć określonych wartości domyślnych – jeżeli metoda ma przyjąć trzy argumenty, to tyle ich musimy podać podczas wywoływania takiej metody,
- podczas wywoływania muszą być podane w takiej samej kolejności, w jakiej zostały zdefiniowane w metodzie.

Spójrzmy na poniższy przykład:

Nazwa pliku: PrzykladArgumentowMetod.java

```
import java.util.Scanner;

public class PrzykladArgumentowMetod {
  public static void main(String[] args) {
    System.out.println("Ile gwiazdek wypisac?");

    int liczbaGwiazdek = getInt(); // (1)

    wypiszWielokrotnosc("*", liczbaGwiazdek); // (2)
    // wypiszWielokrotnosc(liczbaGwiazdek, "*"); // (3)
  }

  public static int getInt() { // (4)
    return new Scanner(System.in).nextInt();
  }

  public static void wypiszWielokrotnosc(String tekst, int ileRazy) { // (5)
    for (int i = 0; i < ileRazy; i++) {
      System.out.print(tekst);
    }
  }
}
```

- Metoda `getInt` nie przyjmuje żadnych argumentów *(4)* – pomimo tego, wywołując ją musimy użyć pustych nawiasów *(1)*.
- Metoda `wypiszWielokrotnosc` przyjmuje dwa argumenty *(5)* – jeden typu złożonego `String`, a drugi typu podstawowego `int`.
- Wywołując *(2)* metodę `wypiszWielokrotnosc`, musimy podać oba argumenty tej metody.
- Gdybyśmy spróbowali wywołać metodę `wypiszWielokrotnosc` podając argumenty w złej kolejności *(3)*, powyższy kod źródłowy by się nie skompilował.

7.4.1 Modyfikacja argumentów metod

O poniższym zagadnieniu opowiemy sobie bardzo dokładnie w kolejnym rozdziale, gdy zaczniemy uczyć się czym są klasy w języku Java.

W Javie istnieją dwa rodzaje typów – prymitywne oraz referencyjne (zwane *typami złożonymi*). W rozdziale trzecim poznaliśmy wszystkie osiem typów prymitywnych, którymi są:

- **boolean**
- **byte**
- **short**
- **int**
- **long**
- **float**
- **double**
- **char**

Typy prymitywne są elementami budującymi, z których składają się typy złożone, definiowane przez programistów.

Typy referencyjne – to typy złożone, zdefiniowane w bibliotekach standardowych języka Java, oraz tworzone przez programistów. Do typów referencyjnych zaliczamy także tablice. Przykładem typu złożonego jest `String`. Biblioteka standardowa Java udostępnia tysiące tego rodzaju typów.

Oba te rodzaje typów mają kilka bardzo istotnych różnic, o których opowiemy sobie w rozdziale siódmym. Na razie skupimy się na jednej z nich – przekazywaniu i modyfikowaniu ich w metodach.

Spójrzmy na poniższy przykład – co zostanie wypisane na ekranie?

Nazwa pliku: ZmianaArgumentowWMetodzie.java

```
public class ZmianaArgumentowWMetodzie {
  public static void main(String[] args) {
    int x = 5; // (1)
    int[] tablicaWartosci = { 1, 2, 3 }; // (2)

    doKwadratu(x); // (3)
    wszystkieDoKwadratu(tablicaWartosci); // (4)

    System.out.println("X wynosi " + x); // (5)

    for (int i = 0; i < tablicaWartosci.length; i++) { // (6)
      System.out.println("Element tablicy o indeksie " + i +
          " wynosi " + tablicaWartosci[i]); // (7)
    }
  }

  public static void doKwadratu(int liczba) {
    liczba = liczba * liczba; // (8)
  }

  public static void wszystkieDoKwadratu(int[] tablicaLiczb) {
    for (int i = 0; i < tablicaLiczb.length; i++) { // (9)
      tablicaLiczb[i] = tablicaLiczb[i] * tablicaLiczb[i]; // (10)
    }
  }
}
```

Niespodziewanie, na ekranie zobaczymy następujące wartości:

```
X wynosi 5
Element tablicy o indeksie 0 wynosi 1
Element tablicy o indeksie 1 wynosi 4
Element tablicy o indeksie 2 wynosi 9
```

Przeanalizujmy powyższy program:

- Na początku metody `main` tworzymy po jednej zmiennej typu prymitywnego *(1)* i referencyjnego *(2)* (tablicę liczb typu **`int`**).
- Następnie, wywołujemy metody `doKwadratu` *(3)* i `wszystkieDoKwadratu` *(4)*, przekazując jako argumenty utworzone wcześniej zmienne.
- W obu metodach modyfikujemy przesłane do tych metod argumenty *(8)* *(9, 10)*.
- Po powrocie z obu metod wypisujemy na ekran wartości obu zmiennych *(5)* *(6, 7)*.

Dlaczego zmienna `x` zachowała po wywołaniu metody `doKwadratu` swoją pierwotną wartość `5`, natomiast elementy tablicy `tablicaWartosci` po powrocie z metody `wszystkieDoKwadratu` mają wartości podniesione do kwadratu?

Powodem są różne rodzaje typów tych zmiennych – zmiany argumentów prymitywnych typów (jak **`int`**) nie zmieniają oryginalnej zmiennej, podczas gdy argumenty typów referencyjnych operują bezpośrednio na przesłanym obiekcie – w tym przypadku, na tablicy. Dlatego zmienna `x` zachowała swoją pierwotną wartość, a zmiany wykonane na tablicy, którą podaliśmy jako argument metody `wszystkieDoKwadratu`, "przetrwały" po zakończeniu tej metody.

Jest to bardzo ważna różnica i zawsze trzeba mieć świadomość, że modyfikacja argumentów typów referencyjnych pociąga za sobą zmiany w oryginalnym obiekcie. W rozdziale o klasach poznamy powód takiego zachowania zmiennych typów referencyjnych.

7.5 Metody typu String

Typ `String` oferuje wiele przydatnych metod – zaraz zaznajomimy się z kilkoma z nich.

> *Nie mylmy stringów ze znakami! Stringi zapisujemy w cudzysłowach, np.* `"Wijtacie!"`*, a znaki (typ* **`char`***) w apostrofach, na przykład* `'a'`*!*

7.5.1 Przypomnienie metod length i charAt oraz indeksów znaków

Jednym z typów referencyjnych, z którego do tej pory korzystaliśmy, jest typ `String`, który służy do przechowywania ciągu znaków.

> *Typ* `String` *jest szczególnym typem w języku Java z kilku powodów – o typie* `String` *opowiemy sobie dokładniej w rozdziale o klasach.*

Typ `String` oferuje wiele przydatnych metod, które możemy używać w naszych programach. Korzystaliśmy już z metod:

- `length`, aby poznać liczbę znaków, z których składał się `String`,
- `charAt`, metody, która zwracała znak na danej pozycji w stringu (**pamiętajmy, że indeks pierwszego znaku to** `0`**, a nie** `1`!).

Aby skorzystać z którejś z metod typu `String`, po nazwie zmiennej należy napisać kropkę, a następnie nazwę metody z nawiasami i ewentualnymi argumentami – zasady są takie same, jak przy wywoływaniu napisanych przez nas metod z poprzedniech podrozdziałów:

Nazwa pliku: StringLengthCharAt.java

```
public class StringLengthCharAt {
  public static void main(String[] args) {
    String komunikat = "Witaj Swiecie!";

    int liczbaZnakow = komunikat.length();
    System.out.println("Liczba znakow to: " + liczbaZnakow);

    // musimy odjac 1 od wyniku length(), poniewaz
    // indeks znakow zaczyna sie od 0, a nie 1!
    char ostatniZnak = komunikat.charAt(komunikat.length() - 1);
    System.out.println("Ostatni znak to: " + ostatniZnak);
  }
}
```

1. W programie tworzymy zmienną typu `String`.
2. Następnie, przypisujemy do zmiennej `liczbaZnakow` wynik wywołania metody `length` na zmiennej `komunikat`, która zwróci liczbę znaków w stringu `"Witaj Swiecie!"`, po czym wypisujemy liczbę znaków na ekran.
3. Do zmiennej `ostatniZnak` zapisujemy ostatni znak w stringu. Aby to zrobić, używamy metody `charAt`, która przyjmuje jako argument indeks znaku, który chcemy pobrać ze stringa. Jako, że indeks znaków zaczyna się od `0`, a nie `1`, musimy od wyniku metody `length` odjąć `1`, aby nie wyjść poza zakres stringa. Jest to zobrazowane poniżej:

Numer znaku	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Indeks znaku	**0**	1	2	3	4	5	6	7	8	9	10	11	12	**13**
	W	i	t	a	j		S	w	i	e	c	i	e	!

Metoda `charAt` (a także, jak zobaczymy za chwilę – wiele innych metod typu `String`) przyjmuje jako argument indeks znaku w stringu, a nie jego numer w kolejności – różnica pomiędzy tymi dwoma pojęciami jest taka, że indeks zaczynamy liczyć od `0`, zamiast od `1`.

Metoda `length` zwraca liczbę znaków, których w stringu `Witaj Swiecie!` jest `14` – gdybyśmy spróbowali użyć wyniku tej metody jako argument do metody `charAt`, to zapytalibyśmy o zwrócenie znaku o indeksie `14`, który nie istnieje! Zgodnie z powyższą tabelą, indeksem ostatniego znaku jest `13`, dlatego od wyniku metody `length` musieliśmy odjąć `1`.

W wyniku działania program, na ekranie zobaczymy:

```
Liczba znakow to: 14
Ostatni znak to: !
```

7.5.2 Przykłady użycia metod typu String

Poza metodami `length` i `charAt`, typ `String` oferuje wiele innych, przydatnych metod. Poniżej znajdują się opisy kilku z nich.

7.5.2.1 toLowerCase, toUpperCase

Te dwie metody służą do zwrócenia stringa, na rzecz którego metoda została wykonana, z, odpowiednio, wszystkie literami zamienionymi na małe litery (`toLowerCase`), lub na wielkie litery (`toUpperCase`). Ich sygnatury są następujące:

- `String toLowerCase()`
- `String toUpperCase()`

Uwaga: te metody nie modyfikują stringa, na rzecz którego zostały wywołane – zamiast tego, zwracają jego kopię ze wszystkimi literami zamienionymi na małe/wielkie litery. Co to w zasadzie oznacza? Spójrzmy na poniższy przykład:

Nazwa pliku: StringToUpperCase.java

```
public class StringToUpperCase {
  public static void main(String[] args) {
    String komunikat = "Witajcie po raz 20!";

    komunikat.toUpperCase();

    System.out.println(komunikat);
  }
}
```

Który z komunikatów zobaczymy na ekranie?

- **`Witajcie po raz 20!`**

  czy może:

- **`WITAJCIE PO RAZ 20!`**

Na ekranie zobaczymy tekst `Witajcie po raz 20!`. Stało się tak dlatego, że użycie metody `toUpperCase` nie spowodowało zmiany zawartości zmiennej `komunikat` – zamiast tego, metoda ta zwróciła tekst `WITAJCIE PO RAZ 20!`, ale ani nie przypisaliśmy wyniku tej metody do żadnej zmiennej, ani nigdzie z tej wartości nie skorzystaliśmy.

> Jest to bardzo ważna informacja – **metody `toUpperCase` i `toLowerCase` nie zmieniają w żaden sposób oryginalnego stringa, na którym zostały wywołane**. **Mało tego – żadna z metod typu String tego nie robi!** Dlaczego tak się dzieje – dowiemy się w rozdziale o klasach.

W kolejnym przykładzie nie popełaniamy już tego błędu:

Nazwa pliku: StringToUpperLowerCase.java

```
public class StringToUpperLowerCase {
  public static void main(String[] args) {
    String komunikat = "Witajcie po raz 20!";

    String wielkieLitery = komunikat.toUpperCase();

    System.out.println(wielkieLitery);
    System.out.println(komunikat.toLowerCase());
  }
}
```

W tym przykładzie, wynik działania metody `toUpperCase`, użytej na zmiennej `komunikat`, przypisujemy do zmiennej `wielkieLitery`. Następnie, wypisujemy zawartość zmiennej `wielkieLitery` – tym razem na ekranie zobaczymy tekst z wielkimi literami.

Na końcu programu wypisujemy także na ekran wynik działania `toLowerCase`, dzięki czemu zobaczymy na ekranie ponownie wiadomość, jednak będzie ona miała pierwszą literę zamienioną na małą literę:

```
WITAJCIE PO RAZ 20!
witajcie po raz 20!
```

Warto tutaj także zwrócić uwagę, że liczba `20` oraz wykrzyknik `!` nie zmieniły się – metody `toLowerCase` i `toUpperCase` zmieniają jedynie litery.

> **Pamiętajmy**: wszelkie operacje wykonywane za pomocą metod typu `String` nie powodują zmiany stringa, na rzecz którego metoda została wykonana – zamiast tego, zwracana jest zmodyfikowana kopia!

7.5.2.2 endsWith, startsWith, contains

Sygnatury tych trzech metod są następujące:

1. `boolean startsWith(String prefix)`
2. `boolean endsWith(String suffix)`
3. `boolean contains(CharSequence s)`

Każda z powyższych metod odpowiada na pewne pytanie, zwracając *prawdę* bądź *fałsz* za pomocą wartości o typie **`boolean`** (**`true`** / **`false`**).

1. Pierwsza metoda, `startsWith`, zwraca **`true`**, jeżeli string, na rzecz którego metoda została

wywołana, zaczyna się od podanego jako argument ciągu znaków. W przeciwnym razie zwracana wartość to **false**.

2. Metoda `endsWith` działa podobnie jak `startsWith` z tym, że sprawdzany jest koniec stringu.
3. Metoda `contains` działa podobne, jak dwie poprzednie metody z tym, że sprawdzany jest cały string – jeżeli przesłany jako argument ciąg znaków jest zawarty w stringu, na rzecz którego wywołujemy tę metodę, zwrócona zostanie wartość **true**. Jeżeli nie, to **false**.

Spójrzmy na przykład użycia powyższych metod:

Nazwa pliku: StringStartsWithEndsWithContains.java

```
public class StringStartsWithEndsWithContains {
  public static void main(String[] args) {
    String s = "Ala ma kota";

    System.out.println("Czy string zaczyna sie od 'Ala'? " + s.startsWith("Ala"));
    System.out.println("Czy string zaczyna sie od 'ala'? " + s.startsWith("ala"));

    System.out.println("Czy string konczy sie na 'kota'? " + s.endsWith("kota"));

    System.out.println("Czy zawiera 'ma'? " + s.contains("ma"));
    System.out.println("Czy zawiera 'kot'? " + s.contains("kot"));
    System.out.println("Czy zawiera 'ala'? " + s.contains("ala"));
    System.out.println("Czy zawiera 'ala'? " + s.toLowerCase().contains("ala"));
  }
}
```

W powyższym kodzie kilkukrotnie używamy przedstawionych wcześniej metod. Spójrzmy na wyniki działania programu:

```
Czy string zaczyna sie od 'Ala'? true
Czy string zaczyna sie od 'ala'? false
Czy string konczy sie na 'kota'? true
Czy zawiera 'ma'? true
Czy zawiera 'kot'? true
Czy zawiera 'ala'? false
Czy zawiera 'ala'? true
```

Interesujące są dwa następujące wyniki:

```
Czy string zaczyna sie od 'ala'? false
Czy zawiera 'ala'? false
```

Dlaczego otrzymaliśmy takie wyniki? Metoda `startsWith` zwróciła **true** dla argumentu `Ala`, ale **false** dla argumentu `ala`. Podobnie metoda `contains` – zwróciła **false** dla argumentu `ala`, chociaż taki ciąg znaków znajduje się w sprawdzanym stringu.

Takie wyniki tych metod wynikają z faktu, że małe i wielkie litery w stringach są rozróżniane i traktowane są jako różne znaki – dlatego zobaczyliśmy na ekranie takie a nie inne wyniki.

Jeżeli chcemy zignorować małe i wielkie litery, możemy skorzystać z poznanych już metod `toLowerCase` i `toUpperCase`, by sprawdzić, czy, na przykład, string zawiera pewien ciąg znaków, tak jak w powyższym programie w ostatnim przykładzie:

```
System.out.println("Czy zawiera 'ala'? " + s.toLowerCase().contains("ala"));
```

Skorzystaliśmy z metody `toLowerCase`, która zwróciła zawartość zmiennej `s` ze wszystkimi

literami zamienionymi na małe litery – nastepnie, od razu wywołaliśmy na zwróconym stringu metodę `contains` (użycie metod w ten sposób, jedna po drugiej, to tzw. *method chaining*). Dzięki temu, wywołaliśmy metodę `contains` z argumentem `ala` nie na stringu `Ala ma kota`, lecz na stringu `ala ma kota` – dlatego na ekranie zobaczyliśmy wynik `true`.

7.5.2.3 equals, equalsIgnoreCase

Metody `equals` oraz `equalsIgnoreCase` służą do porównywania stringów – ich sygnatury są następujące:

- `boolean equals(Object anObject)`
- `boolean equalsIgnoreCase(String anotherString)`

Jeżeli string, który prześlemy jako argument do `equals` lub `equalsIgnoreCase` jest taki sam, jak string, na rzecz którego wywołujemy jedną z tych metod, to zwrócona zostanie wartość `true`, a w przeciwnym razie `false`. Metody te różnią się tym, że druga z nich, `equalsIgnoreCase`, ignoruje wielkość znaków (stringi `"Ala"` i `"ala"` są dla niej taki same, w przeciwieństwie do meteody `equals`).

Te dwie metody mają jeszcze kilka różnic – na przykład, czym jest tajemniczy typ `Object`, którego oczekuje metoda `equals`? Dowiemy się w kolejnym rozdziale! Na ten moment wystarczy nam informacja, że do metody `equals` możemy po prostu przesłać string.

Zobaczmy przykład użycia dwóch powyższych metod:

Nazwa pliku: StringEqualsEqualsIgnoreCase.java

```
import java.util.Scanner;

public class StringEqualsEqualsIgnoreCase {
  public static void main(String[] args) {
    System.out.print("Podaj slowo: ");

    String slowo = getString();

    if (slowo.equals("kot")) {
      System.out.println("Podane slowo to kot.");
    } else {
      System.out.println("Podano slowo inne niz kot");
    }

    if (slowo.equalsIgnoreCase("Kot")) {
      System.out.println(
        "Podane slowo (bez uwzglednienia wielkosci znakow) to kot."
      );
    } else {
      System.out.println(
        "Podano slowo inne niz kot (nawet po nie braniu pod uwage wielkosci znakow)."
      );
    }
  }

  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

W tym programie pobieramy od użytkownika słowo i porównujemy je do słowa `kot` dwukrotnie –

raz za pomocą `equals`, a drugi raz za pomocą `equalsIgnoreCase`, by nie uwzględnić przy porównaniu wielkości znaków. Spójrzmy na wyniki dwukrotnego uruchomienia tego programu:

```
Podaj slowo: KOT
Podano slowo inne niz kot
Podane slowo (bez uwzglednienia wielkosci znakow) to kot.
```

```
Podaj slowo: pies
Podano slowo inne niz kot
Podano slowo inne niz kot (nawet po nie braniu pod uwage wielkosci znakow).
```

Pytanie: po co nam dwie powyższe metody? Czy nie możemy po prostu porównywać stringów za pomocą operatora porównania ==, tak jak porównujemy np. liczby?

Jest to podchwytliwe pytanie, ponieważ możemy używać operatora == by porównywać stringi, ale **nigdy nie powinniśmy tego robić** – dlaczego tak jest dowiemy się w rozdziale o klasach. Użycie operatora porównania do porównywania stringów zazwyczaj zwraca inne wyniki, niż byśmy się spodziewali:

Nazwa pliku: StringOperatorPorownania.java

```java
import java.util.Scanner;

public class StringOperatorPorownania {
  public static void main(String[] args) {
    System.out.print("Podaj slowo: ");

    String slowo = getString();

    if (slowo == "kot") {
      System.out.println("Wpisales slowo kot.");
    } else {
      System.out.println("Wpisales slowo inne niz kot.");
    }
  }

  public static String getString() {
    return new Scanner(System.in).next();
  }
}
```

W powyższym programie pobieramy od użytkownika słowo i zapisujemy je w zmiennej o nazwie `slowo`. Następnie, używamy operatora porównania by sprawdzić, czy słowo, jakie podał użytkownik, to `kot`.

Zobaczmy co się stanie, gdy podamy słowo `kot`:

```
Podaj slowo: kot
Wpisales slowo inne niz kot.
```

Skoro podaliśmy słowo `kot`, to dlaczego zobaczyliśmy na ekranie niespodziewany komunikat? Wynika to z nieprawidłowego sposobu porównywania stringów – zamiast skorzystać z metody `equals` bądź `equalsIgnoreCase`, użyliśmy operatora `==`, którego nigdy nie powinniśmy stosować do porównywania stringów. W rozdziale o klasach dowiemy się z czego wynika ten wymóg.

Zapamiętajmy – gdy będziemy musieli porównywać dwa łańcuchy tekstowe (stringi), powinniśmy używać metody `equals` (lub `equalsIgnoreCase`) zamiast używać operator

porównania == .

Pamiętajmy także, by nie mylić stringów ze znakami – stringi zapisujemy w cudzysłowach " ", a znaki w apostrofach ' ' – porównywanie znaków za pomocą operatora porównania ==, w przeciwieństwie do porównywania stringów, *jest* poprawne.

7.5.2.4 replace, substring, split

Trzema ostatnimi metodami typu `String`, które poznamy, są:

1. `String replace(CharSequence oldChar, CharSequence newChar)`
2. `String substring(int beginIndex, int endIndex)`
3. `String[] split(String regex)`

1. Metoda `replace` zwraca kopię stringa, na rzecz którego ją wywołujemy, ze wszystkimi wystąpieniami tekstu przesłanego jako pierwszy argument zastąpionymi tekstem podanym jako drugi argument.
2. Metoda `substring` zwraca fragment stringa, na rzecz którego ją wywołujemy. Fragment ten zaczyna się od **indeksu** przesłanego jako pierwszy argument, a kończy na indeksie przesłanym jako drugi argument. Zwracany fragment nie zawiera znaku o indeksie wskazywanym jako drugi argument.

   (przypomnijmy: indeks pierwszego znaku to `0`)
3. Ostatnia metoda, `split`, zwraca tablicę stringów – dzieli ona na części string, na rzecz którego wywołujemy tę metodę, używając przesłanego argumentu jako separator. Na przykład, wywołanie `split` na stringu `raz,dwa,trzy` z argumentem , (przecinek) zwróciłoby tablicę stringów z trzema elementami: "`raz`" "`dwa`" "`trzy`".

Spójrzmy na użycie każdej z tych metod:

Nazwa pliku: StringReplace.java

```
public class StringReplace {
  public static void main(String[] args) {
    String tekst = "Ala ma kota";

    String zmianaImienia = tekst.replace("Ala", "Jola");
    String bezSpacji = tekst.replace(" ", "");

    System.out.println("Tekst z innym imieniem: " + zmianaImienia);
    System.out.println("Tekst bez spacji: " + bezSpacji);
  }
}
```

Do zmiennej `zmianaImienia` zapisujemy zawartość zmiennej `tekst` z zamienionym ciągiem znaków `Ala` na ciąg znaków `Jola`.

Kolejna zmienna, `bezSpacji`, otrzymuje wynik działania metody `replace`, do której jako pierwszy argument podaliśmy spację, którą chcemy zamienić na... pustego stringa – w takim przypadku wszelkie spacje zostaną po prostu usunięte w wynikowym stringu i na ekranie zobaczymy:

```
Tekst z innym imienie: Jola ma kota
Tekst bez spacji: Alamakota
```

Uwaga: małe i wielkie litery mają znaczenie podczas szukania tekstu do zastępienia! Gdybyśmy

przy pierwszym z powyższych wywołań metody `replace` podali jako pierwszy argument `"ala"`, to zostałby zwrócony niezmieniony string `Ala ma kota`.

Czas na przykład użycia metody `substring`:

Nazwa pliku: StringSubstring.java

```
public class StringSubstring {
  public static void main(String[] args) {
    String tekst = "Ala ma kota";

    String pierwszeSlowo = tekst.substring(0, 3);
    String drugieSlowo = tekst.substring(4, 6);
    String trzecieSlowo = tekst.substring(7, tekst.length());

    System.out.println("Pierwsze slowo: " + pierwszeSlowo);
    System.out.println("Drugie slowo: " + drugieSlowo);
    System.out.println("Trzecie slowo: " + trzecieSlowo);
  }
}
```

W wyniku działania programu na ekranie zobaczymy:

```
Pierwsze slowo: Ala
Pierwsze slowo: ma
Pierwsze slowo: kota
```

Każda z trzech zmiennych otrzymuje po jednym słowie ze stringa zapisanego w zmiennej `tekst`. Osiągamy to poprzez użycie metody `substring` z odpowiednimi argumentami – indeksem pierwszego znaku, od którego chcemy pobrać fragment oryginalnego stringa, oraz indeksu znaku końcowego, który nie będzie w zwróconym fragmencie (substringu).

Poniższa tabela obrazuje, dlaczego użyliśmy takich a nie innych argumentów w każdym z wywołań metody `substring`:

Indeks	0	1	2	3	4	5	6	7	8	9	10
Znak	**A**	**l**	**a**		**m**	**a**		**k**	**o**	**t**	**a**

Aby pobrać pierwsze słowo, zaczynamy od pierwszego znaku – jego indeks to `0`. Chcemy pobrać trzy pierwsze znaki. Metoda `substring` jako drugi argument oczekuje indeksu znaku, przed którym ma zakończyć zwracany fragment – **ten znak pod indeksem podanym jako drugi argument nie jest włączany do zwracanego wyniku**. Dlatego, aby pobrać pierwsze słowo, jako drugi argument podajemy indeks spacji (który wynosi `3`), a nie indeks małej litery `a` (indeks `2`).

Analogicznie dzieje się z drugim słowem – zaczynamy od indeksu `4`, ponieważ pod tym indeksem znajduje się pierwsza litera drugiego słowa (`m`). Jako koniec podajemy indeks spacji pod indeksem `6` – ponownie, ta spacja nie będzie w zwróconym fragmencie tekstu.

Ostatni przykład jest najbardziej interesujący – otóż nie odejmujemy od wartości zwróconej przez metodę `length` liczby `1` (tak jak robiliśmy to już kilkukrotnie w innych przykładach w tym rozdziale)! Czy nie spowoduje to błędu? Nie – w końcu metoda `substring` nie zwraca znaku znajdującego się pod indeksem przekazanym jako drugi argument. Argument ten służy jedynie do wyznaczenia granicy zwracanego fragmentu tekstu. Tak się składa, że dla fragmentu, który jest końcowym fragmentem oryginalnego tekstu, granicą jest indeks następujący po ostatnim indeksie

w oryginalnym stringu. Ten indeks to `11`, ponieważ ostatni znak ma indeks równy `10`. Taką właśnie wartość zwraca metoda `length` (ponieważ string `Ala ma kota` składa się z `11` znaków).

Na końcu przyjrzyjmy się metodzie `split`:

Nazwa pliku: StringSplit.java

```
public class StringSplit {
  public static void main(String[] args) {
    String tekst = "Ala ma kota";

    // podziel tekst - jako separator uzyj spacji
    String[] slowa = tekst.split(" ");

    for (int i = 0; i < slowa.length; i++) {
      System.out.println("Slowo nr " + (i + 1) + " to: " + slowa[i]);
    }


    String zwierzetaPoPrzecinku = "kot,pies,,chomik";

    // podziel tekst - jako separator uzyj przecinka
    String[] zwierzeta = zwierzetaPoPrzecinku.split(",");

    for (int i = 0; i < zwierzeta.length; i++) {
      System.out.println("Zwierze nr " + (i + 1) + " to: " + zwierzeta[i]);
    }
  }
}
```

W tym przykładzie najpierw używamy metody `split` by podzielić tekst `Ala ma kota` na słowa, które będą zwrócone w postaci tablicy stringów. Jako separator do oddzielenia słów od siebie podajemy spację jako argument do metody `split`.

W kolejnym przykładzie rozdzielamy tekst `kot,pies,,chomik` używając jako separatora przecinka. Zauważmy, że pomiędzy drugim a trzecim przecinkiem nie ma żadnego znaku – spowoduje to, że w tablicy stringów, która zostanie zwrócona, trzeci element będzie pustym stringiem, co widzimy na ekranie po uruchomieniu powyższego programu:

```
Slowo nr 1 to: Ala
Slowo nr 2 to: ma
Slowo nr 3 to: kota
Zwierze nr 1 to: kot
Zwierze nr 2 to: pies
Zwierze nr 3 to:
Zwierze nr 4 to: chomik
```

Trzeci element tablicy `zwierzeta` to pusty string.

Powyżej przedstawionych zostało jedynie kilka z metod, jakie oferuje typ `String`. Informacje o tych i pozostałych metodach można znaleźć w oficjalnej dokumentacji typu `String` na stronie:

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Dokumentacja znajdująca się na podanej wyżej stronie została wygenerowana automatycznie – czy i my możemy (i jak to zrobić) dokumentować nasze metody?

7.6 Dokumentowanie metod

Wiele nowoczesnych języków programowania udostępnia rodzaj komentarza, który możemy stosować w naszym kodzie do dokumentowania działania metod naszego programu (Rozdział II). Komentarz taki zaczynamy od znaków `/**` a kończymy znakami `*/`

Komentarze dokumentujące służą do:

- pomocy w zrozumieniu działania metody,
- opisie parametrów metody,
- opisie zwracanej wartości,
- informacji, czy metoda rzuca jakieś wyjątki i w jakich sytuacjach (o wyjątkach będziemy się uczyć wkrótce),
- dodatkowo, można w komentarzu poinformować o m. in. autorze metody i wersji, od której jest dostępna.

Dokładny opis jak pisać komentarze tego typu można znaleźć na oficjalnej stronie firmy Oracle (właściciela języka Java):
https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Istnieją narzędzia, które potrafią przejść przez kod źródłowy naszego programu i wygenerować do niego dokumentację na podstawie komentarzy tego specjalnego typu. Przykładem takiego narzędzia jest oficjalny program do tworzenia dokumentacji dla języka Java o nazwie *JavaDoc*.

Wygenerowaną w ten sposób dokumentację możemy zobaczyć na np. oficjalnej stronie dokumentacji języka Java pod adresem:

https://docs.oracle.com/javase/8/docs/api/

Bardzo wiele frameworków, bibliotek itp. ma własne, wygenerowane w ten sposób dokumentacje dostępne w Internecie – programiści często z nich korzystają.

Spójrzmy na przykład użycia tego rodzaju komentarzy:

Nazwa pliku: JavaDocPrzyklad.java

```
import java.util.Scanner;

public class JavaDocPrzyklad {
  public static void main(String[] args) {
    System.out.println("Podaj liczbe");

    int x = getInt();

    System.out.println("Ta liczba do kwadratu to " + policzKwadrat(x));
  }

  /**
   * Metoda czeka na podanie przez uzytkownika wartosci, po czym ja zwraca.
   * @return Wartosc podana przez uzytkownika.
   */
  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }

  /**
   * Zwraca przeslana liczbe podniesiona do kwadratu.

   * Przyklad: dla argumentu liczba rownego 5, zwroci 25.
   *
   * @param liczba Liczba, ktora chcemy podniesc do kwadratu.
   * @return Liczba podniesiona do kwadratu.
   */
  public static double policzKwadrat(int liczba) {
    return liczba * liczba;
  }
}
```

Przed metodami `getInt` oraz `policzKwadrat` znajdują się komentarze dokumentacyjne, opisujace co robi każda z tych metod. Dodatkowo, metoda `policzKwadrat` ma jeden argument, który opisujemy w następujący sposób: najpierw piszemy słowo ***param*** poprzedzone znakiem *@* (małpa), po którym następuje nazwa argumentu, a po spacji jego opis (może on być dłuższy niż jedna linia).

Możemy w ten sposób opisać więcej, niż jeden parametr:

```
@param x Pozycja X.
@param y Pozycja Y.
```

Obie powyższe metody zwracają wartość – do opisu zwracanej wartości używamy słowa ***@return***.

Jeżeli modyfikujemy czyjś kod, który ma komentarz dokumentacyjny, należy po wprowadzeniu zmian do metody zaktualizować także komentarz dokumentujący – jest to bardzo ważne, aby zachować spójność.

Pytanie: czy powinniśmy dokumentować wszystkie metody? Raczej nie – tylko te bardziej skomplikowane lub takie, których opis będzie przydatny w wygenerowanej dokumentacji, którą, być może, dostarczymy osobom, które będą korzystały z naszego kodu.

7.7 Podsumowanie, pytania i zadania do argumentów metod i metod typu String

7.7.1 Argumenty metod

- Metody mogą przyjmować zero, jeden lub więcej argumentów.
- Argumenty metody:
  - mają określony typ (prymitywny, jak na przykład `int`, bądź złożony, jak `String`),
  - mają nazwę (zgodną z zasadami nazwnictwa obiektów w Javie),
  - są rozdzielone przecinkami i nie muszą być tego samego typu,
  - nie mogą mieć określonych wartości domyślnych – jeżeli metoda ma przyjąć trzy argumenty, to tyle ich musimy podać podczas wywoływania takiej metody,
  - podczas wywoływania muszą być podane w takiej samej kolejności, w jakiej zostały zdefiniowane w metodzie.
- W Javie istnieją dwa rodzaje typów: prymitywne (jest ich osiem: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, oraz `char`) oraz referencyjne (tablice, `String`, wiele innych).
- Zmiany argumentów prymitywnych typów (jak `int`) w metodach nie zmieniają oryginalnej zmiennej:

```
public class ZmianaArgumentuTypuPrymitywnego {
  public static void main(String[] args) {
    int x = 5;
    doKwadratu(x);

    System.out.println("Po wywolaniu doKwadratu, x wynosi: " + x);
  }

  public static void doKwadratu(int liczba) {
    liczba = liczba * liczba;
  }
}
```

Na ekranie zobaczymy `Po wywolaniu doKwadratu, x wynosi: 5`, ponieważ w metodzie `doKwadratu` nie wykonaliśmy zmiany na oryginalnej zmiennej, lecz na jej kopii.

- Argumenty typów referencyjnych operują bezpośrednio na przesłanym obiekcie – na ekranie zobaczymy `Drugi element tablicy to: 25`, gdyż w metodzie `wszystkieDoKwadratu` działamy nie na kopii tablicy, lecz na oryginalnym obiekcie:

```
public class ZmianaArgumentuTypuReferencyjnego {
  public static void main(String[] args) {
    int[] tablicaWartosci = { 5, 10, 15 };

    wszystkieDoKwadratu(tablicaWartosci);

    System.out.println("Drugi element tablicy to: " + tablicaWartosci[0]);
  }

  public static void wszystkieDoKwadratu(int[] tablicaLiczb) {
    for (int i = 0; i < tablicaLiczb.length; i++) {
      tablicaLiczb[i] = tablicaLiczb[i] * tablicaLiczb[i];
    }
  }
}
```

7.7.2 Metody typu String

- Typ `String` służy do przechowywania ciągu znaków.
- Pamiętajmy! Indeks liter w stringach zaczyna się od `0`, a nie `1`, więc pierwszy znak w stringu ma indeks `0`, a ostatni – liczba znaków w stringu - 1 (czyli `pewnyString.length() - 1`).
- Typ `String` oferuje wiele przydatnych metod, m. in.:

  (poniżej zakładamy, że zmienna `tekst` to `String tekst = "Ala ma kota";`):

  - `length` – zwraca liczbę znaków w stringu,

    ```
    int liczba = tekst.length();
    ```

  - `charAt` – zwraca znak o podanym indeksie w stringu (**indeksy zaczynają się od** `0`),

    ```
    char pierwszyZnak = tekst.charAt(0);
    ```

  - `toLowerCase` – zwraca string ze wszystkimi literami zamienionymi na małe litery,

    ```
    String tekstMalymiLiterami = tekst.toLowerCase();
    ```

  - `toUpperCase` – zwraca string ze wszystkimi literami zamienionymi na wielkie litery,

    ```
    String tekstWielkimiLiterami = tekst.toUpperCase();
    ```

  - `endsWith` – sprawdza, czy string kończy się na dany ciąg znaków,

    ```
    boolean czyKonczySieNaPsa = tekst.endsWith("psa");
    ```

  - `startsWith` – sprawdza, czy string zaczyna się na dany ciąg znaków,

    ```
    boolean czyZaczynaSieOdAla = tekst.startsWith("Ala");
    ```

  - `contains` – sprawdza, czy string zawiera dany ciąg znaków,

    ```
    boolean czyZawieraMaKota = tekst.contains("ma kota");
    ```

  - `equals` – porównuje stringi – jeżeli są sobie równe, zwaraca `true`. W przeciwnym razie zwraca `false`.

    ```
    boolean czyAlaMaPsa = tekst.equals("Ala ma psa")
    ```

  - `equalsIgnoreCase` – porównuje stringi bez brania pod uwagę wielkości znaków, tzn. małe i wielkie litery traktuje jako takie same litery. Jeżeli stringi są sobie równe, zwaraca `true`. W przeciwnym razie zwraca `false`.

    ```
    boolean czyAlaMaKota = tekst.equalsIgnoreCase("ALA MA KOTA")
    ```

  - `replace` – zamienia wszystkie wystąpienia podanego ciągu znaków na inny, podany ciąg znaków.

    ```
    String nowyTekst = tekst.replace("kota", "psa");
    ```

  - `substring` – zwraca fragment tekstu pomiędzy dwoma indeksami, nie włączając w wynikowym fragmencie litery znajdującym się pod ostatnim indeksem.

    ```
    String al = tekst.substring(0, 2); // al ma wartosc "Al"
    ```

  - `split` – dzieli tekst na słowa za pomocą podanego separatora – zwraca tablicę Stringów.

    ```
    String[] slowaTekstu = tekst.split(" "); // separator to spacja
    ```

- Żadna z powyższych metod nie zmienia w żaden sposób oryginalnego stringa. Dla przykładu: `jakisString.toLowerCase()` nie spowoduje zmiany liter w stringu `jakisString` na małe litery, lecz zwróci jego kopię ze wszystkimi literami zamienionymi na małe litery.
- **Uwaga**: małe i wielkie litery są w stringach rozróżniane, czyli stringi `"kot"` i `"KOT"` nie są sobie równe. Ma to znaczenie podczas używania opisanych powyżej metod – sprawdzanie, czy string zawiera ciąg znaków `"ala"` i `"Ala"` to dwa różne sprawdzenia.
- Nigdy nie powinniśmy porównywać stringów za pomocą operatora porównania == – zamiast niego, powinniśmy stosować metodę `equals` lub `equalsIgnoreCase`.
- Nie powinniśmy mylić stringów ze znakami – stringi zapisujemy w cudzysłowach, np. `"Witajcie!"`, natomiast znaki w apostrofach, np. `'a'`. Znaki należy, w przeciwieństwie do stringów, porównywać za pomocą operatora ==.
- Typ `String` jest dobrze udokumentowany – oficjalną dokumentację można znaleźć pod adresem: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

7.7.3 Dokumentowanie metod

- Komentarz w kodzie zaczynający się od znaków `/**` i kończący znakami `*/` to komentarz dokumentacyjny.
- Komentarze dokumentujące służą między innymi do pomocy w zrozumieniu działania metody, opisie jej parametrów, zwracanej wartości itp.
- Z takich komentarzy można wygenerować dokumentację opisującą metody.
- Przykład komentarza dokumentującego:

```
/**
 * Zwraca przeslana liczbe podniesiona do kwadratu.
 *
 * Przyklad: dla argumentu liczba rownego 5, zwroci 25.
 *
 * @param liczba Liczba, ktora chcemy podniesc do kwadratu.
 * @return Liczba podniesiona do kwadratu.
 */
public static double policzKwadrat(int liczba) {
  return liczba * liczba;
}
```

7.7.4 Pytania

1. Do czego służą argumenty metod?
2. Czy metody mogą przyjmować zero argumentów?
3. Czy metodę, która przyjmuje jeden argument – liczbę typu `int`, możemy wywołać bez podania żadnego argumentu?
4. Czy kolejność argumentów ma znaczenie?
5. Jeżeli w metodzie zmodyfikujemy argument typu prymitywnego, to czy po powrocie z tej metody wartość zmiennej użytej jako argument do metody zachowa wartość ustawioną w wywołanej metodzie, czy będzie miała swoją oryginalną wartość?
6. Jak można udokumentować metodę, opisując jej działanie, parametry, zwracany typ itp.?
7. Czy poniższy kod jest poprawny?

```
public class Pytanie {
  public static void main(String[] args) {
    wypiszKomunikat;
  }

  public static void wypiszKomunikat() {
    System.out.println("Witajcie!");
  }
}
```

8. Jaka wartość zostanie wypisana na ekran w poniższym programie?

```
public class Pytanie {
  public static void main(String[] args) {
    int[] tab = { 7, 8, 9 };

    metoda(tab);

    System.out.println(tab[0]);
  }

  public static void metoda(int[] tab) {
    int[] tablica = tab;

    for (int i = 0; i < tab.length; i++) {
      tablica[i] = tab[i] * tab[i];
    }
  }
}
```

9. Które z poniższych sygnatur metod (pomijamy brak ciała metod) są nieprawidłowe i dlaczego?
   - `public static void metoda(wiadomosc String)`
   - `public static void metoda`
   - `public static void metoda(int byte, char znak)`

- `public static void metoda()`
- `public static void metoda(int #numerPracownika)`
- `public static void metoda(string wiadomosc)`
- `public static void metoda(int)`
- `public static void metoda(int liczba String wiadomosc)`
- `public static void metoda(int _numerPracownika)`
- `public static void metoda(double pi = 3.14)`

10. Jak będzie wynik wykonania poniższego programu?

```
public class Pytanie {
  public static void main(String[] args) {
    metoda(3.14, 20);
  }

  public static void metoda(int liczba, double drugaLiczba) {
    System.out.println("Liczba = " + liczba);
    System.out.println("Druga liczba = " + drugaLiczba);
  }
}
```

11. Wymień kilka metod, które udostepnia typ `String`.
12. Jaki jest indeks pierwszego znaku w każdym stringu? A jaki ostatniego?
13. Czy małe i wielkie litery są rozróżniane w stringach?
14. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";
tekst.replace("Wi", "Pamie");
System.out.println(tekst);
```

15. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";
System.out.println(tekst.charAt(tekst.length()));
```

16. Jaki będzie wynik działania poniższego kodu? Co zawiera tablica `tab`?

```
String tekst = "Witajcie!";
String[] tab = tekst.split(",");
```

17. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";

if (tekst.contains("witajcie")) {
  System.out.println("Zmienna zawiera slowo witajcie.");
}
```

18. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Ala ma kota";
String[] slowa = tekst.split(" ");

if (slowa[0] + " " + slowa[1] + " " + slowa[2] == tekst) {
  System.out.println("Rowne!");
}
```

19. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";
String fragment = tekst.substring(0, 5);

System.out.println(fragment);
```

20. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";

if (tekst.equals("witajcie!")) {
  System.out.println("Rowne!");
}
```

7.7.5 Zadania

7.7.5.1 Metoda zwracająca ostatni znak

Napisz metodę, która zwróci ostatni znak w przesłanym jako argument stringu.

Dla przykładu, dla argumentu `"Witaj"`, metoda powinna zwrócić literę `j`.

7.7.5.2 Metoda czyPalindrom

Napisz metodę, która odpowiada na pytanie, czy podany string jest palindromem. Palindromy to słowa, które są takie same czytane od początku i od końca, np. **kajak**.

Dla przykładu, dla argumentu `"kajak"` (a także `"Kajak"`), metoda ta powinna zwrócić **`true`**, a dla argumentu `"kot"` – **`false`**.

7.7.5.3 Metoda sumująca liczby w tablicy

Napisz metodę, która przyjmuje tablicę liczb całkowitych i zwraca sumę wszystkich elementów tej tablicy.

Dla przykładu, dla tablicy o elementach { `1`, `7`, `20`, `100` } metoda powinna zwrócić liczbę `128`.

7.7.5.4 Metoda zliczająca znak w stringu

Napisz metodę, która przyjmuje jako argument string i znak (**`char`**) i zwraca liczbę równą liczbie wystąpień podanego znaku w danym stringu.

Dla argumentów: `"Ala ma kota"`, `'a'`, metoda powinna zwrócić `3`, ponieważ string zawiera trzy małe litery `a`. **Uwaga**: znaki zapisujemy w apostrofach, a stringi w cudzysłowach. Przykładowe wywołanie metody, ktorą należy napisać w tym zadaniu:

```
int liczbaLiterA = zliczWystapienia("Ala ma kota", 'a');
```

7.8 Przeładowywanie metod

W poprzednich rozdziałach zapoznaliśmy się z podstawami tworzenia metod. W tym rozdziale nauczymy się, czym jest *przeładowywanie* metod.

Spójrzmy na poniższy przykład – który z komunikatów zobaczymy na ekranie?

Nazwa pliku: DwieMetodyTaSamaNazwa.java

```
public class DwieMetodyTaSamaNazwa {
  public static void main(String[] args) {
    wypisz(100);
  }

  public static void wypisz(int liczba) {
    System.out.println("Przeslana liczba: " + liczba);
  }

  public static void wypisz(int wartosc) {
    System.out.println("Wartosc wynosi: " + wartosc);
  }
}
```

Komunikat, który zobaczymy, to komunikat z błędem kompilacji od kompilatora:

```
DwieMetodyTaSamaNazwa.java:10: error: method wypisz(int) is already defined
in class DwieMetodyTaSamaNazwa
    public static void wypisz(int wartosc) {
                       ^
1 error
```

Stało się tak dlatego, że kompilator nie wie, którą z dwóch metod `wypisz` miałyby wywołać i nie pozwala w ogóle na skompilowanie powyższego kodu. Zauważmy także, że różnica w nazwach argumentów obu metod `wypisz` nie ma znaczenia.

A gdybyśmy chcieli napisać metodę `wypisz` wypisującą liczbę całkowitą, rzeczywistą, przesłanego Stringa czy też tablicę, to czy jesteśmy skazani na napisanie metod o różnych nazwach, jak poniżej?

- `public static void wypiszCalkowita(int liczba) { ... }`
- `public static void wypiszRzeczywista(double liczba) { ... }`
- `public static void wypiszString(String tekst) { ... }`
- `public static void wypiszTablice(int[] tablica) { ... }`

Na szczęście nie – w języku Java (a także innych językach, jak np. C#) mamy możliwość skorzystania z mechanizmu nazywanego *przeładowywaniem metod* (*method overloading*). Pozwala on na tworzenie metod o tej samej nazwie, o ile:

- różnią się one liczbą argumentów **i / lub**
- argumenty różnią się typem.

Oznacza to, że możemy mieć wiele wersji funkcji `wypisz`, o ile ich argumenty będą się różnić:

Nazwa pliku: PrzeladowanieWypisz.java

```
public class PrzeladowanieWypisz {
  public static void main(String[] args) {
    wypisz(10);
    wypisz(7, 128);
    wypisz(3.14);
    wypisz("Witaj!");
    wypisz(new int[] {2, 40, 500});
  }

  public static void wypisz(int liczba) {
    System.out.println("Liczba calkowita: " + liczba);
  }

  public static void wypisz(int pierwsza, int druga) {
    System.out.println("Pierwsza liczba: " + pierwsza +
        ", druga liczba: " + druga);
  }

  public static void wypisz(double liczba) {
    System.out.println("Liczba rzeczywista: " + liczba);
  }

  public static void wypisz(String tekst) {
    System.out.println("Tekst: " + tekst);
  }

  public static void wypisz(int[] tablica) {
    for (int i = 0; i < tablica.length; i++) {
      System.out.println("Element tablica nr " + i +
          " to: " + tablica[i]);
    }
  }
}
```

W powyższym programie zdefiniowaliśmy wiele wersji metody `wypisz`. Różnią się one liczbą i/lub typami argumentów, więc kompilator nie ma problemów z dopasowaniem wywołania odpowiedniej metody `wypisz` w metodzie `main`, więc na ekranie zobaczymy:

```
Liczba calkowita: 10
Liczba rzeczywista: 3.14
Pierwsza liczba: 7, druga liczba: 128
Tekst: Witaj!
Element tablica nr 0 to: 2
Element tablica nr 1 to: 40
Element tablica nr 2 to: 500
```

Kolejność argumentów także ma znaczenie – metoda, która przyjmuje liczbę i `String` to inna metoda, niż metoda przyjmująca `String` i liczbę:

Nazwa pliku: PrzeladowanieKolejnoscArgumentow.java

```
public class PrzeladowanieKolejnoscArgumentow {
  public static void main(String[] args) {
    wypisz("Liczba wynosi: ", 8);
    wypisz(32, "2 do potegi 5 wynosi");
  }

  public static void wypisz(String komunikat, int liczba) {
    System.out.println(komunikat + liczba);
  }

  public static void wypisz(int liczba, String komunikat) {
    System.out.println(komunikat);
    System.out.println(liczba);
  }
}
```

Powyższy kod wykonuje się bez problemów – na ekranie zobaczymy:

```
Liczba wynosi: 8
2 do potegi 5 wynosi
32
```

Kompilator na podstawie liczby argumentów, ich typów, oraz kolejności, jest w stanie jednoznacznie określić, która z dwóch metod `wypisz` ma zostać użyta – najpierw wywołana zostanie metoda `wypisz`, która przyjmuje liczbę jako drugi argument, bo pierwsze wywołanie w metodzie `main`:

```
wypisz("Liczba wynosi: ", 8);
```

ma liczbę właśnie jako drugi argument. Natomiast drugie wywołanie metody `wypisz`:

```
wypisz(32, "2 do potegi 5 wynosi");
```

podaję liczbę jako pierwszy argument, więc użyta zostanie metoda `wypisz`, która oczekuje liczby jako pierwszy argument.

7.8.1 Typ zwracany przez metodę a przeładowywanie metod

Może się tutaj jeszcze nasunąć pytanie: a co ze zwracanym typem? Czy możemy mieć dwie metody o takich samych parametrach, które będą różniły się zwracanym typem?

Nie możemy – zwracany typ nie ma znaczenia – jeżeli spróbowalibyśmy utworzyć takie metody, kompilator ponownie by zaprotestował. Dla kompilatora informacja o zwracanym typie jest nieistotna przy podejmowania decyzji, którą metodę wywołać – przecież możemy całkowicie pominąć zwracaną wartość! Kompilator w takim przypadku nie wiedziałby, którą wersję metody ma użyć, a kompilacja zakończyłaby się błędem:

Nazwa pliku: PrzeladowanieZwracanyTyp.java

```
public class PrzeladowanieZwracanyTyp {
  public static void main(String[] args) {
    int x = podziel(100, 9);
  }

  public static int podziel(int a, int b) {
    return a / b;
  }

  public static double podziel(int a, int b) {
    return a / b;
  }
}
```

Ponownie zobaczymy błąd kompilacji informujący o konflikcie nazw:

```
PrzeladowanieZwracanyTyp.java:10: error: method podziel(int,int) is already
defined in class PrzeladowanieZwracanyTyp
  public static double podziel(int a, int b) {
                       ^
1 error
```

Pomimo, że metody różnią się zwracanym typem, a podczas wywoływania jednej z nich wynik przypisujemy do zmiennej typu `int` (a wartość tego typu zwraca tylko jedna z dwóch metod `podziel`), kompilator protestuje, ponieważ różnica tylko w zwracanym typie nie jest dla kompilatora wystarczająca.

7.8.2 Nazwy parametrów a przeładowywanie metod

A czy nazwy parametrów mają znaczenie? Jak już wspomnieliśmy na początku tego rozdziału, nazwy argumentów (tak jak zwracany typ) nie mają znaczenia dla kompilatora i nie są wystarczające, by móc poprawnie przeładować metodę.

7.8.3 Podsumowanie przeładowywania metod

- W języku Java mamy możliwość skorzystania z mechanizmu nazywanego **przeładowywaniem metod** (*method overloading*).
- Przeładowywanie metod pozwala na tworzenie metod o tej samej nazwie, o ile:
  - różnią się one liczbą argumentów i / lub
  - argumenty różnią się typem.
- Oznacza to, że możemy mieć wiele wersji tej samej metody, dostosowanych do różnych wymagań:

```
public static void wypisz(int liczba) {
  System.out.println("Liczba calkowita: " + liczba);
}

public static void wypisz(int pierwsza, int druga) {
  System.out.println("Pierwsza liczba: " + pierwsza +
      ", druga liczba: " + druga);
}

public static void wypisz(String tekst) {
  System.out.println("Tekst: " + tekst);
}
```

- Kolejność argumentów także ma znaczenie – metoda, która przyjmuje liczbę i string to inna metoda, niż metoda przyjmująca string i liczbę.
- Z kolei zwracany typ nie ma znaczenia – jeżeli spróbowalibyśmy utworzyć metody o takich samych argumentach, ale innych zwracanych typach, to kompilator by zaprotestował – kompilacja zakończyłaby się błędem.
- Dla kompilatora informacja o zwracanym typie jest nieistotna przy podejmowania decyzji, którą metodę wywołać.
- Nazwy parametrów, podobnie jak zwracany typ, nie mają znaczenia i są niewystarczające, by rozróżnić metody o tych samych nazwach i takich samych argumentach.

7.8.4 Pytania do przeładowywania metod

1. Czym jest przeładowywanie metod?
2. Które z poniższych par sygnatur przeładowanych metod są poprawne, a które nie? Wyjaśnij, dlaczego.

   a)

```
public static void metoda(int liczba)
public static int metoda(int liczba)
```

   b)

```
public static void metoda(int liczba, int drugaLiczba)
public static void metoda(int liczba)
```

   c)

```
public static void metoda(int liczba)
public static void metoda(int liczba)
```

   d)

```
public static void metoda(double liczba)
public static void metoda(int liczba)
```

   e)

```
public static void metoda(double liczba, String tekst)
public static void metoda(String tekst, double liczba)
```

   f)

```
public static void metoda(String tekst, double liczba)
public static void metoda(String komunikat, double wartosc)
```

   g)

```
public static void metoda()
public static void metoda(String komunikat)
```

   h)

```
public static void metoda(String komunikat)
public static void Metoda(String komunikat)
```

3. Które z poniższych ma znaczenie podczas przeładowywania metod i pozwoli na utworzenie metod o tej samej nazwie?
   a) typ zwracanej przez metodę wartości,
   b) liczba argumentów,
   c) typy argumentów,
   d) kolejność argumentów,
   e) nazwy argumentów.

7.8.5 Zadania do przeładowywania metod

7.8.5.1 Metoda porównująca swoje argumenty

Napisz metodę, która porównuje dwa przesłane do niej argumenty tego samego typu. Jeżeli wartości tych argumentów są sobie równe, metoda powinna zwrócić wartość `true`, a w przeciwnym razie `false`.

Metoda powinna mieć kilka wersji i przyjmować argumenty typów:

- `int`
- `double`
- `boolean`
- `char`
- `String`
- tablice wartości typu int: `int[]`
- tablice wartości typu String: `String[]`

8 Testowanie kodu

W tym rozdziale:

- nauczymy się testować nasz kod,
- zobaczymy, jak pisać metody, by były testowalne,
- opowiemy sobie o istotności testowaniu kodu,
- dowiemy się, czym jest Test Driven Development.

8.1 Wstęp do testowania

Pisząc kod naszych programów, powinniśmy upewnić się, że działa on prawidłowo – w tym celu przeprowadzamy testy.

Testy najmniejszych jednostek naszych programów, czyli metod, nazywane są *testami jednostkowymi*. Mają one za zadanie sprawdzić, że dana metoda działa tak, jak tego oczekiwaliśmy.

Testy jednostkowe także są metodami – są to metody, które wywołują testowaną przez nie metodę z różnymi parametrami i sprawdzają jej wynik. Jeżeli nie jest zgodny z oczekiwaniami, informują o błędzie.

Pisanie testów jednostkowych jest bardzo ważną umiejętnością. Pisanie testów nie jest proste – kod, który będziemy testować, musi być *testowalny*, tzn. musi być napisany z myślą o tym, że będzie testowany. Może się to wydawać dziwne, ale już teraz wiemy, po siedmiu rozdziałach tego kursu, że to samo w programowaniu prawie zawsze można zapisać na kilka sposobów.

W związku z tym, że pisanie testów jest umiejętnością, która wymaga praktyki, warto od teraz pisać testy jednostkowe do naszych programów. Aby sprawnie pisać testy jednostkowe, musimy zacząć myśleć o tworzeniu kodu pod kątem tego, by był testowalny. Wymaga to innego podejścia do tworzenia kodu, ponieważ zawsze mamy na względzie dostarczenie takiego rozwiązania, które będzie można (łatwo) przetestować.

W tym rozdziale będziemy pisali testy w uproszczony sposób – w jednym z kolejnych rozdziałów o klasach dowiemy się, jak wykorzystać bardzo popularną bibliotekę JUnit do wsparcia tworzenia i uruchamiania testów jednostkowych.

8.2 Pierwsze testy

Jak już wcześniej wspomniano, testy to metody, które wywołują testowaną metodę. Spróbujmy w takim razie napisać nasze pierwsze testy jednostkowe – obiektem naszych testów będzie metoda, która podnosi przesłaną do niej liczbę całkowitą do kwadratu i zwraca wynik. Zacznijmy od napisania tej metody:

Nazwa pliku: TestowanieDoKwadratu.java

```
public class TestowanieDoKwadratu {
  public static int doKwadratu(int x) {
    return x * x;
  }
}
```

Nasz program jest bardzo prosty – nie ma jeszcze nawet metody `main`, ale ma za to metodę `doKwadratu`, która przyjmuje argument typu `int` i zwraca wartość typu `int` – w tym przypadku, jest to wartość argumentu `x` podniesiona do kwadratu.

Jak możemy przetestować, że nasza metoda działa poprawnie? Moglibyśmy dopisać metodę `main` i w niej wywołać metodę `doKwadratu` i wypisać jej wynik na ekran:

Nazwa pliku: TestowanieDoKwadratu.java

```
public class TestowanieDoKwadratu {
  public static void main(String[] args) {
    System.out.println(doKwadratu(20));
  }

  public static int doKwadratu(int x) {
    return x * x;
  }
}
```

Program uruchamia się i wypisuje poprawną wartość na ekranie: `400`. Metoda `main` testuje naszą metodę, jednak nie jest to rozwiązanie docelowe, ponieważ możemy je ulepszyć na trzy sposoby:

1. Po pierwsze, chcielibyśmy umieszczać testy z różnymi danymi wejściowymi w osobnych metodach, a nie w metodzie `main`.
2. Po drugie, lepiej byłoby, gdyby testy informowały nas tylko o przypadkach błędnych – gdy wszystko jest w porządku i testowana metoda zadziałała poprawnie, nie chcemy widzieć nic na ekranie. Interesują nas przypadki, gdy coś poszło nie tak.
3. Po trzecie, chcielibyśmy mieć więcej przypadków testowych, w najlepszym przypadku pokrywających wszystkie możliwości, a na pewno przypadki szczególne.

Dostosujmy nasz program krok po kroku zgodnie z powyższymi wytycznymi.

8.2.1 Testy w osobnych metodach

Przenieśmy test do osobnej metody:

Nazwa pliku: TestowanieDoKwadratu.java

```
public class TestowanieDoKwadratu {
  public static void main(String[] args) {
    doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu(); // (1)
  }

  public static int doKwadratu(int x) {
    return x * x;
  }

  // (2)
  public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu() {
    System.out.println(doKwadratu(20));
  }
}
```

Przenieśliśmy test do osobnej metody o bardzo długiej nazwie `(2)`. Istnieją różne konwencje, które definiują, jak powinny być nazywane metody testowe – jedną z nich jest konwencja, wedle której metody powinny być nazywane w następujący sposób:

`nazwaTestowanejMetody_daneWejściowe_spodziewanyWynik`

- Pierwszy człon nazwy metody testowej to nazwa metody, którą testujemy – w naszym programie testujemy metodę o nazwie `doKwadratu`, więc metoda, która ją testuje, zaczyna się właśnie od "doKwadratu".
- W środku nazwy metody testowej umieszczamy krótki opis danych wejściowych – nasza metoda testowa testuje podnoszenie do kwadratu liczby dodatniej – stąd drugi człon nazwy to "wartoscDodatnia".
- Na końcu umieszczamy krótki opis wyniku, jakiego się spodziewamy, że testowana metoda zwróci dla danych wejściowych – w naszym przypadku oczekujemy, że metoda `doKwadratu` dla wartości dodatniej zwróci "wartoscPodniesionaDoKwadratu".

Dodatkowo, skoro przenieśliśmy test metody `doKwadratu` do osobnej metody, to musimy teraz w jakiś sposób wywołać test tej metody. Test ten znajduje się teraz w nowej metodzie o nazwie `doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu`, więc musimy wywołać tą metodę w metodzie `main` `(1)`, aby przeprowadzić test metody `doKwadratu`.

W jednych z kolejnych rozdziałów tego kursu nauczymy się, jak wydzielać testy do osobnych plików i jak uruchamiać je w prosty sposób przy użyciu biblioteki JUnit.

Zwróćmy uwagę, że metoda testowa nic nie zwraca – jej zwracany typ to **`void`**. Metody testowe nie powinny zwracać wartości, ponieważ nie jest to do niczego wymagane.

8.2.2 Informowanie tylko o błędnym działaniu

Test jest już w osobnej metodzie – teraz zajmiemy się drugim opisanym powyżej punktem: chcielibyśmy, aby nasz test komunikował tylko informacje o przypadkach testowych, które nie przeszły testu – jeżeli wszystko poszło dobrze, to test powinien po prostu "siedzieć cicho".

Jak możemy to osiągnąć? Wystarczy, że sprawdzimy wynik metody – jeżeli będzie inny, niż się spodziewamy, wtedy wypiszemy na ekran komunikat informujący, że testowana przez nas metoda zwróciła dla danego argumentu inny wynik, niż oczekiwaliśmy:

Nazwa pliku: TestowanieDoKwadratu.java

```
public class TestowanieDoKwadratu {
  public static void main(String[] args) {
    doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu();
  }

  public static int doKwadratu(int x) {
    return x * x;
  }

  public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu() {
    int wynik = doKwadratu(20); // (1)

    if (wynik != 400) { // (2)
      // (3)
      System.out.println(
          "Dla liczby 20 wyliczono nieprawidlowy kwadrat: " + wynik
      );
    }
  }
}
```

Nasz program nie wypisuje już po prostu wartości podniesionej do kwadratu. Zamiast tego:

1. Najpierw wyliczamy kwadrat liczby `20` i zapisujemy wynik w zmiennej `wynik` *(1)*.
2. Następnie, w instrukcji warunkowej sprawdzamy otrzymany wynik – jeżeli jest inny, niż się spodziewamy *(2)*, wypiszemy na ekran informację.
3. Jeżeli metoda `doKwadratu` niepoprawnie wyliczy kwadrat liczby `20`, to wypisujemy na ekran komunikat informujący, że testowana metoda zadziałała niepoprawnie *(3)*, bo źle wykonała swoje zadanie (zwróciła błędny wynik). Świadczy to o tym, że test nie zakończył się sukcesem.

Jeżeli metoda `doKwadratu` zadziała poprawnie, to test nic nie wypisze. Dzięki temu będziemy informowani tylko w tych przypadkach, gdy coś w naszym kodzie nie działa bądź przestało działać.

8.2.3 Więcej przypadków testowych

Jeden test to zazwyczaj za mało, by przetestować wszystkie przypadki. Warto zastanowić się zawsze nad przypadkami szczególnymi, np.:

- przypadkami, które wydaje nam się, że nigdy nie wystąpią,
- przypadkami skrajnych danych wejściowych,
- przypadkami z nieprawidłowymi danymi wejściowymi.

Poza tym testujemy też oczywiście metody używając "spodziewanych" argumentów.

Ważne jest także, aby nie duplikować przypadków testowych – jeżeli testujemy metodę `doKwadratu` i napisaliśmy test, który sprawdza, czy metoda ta dla argumentu `5` kończy się sukcesem (tzn. metoda `doKwadratu` działa poprawnie), to nie ma potrzeby pisania testu, który sprawdzi wynik metody `doKwadratu` dla liczby `10` – test z liczbą `5` *pokrył* już podobny przypadek.

Dopiszmy jeszcze dwa testy do metody `doKwadratu`:

- sprawdzimy, jak metoda zachowa się dla argumentu, którym będzie liczbą ujemna,
- sprawdzimy, co stanie się dla argumentu równego zero.

Nazwa pliku: TestowanieDoKwadratu.java

```
public class TestowanieDoKwadratu {
  public static void main(String[] args) {
    doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu();
    doKwadratu_wartoscUjemna_wartoscPodniesionaDoKwadratu(); // (1)
    doKwadratu_liczbaZero_zero(); // (2)
  }

  public static int doKwadratu(int x) {
    return x * x;
  }

  public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu() {
    int wynik = doKwadratu(20);

    if (wynik != 400) {
      System.out.println(
          "Dla liczby 20 wyliczono nieprawidlowy kwadrat: " + wynik
      );
    }
  }

  // (3)
  public static void doKwadratu_wartoscUjemna_wartoscPodniesionaDoKwadratu() {
    int wynik = doKwadratu(-5);

    if (wynik != 25) {
      System.out.println(
          "Dla liczby -5 wyliczono nieprawidlowy kwadrat: " + wynik
      );
    }
  }

  // (4)
  public static void doKwadratu_liczbaZero_zero() {
    int wynik = doKwadratu(0);

    if (wynik != 0) {
      System.out.println(
          "Dla liczby 0 wyliczono nieprawidlowy kwadrat: " + wynik
      );
    }
  }
}
```

W ostatecznej wersji naszego programu dodaliśmy dwa nowe testy `(3)` `(4)` oraz wywołaliśmy je w metodzie `main` `(1)` `(2)`. Po uruchomieniu nasz program nic nie wypisuje – świadczy to o tym, że nasza metoda `doKwadratu` działa poprawnie!

Czy możemy być pewni, że faktycznie tak jest? Czy nie zapomnieliśmy o jeszcze jakimś przypadku testowym? Do zastanowienia jako zadanie!

8.2.4 Duplikacja kodu

Wróćmy jeszcze na chwilę do finalnej wersji kodu z poprzedniego rozdziału – spójrzmy na kod naszych testów:

```
public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu() {
  int wynik = doKwadratu(20);

  if (wynik != 400) {
    System.out.println(
        "Dla liczby 20 wyliczono nieprawidlowy kwadrat: " + wynik
    );
  }
}

public static void doKwadratu_wartoscUjemna_wartoscPodniesionaDoKwadratu() {
  int wynik = doKwadratu(-5);

  if (wynik != 25) {
    System.out.println(
        "Dla liczby -5 wyliczono nieprawidlowy kwadrat: " + wynik
    );
  }
}

public static void doKwadratu_liczbaZero_zero() {
  int wynik = doKwadratu(0);

  if (wynik != 0) {
    System.out.println(
        "Dla liczby 0 wyliczono nieprawidlowy kwadrat: " + wynik
    );
  }
}
```

Widzimy, że każda metoda testowa napisana jest w podobny sposób – najpierw wywołujemy testowaną metodę, a potem sprawdzamy, czy wynik jest poprawny. Czy moglibyśmy w takim razie jakoś uprościć powyższe metody i pozbyć się duplikacji kodu?

Moglibyśmy napisać kolejną metode, której zadaniem będzie sprawdzenie, czy przesłane do niej argumenty są sobie równe. Jeżeli nie, metoda wypisze na ekran komunikat. Następnie moglibyśmy tej nowej metody użyć w naszych testach. Spójrzmy na kolejną wersję naszego programu:

Nazwa pliku: TestowanieDoKwadratu2.java

```
public class TestowanieDoKwadratu2 {
  public static void main(String[] args) {
    doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu();
    doKwadratu_wartoscUjemna_wartoscPodniesionaDoKwadratu();
    doKwadratu_liczbaZero_zero();
  }

  public static int doKwadratu(int x) {
    return x * x;
  }

  public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu() {
    int wynik = doKwadratu(20);
    assertEquals(400, wynik); // (1)
  }

  public static void doKwadratu_wartoscUjemna_wartoscPodniesionaDoKwadratu() {
    int wynik = doKwadratu(-5);
    assertEquals(25, wynik); // (2)
  }

  public static void doKwadratu_liczbaZero_zero() {
    int wynik = doKwadratu(0);
    assertEquals(0, wynik); // (3)
  }

  //                    (4)              (5)          (6)
  public static void assertEquals(int expected, int actual) {
    if (expected != actual) { // (7)
      System.out.println("Spodziewano sie liczby " + actual +
        ", ale otrzymano: " + expected);
    }
  }
}
```

Nowa metoda `assertEquals` *(4)* przyjmuje dwa parametry typu `int` o nazwach `expected` *(5)* oraz `actual` *(6)*. Jej jedynym zadaniem jest wypisanie komunikatu, gdy liczby przesłane jako argumenty nie są sobie równe *(7)*. Dzięki tej metodzie, mogliśmy znacząco skrócić nasze metody testowe – metody `assertEquals` używamy teraz do sprawdzenia wyniku działania metody `doKwadratu` *(1)* *(2)* *(3)*.

Nazwa metody (*assert – zapewnij*) oraz nazwy jej argumentów nie są przypadkowe. Wiele bibliotek wspierających testy jednostkowe udostępnia metodę o właśnie takiej nazwie i takich argumentach – wykonują one bardzo podobne zadanie jak nasza powyższa metoda `assertEquals` (chociaż komunikuje nierówność argumentów w inny sposób, który poznamy w jednym z kolejnych rozdziałów).

8.3 Given .. when .. then

Istnieje popularna konwencja definiująca, jak testy jednostkowe powinny być ustrukturyzowane. Zakłada ona, iż:

1. Na początku testu przygotowujemy dane, które posłużą jako dane wejściowe, na których testowana metoda będzie działać. Ta część testu nazywa się *given*.
2. Nastepnie, wywołujemy testowaną metodę na danych przygotowanych w punkcie pierwszym. Ta cześć testu to *when*.
3. Na końcu testu sprawdzamy, czy wynik zwrócony przez metodę wywołaną w punkcie drugim jest zgodne z naszymi oczekiwaniami. Ta ostatnia część testu to *then*.

Konstruując test w ten sposób dostajemy logiczną całość:

1. dla takich a takich danych (**given**),
2. gdy wykonam taką metodę (**when**),
3. powinienem otrzymać taki a taki rezultat (**then**).

Spójrzmy na przykład jednego z testów z poprzedniego programu napisanego w taki właśnie sposób. Dla porównania dodana została także poprzednia wersja tego testu:

```
public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu() {
  int wynik = doKwadratu(20);
  assertEquals(400, wynik);
}

public static void doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu2() {
  // given
  int liczba = 20;

  // when
  int wynik = doKwadratu(liczba);

  // then
  assertEquals(400, wynik);
}
```

Druga metoda stosuje opisaną powżej konwencję – komentarzami oddzieliliśmy od siebie poszczególne cześci testu. Ze względu na czytelność, testy są często pisane w ten właśnie sposób.

Nasz test jest prosty i w zasadzie nie ma potrzeby na tworzenie zmiennej `liczba` *tylko po to, by przypisać do niej wartość i użyć jej jako argument metody* `doKwadratu`*. Celem było ukazanie przykładu konwencji* given .. when .. then*. Mniej skomplikowane testy możemy zapisywać zwięźlej, nie korzystając z konwencji* given .. when .. then*. Wszystko zależy od tego, którą wersję z testów uważamy za czytelniejszą i łatwiejszą do zrozumienia przez innych programistów.*

8.4 Dlaczego testy jednostkowe są ważne?

Dlaczego testy jednostkowe są ważne? Wynika to z następujących faktów:

- testy jednostkowe pozwalają nam na wyizolowanie i przetestowanie najmniejszych części naszego programu, a co za tym idzie, upewnienie się, że dostarczamy działające rozwiązanie o wysokiej jakości,
- testy stoją na straży przed potencjalnym wprowadzeniem błędów (bugów) w kodzie w przyszłości – po modyfikacji kodu, np. związanej z dodaniej nowej funkcjonalności, uruchamiamy testy i sprawdzamy, czy w wyniku naszych zmian nie zepsuliśmy systemu,
- testy wymuszają styl tworzenia kodu, który jest czytelniejszy, łatwiejszy w utrzymaniu i zmianie,
- posiadanie zestawu testów pozwala na wprowadzanie zmian do kodu bez obawy, że coś zepsujemy – po wprowadzeniu zmiany możemy uruchomić testy i upewnić się, że testy przechodzą bez błędów,
- testy służą jako dokumentacja dla innych programistów – jeżeli otestujemy nasz kod, to można, na podstawie analizy kodu testów, dowiedzieć się, jak działa funkcjonalność dostarczana przez metody, które są przez te testy testowane.

Aby jednak móc pisać testy jednostkowe, musimy pisać *testowalny* kod.

8.5 Testowalny kod

Co zrobić, aby kod był testowalny i co to w ogóle znaczy?

Testowalny kod to taki, do którego można bez problemu przygotować zestaw dokładnych testów jednostkowych. Metody muszą spełniać trzy podstawowe zasady, aby były testowalne:

- muszą być zwięzłe, tzn. krótkie – im krótsze, tym lepsze (kilka – kilkanaście linii kodu),
- muszą robić jedną rzecz,
- nie mogą zależeć od użytkownika.

Dlaczego testowalny kod musi spełniać takie wymogi? Im mniej akcji wykonuje metoda, tym mniej testów jednostkowych będzie wymagała, i tym prostsze one będą. Jeżeli napiszemy metodę na 100 linii kodu, która będzie robiła kilkanaście rzeczy, trudno będzie napisać testy jednostkowe, które sprawdzą wszystkie możliwe ścieżki wykonania i przetestują wszystkie funkcjonalności.

Dodatkowo, metody, które wymagają od programisty wykonania jakiejś akcji, są niewygodne, ponieważ nie chcemy być zmuszeni do interakcji z naszymi testami, które będziemy często uruchamiać i których mogą być tysiące.

Powinniśmy dążyć do rozbicia dużych metod na małe, wydzielone jednostki, które otestujemy, a następnie, będąc upewnieni, że działają, będziemy z nich korzystać w kolejnych metodach.

Małe metody to maksymalnie kilka-kilkanaście linijek kodu.

Spójrzmy na poniższą metodę i zastanówmy się nad następującymi pytaniami:

1. Czy taką metodę łatwo przetestować?
2. Ile operacji wykonuje ta metoda?
3. Czy możemy ją usprawnić, aby można ją było przetestować w łatwy sposób?

Nazwa pliku: TestowanieCzyParzystaWersja1.java

```
import java.util.Scanner;

public class TestowanieCzyParzysta {
  public static void main(String[] args) {
    czyParzysta();
  }

  public static void czyParzysta() {
    System.out.println(
        "Prosze podac liczbe - sprawdze, czy jest parzysta."
    );

    int liczba = getInt();

    if (liczba % 2 == 0) {
      System.out.println("Ta liczba jest parzysta.");
    } else {
      System.out.println("Ta liczba jest nieparzysta");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

1. Bez interakcji użytkownika metoda ta jest nie do przetestowania. Pyta ona użytkownika o liczbę i czeka na jej podanie. W obecnej formie, aby ją przetestować, musielibyśmy uruchomić powyższy program wielokrotnie, wpisując w oknie konsoli różne liczby i sprawdzić wynik wypisany na ekranie.
2. Metoda ta wykonuje wiele operacji:
   1. Wypisuje komunikat z prośbą o podanie liczby.
   2. Pobiera od użytkownika liczbę przy użyciu metody `getInt`.
   3. Sprawdza parzystość liczby.
   4. Na podstawie parzystości liczby wypisuje komunikat.
3. Zastanówmy się, co moglibyśmy usprawnić w naszej metodzie?
   1. Najważniejsze to przenieść pobieranie liczby od użytkownika poza metodę `czyParzysta`. Dzięki temu nasza metoda nie będzie już uzależniona od użytkownika.
   2. Skoro nie będziemy pobierać liczby w metodzie `czyParzysta`, to musimy jakoś przesłać do tej metody liczbę do sprawdzenia – łatwo możemy to osiągnąć, gdy zmienimy metodę, by przyjmowała jeden argument typu **`int`**.
   3. Nasza metoda nie powinna wypisywać na ekran wyniku sprawdzenia parzystości – ktokolwiek będzie używał naszej metody powinien sam zadecydować, co chce z tą informacją (czy liczba jest parzysta czy nie) zrobić. Skoro tak, to nasza metoda powinna zwrócić informację o tym, czy liczba jest parzysta czy nie – osiągniemy to poprzez zmianę zwracanego przez metodę typu z **`void`** na **`boolean`** (prawda / fałsz).

Spójrzmy na usprawnioną wersję powyższego program:

Nazwa pliku: TestowanieCzyParzystaWersja2.java

```
import java.util.Scanner;

public class TestowanieCzyParzystaWersja2 {
  public static void main(String[] args) {
    // (1)
    System.out.println(
        "Prosze podac liczbe - sprawdze, czy jest parzysta."
    );

    int liczba = getInt(); // (2)

    if (czyParzysta(liczba)) {
      System.out.println("Ta liczba jest parzysta."); // (3)
    } else {
      System.out.println("Ta liczba jest nieparzysta"); // (4)
    }
  }

  //        (5)          (6)
  public static boolean czyParzysta(int x) {
    return x % 2 == 0; // (7)
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Wykonane zmiany:

1. Przeniesienie wypisania prośby o podanie liczby *(1)*.
2. Przeniesienie pobierania liczby od użytkownika *(2)*.
3. Przeniesienie wypisywania komunikatu zależnego od parzystości liczby *(3)* *(4)*.
4. Zmiana typu zwracanego przez metodę `czyParzysta` z **`void`** na **`boolean`** *(5)*.
5. Dodanie argumentu do metody `czyParzysta`, który ma być sprawdzony pod kątem parzystości *(6)*.
6. Zwrócenie informacji, czy przesłana w argumencie liczba jest parzysta, czy nie *(7)*.

Dzięki powyższym zmianom, metoda `czyParzysta` bardzo się uprościła. Wykonuje ona teraz jedno konkretne zadanie, którym jest sprawdzenie parzystości podanego do niej argumentu i zwrócenie wyniku. Taką metodę łatwo przetestować – wystarczy napisać testy, które wywołają metodę `czyParzysta` z różnymi wartościami i sprawdzą wynik. Takie testy będą: szybkie, proste, i nie będą wymagały w trakcie wykonywania żadnych akcji od programisty.

8.6 Co charakteryzuje dobre testy jednostkowe?

Dobre testy jednostkowe charakteryzują się następującymi cechami:

- sprawdzają wszystkie możliwe ścieżki wykonania metody – by mieć pewność, że metoda będzie działała w różnych warunkach,
- są krótkie – a dzięki temu łatwe w zrozumieniu i utrzymaniu,
- są czytelne i schludnie napisane – wymóg, który powinien spełniać każdy kod,
- pozwalają na zrozumienie testowanego kodu – dzięki temu będą dodatkowo służyć jako dokumentacja tego, co testują,
- są szybkie – abyśmy nie musieli czekać kilka minut na to, aż się wykonają,
- jeden zestaw danych testowych na jedną metodę testową – zazwyczaj jeden przypadek testowy testowany jest w jednej metodzie testującej – zamiast wykonywać wszystkie testy w jednej metodzie, chcemy odseparować od siebie różne przypadki testowe.

Można zadać teraz dwa pytania:

- a co z testami testów?
- co dzieje się w przypadku, jeżeli musimy zmienić kod, przez co testy przestaną działać?

W pierwszym przypadku odpowiedź jest bardzo prosta:

Testy testują kod produkcyjny – kod produkcyjny testuje testy.

Oznacza to, że jeżeli źle napiszemy testy, to nie będą one działać. Istnieje tutaj zależność działania kodu testów od kodu produkcyjnego i poprawności kodu produkcyjnego od działania testów.

A jeżeli kod produkcyjny się zmienia i testy przestają działać? Wtedy zmieniamy testy by odzwierciedlały zmiany w kodzie produkcyjnym, dodajemy nowe, usuwamy stare, jeżeli są już niepotrzebne. Należy pamiętać, że o testy należy dbać tak samo, jak o kod produkcyjny, jeżeli nie bardziej – w końcu to one stoją na straży naszego kodu.

8.7 Przykłady testów jednostkowych

Poniżej przedstawionych zostało kilka przykładów testów jednostkowych różnych metod.

8.7.1 Wartość bezwzględna

Spójrzmy na prosty przykład testów metody, której zadaniem jest zwrócenie wartości bezwględnej przesłanej liczby.

> *Wartość bezwględna to wartość liczby bez uwzględnienia jej znaku.*
> *Dla liczb parzystych jest to ta sama liczba. Przykład: wartość bezwzględna liczby* `10` *to* `10`*.*
> *Dla liczb ujemnych jest to liczba bez minusa. Przykład: wartość bezwględna liczby* `-5` *to* `5`*.*

Zanim jednak przejdziemy do kodu źródłowego, zastanówmy się jak przetestowalibyśmy taką metodę? Jakie *przypadki testowe* wzięlibyśmy pod uwagę?

Nazwa pliku: TestowanieWartoscBezwzgledna.java

```
public class TestowanieWartoscBezwzgledna {
  public static void main(String[] args) {
    wartoscBezwgledna_liczbaDodatnia_zwrociDodatniaWartosc(); // (1)
    wartoscBezwgledna_liczbaUjemna_zwrociDodatniaWartosc();   // (2)
  }

  public static int wartoscBezwgledna(int x) { // (3)
    return x < 0 ? -x : x;
  }

  public static void
          wartoscBezwgledna_liczbaDodatnia_zwrociDodatniaWartosc() { // (4)

    int rezultat = wartoscBezwgledna(20); // (5)

    if (rezultat != 20) { // (6)
      System.out.println( // (7)
        "Dla wartosci 20 otrzymano nieprawidlowa wartosc: " + rezultat
      );
    }
  }

  public static void
          wartoscBezwgledna_liczbaUjemna_zwrociDodatniaWartosc() { // (8)

    int rezultat = wartoscBezwgledna(-1); // (9)

    if (rezultat != 1) { // (10)
      System.out.println( // (11)
        "Dla wartosci -1 otrzymano nieprawidlowa wartosc: " + rezultat
      );
    }
  }
}
```

Przejdźmy krok po kroku przez powyższy kod:

- Metoda, którą chcemy przetestować, to `wartoscBezwzgledna` *(3)*. Jest to prosta metoda wykonnująco jedno zadanie – dla podanej jako argument liczby zwraca wartość bezwzględną tej liczby.

- Metoda `wartoscBezwzgledna` testowana jest przez dwie metody *(4)* *(8)*.
- Dane testowe pierwszej metody to liczba dodatnia *(5)*, a drugiej metody – liczba ujemna *(9)*.
- Obie metody testowe sprawdzają, czy wynik jest nieprawidłowy *(6)* *(10)* – jeśli tak, to wypisują informację na ekran *(7)* *(11)*.
- Metody testowe wywołujemy w metodzie `main` *(1)* *(2)*.

8.7.2 Metoda sprawdzająca, czy tablica zawiera element

Chcielibyśmy napisać metodę, która będzie przyjmowała jako argument tablicę liczb i odpowiadała na pytanie, czy w tej tablicy znajduje się dana liczba.

Jakie przypadki testowe powinniśmy przygotować?

Powinniśmy na pewno sprawdzić następujące przypadki:

1. *Co sie stanie, jeżeli tablica jest pusta?* – zawsze warto sprawdzić zachowanie na "pustych" danych.
2. *Co się stanie, jeżeli niepusta tablica nie zawiera szukanego elementu?* – podczas wyszukiwania elementów warto sprawdzić co dzieje się, gdy element nie istnieje.
3. *Co się stanie, jeżeli niepusta tablica zawiera szukany element?* – "normalny" przypadek.
4. *Co się stanie, jeżeli niepusta tablica zawiera szukany element wyłącznie na samym początku tablicy?* – w przypadku operacji na tablicach warto brać pod uwagę przypadki testowe sprawdzające zachowanie biorące pod uwagę pierwszy element tablicy.
5. *Co się stanie, jeżeli niepusta tablica zawiera szukany element wyłącznie na samym końcu tablicy?* – jak wyżej, ale tym razem bierzemy pod uwagę element końcowy.

Poniższy kod źródłowy przedstawia, jak moglibyśmy napisać powyższy program wraz z testami:

Nazwa pliku: TestowanieCzyElementWTablicy.java

```
public class TestowanieCzyElementWTablicy {
  public static void main(String[] args) {
    czyZawieraElement_pustaTablica_zwrociFalse();
    czyZawieraElement_brakSzukanegoElementu_zwrociFalse();
    czyZawieraElement_zawieraSukanyElement_zwrociTrue();
    czyZawieraElement_zawieraSukanyElementNaPoczatku_zwrociTrue();
    czyZawieraElement_zawieraSukanyElementNaKoncu_zwrociTrue();
  }

  public static boolean czyZawieraElement(int[] tablica, int liczba) {
    for (int i = 0; i < tablica.length; i++) {
      if (tablica[i] == liczba) {
        return true;
      }
    }

    return false;
  }

  public static void czyZawieraElement_pustaTablica_zwrociFalse() {
    // given
    int[] pustaTablica = {};
    int liczba = 5;
```

```java
    // when
    boolean czyZawiera = czyZawieraElement(pustaTablica, liczba);

    // then
    if (czyZawiera) {
      System.out.println("Blad! Pusta tablica nie powinna nic zawierac.");
    }
  }

  public static void czyZawieraElement_brakSzukanegoElementu_zwrociFalse() {
    // given
    int[] tablica = {-20, 100, 500};
    int liczba = 128;

    // when
    boolean czyZawiera = czyZawieraElement(tablica, liczba);

    // then
    if (czyZawiera) {
      System.out.println("Blad! Element 128 nie powinien byc znaleziony.");
    }
  }

  public static void czyZawieraElement_zawieraSukanyElement_zwrociTrue() {
    // given
    int[] tablica = {2, 4, 8, 16, 32, 64, 128, 256};
    int liczba = 128;

    // when
    boolean czyZawiera = czyZawieraElement(tablica, liczba);

    // then
    if (!czyZawiera) {
      System.out.println("Blad! Element 128 powinien byc znaleziony.");
    }
  }

  public static void czyZawieraElement_zawieraSukanyElementNaPoczatku_zwrociTrue() {
    // given
    int[] tablica = {100, 200, 300};
    int liczba = 100;

    // when
    boolean czyZawiera = czyZawieraElement(tablica, liczba);

    // then
    if (!czyZawiera) {
      System.out.println("Blad! Element 100 powinien byc znaleziony.");
    }
  }

  public static void czyZawieraElement_zawieraSukanyElementNaKoncu_zwrociTrue() {
    // given
    int[] tablica = {100, 200, 300};
    int liczba = 300;

    // when
    boolean czyZawiera = czyZawieraElement(tablica, liczba);
```

```
    // then
    if (!czyZawiera) {
      System.out.println("Blad! Element 300 powinien byc znaleziony.");
    }
  }
}
```

Testy zapisaliśmy zgodnie z notacją *given .. when .. then* – najpierw przygotowujemy dane, następnie wywołujemy testowaną metodą i zapisujemy jej wynik, a na końcu sprawdzamy wynik.

Powyższe testy są proste – mogłyby również być zapisane w bardziej zwięzły sposób, na przykład:

```
public static void czyZawieraElement_zawieraSukanyElementNaKoncu_zwrociTrue() {
  if (!czyZawieraElement(new int[] {100, 200, 300}, 300)) {
    System.out.println("Blad! Element 300 powinien byc znaleziony.");
  }
}
```

Powyższy przykład pomija sekcje "*given*", "*when*" oraz "*then*" i znacząco skraca test – jest on na tyle prosty, że taka postać mogłaby być preferowana, ale w ramach ćwiczenia i przykładu użyta została konwencja *given .. when .. then*.

Pamiętajmy! Testy powinny być czytelne, ale krótsze testy to mniej kodu do zrozumienia i utrzymania. Należy zawsze zastanowić się czy napisany przez nas kod testu zyskałby na rozbiciu go na sekcje *given .. when .. then*, czy może lepiej byłoby napisać krótki i prosty test bez używania tej konwencji.

8.8 TDD – Test Driven Development

Test Driven Development to metodologia tworzenia oprogramowania, w której centralnym punktem są testy.

Testy kodu produkcyjnego piszemy... przed napisaniem kodu produkcyjnego! Najpierw zastanawiamy się, co nasza metoda będzie robić. Następnie piszemy test, który ma to sprawdzić. Uruchamiamy go i widzimy, że zakończył się błędem – nic dziwnego! Testowana przez niego metoda jeszcze nie istnieje. Teraz przystępujemy do pisania testowanej metody tak, aby nasz test przeszedł. Jeżeli już nam się to uda, piszemy kolejny test, testujący kolejny aspekt bądź używający innego zestawu danych, i tak w kółko, aż kod produkcyjny uznamy za gotowy.

Więcej informacji o TDD można znaleźć w internecie.

8.9 Podsumowanie

- Testy mają na celu sprawdzenie, czy nasz kod działa zgodnie z założeniami.
- Testy jednostkowe testują najmniejsze jednostki naszych programów – metody.
- Testy jednostkowe to także metody – są to metody, które uruchamiają metodę, którą testują, z różnymi parametrami i sprawdzają, czy wynik działania testowanej metody jest taki, jak zakładaliśmy.
- Musimy tworzyć nasze metody w taki sposób, by były one testowalne. Oznacza to, że nasze metody:
  - powinny być krótkie (kilka do kilkunastu linii kodu) – im mniej metoda robi, tym łatwiej ją przetestować, ponieważ potrzeba mniejszej liczby przypadków testowych,
  - powinny robić jedną, ustaloną rzecz – wtedy przypadki testowe będą łatwiejsze do przygotowania,
  - nie powinny oczekiwać na żadne akcje użytkownika – ponieważ wtedy wykonanie testów będzie za każdym razem oczekiwało działań od użytkownika.
- Testy jednostkowe są ważne, ponieważ:
  - pozwalają nam na wyizolowanie i przetestowanie najmniejszych części naszego programu,
  - wymuszają styl tworzenia kodu, który jest czytelniejszy i łatwiejszy w utrzymaniu (ponieważ metody są krótsze, a co za tym idzie, łatwiej zrozumieć, co się w nich dzieje),
  - pomagają nam dostarczać działający kod,
  - posiadanie zestawu testów pozwala na wprowadzanie zmian do kodu bez obawy, że coś zepsujemy – po wprowadzeniu zmiany możemy uruchomić testy i upewnić się, że przechodzą one bez błędów
- Testy testują kod produkcyjny – kod produkcyjny testuje testy.
- O testy należy dbać niemniej, niż o kod produkcyjny.
- Dobre testy jednostkowe:
  - sprawdzają wszystkie możliwe ścieżki wykonania metody,
  - są krótkie,
  - są czytelne i schludnie napisane,
  - są szybkie.
- Różne przypadki testowe powinny być obsługiwane w osobnych metodach testujących.
- Testy powinny informować jedynie o błędnych przypadkach, tzn. takich, gdzie zakładana wartość była inna, niż faktyczny rezultat wykonania testowanej metody.
- Powinniśmy przygotowywać wiele przypadków testowych, by mieć pewność, że nasza metoda będzie działała zgodnie z oczekiwaniami niezależnie od danych wejściowych.
- Nie powinniśmy duplikować przypadków testowych – każdy test powinien testować pewien określony przypadek. Dla przykładu – nie ma sensu testować, czy metoda podnosząca liczbę do kwadratu zwróci poprawną wartość dla liczb `1`, `2`, `3` itd. – wystarczy test dla liczby np. `5`.

- Istnieje wiele konwencji nazewczniczych testów – jedna z nich zakłada, że najpierw podajemy nazwę testowanej metody, nastepnie dane wejściowe, a na końcu opisujemy spodziewany wynik:

  `nazwaTestowanejMetody_daneWejściowe_spodziewanyWynik`

  na przykład:

  `doKwadratu_wartoscDodatnia_wartoscPodniesionaDoKwadratu`

- Typem zwracanym metod testowych powinien być **`void`**.
- Podczas pisania testów często używa się tzw. *asercji*. Są to metody, które sprawdzają pewien warunek, na przykład czy dwie liczby są sobie równe. Jeżeli nie, informują o błędzie, a w przeciwnym razie, gdy warunek jest spełniony, nie wykonują żadnych akcji. Przykład:

```
public static void assertEquals(int expected, int actual) {
  if (expected != actual) {
    System.out.println("Spodziewano sie liczby " + actual +
      ", ale otrzymano: " + expected);
  }
}
```

- **Given .. when .. then** to konwencja definiująca, jak testy jednostkowe powinny być ustrukturyzowane. Najpierw przygotowujemy dane wejściowe dla testu (**given**), nastepnie wywołujemy testowaną metodę z użyciem przygotowanych danych (**when**), a na końcu sprawdzamy wynik (**then**):
  - dla takich a takich danych (**given**),
  - gdy wykonam taką metodę (**when**),
  - powinienem otrzymać taki a taki rezultat (**then**).

  Przykład testu o strukturze *given .. when .. then*:

```
public static void czyZawieraElement_brakSzukanegoElementu_zwrociFalse() {
  // given
  int[] tablica = {-20, 100, 500};
  int liczba = 128;

  // when
  boolean czyZawiera = czyZawieraElement(tablica, liczba);

  // then
  if (czyZawiera) {
    System.out.println("Blad! Element 128 nie powinien byc znaleziony.");
  }
}
```

- TDD, czyli *Test Driven Development*, to metodologia tworzenia oprogramowania, w której najpierw zaczynamy tworzenie kodu od napisania testu, a dopiero potem piszemy kod, które ten test ma spełnić. Działamy w ten sposób aż do momentu, aż nie uznamy kodu produkcyjnego za gotowy.

8.10 Pytania

1. Spójrz na poniższe metody – czy i jak można by je poprawić pod kątem testowania?

   a)

```
public static int szescian() {
  System.out.print("Podaj liczbe: ");
  int liczba = getInt();

  return liczba * liczba * liczba;
}

public static int getInt() {
  return new Scanner(System.in).nextInt();
}
```

   b)

```
public static void policzSilnie(int x) {
  int wynik = 1;

  for (int i = 1; i <= x; i++) {
    wynik = wynik * i;
  }

  System.out.println("Policzona silnia wynosi: " + wynik);
}
```

2. Czy poniższa metoda działa poprawnie? Jakie przypadki testowe powinniśmy do niej przygotować?

```
public static int policzZnaki(String tekst, char znak) {
  int liczbaZnakow = 1;

  for (int i = 0; i <= tekst.length(); i++) {
    if (tekst.charAt(i) == znak) {
      liczbaZnakow++;
    }
  }

  return liczbaZnakow;
}
```

3. Jakie testy jednostkowe należałoby napisać do metody, która sortuje przesłaną do niej tablicę rosnąco?

8.11 Zadania

Pisząc testy w poniższych zadaniach, pamiętaj o:

- wzięciu pod uwagę różnych przypadków testowych i rozdzieleniu ich na osobne metody testujące,
- odpowiednim nazewnictwie metod testujących,
- ustrukturyzowaniu metod testujących w taki sposób, by były czytelne i jasno przekazywały, na jakim przypadku testowym działają,
- napisaniu metody, którą będziesz testował, w taki sposób, by była testowalna.

8.11.1 Testy czyParzysta

Napisz testy oraz metodę, która odpowiada na pytanie, czy podana liczba jest parzysta.

8.11.2 Testy sprawdzania znaku liczby

Napisz testy oraz metodę, która przyjmuje liczbę całkowitą jako argument i zwraca:

1. `-1`, jeżeli podana liczba jest ujemna,
2. `0`, jeżeli podana liczba jest równa `0`,
3. `1`, jeżeli podana liczba jest dodatnia.

8.11.3 Testy zwracania indeksu szukanego elementu

Napisz testy oraz metodę, która przyjmuje jako argument tablicę liczb oraz liczbę i zwraca indeks w tej tablicy, pod którym znajduje się liczba podana jako drugi argument. Jeżeli podanej liczby nie ma w tablicy, metoda powinna zwrócić liczbę `-1`. Przykłady:

1. Dla argumentów `{ 1, 10, 200, 1000 }`, `200` – metoda powinna zwrócić `2`, ponieważ liczba `200` jest trzeciem elementem podanej tablicy, a jej indeks to `2` (bo, jak na pewno pamiętamy, indeksy zaczynamy liczyć od `0`).
2. Dla argumentów `{ 1, 10, 200, 1000 }`, `500` – metoda powinna zwrócić `-1`, ponieważ liczby `500` nie ma w podanej tablicy.

9 Rodział IX – Klasy

W tym rozdziale rozpoczniemy poznawanie klas i *programowania zorientowanego obiektowo* (*OOP – Object-Oriented Programming*).

Każdy tworzony przez nas tej pory program składał się z jednej klasy publicznej. Zazwyczaj nadawaliśmy mu nazwę zgodną z zadaniem, które wykonywał. Przypomnijmy kod naszego pierwszego programu:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

Program ten składa się z definicji publicznej klasy `HelloWorld`, której ciało zawarte jest między nawiasami klamrowymi. Klasa ta ma w swoim ciele jedną metodę – `main`. Jak wiemy, `main` to specjalna metoda, która jest punktem wejścia do wykonania naszego programu, która zawsze ma taką samą *sygnaturę* (tzn. modyfikatory, zwracany typ, nazwę, argumenty).

Nadszedł czas, by wyjaśnić, czym są i jak tworzyć bardziej skomplikowane klasy na nasz użytek – w tym rozdziale m. in.:

- zobaczymy, jak tworzy się i używa klas,
- dowiemy się, jak implementować metody `toString` oraz `equals`,
- poznamy kolejne różnice pomiędzy typami złożonymi i prymitywnymi,
- nauczymy się, do czego służą modyfikatory **`public`**, **`private`**, oraz **`static`**,
- dowiemy się, czym są pola i metody klas,
- zobaczymy, do czego służą konstruktory i słowo kluczowe **`this`**,
- nauczymy się, jak zawierać nasze klasy w pakietach i jak je importować.

W tym rozdziale, wiele z podrozdziałów ma własne sekcje z podsumowaniem, pytaniami sprawdzającymi znajomość materiału, oraz zadania.

9.1 Czym są klasy i do czego służą?

Klasy w programowaniu służą m. in. do opisywania otaczających nas przedmiotów, pojęć, zdarzeń, zadań, wymagań biznesowych, relacji itd. Za pomocą klas definiujemy zupełnie nowe typy, z których możemy korzystać w naszych programach tworząc zmienne tych typów. Typy definiowane za pomocą klas nazywamy *typami złożonymi*, czy też *typami referencyjnymi*.

Klasy mają dwie składowe:

- Pola – to nic innego jak poznane już przez nas zmienne. Kiedy jednak mamy na myśli zmienne należące do pewnej klasy, nazywamy je wtedy *polami* tej klasy.
- Metody – klasy definiują, jakie operacje klasa udostępnia dla zewnętrznego świata i jak można z niej korzystać.

Można powiedzieć, że klasa to pewien schemat, na który składają się jej pola i metody.

Na podstawie tego schematu możemy tworzyć *instancje* (czyli egzemplarze) klasy. Te egzemplarze klasy będą zawierały pola, które zdefiniowaliśmy w definicji klasy, oraz będzie można na nich wywoływać zdefiniowane w tej klasie metody. Te utworzone egzemplarze (instancje) klasy nazywamy *obiektami*.

Na rozmowach kwalifikacyjnych często zadawane jest pytanie: ***Czym różni się klasa od obiektu?***

Bardzo istotne jest by zrozumieć różnicę pomiędzy tymi dwoma pojęcia:

1. Jeżeli klasa zostałaby porównana do schematu samochodu, wówczas *obiektem tej klasy* byłby konkretny samochód wyprodukowany na podstawie tego schematu.
2. Innym przykładem mógłby być przepis na ciasto (klasa) oraz konkretne ciasto upieczone wedle tego przepisu (obiekt). Mając jeden przepis (jedną definicję klasy), możemy upiec wiele ciast (obiektów).

W poprzednich rozdziałach korzystaliśmy z typu `String`. Typ `String` to właśnie klasa, napisana przez twórców języka Java i udostępniona nam, programistom, byśmy mogli w prosty sposób operować na ciągach znaków. Za każdym razem, gdy tworzyliśmy zmienną typu `String`, tworzyliśmy obiekt klasy `String` – spójrzmy na przykład z poprzedniego rozdziału o metodach:

Nazwa pliku: Rozdzial_07__Metody/StringReplace.java

```
public class StringReplace {
  public static void main(String[] args) {
    String tekst = "Ala ma kota";

    String zmianaImienia = tekst.replace("Ala", "Jola");
    String bezSpacji = tekst.replace(" ", "");

    System.out.println("Tekst z innym imieniem: " + zmianaImienia);
    System.out.println("Tekst bez spacji: " + bezSpacji);
  }
}
```

W powyższym programie, w którym zaprezentowana została metoda `replace` klasy `String`, utworzyliśmy trzy obiekty typu (klasy) `String` o nazwach: `tekst`, `zmianaImienia`, oraz `bezSpacji` – obiekty te to konkretne instancje klasy `String`.

Jeszcze raz: klasa *definiuje jakie* pola *i* metody *będą miały jej* instancje, *czyli* obiekty *tej klasy. Obiekty to konkretne, utworzone egzemplarze danej klasy.*

9.1.1 Przykład pierwszej klasy z polami

Spójrzmy na przykład klasy `Samochod` z kilkoma polami oraz metodami:

Nazwa pliku: Samochod.java

```
public class Samochod {
  private int predkosc; // (1)
  private String kolor; // (2)

  public void ustawPredkosc(int nowaPredkosc) {
    predkosc = nowaPredkosc; // (3)
  }

  public void ustawKolor(String nowyKolor) {
    kolor = nowyKolor; // (4)
  }

  public void wypiszInformacje() {
    System.out.println("Jestem samochodem! Moj kolor to " + kolor +
        ", jade z predkoscia " + predkosc); // (5)
  }
}
```

1. Nasza klasa `Samochod` ma **dwa pola**: `predkosc` oraz `kolor`, a także **trzy metody**: `ustawPredkosc`, `ustawKolor`, oraz `wypiszInformacje`.
2. Jak widzimy, pola klasy *(1)* *(2)* to po prostu zmienne zdefiniowane poza jakąkolwiek metodą – właśnie ta właściwość powoduje, że są to pola klasy, a nie zmienne lokalne jednej z metod. Dodatkowo, typ pól klasy poprzedza modyfikator **`private`** – wkrótce opowiemy sobie dokładniej o tej i innych różnicach pomiędzy zmiennymi lokalnymi a polami klas, a także o znaczeniu modyfikatorów.
3. Zadaniem dwóch metod zdefiniowanych w klasie `Samochod` jest przypisanie wartości polom klasy – metoda `ustawPredkosc` ustawia wartość pola `predkosc` *(3)*, natomiast `ustawKolor` – pola `kolor` *(4)*. Trzecia metoda, `wypiszInformacje`, wypisuje dane o samochodzie *(5)*.

Taka definicja klasy informuje nas, że obiekty tej klasy będą miały po dwa pola: `predkosc` oraz `kolor`, oraz będziemy mogli na rzecz obiektów tej klasy wywoływać trzy metody: `ustawPredkosc`, `ustawKolor`, oraz `wypiszInformacje`.

Pytanie: czy metody napisane w klasie `Samochod` różnią się czymś od metod, które pisaliśmy do tej pory?

Metody te nie mają modyfikatora **`static`**, którego do tej pory zawsze używaliśmy tworząc nasze metody. Brak użycia słowa kluczowego **`static`** jest celowe – dlaczego i jaka jest różnica, gdy jest użyte – dowiemy się w dalszej części rozdziału. Na razie nie będziemy korzystać z modyfikatora **`static`** – jedynym wyjątkiem będzie metoda `main`, której sygnatura będzie go zawierała.

9.1.2 Użycie pierwszej klasy

Jak, mając tak zdefiniowaną klasę `Samochod`, możemy z niej skorzystać? W jaki sposób *stworzyć obiekty* (instancje) wedle powyższego *schematu*? Spójrzmy na przykład użycia powyższej klasy:

Nazwa pliku: UzycieSamochodu.java

```
public class UzycieSamochodu {
  public static void main(String[] args) {
    Samochod samochod1 = new Samochod(); // 1
    samochod1.ustawKolor("Zielony");
    samochod1.ustawPredkosc(50);

    Samochod samochod2 = new Samochod(); // 2
    samochod2.ustawKolor("Niebieski");
    samochod2.ustawPredkosc(60);

    samochod1.wypiszInformacje();
    samochod2.wypiszInformacje();
  }
}
```

Powyższa klasa o nazwie `UzycieSamochodu` zawiera metodę `main`, w której widzimy dwie zmienne `samochod1` oraz `samochod2` typu `Samochod`, czyli klasy, którą wcześniej zdefiniowaliśmy w osobnym pliku `Samochod.java`. Jak widzimy, możemy korzystać z klasy `Samochod` w innej, napisanej przez nas klasie.

Zmienne `samochod1` oraz `samochod2` przechowują dwa utworzone egzemplarze (obiekty) klasy `Samochod`. Tworzenie obiektów *(1)* *(2)* odbywa się poprzez użycie słowa kluczowego **`new`**, o którym opowiemy sobie dokładniej w kolejnym podrozdziale.

Klasa `UzycieSamochodu` to także nowy, zdefiniowany przez nas typ, jednak jedyne, do czego go wykorzystujemy, to punkt wejścia do naszego programu, ponieważ to w tej klasie zdefiniowaliśmy metodę `main`.

Na rzecz obu utworzonych obiektów wywołujemy metody `ustawKolor` oraz `ustawPredkosc` – tak samo, jak robiliśmy to w rozdziale o metodach typu `String` – tam także wywoływaliśmy różne metody (np. `toLowerCase`, `replace`) na zmiennych typu `String`. Wywołanie metod odbyło się poprzez napisanie kropki po nazwie zmiennej, po której nastąpiła nazwa metody oraz nawiasy wraz z argumentem dla każdej z metod:

```
samochod1.ustawKolor("Zielony");
samochod1.ustawPredkosc(50);
```

Na końcu programu wywołujemy metodę `wypiszInformacje` na rzecz każdego z naszych obiektów typu `Samochod`, w wyniku czego na ekranie zobaczymy:

```
Jestem samochodem! Moj kolor to Zielony, jade z predkoscia 50
Jestem samochodem! Moj kolor to Niebieski, jade z predkoscia 60
```

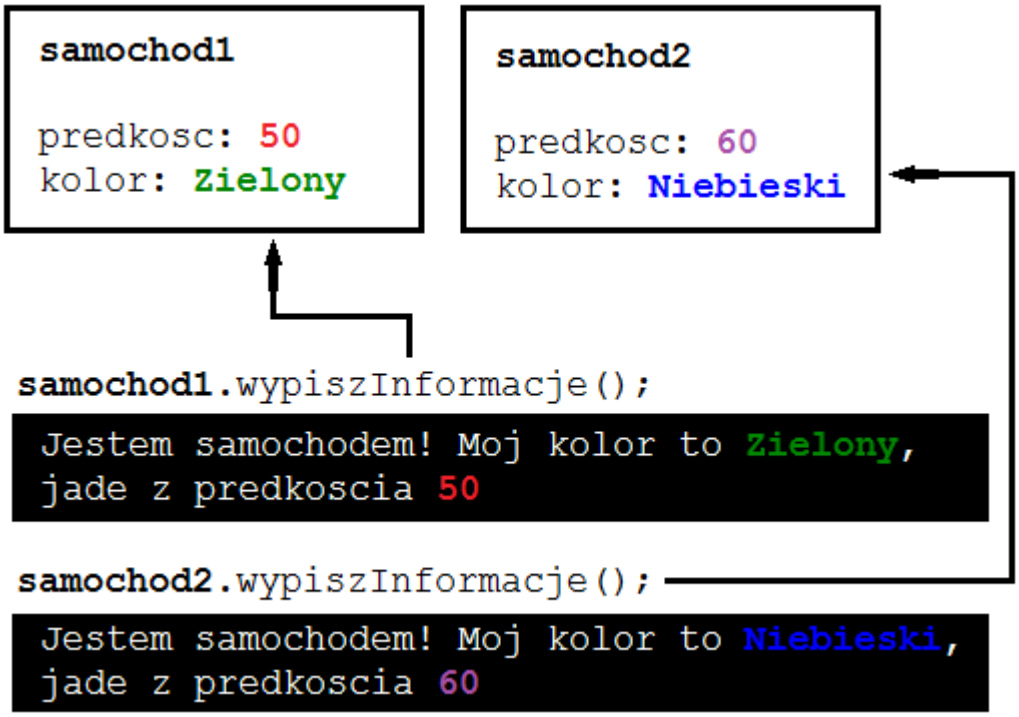
Zauważmy bardzo ważną cechę naszych obiektów – każdy z nich ma swój własny stan. Pierwszy samochód jest *zielony*, a drugi *niebieski*. Wywoływanie metody `ustawKolor` spowodowało zmianę jedynie tego obiektu, na rzecz którego została wywołana. Każdy z egzemplarzy (instancji) klasy `Samochod` ma swój własny zestaw niezależnych pól `kolor` oraz `predkosc`. Zmiana jednego obiektu typu `Samochod` nie ma wpływu na drugi obiekt.

Tak samo jak w życiu – pomalowanie naszego samochodu na zielono nie spowoduje, że wszystkie samochody tej samej marki i modelu na świecie staną się zielone.

Jest to bardzo ważna cecha obiektów, więc powtórzmy: każda obiekt ma swój własny stan, oddzielony od innych instancji tej samej klasy. Na ten stan składają się pola, które zostały zdefiniowane w klasie – w przypadku klasy `Samochod` są to pola:

- `predkosc`, typu `int`, oraz
- `kolor`, typu `String`.

Z drugiej jednak strony, wszystkie obiekty współdzielą metody, które zostały zdefiniowane w klasie. Metody te operują na stanie (polach) tego obiektu, na rzecz którego zostały wywołane. To dlatego wywołanie metody `wypiszInformacje` na różnych obiektach powoduje wypisanie innych danych na ekran – wypisywane są wartości pól obiektu, na rzecz którego metoda została wywołana:



Obiekty `samochod1` i `samochod2` mają własne egzemplarze pól `predkosc` oraz `kolor`. Wywołanie metody `wypiszInformacje` na każdym z obiektów powoduje, że metoda ta za pierwszym razem działa na polach `predkosc` oraz `kolor` o wartościach `50` i `"Zielony"`, a za drugim razem – `60` i `"Niebieski"`. Dzieje się tak dlatego, że metoda `wypiszInformacje` wie, na rzecz którego obiektu została wywołana.

Oba pliki (`Samochod` oraz `UzycieSamochodu`) z powyższego przykładu powinny być w tym samym katalogu. W jednym z kolejnych podrozdziałów nauczymy się, jak możemy odnosić się do klas zdefiniowanych w innych katalogach i używać ich w naszych programach.

***Pytanie**: co by się stało, gdybyśmy spróbowali skompilować i uruchomić klasę* `Samochod`*? Kompilacja przebiegłaby bez problemów, jednak zaraz po uruchomieniu program zakończyłby się błędem, ponieważ nie ma w nim metody* `main`*, a to ona definiuje, co nasz program będzie robił! Dlatego potrzebowaliśmy drugiej klasy,* `UzycieSamochodu`*, by zobaczyć naszą klasę* `Samochd` *w akcji.*

*Nasuwa się teraz **kolejne pytanie** – czy moglibyśmy przenieść metodę* `main` *z klasy* `UzycieSamochodu` *do klasy* `Samochod` *i wtedy spróbować ją uruchomić?*

Oczywiście! Program zadziałałby tym razem bez problemu – na ekranie zobaczylibyśmy komunikaty "Jestem samochodem (...)".

9.1.3 Nazewnictwo klas

W jednym z pierwszych rozdziałów kursu podane były dwie informacje dotyczące nazewnictwa klas i plików, w których są zawarte – przypomnijmy je sobie teraz:

1. Plik z kodem źródłowym musi nazywać się tak samo, jak nazwa publicznej klasy, która jest w nim zawarta – dlatego plik z kodem źródłowym klasy `Samochod` musieliśmy nazwać `Samochod.java`.
2. Nazwę klasy zawsze zaczynamy z wielkiej litery – nie jest to wymóg, lecz ogólnie przyjęta konwencja.

Klasy nie muszą być publiczne – opowiemy sobie dokładniej o takim przypadku w podrozdziale o pakietach klas. Na razie pamiętajmy, żeby zawsze nazywać plik z klasą tak, jak nazywa się klasa (z dodatkiem rozszerzenia `.java`), oraz by nazwa klasy zawsze zaczynała się z wielkiej litery.

Dodatkowo, w jednym pliku może być więcej niż jedna klasa – ale tylko jedna z nich może być publiczna. My będziemy w tym kursie zawsze umieszczali klasy w osobnych plikach, ponieważ tak najczęściej definiuje się klasy.

9.1.4 Jak tworzyć nowe instancje (obiekty) klas?

Aby utworzyć nowy obiekt danej klasy, używamy słowa kluczowego **`new`**, po którym powinny nastąpić: nazwa *konstruktora* klasy, której obiekt chcemy utworzyć, nawiasy oraz średnik. Konstruktory to specjalne metody służące do inicjalizacji tworzonych przez nas obiektów – poświęcony jest im jeden z kolejnych rozdziałów. Na razie wystarczy nam informacja, że konstruktory mają taką samą nazwę, jak klasa, w której się znajdują.

W przykładzie, w którym pokazywaliśmy, jak używać klasy `Samochod`, utworzyliśmy dwa obiekty typu `Samochod` w następujący sposób:

```
Samochod samochod1 = new Samochod();
Samochod samochod2 = new Samochod();
```

Każdy z utworzonych obiektów jest osobnym bytem, do którego możemy odnosić się za pomocą zmiennych `samochod1` oraz `samochod2`. Utworzone obiekty mają własny zestaw pól zdefiniowanych w klasie – ustawienie pola `kolor` obiektu `samochod1` nie ma wpływu na `kolor` obiektu `samochod2`.

Jeżeli w powyższym przykładzie zapomnielibyśmy o słowie kluczowym **`new`**, nasz kod źródłowy w ogóle by się nie skompilował – kompilator oczekiwałby, że `Samochod` będzie metodą bez argumentów, która zwraca obiekt typu `Samochod` – a takiej metody nasz przykład nie zawierał.

Może teraz nasunąć się pytanie: czy możemy już podczas tworzenia nowego obiektu nadać jego polom jakieś wartości? Możemy, używając, wspomnianych już, specjalnych metod nazywanych *konstruktorami*, o których wkrótce sobie opowiemy.

9.1.5 Metoda toString

W klasie `Samochod` napisaliśmy metodę `wypiszInformacje`, której zadaniem było wypisanie na ekran informacji na temat obiektu tej klasy:

```
public void wypiszInformacje() {
  System.out.println("Jestem samochodem! Moj kolor to " + kolor +
      ", jade z predkoscia " + predkosc);
}
```

Ta metoda jest przydatna, ponieważ możemy wypisać na ekran aktualny stan obiektu, to znaczy wartości przechowywane w danym momencie w jego polach (`kolor` i `predkosc`). Często się to przydaje, gdy np. zapisujemy do pliku logu różne informacje o statusie wykonywania naszego programu.

Okazuje się, że ta operacja jest na tyle często używana i przydatna, że możemy w każdej klasie napisać specjalną metodę `toString`, która będzie zwracała tekstową reprezentację danego obiektu.

Metoda `toString` musi mieć konkretną sygnaturę, jak pokazano poniżej:

```
// 1     2        3    4
public String toString() {
  return "Jestem samochodem! Moj kolor to " + kolor +
      ", jade z predkoscia " + predkosc; // 5
}
```

Powyższa metoda `toString` zastępuje metodę `wypiszInformacje`, której do tej pory używaliśmy w klasie `Samochod`. Metoda `toString` musi spełniać następujące wymagania:

1. Musi nazywać się `toString` *(3)*.
2. Musi zwracać `String` *(2)* *(5)*.
3. Nie może przyjmować żadnych argumentów *(4)*.
4. Musi być publiczna *(1)* (tzn. należy użyć modyfikatora **`public`**) i nie może być statyczna (nie może mieć modyfikatora **`static`**).

Jeżeli nie spełnimy powyższych wymagań, nasz program nie będzie albo działał w ogóle (nie skompiluje się), albo metoda nie będzie działała zgodnie z naszymi oczekiwaniami – z czego te wymagania wynikają, dowiemy się w rozdziale o dziedziczeniu.

Pytanie: jaka jest w takim razie zaleta metody `toString`?

Metoda `toString` jest automatycznie wywoływana na rzecz obiektu, gdy będzie on użyty w wyrażeniu, w którym znajdują się łańcuchy tekstowe (stringi). Metoda `println` (używana poprzez `System.out.println`) także jest w stanie skorzystać z tej metody.

Dzięki temu, nie musimy wywoływać tej metody sami za każdym razem, kiedy chcielibyśmy dostać tekstową reprezentację naszego obiektu. Spójrzmy na przykład klasy `Samochod` oraz `UzycieSamochodu` po dodaniu do klasy `Samochod` metody `toString`:

Nazwa pliku: Samochod.java

```
public class Samochod {
  private int predkosc;
  private String kolor;

  public void ustawPredkosc(int nowaPredkosc) {
    predkosc = nowaPredkosc;
  }

  public void ustawKolor(String nowyKolor) {
    kolor = nowyKolor;
  }

  public String toString() {
    return "Jestem samochodem! Moj kolor to " + kolor +
        ", jade z predkoscia " + predkosc;
  }
}
```

Zastąpiliśmy metodę `wypiszInformacje` metodą `toString`, która nie wypisuje już informacji na ekran, a zamiast tego zwraca opis naszego obiektu w postaci stringu. Spójrzmy na użycie tej klasy:

Nazwa pliku: UzycieSamochodu.java

```
public class UzycieSamochodu {
  public static void main(String[] args) {
    Samochod samochod1 = new Samochod();
    samochod1.ustawKolor("Zielony");
    samochod1.ustawPredkosc(50);

    Samochod samochod2 = new Samochod();
    samochod2.ustawKolor("Niebieski");
    samochod2.ustawPredkosc(60);

    System.out.println(samochod1); // 1
    System.out.println(samochod2); // 2

    String opisSamochodu1 = samochod1.toString(); // 3
    String dokladniejszyOpis = "Opis zmiennej samochod1 to: " + samochod1; // 4

    System.out.println(opisSamochodu1);
    System.out.println(dokladniejszyOpis);
  }
}
```

W klasie `UzycieSamochodu` nie korzystamy już z metody `wypiszInformacje`. Zamiast tego wypisujemy na ekran informacje o zmiennych `samochod1` oraz `samochod2` bezpośrednio przesyłając je jako argumenty do `println` *(1)* *(2)*.

`toString` to metoda jak każda inna – możemy z niej skorzystać, tzn. wywołać ją na rzecz danego obiektu – tak jak robimy to w linijce oznaczonej *(3)*, w której przypisujemy wynik wywołania `toString` na obiekcie `samochod1` do zmiennej typu `String` o nazwie `opisSamochodu1`.

Dodatkowo, jak wspomnieliśmy powyżej, `toString` jest automatycznie wywoływane na rzecz obiektów, które znajdują się w wyrażeniu zwierającym łańcuchy tekstowe. Spójrzmy na linijkę *(4)*:

```
String dokladniejszyOpis = "Opis zmiennej samochod1 to: " + samochod1; // 4
```

Do zmiennej `dokladniejszyOpis` przypisujemy wynik konkatenacji (złączenia) stringu `"Opis`

`zmiennej samochod1 to: "` oraz zmiennej `samochod1`. Zmienna `samochod1` nie ma żadnego związku z łańcuchami tekstowymi – w końcu jest to obiekt typu `Samochod`, a nie `String`! Jednakże, w tym przypadku, kompilator ułatwia nam życie i automatycznie wywołuje za nas metodę `toString` na zmiennej `samochod1`. Zamiast więc przypisać do zmiennej `dokladniejszyOpis` string i `Samochod`, tak naprawdę przypisujemy do niej złączenie stringu `"Opis zmiennej samochod1 to: "` oraz stringu `"Jestem samochodem! Moj kolor to Zielony, jade z predkoscia 50"`, który zwrócony zostanie przez metodę `toString` wywołaną na rzecz obiektu `samochod1`.

W wyniku działania programu na ekranie zobaczymy:

```
Jestem samochodem! Moj kolor to Zielony, jade z predkoscia 50
Jestem samochodem! Moj kolor to Niebieski, jade z predkoscia 60
Jestem samochodem! Moj kolor to Zielony, jade z predkoscia 50
Opis zmiennej samochod1 to: Jestem samochodem! Moj kolor to Zielony, jade z
predkoscia 50
```

Pytanie: a co by się stało, gdyby klasa `Samochod` nie miała metody `toString`, a spróbowalibyśmy ją wypisać na ekran za pomocą `System.out.println`? Czy kompilacja zakończyłaby się błędem? Nie – program wykonałby się bez błędów, a na ekranie zobaczylibyśmy komunikat podobny do poniższego (dla obiektu `samochod1`):

```
Samochod2@4554617c
```

Jest to domyślny sposób tekstowej reprezentacji obiektów, gdy w ich klasie nie została zdefiniowana metoda `toString`. Dlaczego zapis ten wygląda w taki właśnie sposób – dowiemy się w jednym z kolejnych podrozdziałów.

9.1.6 Podsumowanie

- Klasy w programowaniu służą m. in. do opisywania otaczających nas przedmiotów, pojęć, zdarzeń, zadań, wymagań biznesowych, relacji, itp.
- Za pomocą klas definiujemy nowe typy, których możemy używać w naszych programach.
- Typy definiowane za pomocą klas nazywamy *typami złożonymi*, czy też *typami referencyjnymi*.
- Plik z kodem źródłowym musi nazywać się tak samo, jak nazwa publicznej klasy, która jest w nim zawarta, z dodatkiem rozszerzenia `.java`. Dla przykładu – klasa `Samochod` powinna być zapisana w pliku o nazwie `Samochod.java`.
- Nazwę klasy zaczynamy z wielkiej litery – nie jest to wymóg, lecz przyjęta konwencja.
- Klasy składają się z *pól* oraz *metod*.
- *Pola klasy* to zmienne, która zostały zdefiniowane w tej klasie (poza metodami).
- Metody klas definiują, jakie operacje klasa udostępnia dla zewnętrznego świata i jak można z niej korzystać.
- Klasa to pewien schemat, na który składają się jej pola i metody.
- Aby korzystać z klasy, należy utworzyć jej instancje, czyli *obiekty* tej klasy.
- Klasa od obiektu różni się tym, że klasa to opis zawartości (pól) i dostępnych metod, a obiekty to konkretne egzemplarze tej klasy, mające opisaną w klasie zawartość (pola) i możliwe do wykonania na nich metody.
- Klasę możnaby porównać do schematu samochodu, a konkretne samochody wyprodukowane na podstawie tego schematu, do obiektów tej klasy.
- Na razie pola w klasach definiujemy z modyfikatorem **`private`**, a metody – z **`public`**.
- Modyfikatora **`static`** nie używamy na razie ani przy definicjach pól, ani metod. Wyjątkiem jest tutaj metoda `main`, która nadal powinna korzystać z tego modyfikatora.
- Aby skorzystać z utworzonej klasy, należy utworzyć obiekt tej klasy za pomocą słowa kluczowego **`new`**, po którym następuje *konstruktor* tej klasy (konstruktor to specjalna metoda, która nazywa się tak samo jak klasa, w której jest zawarta), nawiasy oraz średnik:

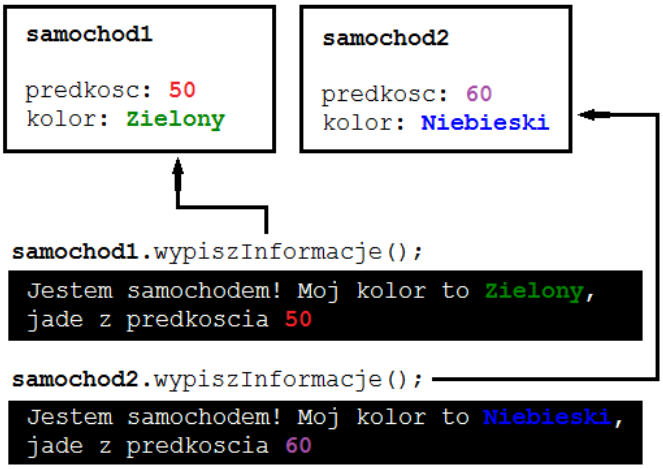
```
Samochod samochod1 = new Samochod();
Samochod samochod2 = new Samochod();
```

- Na rzecz utworzonych obiektów danej klasy możemy wywoływać zdefiniowane w tej klasie metody za pomocą użycia nazwy zmiennej, kropki, nazwy metody, i ewentualnych argumentów.
- Każdy z utworzonych obiektów danej klasy ma swój własny zestaw pól zdefiniowanych w tej klasie – ustawienie pola `kolor` obiektu `samochod1` nie wpływa na pole `kolor` obiektu `samochod2`:

```
samochod1.ustawKolor("Zielony");
samochod1.ustawPredkosc(50);

samochod2.ustawKolor("Niebieski"); // 1
samochod2.ustawPredkosc(60);
```

Po wykonaniu metody `ustawKolor` na obiekcie `samochod2` *(1)*, wartość pola `kolor` w obiekcie `samochod1` to nadal `"Zielony"` – ustawienie koloru na obiekcie `samochod2` nie ma wpływu na obiekt `samochod1`, który ma własny "zestaw" pól `kolor` oraz `predkosc`.

- Gdy wywołujemy metodę na rzecz pewnego obiektu, metoda ta wie na rzecz którego obiektu została wykonana. Wszelkie operacje, które ta metoda wykonuje na polach obiektu, wykonuje na polach dokładnie tego obiektu, na rzecz którego została wywołana. Użycie metody `wypiszInformacje`, którą zdefiniowaliśmy w klasie `Samochod`, która wypisuje wartości pól obiektu, powoduje, że wypisane zostają wartości pól tego obiektu, na rzecz którego ją wywołaliśmy – zagadnienie to opisuje poniższy obrazek:



- W klasach możemy zdefiniować specjalną metodę `toString`, której zadaniem będzie zwracanie tekstowej reprezentacji naszych obiektów:

```
// 1      2        3     4
public String toString() {
  return "Jestem samochodem! Moj kolor to " + kolor +
      ", jade z predkoscia " + predkosc; // 5
}
```

- Metoda ta jest automatycznie wykonywana, gdy nasz obiekt znajduje się w wyrażeniu, w którym występują stringi (łańcuchy tekstowe). Przykład – poniżej metoda `toString` zostanie automatycznie wywołana na obiekcie `samochod1`:

```
String dokladniejszyOpis = "Opis zmiennej samochod1 to: " + samochod1;
```

- `System.out.println` także może skorzystać z tej metody:

```
System.out.println(samochod1);
```

- Metoda `toString` musi spełniać następujące wymagania:
  - Musi nazywać się `toString` *(3)*.
  - Musi zwracać `String` *(2)* *(5)*.
  - Nie może przyjmować żadnych argumentów *(4)*.
  - Musi być publiczna *(1)* (tzn. należy użyć modyfikatora **`public`**) i nie może być statyczna (nie może mieć modyfikatora **`static`**).

9.1.7 Pytania

1. Czym różni się klasa od obiektu?
2. Z czego składają się klasy?
3. Jak utworzyć nowy obiekt klasy?
4. Do czego służy i jakie wymagania powinna spełniać metoda `toString`?
5. Czy poniższa linia kodu jest poprawna?

```
Samochod samochod = Samochod();
```

6. Czy poniższa klasa jest poprawna?

Nazwa pliku: PytaniaOKlasach.java

```
public class Pytanie {
  public static void main(String[] args) {
    System.out.println("Witaj!");
  }
}
```

7. Jaki będzie wynik uruchomienia poniższego programu?

```
public class PrzykladowaKlasa {
  private int x;
}
```

8. Co zostanie wypisane na ekran?

```
public class Punkt {
  private int x, y;

  public void ustawX(int wartoscX) {
    x = wartoscX;
  }

  public void ustawY(int wartoscY) {
    y = wartoscY;
  }

  public String toString() {
    return "X, Y: " + x + ", " + y;
  }

  public static void main(String[] args) {
    Punkt a = new Punkt();
    Punkt b = new Punkt();

    a.ustawX(10);
    a.ustawY(20);

    b.ustawX(0);
    b.ustawY(5);

    System.out.println(a);
    System.out.println(b);
  }
}
```

9. Co zobaczymy na ekranie po uruchomieniu poniższego programu?

```
public class PytanieToString {
  public void toString() {
    System.out.println("Jestem obiektem klasy PytanieToString!");
  }

  public static void main(String[] args) {
    PytanieToString a = new PytanieToString();
    System.out.println(a);
  }
}
```

9.1.8 Zadania

9.1.8.1 Klasa Osoba

Napisz klasę `Osoba`, która będzie zawierała:

1. Trzy pola: `wiek`, `imie`, `nazwisko`. Użyj odpowiednich typów.
2. Trzy metody, w których będziesz ustawiał wartości pól klasy: `ustawWiek`, `ustawImie`, `ustawNazwisko`. **Argumenty tych metod powinny nazywać się `wartoscWieku`, `imieOsoby`, `nazwiskoOsoby`.**
3. Metodę `toString`, która będzie zwracała informacje o imieniu, nazwisko, oraz wieku osoby.
4. Metodę `main`, w której utworzysz jeden obiekt klasy `Osoba` i nadasz mu wartości za pomocą metod `ustawWiek`, `ustawImie`, oraz `ustawNazwisko`, a następnie, za pomocą `System.out.println`, wypiszesz utworzony obiekt typu `Osoba` na ekran.

9.2 Różnice między typami prymitywnymi i typami referencyjnymi

Zanim opowiemy sobie o kolejnych aspektach tworzenia klas, przyjrzymy się dwóm bardzo istotnym różnicom pomiędzy typami prymitywnymi oraz typami referencyjnymi.

Przypomnijmy, czym są oba te rodzaje typów:

- Typy prymitywne – są to typy wbudowane – Java oferuje 8 typów prymitywnych, które już poznaliśmy – są to: **boolean**, **byte**, **short**, **int**, **long**, **float**, **double**, oraz **char**. Są to elementy budujące, z których składają się typy złożone, definiowane przez programistów.
- Typy referencyjne – są to typy złożone, zdefiniowane w bibliotekach standardowych języka Java (jak np. typ `String`) oraz tworzone przez programistów poprzez pisanie klas (jak np. klasa `Samochod` z poprzedniego podrozdziału). Do typów złożonych zaliczają się także tablice.

W rozdziale o metodach poznaliśmy już jedną różnicę – zmiany na obiektach typów złożonych przesłanych jako argumenty do metody wykonywane są na oryginalnych obiektach, a nie na kopiach. Wkrótce wrócimy do tego zagadnienia.

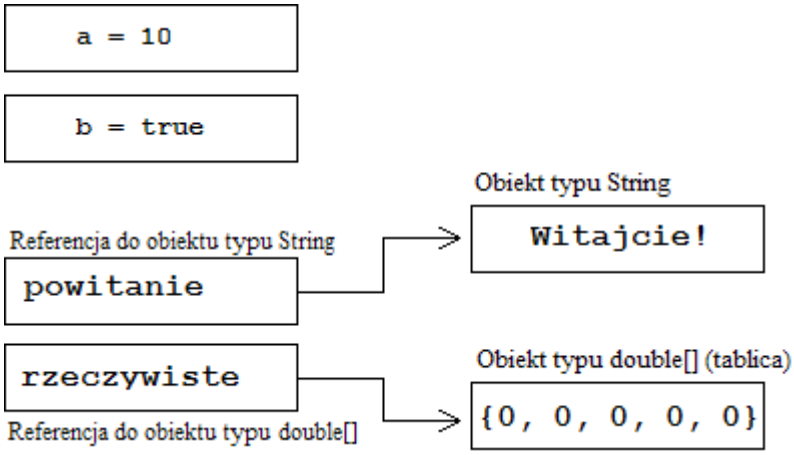
W dalszej części rozdziału o klasach poznamy jeszcze kilka innych, niemniej ważnych, różnic.

9.2.1 Przechowywane wartości

Główną różnicą pomiędzy typami prymitywnymi a typami referencyjnymi jest to, że zmienne typów prymitywnych przechowują w pamięci komputera konkretne wartości, natomiast zmienne typu referencyjnego przechowują adresy obiektów w pamięci:

```
int a = 10; // zmienna a ma wartosc 10
boolean b = true; // zmienna b ma wartosc true

// zmienna powitanie zawiera adres obiektu typu String w pamieci,
//  ktory przechowuje tekst Witajcie!
String powitanie = "Witajcie!";

// zmienna rzeczywiste zawiera adres tablicy w pamieci,
//  ktora przechowuje 5 liczb rzeczywistych
double[] rzeczywiste = new double[5];
```

Zagadnienie to obrazuje poniższy rysunek:



Zmienne referencyjne (zmienne typów złożonych, czyli klas) **nie są tworzonymi obiektami, lecz odniesieniami do utworzonych obiektów, które do nich przypisujemy.** Natomiast zmienne typu prymitywnego przechowują wartości, które do nich przypisaliśmy.

Adresy obiektów to także pewne wartości. Nie operujemy na nich co prawda bezpośrednio, ale możemy się nimi posługiwać:

Nazwa pliku: ZmienneTypowZlozonych.java

```
public class ZmienneTypowZlozonych {
  public static void main(String[] args) {
    Samochod pierwszySamochod = new Samochod(); // 1
    pierwszySamochod.ustawKolor("Czerwony"); // 2
    pierwszySamochod.ustawPredkosc(80);

    Samochod drugiSamochodd = pierwszySamochod; // 3
    drugiSamochodd.ustawKolor("Bialy"); // 4

    System.out.println(pierwszySamochod); // 5
    System.out.println(drugiSamochodd); // 6
  }
}
```

Co zobaczymy na ekranie w wyniku wykonania powyższego programu?

1. Program ten korzysta z klasy `Samochod` z poprzedniego podrozdziału, która zawiera m. in. pole `kolor` oraz metodę `toString`.
2. Najpierw tworzymy nowy obiekt klasy `Samochod` i przypisujemy go do zmiennej `pierwszySamochod` *(1)*, a następnie ustawiamy jego `kolor` i `predkosc` za pomocą metod `ustawKolor` i `ustawPredkosc` *(2)*.
3. W kolejnej linii tworzymy drugą zmienną typu `Samochod` o nazwie `drugiSamochod` i przypisujemy do niej zmienną `pierwszySamochod` *(3)*. Następnie, ustawiamy `kolor` obiektu `drugiSamochod` na `"Bialy"` *(4)*.
4. Na końcu programu wypisujemy na ekran oba obiekty *(5)* *(6)*.

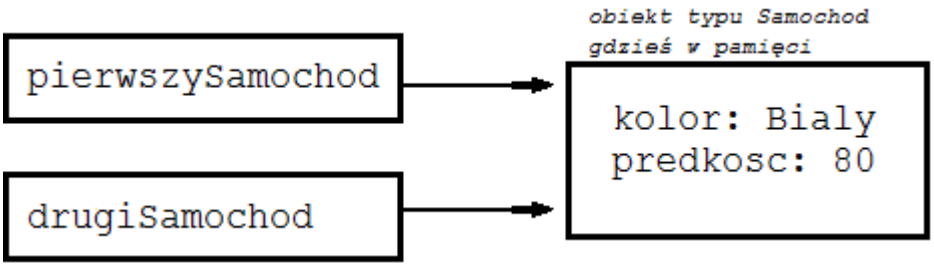
Wynikiem działania programu jest:

```
Jestem samochodem! Moj kolor to Bialy, jade z predkoscia 80
Jestem samochodem! Moj kolor to Bialy, jade z predkoscia 80
```

Chwila, moment! Dlaczego oba obiekty sa białego koloru?

Jak już wspomnieliśmy, zmienne typów referencyjnych to nie obiekty – to *adresy* obiektów. Gdy przypisujemy do jednej zmiennej typu referencyjnego wartość innej zmiennej typu referencyjnego, to kopiujemy adres obiektu docelowego, a nie obiekt docelowy.

Dlatego właśnie na ekranie zobaczyliśmy takie same wartości – ponieważ mamy **jeden** obiekt typu `Samochod` (utworzony w linii *(1)* za pomocą słowa kluczowego **`new`**) oraz dwie zmienne, które na niego wskazują:



Innymi słowy – poniższa linijka nie powoduje stworzenia kopii obiektu typu `Samochod`:

```
Samochod drugiSamochodd = pierwszySamochod;
```

lecz przepisanie (skopiowanie) **adresu** obiektu, na który zmienna po prawej stronie operatora przypisania wskazuje.

Skoro mamy tylko jeden obiekt, to drugie wywołanie `ustawKolor` powoduje nadpisanie poprzedniej wartości koloru, którą ustawiliśmy na początku programu. Obie zmienne – `pierwszySamochod` oraz `drugiSamochod` – wskazują na ten sam obiekt w pamięci.

W takim razie można by pomyśleć, że instrukcja:

```
drugiSamochodd.ustawKolor("Bialy");
```

spowodowała, że zmodyfikowana została zarówno zmienna `pierwszySamochod`, jak i zmienna `drugiSamochod`. Nie jest to jednak prawda – ani zmienna `pierwszySamochod`, ani zmienna `drugiSamochod`, nie została zmodyfikowana. Obiekt, który został zmodyfikowany, to ten, na który obie te zmienne wskazują – czyli obiekt typu `Samochod`, przechowywany gdzieś w pamięci komputera, którego adres przypisany jest do obu zmiennych: `pirwszySamochod` oraz `drugiSamochod`.

Tablice to także typy referencyjne, a skoro tak, to co zobaczymy w wyniku działania poniższego programu?

Nazwa pliku: ZmienneITablica.java

```
public class ZmienneITablica {
  public static void main(String[] args) {
    double[] rzeczywiste = new double[5];
    rzeczywiste[0] = 3.14;

    double[] drugaTablica = rzeczywiste;
    drugaTablica[0] = 5;

    System.out.println("Pierwszy element rzeczywiste: " + rzeczywiste[0]);
    System.out.println("Pierwszy element drugaTablica: " + drugaTablica[0]);
  }
}
```

Na ekranie zobaczymy:

```
Pierwszy element rzeczywiste: 5.0
Pierwszy element drugaTablica: 5.0
```

Dlaczego zobaczyliśmy taki wynik? W tym programie mamy tylko jedną tablicę liczb rzeczywistych, utworzoną w linii:

```
double[] rzeczywiste = new double[5];
```

W poniższej linii nie kopiujemy tablicy, na którą wskazuje zmienna `rzeczywiste`, lecz adres tej tablicy:

```
double[] drugaTablica = rzeczywiste;
```

W wyniku tego, obie zmienne: `rzeczywiste` i `drugaTablica`, wskazują na ten sam obiekt – tablicę mogącą przechowywać pięć liczb rzeczywistych. Przypisanie wartość `5` do pierwszego elementu tej tablicy nadpisuje umieszczoną tam wcześniej wartość `3.14`. Chociaż ustawienie każdej z tych wartości dokonaliśmy za pomocą innej zmiennej wskazującej na tę tablicę, to docelowo operowaliśmy na tej samej tablicy.

Pamiętajmy: zmienne typów referencyjnych (klas) oraz zmienne tablicowe przechowują adresy obiektów tych klas i tablic.

Rozróżnianie adresów i docelowych obiektów jest ważne, ale mało praktyczne. Dlatego, gdy działamy na zmiennej pewnej klasy, operujemy nazwą tej zmiennej, mając na myśli docelowy obiekt. Nie mówimy w praktyce, że "zmienna `samochod1` *zawiera adres pewnego obiektu typu* `Samochod` *w pamięci", lecz, po prostu (ze względu na naszą wygodę), że zmienna* `samochod1` *jest obiektem typu* `Samochod` *(mając jednak nadal świadomość, że nie jest to do końca prawda –* `samochod1` *nie jest obiektem, lecz wskazaniem na obiekt).*

9.2.2 Tworzenie

By stworzyć zmienną typu prymitywnego, piszemy po prostu typ oraz nazwę zmiennej:

```
int a = 10;
boolean b = true;
```

Natomiast tworzenie obiektów, na które będzie wskazywać zmienna typu referencyjnego, odbywa się, jak już wiemy, poprzez użycie słowa kluczowego **new** – ma ono za zadanie utworzyć nowy obiekt, do którego referencja zostanie zapisana w zmiennej, którą definiujemy:

```
Samochod pierwszySamochod = new Samochod();

String powitanie = "Witajcie!";
String java = new String("Java");

double[] rzeczywiste = new double[5];
double[] rzeczywiste2 = { 3.14, 5, -20.5 };
```

Typ `String` jest szczególnym typem złożonym w Javie – nie wymaga on użycia słowa kluczowego **new** dla wygody programistów – wystarczy po prostu przypisać do zmiennej typu `String` wartość, jak w powyższym przykładzie ze zmienną `powitanie`. Mało tego, słowo kluczowe **new** przy pracy ze stringami nie powinno być w ogóle używane ze względów wydajnościowych, ponieważ Java specjalnie przechowuje łańcuchy tekstowe.

Tablice także są szczególne, ponieważ zamiast używać słowa kluczowego **new**, możemy użyć skróconego zapisu z użyciem nawiasów klamrowych, w których umieszczamy elementy tablicy. Rozmiar tablicy będzie wydedukowany na podstawie liczby podanych elementów.

Typ `String` i tablice to dwa wyjątki w świecie typów złożonych, jeśli chodzi o możliwy sposób ich tworzenia.

Ilekroć pracujemy z obiektami typu `String`, nigdy nie twórzmy ich za pomocą słowa kluczowego **new** – korzystajmy zawsze z bezpośredniego przypisywania ich do zmiennych.

*Dlaczego nie powinniśmy tworzyć wartości typu `String` za pomocą słowa kluczowego **new**? Przygotowanie pamięci i tworzenie obiektów typu `String` może być czasochłonne, szczególnie, jeżeli będziemy tworzyli dużo łańcuchów tekstowych w naszych programach. Twórcy języka Java zoptymalizowali sposób przechowywania wartości typu `String` – w uproszczeniu, są one trzymane w tzw.* String pool*, który cache'uje użyte już raz wartości typu String, dzięki czemu ponownie użycie takiego samego stringa nie wymaga ponownego tworzenia.*

*Jeżeli jednak korzystamy ze słowa kluczowego **new** w celu utworzenia wartości typu `String`, to pamięć za każdym razem jest alokowana na nowo dla takiej wartości, nawet, jeżeli była ona już przechowywana w String pool, co może odbić się na wydajności naszej aplikacji – dlatego zawsze powinniśmy po prostu przypisywać do zmiennych typu `String` wartości za pomocą:*

```
String nazwaZmiennej = "pewien tekst";
```

9.3 Modyfikatory dostępu

Od samego początku nauki języka Java używaliśmy słowa kluczowego **public** – poprzedzało one nazwy naszych klas oraz metod. W rozdziale o klasach zaczęliśmy także używać słowa kluczowego **private**. W tym rozdziale dowiemy się, jakie jest ich znaczenie w odniesieniu do pól i metod klas.

W odniesieniu do klas możemy nie stosować **public** *(chociaż dotąd zawsze tak robiliśmy w naszych programach). Jakie to słowo kluczowe ma znaczenie podczas definiowania klas dowiemy się pod koniec rozdziału o klasach.*

9.3.1 Modyfikatory dostępu public oraz private

Słowa kluczowe **public** oraz **private** to *modyfikatory dostępu*.

Modyfikatory dostępu służą do ustawiania zakresu widoczności pól oraz metod klasy. Dzięki nim możemy określić, jak będzie wyglądało użytkowanie naszej klasy oraz kto, i w jakich okolicznościach, będzie mógł z pól i metod tej klasy korzystać.

W Javie istnieją cztery modyfikatory dostępu:

- **public**
- **protected**
- *domyślny*
- **private**

W tym rozdziale skupimy się na modyfikatorach **public** oraz **private**. O modyfikatorze *domyślnym* (który nie ma własnego słowa kluczowego) opowiemy sobie pod koniec rozdziału o klasach, a o modyfikatorze **protected** w rozdziale o dziedziczeniu.

Gdy definiujemy pole bądź metodę z modyfikatorem **public**, to oznajmiamy, że to pole lub metoda jest dostępne dla całego "zewnętrznego świata" – każdy może się do takiego pola odnieść i wywołać taką metodę. Mówimy wtedy, że pole lub metoda jest *publiczne*.

Z drugiej jednak strony, nasza klasa może zawierać także pola i metody, których nie chcemy udostępniać – mają one być *prywatne* dla naszej klasy – tylko z poziomu tej klasy powinniśmy móc tymi polami i metodami zarządzać. Takie pola i metody definiuje się korzystając z modyfikatora dostępu **private**.

Pytanie: wobec kogo stosowane są modyfikatory dostępu? Kogo mamy na myśli mówiąc "zewnętrzny świat"?

Modyfikatory dostępu regulują dostęp do pól i metod klas, gdy odnosimy się do nich z innych klas. Przejdziemy teraz przez kilka prostych przykładów, które wyjaśnią działanie modyfikatorów dostępu.

9.3.2 Modyfikatory public oraz private – przykład – klasa Ksiazka

Przykład będzie bazował na klasie `Ksiazka` – przyjrzyjmy się jej definicji:

Nazwa pliku: Ksiazka.java

```
public class Ksiazka {
  public String tytul; // 1
  public String autor; // 2

  private double cena; // 3

  public void ustawCene(double nowaCena) { // 4
    if (czyCenaJestPoprawna(nowaCena)) { // 5
      cena = nowaCena;
    } else {
      System.out.println("Cena " + nowaCena + " jest nieprawidlowa!");
    }
  }

  public String toString() { // 6
    return "Ksiazka o tytule " + tytul +
        ", ktorej autorem jest " + autor + ", kosztuje " + cena;
  }


  private boolean czyCenaJestPoprawna(double cenaDoSprawdzenia) { // 7
    return cenaDoSprawdzenia > 0;
  }
}
```

Klasa ta zawiera:

- dwa publiczne pola `tytul` *(1)* oraz `autor` *(2)*,
- jedno prywatne pole `cena` *(3)*,
- dwie publiczne metody `ustawCene` *(4)* oraz `toString` *(6)*,
- jedną prywatną metodę `czyCenaJestPoprawna` *(7)*.

Metoda `ustawCene` służy do ustawiania wartości pola `cena`. Nie chcieliśmy bezpośrednio udostępniać pola `cena` światu zewnętrznemu, ponieważ zawsze przed ustawieniem ceny chcemy sprawdzić, czy jest ona poprawna. Osiągamy to przez sprawdzenie poprawności ceny za pomocą prywatnej metody `czyCenaJestPoprawna`, z której korzystamy w metodzie `ustawCene` *(5)*.

Znamy już znaczenie metody `toString` – zwraca ona tekstową reprezentację obiektu klasy `Ksiazka`. Prywatna metoda `czyCenaJestPoprawna` zwraca **`true`**, jeżeli podana cena jest większa od zera, a przeciwnym razie – **`false`**.

9.3.2.1 Użycie pól i metod publicznych klasy Ksiazka

Mając tak zdefiniowaną klasę `Ksiazka`, spójrzmy, jak możemy korzystać z jej pól i metod, biorąc pod uwagę modyfikatory dostępu, które zostały w niej użyte:

Nazwa pliku: Ksiegarnia.java

```
public class Ksiegarnia {
  public static void main(String[] args) {
    Ksiazka lokomotywa = new Ksiazka(); // 1

    lokomotywa.tytul = "Lokomotywa"; // 2
    lokomotywa.autor = "Julian Tuwim"; // 3
    lokomotywa.ustawCene(29.99); // 4

    System.out.println(lokomotywa); // 5
  }
}
```

Ten prosty przykład tworzy nowy obiekt klasy `Ksiazka` *(1)*, a następnie, za pomocą kropki, odnosi się zarówno do pól `tytul` oraz `autor` obiektu `lokomotywa`, a także wywołuje metodę `ustawCena`.

Wiemy już, że aby wywołać metodę na rzecz pewnego obiektu, po nazwie tego obiektu piszemy kropkę, a następnie metodę, którą chcemy wywołać *(4)*. W ten sam sposób możemy odnieść się do *publicznych* pól obiektu i przypisać im wartość, tak jak zrobilśmy to w powyższym przykładzie, nadając polom `tytul` oraz `autor` wartości, odpowiednio, `"Lokomotywa"` *(2)* oraz `"Julian Tuwim"` *(3)*.

Na końcu program wypisuje na ekran tekstową reprezentację obiektu `lokomotywa` (jak wiemy, zostanie tutaj automatycznie użyta metoda `toString`). Na ekranie zobaczymy:

```
Ksiazka o tytule Lokomotywa, ktorej autorem jest Julian Tuwim, kosztuje 29.99
```

Jak widzimy, ustawione przez nas wartości w polach `tytul` oraz `autor` zostały zapamiętane w obiekcie `lokomotywa`.

9.3.2.2 Próba użycia pola i metody private klasy Ksiazka spoza tej klasy

Co by się stało, gdybyśmy spróbowali przypisać wartość do prywatnego pola `cena` lub wywołać prywatną metodę `czyCenaJestPoprawna` z klasy `Ksiazka`? Sprawdźmy:

```
public class Ksiegarnia {
  public static void main(String[] args) {
    Ksiazka lokomotywa = new Ksiazka();

    lokomotywa.tytul = "Lokomotywa";
    lokomotywa.autor = "Julian Tuwim";
    lokomotywa.ustawCene(29.99);

    System.out.println(lokomotywa);

    lokomotywa.cena = -10;
    lokomotywa.czyCenaJestPoprawna(0);
  }
}
```

Próba kompilacji powyższej wersji klasy `Ksiegarnia` kończy się błędem – kompilator wypisuje na

ekran dwa znaleziony problemy:

```
Ksiegarnia.java:13: error: cena has private access in Ksiazka
        lokomotywa.cena = -10;
                  ^
Ksiegarnia.java:14: error: czyCenaJestPoprawna(double) has private access in Ksiazka
        lokomotywa.czyCenaJestPoprawna(0);
                  ^
2 errors
```

Kompilator zaprotestował, ponieważ próbowaliśmy odnieść się do prywatnego pola i prywatnej metody klasy `Ksiazka`. Widzimy tutaj modyfikatory dostępu w akcji – w klasie `Ksiazka` zdefiniowaliśmy pole `cena` oraz metodę `czyCenaJestPoprawna` z użyciem modyfikatora **`private`**. To pole i metoda są więc prywatne dla klasy `Ksiazka` i nikt nie powinien mieć do nich dostępu spoza klasy `Ksiazka`.

Użycie modyfikatora **`private`** chroni nas przed potencjalnymi, niechcianymi zmianami pola `cena` w obiektach typu `Ksiazka`, a także korzystania z metody `czyCenaJestPoprawna`, która jest własnością klasy `Ksiazka` i nie powinna być stosowana przez inne klasy.

9.3.2.3 Dostęp do prywatnych pól oraz metod w klasie Ksiazka

Zauważmy jednak, że w klasie `Ksiazka` mamy dostęp zarówno do prywatnego pola `cena`, jak i do prywatnej metody `czyCenaJestPoprawna` – ma to oczywiście całkowity sens, ponieważ to właśnie klasa `Ksiazka` definiuje to pole i tę metodę, więc wewnątrz tej klasy mamy dostęp do jej prywatnych pól oraz metod.

Co by się jednak stało, gdyby klasa `Ksiazka` miała metodę, która jak argument przyjmowałaby obiekt tej samej klasy (tzn. obiekt klasy `Ksiazka`)? Czy moglibyśmy w tej metodzie odnieść się do prywatnych pól i metod?

Spójrzmy na omawiany powyżej przypadek – dodamy do klasy `Ksiazka` metodę, której argumentem będzie obiekt klasy `Ksiazka`:

Nazwa pliku: Ksiazka.java

```java
public class Ksiazka {
  public String tytul;
  public String autor;

  private double cena;

  public boolean czyTakaSamaCena(Ksiazka innaKsiazka) {
    return cena != innaKsiazka.cena;
  }

  // definicje metod ustawCene, toString, oraz czyCenaJestPoprawna,
  // zostaly pominiete
}
```

Nowa metoda klasy `Ksiazka` o nazwie `czyTakaSamaCena` przyjmuje jako argument obiekt tej samej klasy – klasy `Ksiazka`. Wewnątrz tej metody próbujemy odnieść się do prywatnego pola `cena` obiektu `innaKsiazka`, by porównać jego cenę z wartością pola `cena` obiektu, na rzecz którego tę metodę wywołamy.

Pytanie: czy powyższy kod jest poprawny?

Tak – kod skompiluje się bez problemów i będzie działał zgodnie z założeniami – spójrzmy na przykład użycia nowej metody:

Nazwa pliku: Ksiegarnia.java

```
public class Ksiegarnia {
  public static void main(String[] args) {
    Ksiazka lokomotywa = new Ksiazka();
    lokomotywa.tytul = "Lokomotywa";
    lokomotywa.autor = "Julian Tuwim";
    lokomotywa.ustawCene(29.99);

    Ksiazka ptasieRadio = new Ksiazka(); // 1
    ptasieRadio.tytul = "Ptasie Radio";
    ptasieRadio.autor = "Julian Tuwim";
    ptasieRadio.ustawCene(19.99);

    if (lokomotywa.czyTakaSamaCena(ptasieRadio)) { // 2
      System.out.println("Ksiazki kosztuja tyle samo.");
    } else {
      System.out.println("Cena ksiazek nie jest taka sama.");
    }
  }
}
```

W klasie `Ksiegarnia` tworzymy drugi obiekt typu `Ksiazka` *(1)* o nazwie `ptasieRadio`. Następnie, na rzecz obiektu `lokomotywa`, wywołujemy metodę `czyTakaSamaCena` *(2)*, której argumentem będzie utworzony właśnie obiekt `ptasieRadio`. Na podstawie wyniku wykonania tej metody wypisujemy na ekran komunikat – w tym przypadku zobaczymy:

```
Cena ksiazek nie jest taka sama.
```

Spójrzmy jeszcze raz na klasę `Ksiazka`:

```
public class Ksiazka {
  public String tytul;
  public String autor;

  private double cena;

  public boolean czyTakaSamaCena(Ksiazka innaKsiazka) {
    return cena != innaKsiazka.cena;
  }

  // definicje metod ustawCene, toString, oraz czyCenaJestPoprawna,
  // zostaly pominiete
}
```

Dlaczego metoda `czyTakaSamaCena` nie powoduje błędów kompilacji takich jak wtedy, gdy próbowaliśmy z klasy `Ksiegarnia` (w poprzednim podrozdziale) odnieść się do pola `cena` oraz metody `czyCenaJestPoprawna`?

Wynika to z faktu, że do prywatnego pola `cena` obiektu przesłanego jako argument odnosimy się z kontekstu klasy `Ksiazka`, a obiekt ten jest właśnie typu `Ksiazka`.

W obrębie tej samej klasy mamy dostęp do prywatnych pól oraz metod, gdy działamy na obiektach tej klasy.

W poprzednim rozdziale próbowaliśmy użyć prywatnego pola i metody obiektu `Ksiazka` w

metodzie `main` klasy `Ksiegarnia` – taka operacja jest zabroniona, ze względu na to, że użyliśmy modyfikatora **`private`**.

Jednakże w powyższym przypadku w metodzie `czyTakaSamaCena`, działamy na prywatnym polu wewnątrz klasy `Ksiazka` – tej samej klasie, która jest typem przesłanego jako argument obiektu. W takim przypadku, mamy prawo dostępu zarówno do prywatnych pól, jak i metod.

9.3.3 Kiedy stosować modyfikatory dostępu

Do czego są nam potrzebne modyfikatory dostępu? Czy nie moglibyśmy ustawić wszystkich pól w naszych klasach jako publiczne? Moglibyśmy. Ale czy jest to dobry pomysł?

Spójrzmy ponownie na definicję metody `czyCenaJestPoprawna` z klasy `Ksiazka`:

```
private boolean czyCenaJestPoprawna(double cenaDoSprawdzenia) {
  return cenaDoSprawdzenia > 0;
}
```

Wyobraźmy sobie, że postanawiamy zmienić metodę `czyCenaJestPoprawna`, by nie pozwalała na podanie ceny niższej niż 10 zł. Jeżeli metoda ta byłaby publiczna – czyli dalibyśmy przyzwolenie na korzystanie z niej przez inne klasy – to moglibyśmy, zmieniając zachowanie metody `czyCenaJestPoprawna`, spowodować, że inne miejsca naszego programu, które polegały na konkretnym działaniu tej metody, przestałyby działać poprawnie.

Z polami klas jest podobnie – dla przykładu, gdybyśmy w klasie `Ksiazka` zmienili dostęp z **`private`** na **`public`** pola `cena`, to nie bylibyśmy w stanie zapewnić, że każdy obiekt klasy `Ksiazka` będzie miał zawsze poprawną (tzn. powyżej zera) cenę. Gdyby definicja pola `cena` wyglądała następująco:

```
public double cena;
```

to nic nie stałoby na przeszkodzie, aby klasy, które tworzyłyby obiekty klasy `Ksiazka`, napisały kod podobny do poniższego:

```
Ksiazka pewnaKsiazka = new Ksiazka();

pewnaKsiazka.cena = -10;
```

Dzięki temu, że pole `cena` jest polem prywatnym, jedyny sposób na ustawienie jego wartości to użycie dedykowanej do tego celu metody `ustawCene`, która sprawdza, czy cena, którą chcemy ustawić, jest poprawna:

```
public void ustawCene(double nowaCena) {
  if (czyCenaJestPoprawna(nowaCena)) {
    cena = nowaCena;
  } else {
    System.out.println("Cena " + nowaCena + " jest nieprawidlowa!");
  }
}
```

To, co jest tutaj także ważne, to to, że ukrywamy przed użytkownikiem wnętrze naszej klasy, jej *implementację*, sposób, w jaki robi to, co do niej należy. Jeżeli w pewnym momencie chcielibyśmy przechowywać cenę w euro zamiast w złotówkach, to możemy zmienić metodę `ustawCene`, by wykonała wymagane w tym celu przeliczenia. Ta zmiana byłaby transparentna dla użytkownika i nie powinna spowodować żadnych problemów. Z drugiej strony, gdyby pole `cena` było polem publicznym, to odebralibyśmy sobie możliwość wszelkich zmian w naszej klasie odnośnie

przetrzymywanej ceny bez bardzo prawdopodobnego zepsucia działania wszystkich klas w naszych programch, które z klasy `Ksiazka` korzystały.

W takim razie co powinno być w naszej klasie publiczne, a co prywatne? Ogólne zasady są proste i powinniśmy się do nich stosować:

1. Wszystkie pola klas[4] powinny być prywatne – klasa powinna udostępniać publiczne metody, których będzie można używać w celu ustawiania i pobierania wartości pól klasy (o ile jest takie wymaganie) – w kolejnym podrozdziale zobaczymy przykłady takich metod.
2. Wszystkie metody, które są używane wewnętrznie przez klasę, powinny być prywatne.
3. Tylko metody, z których mają korzystać użytkownicy klas, mogą być publiczne.

Podsumowując:

> Powinno nam zależeć, by nasze klasy udostępniały tak mało ze swoich pól (czyli żadnych, zgodnie z regułą numer 1), jak to możliwe, oraz tylko te metody, które są wymagane, by nasze klasy spełniały swoje zadania (były używalne przez inne części naszego programu).

Takie podejście do tworzenia klas, w którym ukrywamy przed użytkownikami tych klas ich wewnętrzną implementację i udostępniamy zestaw metod, z których użytkownicy tych klas mają korzystać, nazywamy *enkapsulacją*. Jest to powszechny sposób projektowania klas (nie tylko w języku Java) i na rozmowach kwalifikacyjnych często pada pytanie *"Czym jest enkapsulacja?"*. Do enkapsulacji wrócimy w kolejnym podrodziale.

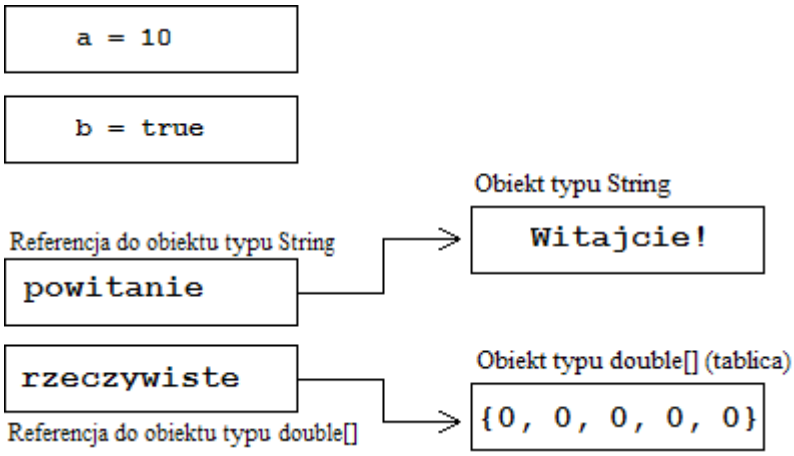
4 Wyjątkiem są pola i metody statyczne, które poznamy w jednym z kolejnych podrozdziałów.

9.4 Podsumowanie i pytania – typy prymitywne i referencyjne, modyfikatory dostępu

9.4.1 Typy prymitywne i referencyjne

- W Javie występują dwa rodzaje typów:
  - *typy prymitywne* – **boolean**, **byte**, **short**, **int**, **long**, **float**, **double**, oraz **char**. Są to elementy budujące, z których składają się *typy złożone*.
  - *typy referencyjne* – są to *typy złożone*, tworzone przez programistów poprzez pisanie klas, jak na przykład `Samochod` lub `String`. Do typów złożonych zaliczają się także tablice.
- Zmienne typów prymitywnych przechowują w pamięci konkretne wartości, natomiast zmienne typu referencyjnego – przechowują adresy obiektów w pamięci:

```
int a = 10;
boolean b = true;

String powitanie = "Witajcie!";
double[] rzeczywiste = new double[5];
```



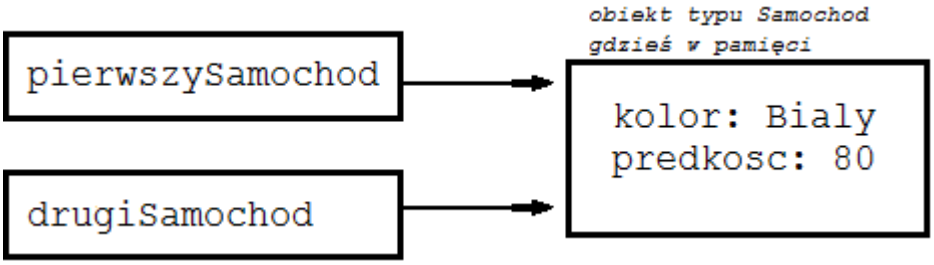
- Zmienne typów referencyjnych to nie obiekty – to *adresy* obiektów. Gdy przypisujemy do jednej zmiennej typu referencyjnego wartość innej zmiennej typu referencyjnego, to kopiujemy adres obiektu docelowego, a nie obiekt docelowy:

```
Samochod pierwszySamochod = new Samochod(); // 1
pierwszySamochod.ustawKolor("Czerwony");
pierwszySamochod.ustawPredkosc(80);

Samochod drugiSamochodd = pierwszySamochod;
drugiSamochodd.ustawKolor("Bialy");

System.out.println(pierwszySamochod);
System.out.println(drugiSamochodd);
```

Wynikiem działania programu jest:

```
Jestem samochodem! Moj kolor to Bialy, jade z predkoscia 80
Jestem samochodem! Moj kolor to Bialy, jade z predkoscia 80
```

- Powyżej mamy jeden obiekt typu `Samochod` (utworzony w linii `(1)` za pomocą słowa kluczowego **new**) oraz dwie zmienne, które na niego wskazują:



- Innymi słowy – poniższa linijka nie powoduje stworzenia kopii obiektu typu `Samochod`:

```
Samochod drugiSamochodd = pierwszySamochod;
```

  lecz przepisanie (skopiowanie) adresu obiektu, na który zmienna po prawej stronie operatora przypisania = wskazuje.

- **Pamiętajmy**: zmienne typów referencyjnych (klas) oraz zmienne tablicowe przechowują adresy obiektów tych klas i tablic.
- Zmienne typu złożonego, w przeciwieństwie do zmiennych typu prymitywnego, tworzymy za pomocą słowa kluczowego **new**:

```
Samochod pierwszySamochod = new Samochod();
```

- Tworząc nowe obiekty typu `String` nie musimy (a nawet nie powinniśmy) używać słowa kluczowego **new** – wystarczy przypisać do zmiennej łańcuch tekstowy zawarty w cudzysłowach:

```
String powitanie = "Witajcie!";
```

- Tablice także są szczególne, ponieważ zamiast używać słowa kluczowego **new**, możemy użyć skróconego zapisu z użyciem nawiasów klamrowych, w których umieszczamy elementy tablicy. Rozmiar tablicy będzie wydedukowany na podstawie liczby podanych elementów:

```
// oba sposoby na utworzenie tablicy sa poprawne
double[] rzeczywiste = new double[5];
double[] rzeczywiste2 = { 3.14, 5, -20.5 };
```

9.4.2 Typy prymitywne i referencyjne – pytania

1. Jakie typy prymitywne są zdefiniowane w Javie?
2. Czym są typy referencyjne (złożone)? Jak się je definiuje?
3. Ile obiektów jest tworzonych w metodzie `main` poniższej klasy?

```
public class PytanieZagadka {
  public int liczba;
  public int[] liczby;

  public static void main(String[] args) {
    PytanieZagadka zagadka = new PytanieZagadka();

    zagadka.liczba = 5;
    zagadka.liczby = new int[2];

    PytanieZagadka innaZagadka = zagadka;
  }
}
```

4. Który z poniższych zapisów jest niepoprawny i dlaczego?

```
PytanieZagadka zagadka = PytanieZagadka();

int[] tablica1 = new int[];

String tekst = new String("Witajcie!");

int[] tablica2 = [];

String innyTekst = "Halo?!";

int[] tablica3 = new int[] {1, 2};

int[] tablica4 = {1, 2, 3};

int[] tablica5 = new int[2] {1, 2};
```

5. Jakie wartości zostaną wypisane na ekran?

```
public class PytanieZagadka {
  public int liczba;
  public int[] liczby;

  public static void main(String[] args) {
    PytanieZagadka zagadka = new PytanieZagadka();

    zagadka.liczba = 5;
    zagadka.liczby = new int[2];

    PytanieZagadka innaZagadka = zagadka;

    int calkowita = zagadka.liczba;
    int[] tablica = innaZagadka.liczby;

    calkowita = 1;
    tablica[0] = 10;
    tablica[1] = 100;

    System.out.println(zagadka.liczba);
    System.out.println(zagadka.liczby[0]);
    System.out.println(innaZagadka.liczby[1]);
  }
}
```

9.4.3 Modyfikatory dostępu

- Modyfikatory dostępu służą do ustawienia zakresu widoczności pól oraz metod klasy.
- Modyfikatory dostępu pozwalają określić, jak będzie wyglądało użytkowanie klasy oraz kto, i w jakich okolicznościach, będzie mógł z pól i metod tej klasy korzystać.
- W Javie istnieją cztery modyfikatory dostępu:
  - **`public`**,
  - **`protected`**,
  - *domyślny* (nie mający własnego słowa kluczowego),
  - **`private`**.
- Gdy definiujemy pole (bądź metodę) z modyfikatorem **`public`**, to oznajmiamy, że to pole (lub metoda) jest dostępne dla całego *"zewnętrznego świata"* – każdy może się do takiego pola odnieść i wywołać taką metodę.
- Pola i metody zdefiniowane z modyfikatorem **`public`** to pola i metody *publiczne*.
- Używając modyfikatora **`private`**, możemy zdefiniować, że pola i metody nie mają być dostępne poza klasą – mają one być *prywatne*. Dostęp powinien być możliwy jedynie z poziomu tejże klasy i tylko ona powinna móc tymi polami i metodami zarządzać.
- Modyfikatory dostępu regulują dostęp do pól i metod klas, gdy odnosimy się do nich z innych klas.
- Gdy korzystamy z obiektu danej klasy w inne klasie, to do pól i metod publicznych możemy odnieść się bezpośrednio *(1)* *(2)* *(3)*:

```
public class Ksiazka {
  public String tytul; // 1
  public String autor; // 2

  private double cena;

  public void ustawCene(double nowaCena) { // 3
    if (czyCenaJestPoprawna(nowaCena)) {
      cena = nowaCena;
    } else {
      System.out.println("Cena " + nowaCena + " jest nieprawidlowa!");
    }
  }

  private boolean czyCenaJestPoprawna(double cenaDoSprawdzenia) {
    return cenaDoSprawdzenia > 0;
  }
}
```

```
public class Ksiegarnia {
  public static void main(String[] args) {
    Ksiazka lokomotywa = new Ksiazka();

    lokomotywa.tytul = "Lokomotywa"; // 1
    lokomotywa.autor = "Julian Tuwim"; // 2
    lokomotywa.ustawCene(29.99); // 3

    System.out.println(lokomotywa);
  }
}
```

- Jeżeli spróbujemy odnieść się do pól prywatnych, to kod w ogóle się nie skompiluje:

```
lokomotywa.cena = -10; // pole prywatne!
lokomotywa.czyCenaJestPoprawna(0); // metoda prywatna!
```

Kompilacja powyższego kodu (fragmentu metody `main` klasy `Ksiegarnia`) zakończyłaby się następującym błędem:

```
Ksiegarnia.java:13: error: cena has private access in Ksiazka
        lokomotywa.cena = -10;
                  ^
Ksiegarnia.java:14: error: czyCenaJestPoprawna(double) has private access in Ksiazka
        lokomotywa.czyCenaJestPoprawna(0);
                  ^
2 errors
```

- Korzystanie z prywatnych pól i metod spoza klasy jest zabronione, ale w ramach klasy, w której te pola i metody są zdefiniowane, możemy się do takich pól odnosić:

```
public class Ksiazka {
  public String tytul;
  public String autor;

  private double cena;

  public boolean czyTakaSamaCena(Ksiazka innaKsiazka) {
    return cena != innaKsiazka.cena; // poprawne odniesienie do prywatnego pola
  }

  // definicje metod ustawCene oraz czyCenaJestPoprawna zostaly pominiete
}
```

Powyższy kod działa poprawnie pomimo, że odnosimy się w metodzie `czyTakaSamaCena` do prywatnego pola `cena` obiektu `innaKsiazka`, ponieważ odnosimy się do tego pola z kontekstu klasy `Ksiazka`, która jest typem obiektu `innaKsiazka`.

- Czy nie moglibyśmy ustawić wszystkich pól w naszych klasach jako publiczne? Moglibyśmy, ale nie jest to dobry pomysł.
  1. Jeżeli udostępnimy prywatne pola klasy, to nie będziemy w stanie zapewnić ich poprawnej wartości – każdy będzie mógł zmienić np. pole `cena` obiektu klasy `Ksiazka` na wartość ujemną.
  2. Jeżeli udostępnimy prywatne metody klasy, to każdy będzie mógł z nich korzystać – wszelkie zmiany wprowadzane do metody będą musiały brać pod uwagę, że być może jest ona używana w setkach innych klas – każda zmiana może potencjalnie spowodować, że inne fragmenty kodu, zależne od tej metody, przestaną działać.
- Użycie modyfikatora **`private`** chroni nas przed potencjalnymi, niechcianymi zmiana pól prywatnych, a także korzystania z prywatnych metod, które są własnością klasy i nie powinny być stosowane przez inne klasy.
- Zasady dotyczące używania modyfikatorów dostępu **`public`** i **`private`**:
  1. Wszystkie pola klas powinny być prywatne.
  2. Wszystkie metody, które są używane wewnętrznie przez klasę, powinny być prywatne.
  3. Tylko metody, z których mają korzystać użytkownicy klas, mogą być publiczne.

- Nasze klasy powinny udostępniać tak mało ze swoich pól (czyli żadnych, zgodnie z regułą numer 1), jak to możliwe, oraz tylko te metody, które są wymagane, by nasze klasy spełniały swoje zadania (były używalne przez inne części naszego programu).
- Podejście do tworzenia klas, w którym ukrywamy przed użytkownikami tych klas ich wewnętrzną implementację, nazywamy *enkapsulacją*.

9.4.4 Modyfikatory dostępu – pytania

1. Do czego służą modyfikatory dostępu w Javie?
2. Czy modyfikatory dostępu można używać tylko podczas definicji pól klasy?
3. Czym różnią się modyfikatory `public` i `private`?
4. Kto ma dostęp do prywatnych pól i metod klasy?
5. Czy moglibyśmy wszystkie pola i metody zawsze definiować z dostępem `public`? Jeżeli tak, to czy jest to dobry pomysł?
6. Czy poniższy kod się skompiluje?

```
public class KlasaZagadka {
  private int liczba;

  public static void main(String[] args) {
    KlasaZagadka obiekt = new KlasaZagadka();

    obiekt.liczba = 5;

    System.out.println(obiekt.liczba);
  }
}
```

7. Mając następujące klasy, do jakich pól i metody typu `Osoba` mamy dostęp w miejscach zaznaczonych jako `(1?)` i `(2?)`?

```
public class Osoba {
  public String nazwisko;
  private int wiek;

  public void ustawWiek(int wiekOsoby) {
    wiek = wiekOsoby;
  }

  public boolean czyWTymSamymWieku(Osoba innaOsoba) {
    // 1?
  }

  private void wypiszNazwisko() {
    System.out.println(nazwisko);
  }
}
```

```
public class UzycieTypuOsoba {
  public static void main(String[] args) {
    Osoba osoba = new Osoba();
    // 2?
  }
}
```

8. Czy poniższa klasa `KlasaZagadka` się skompiluje? Czy klasa `UzycieZagadki` się skompiluje?

```
public class KlasaZagadka {
  private int liczba;
}
```

```
public class UzycieZagadki {
  public static void main(String[] args) {
    KlasaZagadka obiekt = new KlasaZagadka();

    obiekt.liczba = 5;

    System.out.println(obiekt.liczba);
  }
}
```

9. Jakie modyfikatory dostępu możemy, a jakie *powinniśmy* (i dlaczego), wstawić na miejsca znaków zapytania, aby kod był poprawny?

```
public class InnaZagadka {
  ? int liczba;

  ? void ustawLiczbe(int wartosc) {
    liczba = tajnyAlgorytm(wartosc);
  }

  ? int tajnyAlgorytm(int x) {
    return (x + 2) * 11;
  }

  ? String toString() {
    return "Tajna liczba to " + liczba;
  }
}
```

```
public class UzycieInnejZagadki {
  public static void main(String[] args) {
    InnaZagadka obiekt = new InnaZagadka ();

    obiekt.ustawLiczbe(10);

    System.out.println(obiekt);
  }
}
```

9.5 Pola klas

Jak już wiemy, pola klas to pewne właściwości, które opisują obiekty danej klasy. Dla przykładu, klasa `Ksiazka` miała pola `tytul`, `autor`, oraz `cena`, a klasa `Samochod` – pola `kolor` i `predkosc`.

Każdy obiekt klasy (tzn. każda jej instancja, czyli egzemplarz) posiada własny zestaw zdefiniowanych w tej klasie pól – ustawienie `koloru` obiektu `czerwonySamochod` nie ma wpływu na `kolor` samochodu `zielonySamochod`:

```
Samochod zielonySamochod = new Samochod();
zielonySamochod.ustawKolor("Zielony");

Samochod czerwonySamochod = new Samochod();
czerwonySamochod.ustawKolor("Czerwony");
```

Pola klas to po prostu zmienne – mają one swój typ oraz nazwę, oraz mogą mieć różnego rodzaju modyfikatory. Przykładem są modyfikatory dostępu **`public`** oraz **`private`**, które poznaliśmy w poprzednim podrozdziale. Wyznaczają one, czy pole klasy jest publiczne (dostępne dla całego zewnętrzenego świata), czy prywatne (dostępne tylko wewnątrz klasy).

Istnieje wiele modyfikatorów, którymi można poprzedzić definicję pól klasy – kilka z nich jeszcze poznamy, a pozostałe pominiemy, ponieważ są związane z bardziej zaawansowanym wykorzystaniem języka Java, jak na przykład modyfikatory **`synchronized`** *oraz* **`transient`**.

Typem pól klasy mogą być zarówno typy prymitywne (na przykład **`int`** czy **`boolean`**), jak i typy złożone, czyli klasy zdefiniowane przez nas lub innych programistów. Korzystaliśmy już z tej możliwości definiując pola typu `String` w klasie `Ksiazka`, która miała pole `autor` oraz w klasie `Samochod`, która miała pole `kolor`.

9.5.1 Pola klas a zmienne definiowane w metodach

Chociaż pola klas to zmienne, to jest kilka istotnych różnic pomiędzy polami klas a zmiennymi definiowanymi wewnątrz metod.

9.5.1.1 Zapamiętywanie wartości

W rozdziale o metodach dowiedzieliśmy się, że zmienne zdefiniowane w metodach tracą swoje wartości po zakończeniu metody – są one lokalne dla metody, w której zostały zdefiniowane.

Pola klas, natomiast, zapamiętują swoje wartości. Wielokrotnie korzystaliśmy z tej właściwości w poprzednich rozdział, na przykład, w poniższym programie:

Fragment z pliku Ksiegarnia.java

```
Ksiazka lokomotywa = new Ksiazka();

lokomotywa.tytul = "Lokomotywa"; // 1
lokomotywa.autor = "Julian Tuwim"; // 2
lokomotywa.ustawCene(29.99); // 3

System.out.println(lokomotywa); // 4
```

Ustawiamy trzy pola obiektu `lokomotywa` – dwa z nich bezpośrednio `(1)` `(2)`, a jedno poprzez metodę `ustawCene` `(3)`:

Fragment z pliku Ksiazka.java

```
public void ustawCene(double nowaCena) {
  if (czyCenaJestPoprawna(nowaCena)) {
    cena = nowaCena;
  } else {
    System.out.println("Cena " + nowaCena + " jest nieprawidlowa!");
  }
}
```

Na końcu pierwszego z powyższych fragmentów kodu wypisujemy na ekran tekstową reprezentację obiektu `lokomotywa` *(4)* i widzimy, że wartość pola `cena` ustawiona w metodzie `ustawCene`, po zakończeniu tej metody, nadal jest przechowywana (jest "zapamiętana") w polu `cena` obiektu `lokomotywa`:

```
Ksiazka o tytule Lokomotywa, ktorej autorem jest Julian Tuwim, kosztuje 29.99
```

Dopóki obiekt istnieje, wartości nadane jego polom są zachowane.

W dalszej częsci rozdziału o klasach dowiemy się, co wyznacza długość istnienia tworzonych przez nas obiektów.

9.5.1.2 Inicjalizacja i domyślne wartości typów prymitywnych

Podobnie jak w przypadku zmiennych, polom klas możemy nadać wstępne wartości, tzn. zainicjalizować je. Utworzone obiekty będą miały od razu przypisane te wartości do odpowiednich pól:

Nazwa pliku: Statek.java

```
public class Statek {
  public int liczbaZagli = 2; // 1
  public String nazwa = "nienazwany statek"; // 2

  public String toString() {
    return "Statek o nazwie '" + nazwa + "' ma " + liczbaZagli + " zagle.";
  }

  public static void main(String[] args) {
    Statek nieznanyStatek = new Statek(); // 3

    Statek hammond = new Statek(); // 4
    hammond.nazwa = "Hammond"; // 5

    System.out.println(nieznanyStatek);
    System.out.println(hammond);
  }
}
```

Klasa `Statek` ma dwa pola publiczne `liczbaZagli` oraz `nazwa` – oba pola inicjalizujemy wartościami, odpowiednio, `2` *(1)* oraz `"nienazwany statek"` *(2)*. Tworzymy dwa nowe obiekty klasy `Statek` *(3)* *(4)*. Drugiemy z nich przypisujemy nową nazwę *(5)*, a następnie wypisujemy na ekran ich tekstowe reprezentacje, w wyniku czego na ekranie zobaczymy:

```
Statek o nazwie 'nienazwany statek' ma 2 zagle.
Statek o nazwie 'Hammond' ma 2 zagle.
```

Jak widać, pomimo, że nie ustawiliśmy pól `liczbaZagli` obu obiektów po ich utworzeniu, oraz

pola `nazwa` pierwszego statku, to i tak oba statki mają już w tych polach wartości – te, którymi je zainicjalizowaliśmy w *(1)* i *(2)*.

Z rozdziału o zmiennych wiemy, że w metodach nie możemy korzystać ze zmiennych, którym nie nadamy wstępnej wartości:

```
public class UzycieNiezainicjalizowanejZmiennej {
  public static void main(String[] args) {
    int x;

    System.out.println("Wartosc x wynosi: " + x);
  }
}
```

Próba kompilacji powyższego programu kończy się następującym błędem:

```
UzycieNiezainicjalizowanejZmiennej.java:5: error: variable x might not have
been initialized
      System.out.println("Wartosc x wynosi: " + x);
                                                 ^
1 error
```

Zobaczmy co się stanie, gdy spróbujemy odnieść się do pól klasy, którym nie nadaliśmy żadnych wartości:

Nazwa pliku: Adres.java

```
public class Adres {
  private String miasto;
  private String ulica;
  private int nrDomu;
  private int nrMieszkania;

  public String toString() {
    return "Adres " + miasto + ", ulica " + ulica +
      " " + nrDomu + " m. " + nrMieszkania;
  }

  public static void main(String[] args) {
    Adres mojAdres = new Adres(); // 1
    System.out.println(mojAdres); // 2
  }
}
```

Klasa `Adres` przechowuje dane adresowe w czterech polach – dwóch stringach i dwóch liczbach typu `int`. Tworzymy nowy obiekt tej klasy *(1)*, a następnie, bez przypisania wartości do pól tego obiektu, wypisujemy na ekran jego tekstową reprezentację *(2)* – spowoduje to automatyczne użycie metody `toString`, która używa pól `miasto`, `ulica`, `nrDomu`, oraz `nrMieszkania`.

Jaki będzie efekt próby kompilacji powyższego kodu?

Kod skompiluje się bez błędów, a po uruchomieniu zobaczymy na ekranie:

```
Adres null, ulica null 0 m. 0
```

Skąd w wyniku wzięły się zera jako `nrDomu` oraz `nrMieszkania`, czym jest dziwny tekst *`null`*, a także dlaczego powyższy kod w ogóle się skompilował, skoro nie nadaliśmy polom obiektu `mojAdres` żadnych wartości?

Pola klas różnią się tym od zmiennych lokalnych (zdefiniowanych w metodach),

że mają wartości domyślne, nadawane im zawsze automatycznie, jeżeli my, jako programiści, nie zainicjalizujemy ich początkowymi wartościami (tak jak zrobiliśmy to w przypadku pól klasy `Statek`).

Każdy z 8 typów prymitywnych ma swoją wartość domyślną:

Typ	Wartość domyślna
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000'`
`boolean`	**`false`**

Wartość domyślna liczb typu **`byte`**, **`short`**, oraz **`int`**, to **zero**. Wartości typu **`long`** definiujemy dopisując na końcu liczby znak **L** (jak `Long`).

Domyślne wartości pól typów rzeczywistych to, odpowiednio, `0.0f` i `0.0d` – **f** używamy dla oznaczenia liczb o mniejszej precyzji, typu **`float`**, a literą **d** kończymy wartości typu **`double`** (chociaż nie jest to wymagane – liczby rzeczywiste w Javie są domyślnie typu **`double`**).

Pola typu **`boolean`** mają zawsze domyślnie ustawioną wartość **`false`**. Pola typu **`char`**, natomiast, mają wartość `'\u0000'`, która jest zapisem znaku, który reprezentuje *"brak wartości"* w systemie *Unicode* (jest to system, który definiuje, jak zapisywać znaki ze wszystkich alfabetów języków na świecie w świecie komputerów).

9.5.1.3 Wartość null

A jaką wartość domyślną mają pola typów referencyjnych (tzn. typów złożonych, definiowanych przez klasy), jak na przykład w klasie `Adres` pola `miasto` i `ulica`, które są typu `String`?

Ta wartość to **`null`**, jak widać na ekranie po wypisaniu obiektu `mojAdres` w przykładzie z poprzedniego podrozdziału:

```
Adres null, ulica null 0 m. 0
```

Null w programowaniu to wartość szczególna – występuje w większości języków programowania i w bazach danych – jego znaczenie to **"brak wartości"** (nie mylić z pustą wartością!).

Pola klas typu referencyjnego mają domyślną wartość **`null`**, jeżeli nic do nich nie przypiszemy – oznacza to po prostu, że nie wskazują one na żaden obiekt. Jak pamiętamy z rozdziału 9.2.1. Przechowywane wartości, zmienne typów złożonych (czyli klas) to tak naprawdę referencje (odniesienia) do utworzonych obiektów. Gdy zmienna typu złożonego nie wskazuje na żaden obiekt, oznacza to, że jej wartość to **`null`**. Przypomnijmy także, iż tablice to także typ złożony i pola klas typu tablicowego także mają domyślną wartość **`null`**.

Próba odniesienia się do pola klasy lub wywołanie metody na rzecz obiektu, który *jest nullem* (nie ma przypisanego żadnego obiektu) kończy się błędem wykonywania programu. Spójrzmy na poniższy przykład:

Nazwa pliku: Gora.java

```
public class Gora {
  public int wysokosc;
  public String nazwa;

  public double policzWysokoscWKm() {
    return wysokosc / 1000.0;
  }

  public static void main(String[] args) {
    Gora pewnaGora = new Gora();

    System.out.println("Wysokosc gory: " + pewnaGora.wysokosc); // 1
    System.out.println("Nazwa gory: " + pewnaGora.nazwa); // 2

    // blad wykonywania programu!
    System.out.println("Nazwa gory wielkimi literami: " +
        pewnaGora.nazwa.toUpperCase()); // 3
  }
}
```

Powyższy program kompiluje się bez problemów, ale po uruchomieniu jego działanie kończy się błędem – na ekranie zobaczymy następujący komunikat:

```
Wysokosc gory: 0
Nazwa gory: null
Exception in thread "main" java.lang.NullPointerException
     at Gora.main(Gora.java:16)
```

W metodzie `main` tworzymy nowy obiekt typu `Gora`. Wypisujemy najpierw na ekran wartości pól tego obiektu: `wysokosc` *(1)* oraz `nazwa` *(2)*. Ważne jest tutaj to, że tym polom nie nadajemy żadnych wartości – zostaną więc one zainicjalizowane wartościami domyślnymi podczas tworzenia obiektu typu `Gora` – będą to wartości: `0` dla pola `wysokosc` oraz **`null`** dla pola `nazwa`.

Do tego momentu program działa poprawnie, ale w kolejnej linii *(3)* probujemy wywołać metodę `toUpperCase` na rzecz pola typu `String` o nazwie `nazwa` – skoro do tego pola nie przypisaliśmy żadnej wartości, to na rzecz jakiego obiektu metoda `toUpperCase` ma zostać wywołana?

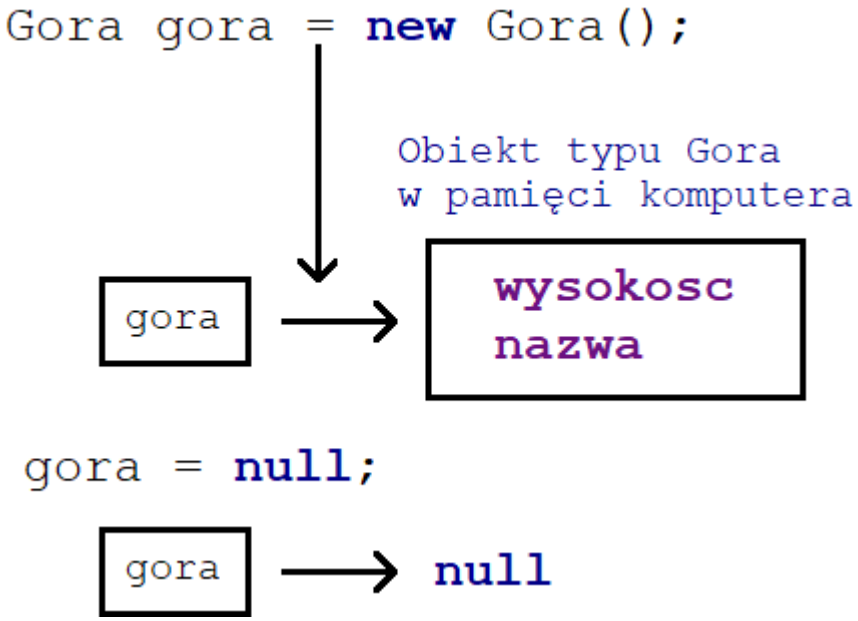
To jest właśnie powodem błędu (który widać powyżej) wykonania naszego programu – próbujemy wywołać metodę `toUpperCase` na nieistniejącym obiekcie – jest to "słynny" błąd *`Null Pointer Exception`* – będziemy się na niego natykać od czasu do czasu. Zawsze, gdy zobaczymy ten błąd, będzie to świadczyło o tym, że próbujemy działać na obiekcie, który *jest nullem*.

Wartością **`null`** można operować jak każdą inną wartością – można ją, na przykład, przypisać do zmiennej typu złożonego:

Fragment pliku: Gora.java

```
public static void main(String[] args) {
  Gora pewnaGora = new Gora();

  System.out.println("Wysokosc gory: " + pewnaGora.wysokosc);
  System.out.println("Nazwa gory: " + pewnaGora.nazwa);

  // zakomentowane, bo powoduje blad NullPointerException!
  // System.out.println("Nazwa gory wielkimi literami: " +
  //     pewnaGora.nazwa.toUpperCase());

  pewnaGora = null; // 1
  System.out.println("Wysokosc gory w kilometrach: " +
      pewnaGora.policzWysokoscWKm()); // 2
}
```

W zmienionej metodzie `main` z klasy `Gora`, po wypisaniu domyślnych wartości pól obiektu `pewnaGora`, przypisujemy do tej zmiennej `null` *(1)* – od tej pory, `pewnaGora` nie wskazuje już na żaden obiekt, a do obiektu typu `Gora`, który utworzyliśmy na początku metody `main`, tracimy dostęp! Nic już na ten obiekt nie wskazuje:



Po przypisaniu `null`a do obiektu `pewnaGora`, próbujemy wypisać na ekran wysokość w kilometrach *(2)*. Jaki będzie teraz efekt uruchomienia tego programu?

Ponownie zobaczymy na ekranie błąd `Null Pointer Exception` – w końcu próbujemy wywołać metodę `policzWysokoscWKm` na rzecz *nullowego* obiektu `pewnaGora`!

```
Wysokosc gory: 0
Nazwa gory: null
Exception in thread "main" java.lang.NullPointerException
     at Gora.main(Gora.java:22)
```

Czy możemy w takim razie w jakiś sposób uchronić się przed działaniem na nullowych obiektach?

Tak – zmienne typu referencyjnego można przyrównywać do wartości `null`, aby sprawdzić, czy wskazują na jakiś obiekt, czy nie – możemy poprawić poprzedni przykład, by korzystał tej funkcjonalności:

Fragment pliku: Gora.java

```
pewnaGora = null;

if (pewnaGora != null) { // 1
  System.out.println("Wysokosc gory w kilometrach: " +
      pewnaGora.policzWysokoscWKm());
} else { // 2
  System.out.println("Obiekt pewnaGora to null!");
}
```

Do poprzedniego przykładu dodaliśmy instrukcję `if` w której sprawdzamy, czy `pewnaGora` jest różna od `null`a *(1)*, czyli sprawdzamy "*czy* `pewnaGora` *wskazuje na jakiś obiekt?*". Jeżeli tak, to wypiszemy na ekran wysokość w kilometrach, a jeżeli nie *(2)*, to wypiszemy drugi komunikat.

Porównując zmienne typu referencyjnego do wartości **null** możemy używać zarówno operatora != (nierówne) jak i operatora == (równe). Pierwszy sprawdzi w tym kontekście, czy zmienna typu referencyjnego pokazuje na jakiś obiekt (bo pokazuje na coś innego niż **null**), a drugi (operator ==) odpowie na pytanie: *"Czy zmienna pokazuje na nic?"*:

Dane	nazwa == null	nazwa != null
`String nazwa = "Giewont";`	**false**	**true**
`String nazwa;` `nazwa = null;`	**true**	**false**

W efekcie działania programu z dodanym powyższym warunkiem, na ekranie zobaczymy:

```
Wysokosc gory: 0
Nazwa gory: null
Obiekt pewnaGora to null!
```

Tym razem program nie zakończył się błędem, ponieważ zabezpieczyliśmy się przed potencjalną próbą działania na nullowym obiekcie.

Pytanie: co by się stało, gdybyśmy spróbowali przypisać do zmiennej typu prymitywnego wartość **null**? Zmienne typu prymitywnego nigdy nie mogą mieć wartości **null** – próba przypisania **null** do zmiennej, na przykład, typu **int**, spowodowałaby błąd kompilacji:

Nazwa pliku: NullZmiennaPrymitywna.java

```
public class NullZmiennaPrymitywna {
  public static void main(String[] args) {
    int x = null; // 1
    System.out.println(x);
  }
}
```

W powyższym programie próbujemy przypisać do zmiennej typu **int** wartość **null** *(1)*. Próba kompilacji powyższego programu zakończy się następującym błędem:

```
NullZmiennaPrymitywna.java:3: error: incompatible types: <null> cannot be
converted to int
      int x = null;
              ^
1 error
```

Zapamiętajmy: pola typów prymitywnych (jak **int** i **boolean**) mają swoje domyślne wartości, natomiast pola typów złożonych (czyli klas) mają wartość domyślną **null**, co oznacza, że nie wskazują one na żaden obiekt.

Opisane powyżej zagadnienia związane z wartościami domyślnymi oraz wartością **null** są ważne i mogą na początku wydawać się skomplikowane, dlatego spojrzymy na jeszcze jeden przykład, w którym przećwiczymy sobie te zagadnienia.

W kolejnych fragmentach kodu będziemy bazować na poniższej klasie `Uczen`:

Nazwa pliku: Uczen.java

```
public class Uczen {
  public String nazwisko;
  public int[] oceny;
  public int rokUrodzenia;

  public double policzSredniaOcen() {
    double sumaOcen = 0;
    for (int i = 0; i < oceny.length; i++) {
      sumaOcen += oceny[i];
    }
    return sumaOcen / oceny.length;
  }
}
```

Jakiego rodzaju pola zawiera ta klasa? Ile jest pól typu prymitywnego, a ile złożonego?

Klasa `Uczen` ma jedno pole typu prymitywnego – `rokUrodzenia`, oraz dwa pola typów złożonych `nazwisko` i `oceny`. Jak już wspominaliśmy, typy tablicowe także są typami referencyjnymi (złożonymi), więc zarówno pole `nazwisko` typu `String`, jak i pole `oceny` typu `int`[], będą miały domyślną wartość `null`.

Skoro pole `oceny` ma domyślnie przypisaną wartość `null`, to czy powyższy kod klasy `Uczen` jest poprawny? Czy moglibyśmy w nim coś poprawić?

Spróbujmy policzyć średnią ocen używając metody `policzSredniaOcen` obiektu typu `Uczen`:

Nazwa pliku: Szkola.java

```
public class Szkola {
  public static void main(String[] args) {
    Uczen pewnyUczen = new Uczen();

    System.out.println("Srednia ocen: " + pewnyUczen.policzSredniaOcen());
  }
}
```

Co zostanie wypisane na ekranie?

Wykonanie naszego programu zakończy się błędem *Null Pointer Exception*, który już poznaliśmy:

```
Exception in thread "main" java.lang.NullPointerException
     at Uczen.policzSredniaOcen(Uczen.java:11)
     at Szkola.main(Szkola.java:7)
```

Co spowodowało taki wynik? Przyjrzyjmy się jeszcze raz metodzie `policzSredniaOcen`:

```
public double policzSredniaOcen() {
  double sumaOcen = 0;
  for (int i = 0; i < oceny.length; i++) { // 1
    sumaOcen += oceny[i];
  }
  return sumaOcen / oceny.length;
}
```

Problem wystąpił w pierwszej linii pętli `for` *(1)*. Nie przypisaliśmy żadnej wartości polu `oceny` obiektu `pewnyUczen`, więc `oceny` nie wskazują na żadną tablicę, lecz na `null`. Gdy wywołujemy metodę `policzSredniaOcen` na rzecz obiektu `pewnyUczen`, to w pętli *(1)* próbujemy odczytać liczbę elementów tablicy `oceny`, która nie wskazuje na żadną tablicę! Dlatego wykonanie programu zakończyło się błędem.

Jak już wiemy, może sprawdzić, czy zmienna "wskazuje na nic" (`null`) sprawdzając, czy jest równa `null` – poprawiona wersja metody `policzSredniaOcen` wygląda następująco:

```
public double policzSredniaOcen() {
  if (oceny == null) { // 1
    return 0;
  }

  double sumaOcen = 0;

  for (int i = 0; i < oceny.length; i++) {
    sumaOcen += oceny[i];
  }

  return sumaOcen / oceny.length;
}
```

Dodaliśmy warunek sprawdzający, czy `oceny` *są nullem* `(1)`. Jeżeli tak, to kończymy działanie metody, używając instrukcji `return`, by zwrócić `0`. Teraz, po uruchomieniu klasy `Szkola`, na ekranie zobaczymy `"Srednia ocen: 0.0"`.

A co zobaczymy na ekranie, gdy spróbujemy wypisać rok urodzenia ucznia?

```
Uczen pewnyUczen = new Uczen();

System.out.println("Rok urodzenia: " + pewnyUczen.rokUrodzenia);
```

Jako, że `rokUrodzenia` to zmienna typu prymitywnego, to zostaje jej nadana wartość domyślna – dla typu `int` jest to `0`. Na ekranie zobaczymy:

```
Rok urodzenia: 0
```

Spróbujemy wypisać nazwisko ucznia wielkimi literami:

```
Uczen pewnyUczen = new Uczen();

System.out.println("Nazwisko: " + pewnyUczen.nazwisko.toUpperCase());
```

Tym razem program ponownie zakończy się błędem wykonania i ponownie zobaczymy błąd `Null Pointer Exception` – nie nadaliśmy polu `nazwisko` żadnej wartości – jako, że jest to pole typu złożonego (`String`), będzie automatycznie zainicjalizowane wartością `null`. W powyższym fragmencie kodu próbujemy wywołać metodę `toUpperCase` na *nullowym* obiekcie, co jest niedozowolne, o czym świadczy błąd wykonania programu:

```
Exception in thread "main" java.lang.NullPointerException
     at Szkola.main(Szkola.java:9)
```

Spróbujmy na koniec nadać wartości polom obiektu `pewnyUczen`:

Nazwa pliku: Szkola.java

```
public class Szkola {
  public static void main(String[] args) {
    Uczen pewnyUczen = new Uczen();

    pewnyUczen.nazwisko = "Kowalski";
    pewnyUczen.rokUrodzenia = 2000;
    pewnyUczen.oceny = new int[] { 4, 5, 5, 4, 5};

    System.out.println("Nazwisko: " + pewnyUczen.nazwisko.toUpperCase());
    System.out.println("Rok urodzenia: " + pewnyUczen.rokUrodzenia);
    System.out.println("Srednia ocen: " + pewnyUczen.policzSredniaOcen());
  }
}
```

Tym razem nie zobaczymy na ekranie żadnych błędów:

```
Nazwisko: KOWALSKI
Rok urodzenia: 2000
Srednia ocen: 4.6
```

9.5.1.4 Brak wymagania definicji przed użyciem

W rozdziale o instrukcjach warunkowych dowiedzieliśmy się, że zmienne muszą być zdefiniowane w metodach przed pierwszym użyciem zmiennej:

Nazwa pliku: Rozdzial_04__Instrukcje_warunkowe/UzyciePrzedDefinicja.java

```
public class UzyciePrzedDefinicja {
  public static void main(String[] args) {
    System.out.println("Liczba = " + liczba);

    int liczba = 5;
  }
}
```

Próba kompilacji powyższego programu kończy się następującym błędem:

```
UzyciePrzedDefinicja.java:3: error: cannot find symbol
        System.out.println("Liczba = " + liczba);
                                        ^
  symbol:   variable liczba
  location: class UzyciePrzedDefinicja
```

Z drugiej jednak strony, w rozdziale o metodach dowiedzieliśmy się, że metody w klasach nie muszą być w zdefiniowane w kolejności użycia – kompilator analizuje cały plik źródłowy i wie, jakie metody utworzyliśmy:

Nazwa pliku: WypiszSume.java

```
public class WypiszSume {
  public static void main(String[] args) {
    wypiszSume(100, 200);
    wypiszSume(-5, -20);
    wypiszSume(0, 0);
  }

  public static void wypiszSume(int a, int b) {
    System.out.println(a + b);
  }
}
```

Kompilator nie protestuje widząc linię:

```
wypiszSume(100, 200);
```

ponieważ, po analizie całego pliku, wie, że `wypiszSume` jest metodą z dwoma argumentami, która nie zwraca żadnej wartości.

Pytanie: a jak jest z polami klasy? Czy muszą one być zdefiniowane przed użyciem? **To zależy.**

Pola klas mogą być zdefiniowane w dowolnym miejscu klasy i będą dostępne we wszystkich metodach tej klasy. Z drugiej jednak strony, pola klasy muszą być zdefiniowane przed użyciem ich do nadania wartości innym polom tej klasy. Spójrzmy na dwa przykłady, które to wyjaśnią:

Nazwa pliku: Punkt.java

```
public class Punkt {
  public void ustawX(int wartoscX) {
    x = wartoscX; // 1
  }

  private int x; // 2

  public void ustawY(int wartoscY) {
    y = wartoscY; // 3
  }

  public String toString() {
    return "X: " + x + ", Y: " + y; // 4
  }

  private int y; // 5
}
```

Klasa `Punkt` jest poprawna i kompiluje się bez problemów pomimo, że pola `x` *(2)* oraz `y` *(5)* zdefiniowane są po metodach, które z nich korzystają *(1)* *(3)* *(4)*. Powyższy kod nie stanowi problemów, ponieważ kompilator analizuje całą klasę i wie, czym są pola `x` oraz `y`, do których odosimy się w metodach tej klasy.

Spójrzmy na zmodyfikowany przykład klasy `Punkt`, w której inicjalizujemy oba pola wartościami:

```
public class PunktBlad {
  private int x = y; // blad! (1)
  private int y = 0;

  public void ustawX(int wartoscX) {
    x = wartoscX;
  }

  public void ustawY(int wartoscY) {
    y = wartoscY;
  }

  public String toString() {
    return "X: " + x + ", Y: " + y;
  }
}
```

Próba skompilowania powyższej klasy kończy się następującym błędem:

```
Punkt.java:2: error: illegal forward reference
  private int x = y;
                  ^
```

Do pola `x` próbujemy przypisać wartość pola `y` – jednakże pole `y` dla kompilatora nie ma jeszcze w tym momencie nadanej żadnej wartości i próba odniesienia się do niego w miejscu `(1)` jest traktowana jako błąd – kompilator protestuje.

Podsumowując: pola klas nie muszą być umieszczone przed metodami, w których z nich korzystamy, ale jeżeli są one używane do inicjalizacji innych pól, to muszą wystąpić przed nimi.

To, że możemy umieszczać pola w różnych miejscach w klasie nie oznacza, że powinniśmy – ogólna konwencja mówi o tym, iż:

- publiczne pola powinny być zawsze na początku definicji klasy – ale jak już sobie powiedzieliśmy w rozdziale o modyfikatorach dostępu – takich pól nie powinno w ogóle być w naszych klasach (poza statycznymi polami, o których opowiemy sobie na końcu rozdziału o klasach),
- po nich powinny następować pola prywatne,
- następnie metody publiczne metody,
- na końcu powinny znajdować się metody prywatne.

Stosujmy się do powyższej konwencji – ułatwi to analizę naszego kodu przez innych programistów, a także nas samych, w przyszłości!

Dlaczego taka kolejność?

Pola trzymane w jednym miejscu, na początku klasy, są łatwe do znalezienia. Metody publiczne są zawsze punktem wejścia i sposobem na interakcję z obiektami danej klasy i to one zazwyczaj najbardziej nas interesują. Metody prywatne stanowią wewnętrzną implementację logiki w klasie i zazwyczaj nie mamy potrzeby ich analizować (o ile, oczywiście, nie szukamy błędu lub nie mamy za zadanie zmodyfikować daną klasę).

9.5.2 Gettery i settery oraz enkapsulacja

Z pojęciem *enkapsulacji* zetknęliśmy się w poprzednim podrozdziale o modyfikatorach dostępu.

Enkapsulacja to taki sposób tworzenia klas, w którym przed światem zewnętrznym ukrywamy wewnętrzną implementację klas, udostępniając zamiast tego zestaw metod, z których użytkownicy klas mają korzystać. Enkapsulacja jest ogólnie przyjętym sposobem na tworzenie klas w języku Java.

Aby zapewnić enkapsulację, pola naszych klas zawsze powinny być prywatne, tzn. powinniśmy korzystać z modyfikatora dostępu **`private`** podczas ich definiowania. W związku z tym będziemy potrzebować zestaw metod, za pomocą których będzie można korzystać z pól naszych klas – takie metody nazywamy *akcesorami*.

Metody-*akcesory* dzielą się na dwa rodzaje ze względu na ich zadanie:

- *gettery* – służą do pobierania wartości danego pola,
- *settery* – służą do ustawiania wartości danego pola.

Oba rodzaje akcesorów mają konwencje nazewnicze – metody-*gettery* zaczynamy od słowa *get*, a metody-*settery* – od słowa *set*.

Druga część nazwy stanowi zawsze nazwa ustawianego/pobieranego pola, zapisana, znanym nam już, camelCasem. Dla przykładu, mając pola `kolor` oraz `predkosc` z klasy `Samochod`, moglibyśmy napisać do nich następujące akcesory:

1. `getKolor` oraz `setKolor`
2. `getPredkosc` oraz `setPredkosc`

Od tej pory nie będziemy już korzystać z publicznych pól w naszych klasach – zawsze będziemy pisać do nich *gettery* oraz *settery*.

> Pamiętajmy, że nazwy klas, zmiennych, metod itd. powinniśmy nazywać po angielsku.
>
> W kursie używamy polskich nazw ze względu na wygodę. We wszelkich projektach komercyjnych, jak i open source, "językiem" kodu jest język angielski.

Spójrzmy na przykład klasy z setterami i getterami:

Nazwa pliku: Produkt.java

```
public class Produkt {
  private double cena; // 1
  private String nazwa; // 2

  public void setCena(double nowaCena) { // 3
    cena = nowaCena;
  }

  public double getCena() { // 4
    return cena;
  }

  public void setNazwa(String nowaNazwa) { // 5
    nazwa = nowaNazwa;
  }

  public String getNazwa() { // 6
    return nazwa;
  }

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

Klasa `Produkt` zawiera dwa *prywatne* pola: `cena` `(1)` oraz `nazwa` `(2)`. Dla każdego z tych pól przygotowaliśmy po dwa akcesory – po jednym setterze `(3)` `(5)` oraz po jednym getterze `(4)` `(6)`. Są to bardzo proste metody – ich jedynym zadaniem jest albo ustawienie danego pola, albo zwrócenie jego wartości. Klasa zawiera także użyteczną metodę `toString`. Spróbujmy użyć klasy `Produkt`:

Nazwa pliku: Sklep.java

```
public class Sklep {
  public static void main(String[] args) {
    Produkt czeresnie = new Produkt();
    Produkt herbata = new Produkt();

    czeresnie.setCena(8.0); // 1
    czeresnie.setNazwa("Czeresnie"); // 2

    herbata.setCena(12.0); // 1
    herbata.setNazwa("Herbata czarna"); // 2

    // 3
    System.out.println("Nazwa pierwszego produktu to: " + czeresnie.getNazwa());
    System.out.println("Cena pierwszego produktu to: " + czeresnie.getCena());
    System.out.println(herbata);
  }
}
```

W klasie `Sklep` tworzymy dwa obiekty typu `Produkt`. Za pomocą setterów ustawiamy ceny obu produktów `(1)` oraz ich nazwy `(2)`.

Wykorzystując gettery, wypisujemy na ekran nazwę oraz cenę obiektu `czeresnie` `(3)`. Na końcu wypisujemy informacje o obiekcie `herbata` (wykorzystana zostanie metoda `toString`) – na ekranie zobaczymy:

```
Nazwa pierwszego produktu to: Czeresnie
Cena pierwszego produktu to: 8.0
Produkt o nazwie Herbata czarna kosztuje 12.0
```

Zauważmy, że nasze settery to jedyny sposób na ustawienie wartości w polach `cena` oraz `nazwa` – próba bezpośredniego odniesienia się do któregoś z tych pól zakończyłaby się błędem kompilacji:

```
czeresnie.cena = -20;
```

```
Sklep.java:6: error: cena has private access in Produkt
      czeresnie.cena = -20;
               ^
```

Ukrywając pola `nazwa` oraz `cena` zostawiamy sobie możliwość zmiany sposobu zapisywania i odczytywania tych pól – może w przyszłości będziemy chcieli wykonywać na polach walidację, może będziemy chcieli gdzieś zapisać informację o ustawianej cenie, lub przechowywać ją w wielu walutach? Gdybyśmy udostępnili je używając modyfikatora **`public`**, to odebralibyśmy sobie pole manewru. Powyższy przykład jest prosty, ale wyobraźmy sobie programy z tysiącami klas – zapewnienie, że dostęp do pól klas odbywa się poprzez settery i gettery staje się wtedy bardzo ważny.

9.5.2.1 this

Pytanie: czy moglibyśmy nazwać argumenty setterów tak samo, jak ustawiane pole? Nasz kod wyglądałby lepiej, gdybyśmy nie musieli za każdym razem poprzedzać argumentu settera np. słowem "*nowa*", jak np. `nowaCena`, `nowaNazwa`. Spróbujmy:

Nazwa pliku: Produkt.java

```
public class Produkt {
  private double cena;
  private String nazwa;

  public void setCena(double cena) { // 1
    cena = cena;
  }

  public double getCena() {
    return cena;
  }

  public void setNazwa(String nazwa) { // 2
    nazwa = nazwa;
  }

  public String getNazwa() {
    return nazwa;
  }

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

Zmieniliśmy nazwy argumentów setterów `setCena` oraz `setNazwa` z `nowaCena` i `nowaNazwa` na, po prostu, `cena` *(1)* i `nazwa` *(2)*. Przypomnijmy klasę `Sklep`, która korzysta z obiektów typu `Produkt`:

Nazwa pliku: Sklep.java

```
public class Sklep {
  public static void main(String[] args) {
    Produkt czeresnie = new Produkt();
    Produkt herbata = new Produkt();

    czeresnie.setCena(8.0);
    czeresnie.setNazwa("Czeresnie");

    herbata.setCena(12.0);
    herbata.setNazwa("Herbata czarna");

    System.out.println("Nazwa pierwszego produktu to: " + czeresnie.getNazwa());
    System.out.println("Cena pierwszego produktu to: " + czeresnie.getCena());
    System.out.println(herbata);
  }
}
```

Gdy teraz uruchomimy powyższy program, to na ekranie zobaczmy:

```
Nazwa pierwszego produktu to: null
Cena pierwszego produktu to: 0.0
Produkt o nazwie null kosztuje 0.0
```

Co się stało? Dlaczego ustawiane przez nas wartości nie zostały wypisane? Wygląda na to, że oba obiekty, `czeresnie` oraz `herbata`, mają ustawione wartości domyślne w swoich polach `cena` oraz `nazwa`!

Spójrzmy jeszcze raz na setter `setCena`:

```
public void setCena(double cena) {
  cena = cena;
}
```

Zmieniliśmy nazwę argument metody `setCena` na `cena`. W związku z tym, argument `cena` zasłania pole klasy `Produkt` o tej samej nazwie – powoduje to, że linia:

```
cena = cena;
```

przypisuje argumentowi `cena` wartość.. argumentu `cena`! Nie ma to sensu – nie to chcieliśmy osiągnąć. W wyniku takiego działania, wywołanie settera nie przynosi żadnych efektów – pole klasy nie zostaje w ogóle zmienione.

Czy istnieje zatem jakaś możliwość, by odwołać się do pola `cena`, które jest polem klasy, a nie argumentem metody `setCena`, by nadać temu polu wartość?

Tak – służy do tego specjalne słowo kluczowe **`this`**.

`this` jest aktualnym obiektem – obiektem, na rzecz którego metoda została wywołana. Za pomocą **`this`** możemy odnieść się do pól i metod obiektu, na rzecz którego wywołujemy metodę. Zobaczymy **`this`** w akcji w poprawionej klasie `Produkt`:

Nazwa pliku: Produkt.java

```
public class Produkt {
  private double cena;
  private String nazwa;

  public void setCena(double cena) {
    this.cena = cena; // 1
  }

  public double getCena() {
    return cena;
  }

  public void setNazwa(String nazwa) {
    this.nazwa = nazwa; // 2
  }

  public String getNazwa() {
    return nazwa;
  }

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

Zmieniliśmy settery pól `cena` *(1)* oraz `nazwa` *(2)*, by korzystały z **`this`**, które traktujemy tak, jakbyśmy działali na obiekcie typu `Produkt`, a konkretniej – tym obiekcie, na rzecz którego wywołaliśmy daną metodę.

Gdy teraz uruchomimy klasę `Sklep`, korzystającą z zaktualizowanej wersji klasy `Produkt`, na ekranie zobaczymy spodziewany komunikat:

```
Nazwa pierwszego produktu to: Czeresnie
Cena pierwszego produktu to: 8.0
Produkt o nazwie Herbata czarna kosztuje 12.0
```

9.5.3 Konwencje dotyczące pisania setterów i getterów

Pisząc metody będące setterami lub getterami, powinniśmy stosować się do kilku konwencji – spójrzmy jeszcze raz na setter i getter pola `cena` z klasy `Produkt`:

```
//      (1)    (2)     (3)    (4)
public void setCena(double cena) {
  this.cena = cena; // 5
}

//      (6)      (7)   (8)
public double getCena() {
  return cena; // 9
}
```

Settery powinny:

1. nie zwracać żadnej wartości `(1)`,
2. zaczynać się od słowa *set*, po którym powinna nastąpić nazwa ustawianego pola, zapisana camelCasem `(2)` – przykładowe nazwy setterów: `setNazwisko` i `setAdresZamieszkania`,
3. przyjmować jeden argument takiego samego typu, jak pole, które ustawia `(3)`,
4. mieć jeden argument o takie samej nazwie, jak nazwa pola, które ustawia `(4)`,
5. przypisać do odpowiedniego pola przesłaną wartość `(5)`.

Gettery powinny:

1. zwracać wartość takiego samego typu, jakiego jest pole, z którego pobierają wartość `(6)`,
2. zaczynać się od słowa *get* (z pewnym wyjątkiem, opisanym poniżej) po którym powinna nastąpić nazwa pola, którego wartość zwracają, zapisana camelCasem `(7)` – przykładowe nazwy getterów: `getNazwisko` i `getAdresZamieszkania`,
3. nie mieć żadnych argumentów `(8)`,
4. zwracać wartość danego pola `(9)`.

Od drugiej reguły getterów istnieje jeden wyjątek – gettery pól typu **`boolean`** powinny zaczynać się od słowa *is*, na przykład `isZamowienieWyslane`. Dozwolone są także zamienniki słowa *is* dla tych przypadków, gdy użycie któregoś z nich lepiej oddaje cel pobieranego pola. Te zamienniki to *has*, *should*, oraz *can*, na przykład, dla pola **`boolean`** `activeAccount`, getter mógłby nazywać się `hasActiveAccount`.

Od tej pory powinniśmy zawsze pisać gettery i settery w naszych klasach zgodnie z powyższą konwencją.

9.5.4 Podsumowanie

9.5.4.1 Pola klas

- Pola klas to pewne właściwości, które opisują obiekty danej klasy, np. pola `tytul` i `autor` klasy `Ksiazka`:

```
public class Ksiazka {
  public String tytul;
  public String autor;
  // pozostala część klasy zostala pominieta
}
```

- Każdy obiekt klasy (każdy jej egzemplarz, czyli instancja) posiada własny zestaw zdefiniowanych w tej klasie pól – ustawienie `koloru` obiektu `czerwonySamochod` nie ma wpływu na `kolor` samochodu `zielonySamochod`:

```
Samochod zielonySamochod = new Samochod();
zielonySamochod.ustawKolor("Zielony"); // 1

Samochod czerwonySamochod = new Samochod();
czerwonySamochod.ustawKolor("Czerwony"); // 2
```

  Po wykonaniu ostatniej linijki `(2)`, pole `kolor` obiektu `zielonySamochod` nadal będzie miało wartość `"Zielony"` ustawioną w linii `(1)` – zmiana wykonana w `(2)` na obiekcie `czerwonySamochod` nie ma wpływu na obiekt `zielonySamochod`.

- Pola klas to po prostu zmienne – mają one swój typ oraz nazwę, oraz mogą mieć różnego rodzaju modyfikatory, jak na przykład **`public`** oraz **`private`**.
- Pola klasy mogą być typów zarówno prymitywnych (na przykład **`int`** czy **`boolean`**), jak i typów złożonych (np. `String`).
- W przeciwieństwie do zmiennych lokalnych, które przestają istnieć po zakończeniu metody, pola klas istnieją, dopóki istnieje obiekt, do którego należą – wartości, które te pola przechowują są dostępne przez całe "życie" obiektu.
- Pola klas nie muszą być umieszczone przed metodami, w których z nich korzystamy, ale jeżeli są one używane do inicjalizacji innych pól, to muszą wystąpić przed nimi.
- Pola i metody w klasach powinny być umieszczane w następującej kolejności (dla lepszej czytelności kodu):
  - publiczne pola powinny być zawsze na początku definicji klasy (chociaż nasze klasy raczej nie powinny posiadać takich pól, poza ewentualnymi polami *statycznymi*, o których opowiemy sobie wkrótce),
  - po nich powinny następować pola prywatne,
  - następnie metody publiczne metody,
  - na końcu powinny znajdować się metody prywatne.

9.5.4.2 Wartości domyślne i null

- Pola klas mają wartości domyślne, które są im automatycznie nadawane, jeżeli nie nadamy im wartości początkowej. Zmienne lokalne nie mają wartości domyślnych.
- Wartości domyślne typów prymitywnych są następujące:

Typ	Wartość domyślna
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000'`
`boolean`	`false`

- Typy referencyjne mają specjalną wartość domyślną – **`null`**.
- Null w programowaniu to wartość szczególna – jego znaczenie to "**brak wartości**" (nie mylić z pustą wartością!).
- Pola klas typu referencyjnego mają domyślną wartość **`null`**, jeżeli nic do nich nie przypiszemy – oznacza to, że nie wskazują one na żaden obiekt.
- Tablice to także typ złożony i pola klas typu tablicowego też mają domyślną wartość **`null`**.
- Próba odniesienia się do pola klasy lub wywołanie metody na rzecz obiektu, który *jest nullem* (nie ma przypisanego żadnego obiektu), kończy się błędem wykonywaniu programu:

```
public class Gora {
  public int wysokosc;
  public String nazwa;

  public double policzWysokoscWKm() {
    return wysokosc / 1000.0;
  }

  public static void main(String[] args) {
    Gora pewnaGora = new Gora();

    // blad wykonywania programu!
    System.out.println("Nazwa gory wielkimi literami: " +
        pewnaGora.nazwa.toUpperCase()); // 1
  }
}
```

na ekranie zobaczymy:

```
Exception in thread "main" java.lang.NullPointerException
     at Gora.main(Gora.java:16)
```

- Gdy zobaczymy błąd `Null Pointer Exception`, będzie to świadczyło o tym, że próbujemy działać na obiekcie, który *jest nullem*. Powyżej, pole `nazwa` ma domyślną wartość **`null`**, a w linii `(1)` próbujemy na jego rzecz wywołać metodę `toUpperCase`.

- Wartością **null** można operować jak każdą inną wartością – można ją, na przykład, przypisać do zmiennej typu złożonego:

```
pewnaGora = null; // 1
System.out.println("Wysokosc gory w kilometrach: " +
    pewnaGora.policzWysokoscWKm()); // 2
```

Powyższy przykład ponownie zakończyłby się błędem Null Pointer Exception.

- Aby uchronić się przed działaniem na nullowych obiektach, zmienne typu referencyjnego możemy przyrównywać do wartości **null**, aby sprawdzić, czy wskazują na jakiś obiekt, czy nie.
- Porównując zmienne typu referencyjnego do wartości **null** możemy używać operatora != (nierówne), jak i operatora == (równe). Pierwszy sprawdzi w tym kontekście, czy zmienna typu referencyjnego pokazuje na jakiś obiekt (bo pokazuje na coś innego niż **null**), a drugi (operator ==) odpowie na pytanie: "Czy zmienna pokazuje na nic?":

Dane	nazwa == null	nazwa != null
`String nazwa = "Giewont";`	**false**	**true**
`String nazwa;` `nazwa = null;`	**true**	**false**

```
pewnaGora = null;

if (pewnaGora != null) {
  System.out.println("Wysokosc gory w kilometrach: " +
      pewnaGora.policzWysokoscWKm());
} else {
  System.out.println("Obiekt pewnaGora to null!");
}
```

- Zmienne typu prymitywnego nigdy nie mogą mieć wartości **null** – próba przypisania **null** do zmiennej, na przykład, typu **int**, spowodowałaby błąd kompilacji:

```
int x = null;
System.out.println(x);
```

```
error: incompatible types: <null> cannot be converted to int
        int x = null;
```

9.5.4.3 Gettery i settery oraz this

- Enkapsulacja to sposób tworzenia klas, w którym ukrywamy ich wewnętrzną implementację.
- Aby zapewnić enkapsulację, pola naszych klas zawsze powinny być prywatne.
- Aby można było korzystać z naszych klas, tworzymy metody nazywane *akcesorami*, które dzielą się na dwa rodzaje ze względu na ich zadanie:
  - gettery – służą do pobierania wartości danego pola,
  - settery – służą do ustawiania wartości danego pola.
- Nazwy metody-getterów zaczynamy od słowa *get*, a metody-settery – od słowa *set*.
- Drugą część nazwy stanowi zawsze nazwa ustawianego/pobieranego pola, zapisana camelCasem.

- Dla przykładu, mając pola `cena` oraz `nazwa` z klasy `Produkt`, moglibyśmy napisać do nich następujące akcesory: `getCena` i `setCena` oraz `getNazwa` i `setNazwa`:

```
public class Produkt {
  private double cena;
  private String nazwa;

  public void setCena(double nowaCena) {
    cena = nowaCena;
  }

  public double getCena() {
    return cena;
  }

  public void setNazwa(String nowaNazwa) {
    nazwa = nowaNazwa;
  }

  public String getNazwa() {
    return nazwa;
  }
}
```

- Nasz kod wyglądałby lepiej, gdybyśmy mogli nazwać argumenty setterów tak samo, jak ustawiane pole – zamiast `nowaCena` – po prostu `cena`. Jeżeli jednak spróbowalibyśmy przypisać wtedy do pola `cena` wartość argumentu o tej samej nazwie, nasz kod nie zadziałałby zgodnie z naszym oczekiwaniem:

```
public void setCena(double cena) {
  cena = cena; // 1
}
```

- Zmieniliśmy powyżej nazwę argumentu metody `setCena` na `cena`. W wyniku tego, argument `cena` zasłania pole klasy `Produkt` o tej samej nazwie. Powoduje to, że linia `(1)` przypisuje argumentowi `cena` wartość argumentu `cena`, przez co wywołanie settera nie przynosi żadnych efektów – pole klasy nie zostaje w ogóle zmienione.
- Aby odwołać się do pola `cena`, które jest polem klasy, a nie argumentem metody `setCena`, korzystamy ze słowa kluczowego **`this`**.
- **`this`** jest aktualnym obiektem – obiektem, na rzecz którego metoda została wywołana.
- Za pomocą **`this`** możemy odnieść się do pól i metod obiektu, na rzecz którego wywołujemy metodę – w tym przypadku, `setCena` i `setNazwa`:

```
public class Produkt {
  private double cena;
  private String nazwa;

  public void setCena(double cena) {
    this.cena = cena;
  }

  public void setNazwa(String nazwa) {
    this.nazwa = nazwa;
  }

  // gettery zostaly pominiete
}
```

9.5.4.4 Konwencje dotyczące pisania setterów i getterów

Pisząc metody będące setterami lub getterami, powinniśmy stosować się do kilku konwencji – spójrzmy jeszcze raz na setter i getter pola `cena` z klasy `Produkt`:

```
//      (1)    (2)     (3)    (4)
public void setCena(double cena) {
  this.cena = cena; // 5
}

//      (6)      (7)   (8)
public double getCena() {
  return cena; // 9
}
```

Settery powinny:

1. nie zwracać żadnej wartości `(1)`,
2. zaczynać się od słowa *set*, po którym powinna nastąpić nazwa ustawianego pola, zapisana camelCasem `(2)` – przykładowe nazwy setterów: `setNazwisko` i `setAdresZamieszkania`,
3. przyjmować jeden argument takiego samego typu, jak pole, które ustawia `(3)`,
4. mieć jeden argument o takie samej nazwie, jak nazwa pola, które ustawia `(4)`,
5. przypisać do odpowiedniego pola przesłaną wartość `(5)`.

Gettery powinny:

1. zwracać wartość takiego samego typu, jakiego jest pole, z którego pobierają wartość `(6)`,
2. zaczynać się od słowa *get* (z pewnym wyjątkiem, opisanym poniżej) po którym powinna nastąpić nazwa pola, którego wartość zwracają, zapisana camelCasem `(7)` – przykładowe nazwy getterów: `getNazwisko` i `getAdresZamieszkania`,
3. nie mieć żadnych argumentów `(8)`,
4. zwracać wartość danego pola `(9)`.

Od drugiej reguły getterów istnieje jeden wyjątek – gettery pól typu **`boolean`** powinny zaczynać się od słowa *is*, na przykład `isZamowienieWyslane`. Dozwolone są także zamienniki słowa *is* dla tych przypadków, gdy użycie któregoś z nich lepiej oddaje cel pobieranego pola. Te zamienniki to *has*, *should*, oraz *can*, na przykład, dla pola **`boolean`** `activeAccount`, getter mógłby nazywać się `hasActiveAccount`.

9.5.5 Pytania

1. Jaka jest wartość domyślna dla typów referencyjnych?
2. Do czego są nam potrzebne gettery i settery?
3. Jak powinny być pisane gettery?
4. Jak powinny być pisane settery?
5. Jakie powinny być nazwy getterów i setterów następujących pól?
   a) `String tytul;`
   b) `double mianownik;`
   c) `boolean uzytkownikZalogowany;`
6. Czym jest i do czego służy `this`?
7. Kiedy możemy zobaczyć błąd `Null Pointer Exception`?
8. Jak uchronić się przed potencjalnym błędem `Null Pointer Exception`?
9. Co zostanie wypisane na ekranie w poniższym programie?

```
public class PytanieZagadka {
  private int liczba;

  public void setLiczba(int liczba) {
    liczba = liczba;
  }

  public int getLiczba() {
    return liczba;
  }

  public static void main(String[] args) {
    PytanieZagadka o = new PytanieZagadka();

    o.setLiczba(100);
    System.out.println("Liczba wynosi: " + o.getLiczba());
  }
}
```

10. Jaki będzie efekt próby kompilacji poniższych klas?
    a)

```
public class PytanieMetody {
  public void setNazwa(String nazwa) {
    this.nazwa = nazwa;
  }

  public String getNazwa() {
    return nazwa;
  }

  private String nazwa;
}
```

b)

```
public class PytaniePola {
  private int x = y;
  private int y = 0;
}
```

11. Co zostanie wypisane na ekranie w poniższym programie?

```
public class PytanieObiekty {
  private String nazwa;

  public void setNazwa(String nazwa) {
    this.nazwa = nazwa;
  }

  public String getNazwa() {
    return nazwa;
  }

  public static void main(String[] args) {
    PytanieObiekty o1 = new PytanieObiekty();
    PytanieObiekty o2 = new PytanieObiekty();

    o1.setNazwa("Pewna nazwa");
    o2.setNazwa("Inna nazwa");

    System.out.println(o1.getNazwa());
    System.out.println(o2.getNazwa());
  }
}
```

12. Co zostanie wypisane na ekranie w poniższym programie?

```
public class PytanieWartosci {
  private int liczba;
  private boolean wartoscLogiczna;
  private String nazwa;

  public String toString() {
    return liczba + " " + wartoscLogiczna + " " + nazwa;
  }

  public static void main(String[] args) {
    PytanieWartosci o = new PytanieWartosci();
    System.out.println(o);
  }
}
```

13. Co zostanie wypisane na ekran w wyniku działania poniższego fragmentu kodu?

```
String tekst = "Witajcie!";

if (tekst == null) {
  System.out.println("tekst jest nullem.");
}

tekst = null;

if (tekst != null) {
  System.out.println("tekst nie jest nullem");
}
```

14. Co zostanie wypisane na ekranie w poniższym programie?

```
public class UzycieWartosci {
  private int liczba;
  private String nazwa;

  private int getLiczba() {
    return liczba;
  }

  private String getNazwa() {
    return nazwa;
  }

  public static void main(String[] args) {
    UzycieWartosci o = new UzycieWartosci();

    System.out.println(o.getLiczba());
    System.out.println(o.getNazwa().toUpperCase());
  }
}
```

9.5.6 Zadania

9.5.6.1 Klasa Punkt

Pamiętając o konwencjach nazewniczych setterów i getterów oraz o użyciu `this`, napisz klasę `Punkt` z:

- dwoma prywatnymi polami `x` oraz `y` typu `int`,
- setterami i getterami do obu pól,
- metodą `toString`.

9.6 Konstruktory

Do tej pory tworzyliśmy obiekty klas używając słowa kluczowego **new**, po którym następowała nazwa konstruktora klasy, której obiekt chcieliśmy utworzyć, nawiasy i średnik:

```
Produkt czeresnie = new Produkt();
```

Po utworzeniu obiektu, nadawaliśmy jego polom wartości albo poprzez settery, albo przez bezpośrednie odwołanie się do tych pól (tego drugiego sposobu nie będziemy już stosować na rzecz wykorzystania setterów):

```
czeresnie.setCena(8.0);
czeresnie.setNazwa("Czeresnie");
```

Takie inicjalizowanie pól obiektu jest co prawda jedną z możliwości, ale niezbyt wygodną – gdybyśmy mieli do ustawienia pięć pól, to musielibyśmy na rzecz danego obiektu wywołać pięć setterów, by ustawić każde z nich. Mamy jednak inną możliwość – napisać własny *konstruktor*.

Język Java, jak i inne języki obiektowe, udostępnia specjalny rodzaj metod nazywanych *konstruktorami*, które służą do inicjalizacji obiektów klasy.

Konstruktory mają dwie cechy specjalne, które wyróżniają je na tle innych metod:

- nazwy konstruktorów są zawsze takie same, jak nazwa klasy, w której się znajdują – dla przykładu, konstruktor klasy `Produkt` będzie nazywał się `Produkt`,
- konstruktory nie zwracają żadnej wartości (nie stosujemy nawet **void**!) – podczas definiowania konstruktora po prostu omijamy zwracany typ.

Na początku tego kursu poznaliśmy konwencję nazewniczą obiektów w Javie – wedle niej, nazwy zmiennych i metod powinny zaczynać się od małej litery – konstruktory są wyjątkiem od tej reguły, ponieważ muszą nazywać się tak samo, jak nazwa klasy – a jak wiemy, nazwę klas, także zgodnie z konwencją, zawsze zaczynamy od wielkiej litery.

Zobaczmy pierwszy przykład konstruktora w klasie `Produkt`:

Nazwa pliku: Produkt.java

```
public class Produkt {
  private double cena;
  private String nazwa;

  //(1) (2)  (3)        (4)
  public Produkt(double cena, String nazwa) {
    this.cena = cena; // 5
    this.nazwa = nazwa; // 6
  }

  // settery i gettery zostaly pominiete

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

Klasa `Produkt` posiada jeden konstruktor – jego cechy to:

- publiczny dostęp `(1)` – każdy będzie mógł tworzyć obiekty typu `Produkt` za pomocą tego konstruktora,

- brak zwracanego typu *(2)* – zazwyczaj, pomiędzy modyfikatorami a nazwą metody, umieszczamy typ wartości, którą metoda zwraca – w przypadku konstruktorów pomijamy całkowicie zwracany typ (nie piszemy nawet słowa **void**),
- nazwa jest taka sama, jak nazwa klasy w której konstruktor się znajduje *(3)* – nazwa konstruktora to `Produkt`, ponieważ zawarty jest w klasie o nazwie `Produkt`,
- dwa argumenty *(4)* – wartości, którymi zostaną zainicjalizowane pola `cena` *(5)* oraz `nazwa` *(6)* – jak widzimy, korzystamy w konstruktorze z poznanego już słowa kluczowego **this**, by odwołać się do pól obiektu, który tworzymy. Dzięki użyciu słowa kluczowego **this** możemy nazwać argumenty konstruktora tak samo, jak nazwy pól naszej klasy.

Zobaczmy powyższy konstruktor klasy `Produkt` w akcji:

Nazwa pliku: Warzywniak.java

```
public class Warzywniak {
  public static void main(String[] args) {
    Produkt papryka = new Produkt(5.0, "Papryka"); // 1

    System.out.println(papryka);
  }
}
```

Tym razem nawiasy po nazwie klasy, której obiekt tworzymy, nie są puste – w nawiasach zawarliśmy argumenty konstruktora *(1)* utworzonego w klasie `Produkt`.

Po uruchomieniu tego programu, na ekranie zobaczymy:

```
Produkt o nazwie Papryka kosztuje 5.0
```

Wartości przesłane jako argumenty konstruktora *(1)* zostały przypisane do pól `cena` i `nazwa`, co widać po wypisaniu tekstowej reprezentacji obiektu `papryka` na ekran.

Konstruktory mogą być prywatne. Jaki jest ich cel dowiemy się w rozdziale o dziedziczeniu.

9.6.1 Domyślny konstruktor

Wróćmy do klasy `Sklep`, z której niedawno korzystaliśmy – tworzyliśmy w niej dwa obiekt typu `Produkt`:

```
public class Sklep {
  public static void main(String[] args) {
    Produkt czeresnie = new Produkt();
    Produkt herbata = new Produkt();

    czeresnie.setCena(8.0);
    czeresnie.setNazwa("Czeresnie");

    herbata.setCena(12.0);
    herbata.setNazwa("Herbata czarna");

    System.out.println("Nazwa pierwszego produktu to: " + czeresnie.getNazwa());
    System.out.println("Cena pierwszego produktu to: " + czeresnie.getCena());
    System.out.println(herbata);
  }
}
```

Spróbujmy jeszcze raz skompilować i uruchomić powyższy program. Niespodziewanie, próba kompilacji kończy się następującym błędem:

```
Sklep.java:3: error: constructor Produkt in class Produkt cannot be applied to given types;
        Produkt czeresnie = new Produkt();
                            ^
  required: double,String
  found: no arguments
  reason: actual and formal argument lists differ in length

Sklep.java:4: error: constructor Produkt in class Produkt cannot be applied to given types;
        Produkt herbata = new Produkt();
                          ^
  required: double,String
  found: no arguments
  reason: actual and formal argument lists differ in length
2 errors
```

Wygląda na to, że dodanie do klasy `Produkt` konstruktora spowodowało, że klasa `Sklep` przestała działać – dlaczego tak się stało?

Każda klasa, którą napiszemy w języku Java, będzie miała konstruktor – **niezależnie od tego, czy go napiszemy, czy nie**.

Taki konstruktor nazywany jest *konstruktorem domyślnym* i jest dla nas generowany automatycznie przez kompilator języka Java w jednym, konkretnym w przypadku: gdy my, jako autorzy klasy, nie dostarczymy sami konstruktora dla klasy.

Konstruktor domyślny:

- nie przyjmuje żadnych argumentów,
- nie wykonuje żadnych instrukcji – jego ciało jest puste,
- występuje tylko w tych klasach, w których nie został zdefiniowany żaden konstruktor przez programistę.

Tworząc obiekty klas, które pisaliśmy w tym rozdziale, korzystaliśmy cały czas, nie wiedząc o tym, z konstruktorów domyślnych, wygenerowanych dla nas przez kompilator, na przykład w klasie `Sklep`:

```
Produkt czeresnie = new Produkt();
Produkt herbata = new Produkt();
```

W powyższym fragmencie kodu, używaliśmy konstruktora domyślnego. Po dodaniu do klasy `Produkt` poniższego konstruktora:

```
public Produkt(double cena, String nazwa) {
  this.cena = cena;
  this.nazwa = nazwa;
}
```

kompilator nie musiał już generować konstruktora domyślnego dla naszej klasy `Produkt`, ponieważ sami dostarczyliśmy konstruktor tej klasy. Z tego powodu, klasa `Sklep` przestała działać, ponieważ wykorzystywany w niej do tej pory konstruktor domyślny (przyjmujący zero argumentów) przestał istnieć w klasie `Produkt`!

Co możemy zatem zrobić, aby kod klasy `Sklep` zaczął działać? Mamy dwa wyjścia:

1. Przepisać tworzenie obiektów w klasie `Sklep`, by korzystało z nowego konstruktora.
2. Dodać do klasy `Produkt` drugi konstruktor, który nie przyjmuje żadnych argumentów i nie wykonuje żadnych operacji – będzie to w zasadzie odwzorowanie konstruktora domyślnego, z którego do tej pory korzystaliśmy.

Spróbujmy dodać drugi konstruktor do klasy `Produkt`:

Nazwa pliku: Produkt.java

```
public class Produkt {
  private double cena;
  private String nazwa;

  public Produkt() { // 1

  }

  public Produkt(double cena, String nazwa) {
    this.cena = cena;
    this.nazwa = nazwa;
  }

  // settery i gettery zostaly pominiete

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

Do klasy `Produkt` dodaliśmy drugi konstruktor `(1)`, który nie przyjmuje żadnych argumentów i nie wykonuje żadnych instrukcji. Gdy teraz spróbujemy skompilować i uruchomić klasę `Sklep`, na ekranie zobaczymy:

```
Nazwa pierwszego produktu to: Czeresnie
Cena pierwszego produktu to: 8.0
Produkt o nazwie Herbata czarna kosztuje 12.0
```

Dodatkowy konstruktor, wzorowany na konstruktorze domyślnym, spowodował, że kod zaczął ponownie działać. Jak widać, klasa może mieć więcej niż jeden konstruktor.

9.6.2 Przeładowanie konstruktora

Klasy mogą mieć wiele konstruktorów – tyle, ile uznamy za stosowne. Z racji tego, że konstruktory są metodami, to aby mieć więcej niż jeden konstruktor, każdy z nich musi różnić się liczbą, typem, lub kolejnością argumentów (co wiemy z rozdziału o przeładowywaniu metod).

Kiedy możemy potrzebować więcej niż jednego konstruktora? W poprzednim przykładzie mieliśmy dwa konstruktory – pierwszy nie przyjmował argumentów i nie wykonywał żadnych operacji, a drugi inicjalizował oba pola obiektu przesłanymi do niego argumentami:

```
public Produkt() {

}

public Produkt(double cena, String nazwa) {
  this.cena = cena;
  this.nazwa = nazwa;
}
```

Czasami możemy chcieć zainicjalizować od razu pola obiektu pewnym wartościami – innym razem możemy nie chcieć od razu podawać wszystkich wartości, tylko uzupełnić je później.

Często spotykanym przypadkiem w klasach, które mają wiele pól, jest posiadanie kilku konstruktorów, z których każdy inicjalizuje inny zestaw pól – od konstruktora, który zainicjalizuje

wszystkie, do takiego, który nie zainicjalizuje żadnych. Często w takich przypadkach polom niezainicjalizowanym przypisywane są pewne domyślne (dla obiektów danej klasy) wartości. Spójrzmy na poniższy przykład:

Nazwa pliku: Film.java

```
public class Film {
  private String tytul;
  private String rezyser;
  private double cenaBiletu;

  public Film() { // 1
    this.tytul = "<nienazwany film>"; // 5
    this.rezyser = "<brak rezysera>"; // 5
    this.cenaBiletu = 20.0; // 5
  }

  public Film(String tytul) { // 2
    this.tytul = tytul;
    this.rezyser = "<brak rezysera>"; // 5
    this.cenaBiletu = 20.0; // 5
  }

  public Film(String tytul, String rezyser) { // 3
    this.tytul = tytul;
    this.rezyser = rezyser;
    this.cenaBiletu = 20.0; // 5
  }

  public Film(String tytul, String rezyser, double  cenaBiletu) { // 4
    this.tytul = tytul;
    this.rezyser = rezyser;
    this.cenaBiletu = cenaBiletu;
  }
}
```

Klasa `Film` ma cztery konstruktory `(1)` `(2)` `(3)` `(4)` – pozwalają one na ustawienie pewnych pól, przesyłając ich wartości jako argumenty, a pozostałe inicjalizują wartościami domyślnymi `(5)`. Mamy więc cztery sposoby na utworzenie obiektu klasy `Film`:

```
Film tajemniczyFilm = new Film();
Film rambo = new Film("Rambo");
Film zrodlo = new Film("Zrodlo", "Darren Aronofsky");
Film cicheMiejsce = new Film("Ciche miejsce", "John Krasinski", 25.0);
```

Powyższy fragment kodu wykorzystuje każdy z czterech konstruktorów klasy `Film`.

Wróćmy do konstruktorów – widzimy w kodzie klasy `Film`, że wielokrotnie powtarzamy te same fragmenty kodu – każdy konstruktor inicjalizuje wszystkie pola tworzonego obiektu.

W poprzednich rozdziałach uczyliśmy się, że duplikacja kodu nie jest dobrą praktyką – na szczęście, w tym przypadku język Java udostępnia nam sposób na zwięźlejsze zapisanie kodu klasy `Film`.

Zamiast w każdym konstruktorze ustawiać wszystkie pola, możemy oddelegować ustawienie ich innemu konstruktorowi tej samej klasy – wywołamy go jak każdą inną metodę, podając argumenty.

Aby wywołać z konstruktora inny konstruktor, korzystamy z poznanego już słowa kluczowego **`this`**, po którym następują nawiasy i ewentualne argumenty – spójrzmy na drugą wersję klasy `Film`:

```
public class Film {
  private String tytul;
  private String rezyser;
  private double cenaBiletu;

  public Film() {
    this("<nienazwany film>", "<brak rezysera>", 20.0); // 1
  }

  public Film(String tytul) {
    this(tytul, "<brak rezysera>", 20.0); // 2
  }

  public Film(String tytul, String rezyser) {
    this(tytul, rezyser, 20.0); // 3
  }

  public Film(String tytul, String rezyser, double cenaBiletu) { // 4
    this.tytul = tytul;
    this.rezyser = rezyser;
    this.cenaBiletu = cenaBiletu;
  }
}
```

W każdym z konstruktorów `(1)` `(2)` `(3)`, poza czwartym, skorzystaliśmy z możliwości wywołania innego konstruktora – konstruktorem, który wywołujemy, jest czwarty konstruktor `(4)`, który inicjalizuje wszystkie pola tworzonego obiektu przesłanymi do niego argumentami. Każdy z pozostałych konstruktorów wywołuje go z odpowiednimi wartościami – część wartości pochodzi z argumentów tych konstruktorów, a cześć ustawiamy "na sztywno" jako wartości domyślne. W ten sposób skróciliśmy znacząco kod.

Istnieją dwie zasady odnośnie wywoływania innych konstruktorów:

1. Możemy wywołać tylko jeden inny konstruktor z danego konstruktora (chociaż możemy z wywoływanego konstruktora wywołać kolejny).
2. Wywołanie innego konstruktora musi być pierwszą instrukcją w danym konstruktorze (poza komentarzami).

Spójrzmy najpierw na punkt 1 – poniższy kod spowodowałby błąd kompilacji:

```
public Film() {
  this("<nienazwany film>", "<brak rezysera>", 20.0);
  // blad kompilacji! nie mozemy wywolac kolejnego konstruktora
  this("<nienazwany film>");
}
```

Jednakże moglibyśmy zapisać konstruktory naszej klasy `Film` w następujący, poprawny sposób:

```
public Film() {
  this("<nienazwany film>"); // 1
}

public Film(String tytul) {
  this(tytul, "<brak rezysera>"); // 2
}

public Film(String tytul, String rezyser) {
  this(tytul, rezyser, 20.0); // 3
}
```

```
public Film(String tytul, String rezyser, double cenaBiletu) { // 4
  this.tytul = tytul;
  this.rezyser = rezyser;
  this.cenaBiletu = cenaBiletu;
}
```

W tym (poprawnie działającym) przypadku pierwszy `(1)` konstruktor wywoływuje drugi `(2)`, drugi korzysta z trzeciego `(3)`, który, finalnie, używa konstruktora czwartego `(4)`.

Druga zasada wspominała o kolejności instrukcji – wywołanie innego konstruktora musi być zawsze pierwszą instrukcją w konstruktorze – poniższy przykład zakończy się błędem kompilacji:

```
public Film() {
  // blad kompilacji! wywolanie innego konstruktora, o ile jest on
  //  uzywany, musi byc pierwsza instrukcja konstruktora
  System.out.println("Wywolales konstruktor bezargumentowy!");
  this("<nienazwany film>", "<brak rezysera>", 20.0);
}
```

Błąd kompilacji, który zobaczymy, to:

```
Film.java:8: error: call to this must be first statement in constructor
      this("<nienazwany film>", "<brak rezysera>", 20.0);
          ^
1 error
```

Nic nie stoi jednak na przeszkodzie, by nasze konstruktory zawierały instrukcje *po* wywołaniu innego konstruktora, na przykład:

```
public Film() {
  // tym razem kod jest poprawny - po wywolaniu innego konstruktora,
  //  ten konstruktor moze zawierac inne instrukcje
  this("<nienazwany film>", "<brak rezysera>", 20.0);
  System.out.println("Wywolales konstruktor bezargumentowy!");
}
```

9.6.3 Inicjalizacja pól finalnych w konstruktorach

W rozdziale trzecim "Zmienne" nauczyliśmy się, czym są *stałe* i jak je definiować.

Stałe to takie zmienne, którym wartość nadajemy tylko raz i nie możemy kolejny raz przypisać im wartości. Stałe definiujemy poprzedzając ich nazwę słowem kluczowym **`final`**:

```
final double LICZBA_PI = 3.14;
final String PONIEDZIALEK = "Poniedzialek";

// blad kompilacji!
// error: cannot assign a value to final variable LICZBA_PI
LICZBA_PI = 5;

// blad kompilacji!
// error: cannot assign a value to final variable PONIEDZIALEK
PONIEDZIALEK = "Piatek";
```

Po zainicjalizowaniu stałych `LICZBA_PI` oraz `PONIEDZIALEK`, wszelkie próby nadania im nowej wartości kończą się przytoczonym powyżej błędem kompilacji.

Pola klas także mogą być stałe – po nadaniu im raz wartości, zachowają ją i nie będziemy mogli przypisać im innej wartości – spójrzmy na poniższy przykład:

Nazwa pliku: Pojazd.java

```
public class Pojazd {
  private final String marka; // 1
  private final String numerRejestracyjny; // 2
  private final int rokProdukcji; // 3

  public String toString() {
    return "Pojazd marki " + marka +
        ", numer rejestracjny " + numerRejestracyjny +
        ", wyprodukowany w " + rokProdukcji + " roku.";
  }

  public String getMarka() {
    return marka;
  }

  public String getNumerRejestracyjny() {
    return numerRejestracyjny;
  }

  public int getRokProdukcji() {
    return rokProdukcji;
  }
}
```

Nasza klasa zawiera:

- trzy *stałe* pola: `marka` *(1)*, `numerRejestracyjny` *(2)*, oraz `rokProdukcji` *(3)*,
- metodę `toString`,
- po jednym *getterze* na każde pole,
- domyślny konstruktor, który zostanie dla nas automatycznie wygenerowany z racji tego, że sami nie dostarczyliśmy dla klasy `Pojazd` żadnego konstruktora.

Zauważmy, że nasza klasa nie ma setterów – nie miałyby sensu, skoro wszystkie pola klasy `Pojazd` *są stałe.*

Pytanie: czy powyższa klasa skompiluje się poprawnie?

Spróbujmy:

```
Pojazd.java:2: error: variable marka not initialized in the default
constructor
    private final String marka;
                         ^
Pojazd.java:3: error: variable numerRejestracyjny not initialized in the
default constructor
    private final String numerRejestracyjny;
                         ^
Pojazd.java:4: error: variable rokProdukcji not initialized in the default
constructor
    private final int rokProdukcji;
                      ^
3 errors
```

Próba kompilacji zakończyła się błędami – kompilator protestuje, ponieważ nigdzie w naszej klasie

nie inicjalizujemy stałych pól tej klasy. Moglibyśmy zmienić kod klasy na następujący:

```
public class Pojazd {
  private final String marka = "Toyota";
  private final String numerRejestracyjny = "123456";
  private final int rokProdukcji = 1997;

  // gettery i metoda toString zostaly pominiete
}
```

W tej wersji klasy `Pojazd` przypisujemy do stałych pól klasy `Pojazd` pewne wartości – powyższa klasa skompiluje się bez problemów.

Problem jednak nadal występuje, ponieważ.. klasa `Pojazd` jest nieużywalna, o ile nie zakładamy, że wszystkie pojazdy, jakie kiedykolwiek będziemy potrzebowali, będą `Toyotami` z numerem rejestracyjnym `123456`, wyprodukowanymi w roku `1997`.

W tej chwili wszystkie obiekty klasy `Pojazd`, jakie byśmy utworzyli, miałyby takie same wartości przypisane do każdego pola – wszystkie te pola są stałe i raz po przypisaniu im wartości nie możemy już nadać im innej wartości.

Aby rozwiązać ten problem, możemy skorzystać z konstruktorów, ponieważ stałe pola, których nie zainicjalizujemy od razu wartością, możemy jeszcze zainicjalizować w konstruktorach – spójrzmy na finalną wersję klasy `Pojazd`:

Nazwa pliku: Pojazd.java

```
public class Pojazd {
  private final String marka;
  private final String numerRejestracyjny;
  private final int rokProdukcji;

  // 1
  public Pojazd(String marka, String numerRejestracyjny, int rokProdukcji) {
    this.marka = marka; // 2
    this.numerRejestracyjny = numerRejestracyjny; // 3
    this.rokProdukcji = rokProdukcji; // 4
  }

  public String toString() {
    return "Pojazd marki " + marka +
        ", numer rejestracyjny " + numerRejestracyjny +
        ", wyprodukowany w " + getRokProdukcji() + " roku.";
  }

  public String getMarka() {
    return marka;
  }

  public String getNumerRejestracyjny() {
    return numerRejestracyjny;
  }

  public int getRokProdukcji() {
    return rokProdukcji;
  }
}
```

Do klasy `Pojazd` dodaliśmy konstruktor `(1)`, który przyjmuje trzy argumenty – wartości tych argumentów użyjemy do zainicjalizowania stałych pól tworzonego obiektu. Będą to, odpowiednio,

pola `marka` *(2)*, `numerRejestracyjny` *(3)*, oraz `rokProdukcji` *(4)*.

Powyższy kod kompiluje się bez błędów – co prawda nie nadajemy stałym polom od razu wartości w liniach, w których je definiujemy, ale robimy to w konstruktorze – kompilator analizując kod naszej klasy wie, że nie ma możliwości na utworzenie obiektu klasy `Pojazd`, który miałby niezainicjalizowane stałe pola – pola, które *muszą* być zainicjalizowane. Wszystkie obiekty tworzone będą za pomocą konstruktora, który te wszystkie stałe pola uzupełnia wartościami.

A gdybyśmy dodali jeszcze jeden konstruktor, który inicjalizowałby tylko dwa z trzech stałych pól? Na przykład, załóżmy, że do klasy `Pojazd` dodajemy jeszcze jeden, następujący konstruktor:

```
public Pojazd(String marka, String numerRejestracyjny) {
  this.marka = marka;
  this.numerRejestracyjny = numerRejestracyjny;
}
```

Gdybyśmy teraz spróbowali skompilować klasę `Pojazd`, to zobaczylibyśmy ponownie błąd o potencjalnym brak inicjalizacji pola `rokProdukcji`:

```
Pojazd.java:9: error: variable rokProdukcji might not have been initialized
    }
    ^
1 error
```

Błąd wynika z faktu, że tym razem istnieje sposób na utworzenie obiektu klasy `Pojazd`, który miałby niezainicjalizowane, stałe pole – kompilator wychwycił ten problem i zasygnalizował go powyższym błędem już na etapie kompilacji.

Spójrzmy jeszcze na wykorzystanie (poprawnej) wersji klasy `Pojazd`:

```
Pojazd motor = new Pojazd("Harley", "978654", 2017);
Pojazd samochod = new Pojazd("Toyota", "123456", 1997);

System.out.println(motor);
System.out.println(samochod);
```

Wynik na ekranie:

```
Pojazd marki Harley, numer rejestracyjny 978654, wyprodukowany w 2017 roku.
Pojazd marki Toyota, numer rejestracyjny 123456, wyprodukowany w 1997 roku.
```

Zgodnie z konwencją, stałe zazwyczaj zapisujemy wielkimi literami ze słowami rozdzielonymi znakami podkreślenia. Ta konwencja dotyczy stałych, które mają niezmienne, w pewien sposób uniwersalne wartości, jak na przykład liczba Pi.
W przypadku pól klas nie stosujemy tej konwencji – pola klas definiujemy jako **`final`**, *by zapobiec potencjalnemu przypisaniu do pola klasy innej wartości, lecz sama wartość, jako taka, nie musi stanowić uniwersalnej, stałej wartości – dlatego stałych pól w klasie* `Pojazd` *nie nazywaliśmy wielkimi literami, lecz, tak jak zawsze, używając camelCase'a.*

9.6.4 Podsumowanie

- Konstruktory to specjalny rodzaj metod, które służą do inicjalizacji obiektów klasy.
- Konstruktory mają dwie cechy specjalne, które wyróżniają je na tle innych metod:
  - nazwy konstruktorów są zawsze takie same, jak nazwa klasy – konstruktor klasy `Produkt` będzie nazywał się `Produkt`,
  - konstruktory nie zwracają żadnej wartości (nie stosujemy nawet **`void`**) – podczas definiowania konstruktora omijamy zwracany typ.
- Poniższa klasa `Produkt` posiada jeden publiczny konstruktor *(1)*, który nie ma zwracanego typu (nie zostało użyte nawet słowo **`void`**). Nazwa jest taka sama, jak nazwa klasy:

```
public class Produkt {
  private double cena;
  private String nazwa;

  public Produkt(double cena, String nazwa) { // 1
    this.cena = cena;
    this.nazwa = nazwa;
  }

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

- Użycie konstruktora *(1)* i wynik działania poniższego kodu:

```
Produkt papryka = new Produkt(5.0, "Papryka"); // 1

System.out.println(papryka);
```

```
Produkt o nazwie Papryka kosztuje 5.0
```

- Każda klasa, którą napiszemy w języku Java, ma konstruktor – niezależnie od tego, czy go napiszemy, czy nie – taki konstruktor nazywamy *domyślnym*.
- *Konstruktor domyślny* jest dla nas generowany automatycznie przez kompilator w przypadku, gdy my, jako autorzy klasy, nie dostarczymy sami konstruktora dla tej klasy.
- Konstruktor domyślny:
  - nie przyjmuje żadnych argumentów,
  - nie wykonuje żadnych instrukcji – jego ciało jest puste,
  - występuje tylko w tych klasach, w których nie został zdefiniowany żaden konstruktor przez programistę.
- Poniższa, pusta klasa, posiada konstruktor domyślny:

```
public class PustaKlasaZDomyslnymKonstruktorem {

}
```

Ponieważ konstruktor domyślny będzie automatycznie wygenerowany, możemy napisać:

```
PustaKlasaZDomyslnymKonstruktorem obiekt = new PustaKlasaZDomyslnymKonstruktorem();
```

- Klasy mogą mieć wiele konstruktorów – muszą się one jednak różnić liczbą, typem, lub kolejnością argumentów.
- Z konstruktora możemy wywołać inny konstruktor tej samej klasy – w takim przpadku korzystamy z poznanego już słowa kluczowego **this**, po którym następują nawiasy i ewentualne argumenty:

```
public class Film {
  private String tytul;
  private String rezyser;
  private double cenaBiletu;

  public Film() {
    this("<nienazwany film>", "<brak rezysera>", 20.0); // 1
  }

  public Film(String tytul) {
    this(tytul, "<brak rezysera>", 20.0); // 2
  }

  public Film(String tytul, String rezyser) {
    this(tytul, rezyser, 20.0); // 3
  }

  public Film(String tytul, String rezyser, double cenaBiletu) { // 4
    this.tytul = tytul;
    this.rezyser = rezyser;
    this.cenaBiletu = cenaBiletu;
  }
}
```

- Istnieją dwie zasady odnośnie wywoływania innych konstruktorów:
  - Możemy wywołać tylko jeden inny konstruktor z danego konstruktora (chociaż możemy z wywoływanego konstruktora wywołać kolejny):

```
public Film() {
  this("<nienazwany film>", "<brak rezysera>", 20.0);
  // blad kompilacji! nie mozemy wywolac kolejnego konstruktora
  this("<nienazwany film>");
}
```

  - Wywołanie innego konstruktora musi być pierwszą instrukcją w danym konstruktorze (poza komentarzami):

```
public Film() {
  // blad kompilacji! wywolanie innego konstruktora, o ile jest on
  //  uzywany, musi byc pierwsza instrukcja konstruktora
  System.out.println("Wywolales konstruktor bezargumentowy!");
  this("<nienazwany film>", "<brak rezysera>", 20.0);
}
```

- Pola klas mogą być stałe, tzn. zdefiniowane z modyfikatorem **final** – raz po przypisaniu im wartości nie będziemy już mogli nadać im innej wartości.
- Pola **final** możemy zainicjalizować w konstruktorach:

```
public class Pojazd {
  private final String marka;
  private final String numerRejestracyjny;
  private final int rokProdukcji;

  public Pojazd(String marka, String numerRejestracyjny, int rokProdukcji) {
    this.marka = marka;
    this.numerRejestracyjny = numerRejestracyjny;
    this.rokProdukcji = rokProdukcji;
  }

  // gettery zostaly pominiete
}
```

- Gdybyśmy dodali jeszcze jeden konstruktor, który inicjalizowałby tylko dwa z trzech stałych pól (taki, jak poniżej), to kod powyższej klasy `Pojazd` przestałby się kompilować:

```
public Pojazd(String marka, String numerRejestracyjny) {
    this.marka = marka;
    this.numerRejestracyjny = numerRejestracyjny;
}
```

```
Pojazd.java:9: error: variable rokProdukcji might not have been initialized
    }
    ^
1 error
```

- Błąd wynika z faktu, że istnieje teraz sposób na utworzenie obiektu klasy `Pojazd`, który miałby niezainicjalizowane, stałe pole – kompilator wychwycił ten problem i zasygnalizował go powyższym błędem na etapie kompilacji.
- Pola **final**, o ile nie są uniwersalnymi stałymi z punktu widzenia naszego programu, nazywamy tak jak inne zmienne, czyli korzystając z konwencji camelCase. Stałe, które wyznaczają pewną niezmienną wartość, jak na przykład liczba Pi, nazywamy wielkimi literami z podkreśleniem jako separator słów, np. `LICZBA_PI`.

9.6.5 Pytania

1. Do czego służą konstruktory?
2. Czy każda klasa posiada konstruktor?
3. Jak zdefiniować konstruktor?
4. Czym jest konstruktor domyślny i kiedy jest definiowany?
5. Czy klasa może mieć wiele konstruktorów?
6. Jak z jednego konstruktora wywołać inny konstruktor?
7. Jaki warunek musi spełniać wywołanie jednego konstruktora z drugiego konstruktora?
8. Jaki będzie wynik kompilacji i uruchomienia poniższego kodu?

```
public class PytanieKonstruktor {
  private int x;

  public PytanieKonstruktor(int x) {
    this.x = x;
  }

  public void setX(int x) {
    this.x = x;
  }

  public String toString() {
    return "x = " + x;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o = new PytanieKonstruktor();

    System.out.println(o);
  }
}
```

9. Ile konstruktorów posiada poniższa klasa?

```
public class PytanieKonstruktor {
  private int pewnePole;
}
```

10. Czy poniższa klasa ma domyślny konstruktor?

```
public class PytanieKonstruktor {
  private int pewnePole;

  public PytanieKonstruktor() {

  }
}
```

11. Jaki będzie wynik kompilacji i uruchomienia poniższej klasy?

```
public class PytanieKonstruktor {
  private final int liczba;
  private final String nazwa;

  public PytanieKonstruktor(int liczba) {
    this.liczba = liczba;
  }

  public PytanieKonstruktor(int liczba, String nazwa) {
    this.liczba = liczba;
    this.nazwa = nazwa;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o = new PytanieKonstruktor(10, "Tekst");
  }
}
```

12. Jaki będzie wynik kompilacji i uruchomienia poniższej klasy?

```
public class PytanieKonstruktor {
  private final int liczba;

  public PytanieKonstruktor() {
    System.out.println("Wywolano konstruktor bez argumentow.");
    this(0);
  }

  public PytanieKonstruktor(int liczba) {
    this.liczba = liczba;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o = new PytanieKonstruktor(10);
  }
}
```

13. Jaki będzie wynik kompilacji i uruchomienia poniższej klasy?

```
public class PytanieKonstruktor {
  private final int liczba;
  private final String nazwa;

  public PytanieKonstruktor(int liczba) {
    this(liczba, "brak nazwy");
    this.liczba = liczba;
  }

  public PytanieKonstruktor(int liczba, String nazwa) {
    this.liczba = liczba;
    this.nazwa = nazwa;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o = new PytanieKonstruktor(10, "Tekst");
  }
}
```

14. Jaki będzie wynik kompilacji i uruchomienia poniższej klasy?

```
public class PytanieKonstruktor {
  private int x;

  public void PytanieKonstruktor(int x) {
    this.x = x;
  }

  public String toString() {
    return "x = " + x;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o = new PytanieKonstruktor();

    System.out.println(o);
  }
}
```

15. Jaki będzie wynik kompilacji i uruchomienia poniższej klasy?

```
public class PytanieKonstruktor {
  private int x;

  public Pytaniekonstruktor(int x) {
    this.x = x;
  }

  public String toString() {
    return "x = " + x;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o = new PytanieKonstruktor();

    System.out.println(o);
  }
}
```

16. Jaki będzie wynik kompilacji i uruchomienia poniższej klasy?

```
public class PytanieKonstruktor {
  private int x;

  public PytanieKonstruktor() {
    x = 10;
  }

  public PytanieKonstruktor(int x) {
    x = x;
  }

  public String toString() {
    return "x = " + x;
  }

  public static void main(String[] args) {
    PytanieKonstruktor o1 = new PytanieKonstruktor();
    PytanieKonstruktor o2 = new PytanieKonstruktor(20);

    System.out.println(o1);
    System.out.println(o2);
  }
}
```

9.6.6 Zadania

9.6.6.1 Klasa Adres

Napisz klasę `Adres`, która będzie miała następujące pola:

- `miejscowosc` typu `String`,
- `kodPocztowy` typu `String`,
- `nazwaUlicy` typu `String`,
- `nrDomu` typu **`int`**.

Do klasy `Adres` dodaj:

- konstruktor, który będzie inicjalizował wszystkie pola obiektów tej klasy,
- metodę `toString`.

9.6.6.2 Klasa Osoba z konstruktorem

Napisz klasę `Osoba`, która będzie miała następujące pola:

- `imie` typu `String`,
- `nazwisko` typu `String`,
- stały (**`final`**) `rokUrodzenia` typu **`int`**,
- `adres` typu `Adres`, który został utworzony w ramach poprzedniego zadania (*Klasa Adres*).

Napisz dwa konstruktory dla klasy `Osoba`:

- pierwszy powinien przyjmować argumenty dla pól `imie`, `nazwisko`, `rokUrodzenia`, oraz `adres`,
- drugi powinien przyjmować argumenty dla pól `imie`, `nazwisko`, `rokUrodzenia`, a także wartości wymagane przez konstruktor klasy `Adres` (`miejscowosc`, `kodPocztowy`, `nazwaUlicy`, oraz `nrDomu`). Ten konstruktor będzie przyjmował 7 argumentów. W ciele konstruktora utwórz nowy obiekt typu `Adres` (na podstawie otrzymanych argumentów) i przypisz go do pola `adres` tworzonego obiektu klasy `Osoba`.

Dodaj do klasy `Osoba` metodę `toString` oraz `main`. Utwórz po jednym obiekcie klasy `Osoba` korzystając z każdego z dostępnych konstruktorów i wypisz je na ekran.

9.7 Equals – porównywanie obiektów

W jednym z podrozdziałów rozdziału o metodach dowiedzieliśmy się, że nie powinniśmy porównywać obiektów typu `String` za pomocą operatora ==, lecz za pomocą metody `equals`.

W tym rozdziale dowiemy się z czego to wymaganie wynika.

9.7.1 Porównywanie wartości zmiennych

Jak już wiemy, w Javie wyróżniamy dwa rodzaje typów: prymitywne oraz referencyjne (złożone). Typów prymitywnych jest osiem, natomiast typy referencyjne możemy definiować sami poprzez pisanie klas.

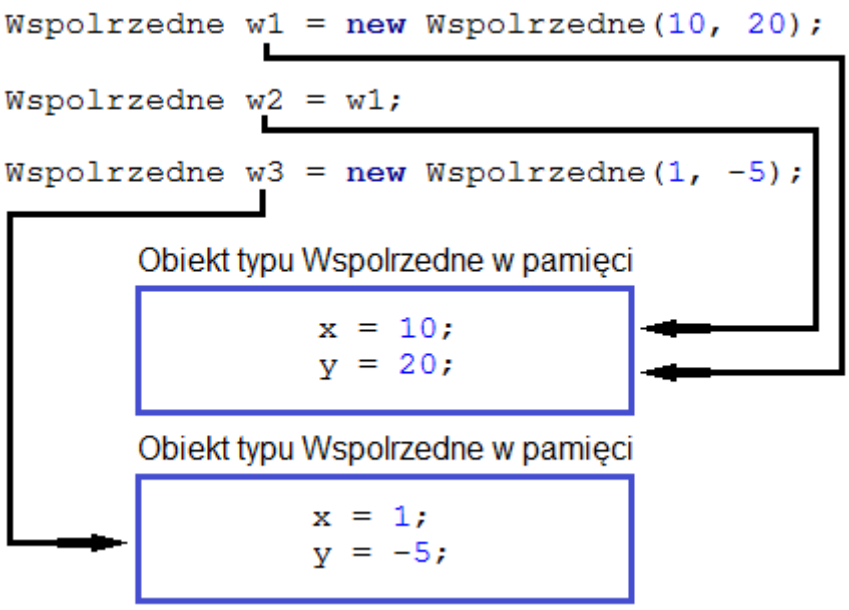
Dowiedzieliśmy się już także, że zmienne typów referencyjnych *wskazują* na obiekty w pamięci – nie są one obiektami, lecz *referencjami* do obiektów.

W poniższym przykładzie tworzymy trzy zmienne typu `Wspolrzedne`, ale tylko dwa obiekty tego typu:

Nazwa pliku: Wspolrzedne.java

```
public class Wspolrzedne {
  private int x, y;

  public Wspolrzedne(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public static void main(String[] args) {
    Wspolrzedne w1 = new Wspolrzedne(10, 20); // 1
    Wspolrzedne w2 = w1; // 2
    Wspolrzedne w3 = new Wspolrzedne(1, -5); // 3
  }
}
```

Poniższy obrazek przedstawia zmienne oraz obiekty z powyższego przykładu:



Zmienne `w1` oraz `w2` wskazują na ten sam obiekt – utworzony w linii *(1)* i w tej samej linii przypisany do zmiennej `w1`. W linii *(2)* przypisujemy do zmiennej `w2` referencję do tego samego obiektu, na który wskazuje `w1`. Zmienna `w3` wskazuje na inny obiekt, utworzony w linii *(3)*.

Spróbujmy teraz wykonać następujące ćwiczenie: zmienimy wartości `x` oraz `y` obiektu, na który wskazuje zmienna `w3`, aby wartości te odpowiadały wartościom `x` i `y` obiektu, na który wskazują zmienne `w1` oraz `w2`.

Następnie, spróbujemy porównać do siebie zmienne `w1`, `w2`, oraz `w3`, używając operatora ==:

Nazwa pliku: Wspolrzedne.java

```
public class Wspolrzedne {
  private int x, y;

  public Wspolrzedne(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }

  public static void main(String[] args) {
    Wspolrzedne w1 = new Wspolrzedne(10, 20);
    Wspolrzedne w2 = w1;
    Wspolrzedne w3 = new Wspolrzedne(1, -5);

    w3.setX(10); // 1
    w3.setY(20); // 1

    if (w1 == w2) {
      System.out.println("w1 jest rowne w2");
    } else {
      System.out.println("w1 nie jest rowne w2");
    }

    if (w1 == w3) {
      System.out.println("w1 jest rowne w3");
    } else {
      System.out.println("w1 nie jest rowne w3");
    }
  }
}
```

W liniach *(1)*, ustawiamy, za pomocą setterów, wartości pól `x` oraz `y` obiektu wskazywanego przez `w3` na takie same wartości, jakie mają pola `x` i `y` obiektu wskazywanego przez zmienne `w1` i `w2`. Mamy teraz w pamięci dwa obiekty typu `Wspolrzedne` – oba mają pola `x` i `y` ustawione na te same wartości: `x` wynosi `10`, a `y` wynosi `20`.

Pytanie: jakie komunikaty zobaczymy na ekranie?

W wyniku działania powyższego programu, na ekranie zobaczymy następujące komunikaty:

```
w1 jest rowne w2
w1 nie jest rowne w3
```

Zobaczyliśmy takie komunikaty, ponieważ użycie operatora porównania == powoduje, że porównywane są do siebie wartości zmiennych. Dlaczego w takim razie wartości zmiennych `w1` i `w3` nie zostały uznane za równe?

Mogłoby się wydawać, że skoro pola `x` i `y` obiektów wskazywanych przez zmienne `w1` i `w3` są takie same (`x` wynosi `10`, a `y` jest równe `20`), to powinniśmy zobaczyć na ekranie komunikat `"w1 jest rowne w3"`:

```
if (w1 == w3) {
  System.out.println("w1 jest rowne w3");
} else {
  System.out.println("w1 nie jest rowne w3");
}
```

Komunikat, który jednak widzimy, to `"w1 nie jest rowne w3"`, ponieważ operator `==` nie porównuje stanu obiektów, na które te zmienne wskazują (to znaczy nie sprawdza, czy wszystkie pola tych obiektów mają takie same wartości), lecz **sprawdza, czy obie te zmienne wskazuję na ten sam obiekt w pamięci**.

Powtórzmy: wartościami zmiennych `w1` i `w3` są *referencje* do dwóch obiektów w pamięci – tak się składa, że oba te obiekty mają pola `x` i `y` ustawione na takie same wartości, jednakże **są to dwa osobne obiekty w pamięci** – dlatego wynik porównania operatorem == zwraca `false` – zmienne te wskazują na dwa różne obiekty typu `Wspolrzedne` w pamięci.

Z tego powodu, z kolei, porównanie zmiennych `w1` i `w2` zwraca `true` – ponieważ obie te zmienne wskazują na ten sam obiekt w pamięci:

```
Wspolrzedne w1 = new Wspolrzedne(10, 20);
Wspolrzedne w2 = w1; // w2 pokazuje teraz na ten sam obiekt, co zmienna w1

if (w1 == w2) { // zwroci true - zmienne pokazuja na ten sam obiekt w pamieci
  System.out.println("w1 jest rowne w2");
} else {
  System.out.println("w1 nie jest rowne w2");
}
```

Zapamiętajmy: jeżeli porównujemy dwie zmienne za pomocą operatora porównania ==, to porównujemy wartości tych zmiennych. Wartościami zmiennych typów złożonych są referencje do obiektów w pamięci – porównując dwie takie zmienne, zadajemy pytanie:

"Czy te zmienne wskazują na ten sam obiekt w pamięci?"

Ze zmiennymi typów prymitywnych jest tak samo – porównujemy wartości tych zmiennych:

```
int x = 5;
int y = 5;
int z = y;

if (x == y) {
  System.out.println("x == y");
} else {
  System.out.println("x != y");
}

if (x == z) {
  System.out.println("x == z");
} else {
  System.out.println("x != z");
}

if (y == z) {
  System.out.println("y == z");
} else {
  System.out.println("y != z");
}
```

W wyniku działania powyższego fragmentu kodu, na ekranie zobaczymy:

```
x == y
x == z
y == z
```

Każda ze zdefiniowanych zmiennych ma taką samą wartość: `5`, dlatego wszystkie trzy komunikaty wskazują, że zmienne są sobie równe.

Nie mamy tutaj do czynienia z referencjami – wszystkie trzy zmienne `x`, `y`, oraz `z`, są zmiennymi typu prymitywnego `int`. Porównujemy wartości tych zmiennych do siebie, a każda z nich ma przypisaną wartość `5`.

Czy jest w takim razie jakiś sposób, by porównywać obiekty danej klasy nie na podstawie tego, czy są tym samym obiektem, lecz na podstawie innych kryteriów, jak na przykład: czy wartości ich pól są takie same?

Tak – służy do tego specjalna metoda `equals`.

9.7.2 Porównywanie obiektów za pomocą equals

Przypomnijmy działanie operatora == (a także !=): gdy używamy operatora porównania == by porównać dwie zmienne typów referencyjnch, porównywane są adresy obiektów, na które te zmienne wskazują. Operator == zwróci **true** tylko w przypadku, gdy obie zmienne wskazują na dokładnie ten sam obiekt w pamięci:

Nazwa pliku: Osoba.java

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private int wiek;

  public Osoba(String imie, String nazwisko, int wiek) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wiek = wiek;
  }
}
```

Nazwa pliku: PorownywanieObiektow.java

```
public class PorownywanieObiektow {
  public static void main(String[] args) {
    Osoba o1 = new Osoba("Jan", "Nowak", 25);
    Osoba o2 = o1;
    Osoba o3 = new Osoba("Jan", "Nowak", 25);

    if (o1 == o2) {
      System.out.println("o1 i o2 sa rowne");
    } else {
      System.out.println("o1 i o2 nie sa rowne");
    }

    if (o1 == o3) {
      System.out.println("o1 i o3 sa rowne.");
    } else {
      System.out.println("o1 i o3 nie sa rowne");
    }
  }
}
```

Uruchomienie powyższej klasy spowoduje, że na ekranie zobaczymy:

```
o1 i o2 są rowne
o1 i o3 nie sa rowne
```

Pomimo tych samych wartości przechowywanych w dwóch utworzonych obiektach, operator == użyty z obiektami `o1` oraz `o3` zwraca **false**, co widzimy na ekranie. Z kolei użycie == z obiektami `o1` i `o2` zwraca **true**, ponieważ wskazują one na ten sam obiekt w pamięci.

Operatorów == oraz != (różne od) powinniśmy używać tylko wtedy, gdy chcemy sprawdzić, czy dwie zmienne wskazują (bądź nie) na ten sam obiekt w pamięci. Jak już także wiemy, możemy je też stosować do sprawdzenia, czy dana zmienna wskazuje na jakiś obiekt, czy nie – poprzez przyrównanie zmiennej do poznanej już wartości **null**.

Z drugiej jednak strony, w naszych programach często będziemy mieli potrzebę, aby porównać dwa obiekty pod względem wartości ich pól. Jest to na tyle powszechne wymaganie, że w języku Java

istnieje specjalny mechanizm służący do właśnie tego celu – metoda `equals`. Spotkaliśmy się już z tą metodą – używaliśmy jej do porównywania zmiennych typu `String`.

Każda klasa, które chce umożliwić porównywanie jej obiektów w określony sposób, może dostarczyć taką metodę. Musi ona jednak spełniać dwa rodzaje wymagań.

9.7.2.1 Sygnatura metody equals

W pierwszej kolejności, metoda `equals` musi:

- przyjmować jako parametr jeden argument typu `Object`,
- być publiczna (posiadać modyfikator **`public`**),
- zwracać wartość typu **`boolean`**.

Sygnatura metody `equals` wygląda więc następująco:

```
public boolean equals(Object o) {
  // implementacja
}
```

W metodzie tej powinniśmy porównać obiekt, na rzecz którego metoda `equals` została wywołana, do obiektu przesłanego jako argument o nazwie `o`. Jeżeli, wedle naszych kryteriów, obiekty zostaną uznane za równe, to metoda powinna zwrócić wartość **`true`**. Jeżeli obiekty są od siebie różne, powinna zwrócić wartość **`false`**. Zaraz zobaczymy przykładową implementację metody `equals` dla klasy `Osoba`, ale najpierw zaznajomimy się z tajemniczym typem `Object`.

9.7.2.2 Typ Object i krótko o dziedziczeniu

Do tej pory nie wspominaliśmy jeszcze o typie `Object`. Język Java, jak wiele innych języków obiektowych, udostępnia mechanizm *dziedziczenia*, któremu poświęcony jest osobny rozdział. Pozwala on na tworzenie hierarchii klas, które dziedziczą, jedna po drugiej, posiadając cechy i metody klas nadrzędnych, oraz wprowadzając nowe pola i metody.

Wszystkie klasy, jeżeli nie zdefiniują swojej klasy-rodzica, dziedziczą automatycznie po specjalnej klasie o nazwie `Object`, która, jako jedyna, nie ma klasy nadrzędnej.

Dzięki mechanizmowi dziedziczenia, wszystkie obiekty klasy podrzędnej mogą być traktowane jak obiekty klasy nadrzędnej. Dla przykładu, klasa `Zwierze` mogłaby mieć klasy po niej dziedziczące o nazwach `Pies` i `Kot` – możnaby wtedy powiedzieć, że każdy obiekt klasy `Pies` (lub `Kot`) jest równocześnie obiektem klasy `Zwierze`. Tego samego nie można jednak powiedzieć o obiektach klasy `Zwierze` – nie każdy obiekt klasy `Zwierze` jest równocześnie obiektem klasy `Pies` – mógłby to być przecież obiekt klasy `Kot`.

Dlaczego w takim razie argumentem metody `equals` musi być obiekt typu `Object`? Metoda `equals` zdefiniowana jest w klasie `Object` – typem argumentu tej metody jest `Object`. Klasy, które dziedziczą po klasie `Object` (czyli wszystkie klasy), chcąc dostarczyć własną wersję metody `equals`, powinny używać takiej samej sygnatury tej metody, jaka użyta jest w klasie `Object`. Więcej na temat rozszerzenia klas i dziedziczenia metod dowiemy się wkrótce.

Podobnie, jak z metodą `equals`, było z poznaną już metodą `toString`. Ona także zdefiniowana jest w klasie `Object`, a my, dodając ją do naszych klas, dostarczamy własną implementację tej metody, którą nasze klasy dziedziczą po klasie `Object`. Jeżeli nie

napiszemy sami metody `toString`, to wykorzystywana będzie jej domyślna implementacja z klasy `Object`, która wypisuje na ekran nazwę obiektu i jego adres, rozdzielone znakiem @ (małpa), co widzieliśmy w jednym z poprzednich podrozdziałów. Dla przykładu, jeżeli wypisalibyśmy na ekran obiekt klasy `Samochod`, a klasa ta nie miałaby własnej metody `toString`, to zobaczylibyśmy na ekranie komunikat w postaci: `Samochod@1b6d3586`

Klasa `Object` dostarcza "domyślną" implementację metody `equals`, którą dziedziczą po niej automatycznie wszystkie klasy, o ile nie dostarczą one własnej implementacji tej metody. Oznacza to więc, że możemy na obiektach typu `Osoba` wywoływać metodę `equals` pomimo, że jeszcze tej metody w klasie `Osoba` nie zdefiniowaliśmy:

Nazwa pliku: PorownywanieObiektow.java

```
public class PorownywanieObiektow {
  public static void main(String[] args) {
    Osoba o1 = new Osoba("Jan", "Nowak", 25);
    Osoba o2 = o1;
    Osoba o3 = new Osoba("Jan", "Nowak", 25);

    if (o1.equals(o2)) { // 1
      System.out.println("o1 i o2 sa rowne");
    } else {
      System.out.println("o1 i o2 nie sa rowne");
    }

    if (o1.equals(o3)) { // 2
      System.out.println("o1 i o3 sa rowne");
    } else {
      System.out.println("o1 i o3 nie sa rowne");
    }
  }
}
```

W powyższym przykładzie korzystamy z metody `equals` *(1)* *(2)*, której klasa `Osoba` co prawda sama nie definiuje, ale dziedziczy ją po klasie `Object`. Jak, w takim razie, działa "domyślna" implementacja metody `equals`, której używamy powyżej?

Spójrzmy na implementację metody `equals` w klasie `Object`:

Fragment klasy Object z biblioteki standardowej Java

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

Pytanie: co robi powyższa metoda i co zobaczymy na ekranie w wyniku uruchomienia klasy `PorownywanieObiektow`?

Domyślna wersja metody `equals` sprawdza, czy przesłany jako argument obiekt jest tym samym obiektem, co `this` (czyli obiekt, na rzecz którego `equals` zostało wywołane). Używany jest tutaj po prostu operator porównania. Domyślna metoda `equals` sprawdza po prostu, czy dwa obiekty są tym samym obiektem w pamięci.

Powyższy program zadziała więc dokładnie tak samo, jak poprzednia wersja, która używała operatora ==. Na ekranie zobaczymy więc:

```
o1 i o2 sa rowne
o1 i o3 nie sa rowne
```

Każda klasa, która nie zdefiniuje własnej metody `equals`, dziedziczy implementację metody `equals` z klasy `Object`, która to implementacja działa tak samo, jak użycie operatora `==`.

9.7.2.3 Implementacja metody equals w klasie Osoba

Metoda `equals` ma jeszcze jeden zestaw wymagań, o którym zaraz sobie opowiemy, ale najpierw zobaczymy, jak mogłaby wyglądać implementacja metody `equals` dla klasy `Osoba`. Przejdziemy krok po kroku po kolejnych aspektach metody `equals`, ponieważ jest tutaj sporo do wyjaśnienia!

Klasa `Osoba` posiada trzy pola: `imie`, `nazwisko`, oraz `wiek`. Chcielibyśmy, aby metoda `equals` oceniała równość obiektów typu `Osoba` nie na podstawie tego, czy są tym samym obiektem w pamięci, lecz na podstawie tego, czy wszystkie trzy pola: `imie`, `nazwisko`, oraz `wiek`, mają takie same wartości.

Mając dwa obiekty:

```
Osoba o1 = new Osoba("Jan", "Nowak", 25);
Osoba o3 = new Osoba("Jan", "Nowak", 25);
```

Operator porównania oraz domyślna implementacja `equals` z klasy `Object` zwróciłyby **`false`**:

```
// czy zmienne wskazuja na ten sam obiekt?
o1 == o3 // false

// equals odziedziczone z klasy Object
// ponownie sprawdzane jest, czy o1 i o3 wskazuja na ten sam obiekt
o1.equals(o3) // false
```

Natomiast wersja `equals`, którą chcemy zaimplementować, w przypadku dwóch powyższych obiektów powinna zwrócić **`true`**:

```
// equals dedykowane dla klasy Osoba (ktorego jeszcze nie napisalismy)
o1.equals(o3) // true
```

Dodajmy do klasy `Osoba` metodę `equals` o odpowiedniej sygnaturze, która porównuje pola obiektów typu `Osoba`. Zwróćmy uwagę, że porównując pola `imie` oraz `nazwisko`, także korzystamy z metody `equals`:

Nazwa pliku: Osoba.java

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private int wiek;

  public Osoba(String imie, String nazwisko, int wiek) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wiek = wiek;
  }

  public boolean equals(Object o) {
    return this.imie.equals(o.imie) &&
        this.nazwisko.equals(o.nazwisko) &&
        this.wiek == o.wiek;
  }
}
```

Pierwsza wersja metody `equals` dla klasy `Osoba` zwraca **`true`**, jeżeli wszystkie pola obiektu **`this`** (czyli obiektu, na rzecz którego wywołamy `equals`) są równe polom obiektu `o`. Jeżeli jednak spróbowalibyśmy teraz skompilować klasę `Osoba`, to zobaczylibyśmy na ekranie następujące błędy:

```
Osoba.java:13: error: cannot find symbol
        return this.imie.equals(o.imie) &&
                                 ^
  symbol:   variable imie
  location: variable o of type Object
Osoba.java:14: error: cannot find symbol
               this.nazwisko.equals(o.nazwisko) &&
                                     ^
  symbol:   variable nazwisko
  location: variable o of type Object
Osoba.java:15: error: cannot find symbol
               this.wiek == o.wiek;
                             ^
  symbol:   variable wiek
  location: variable o of type Object
3 errors
```

Kompilator informuje nas, że obiekt `o` nie posiada pól `imie`, `nazwisko`, oraz `wiek`. W końcu to obiekty klasy `Osoba` mają te pola, a nie obiekty klasy `Object`. Jak w takim razie wykonać porównanie pól?

W rozdziale o zmiennych dowiedzieliśmy się o istnieniu mechanizmu zwanego *rzutowaniem*. Wartość pewnego typu możemy zrzutować na inny typ, o ile ma to sens – liczbę rzeczywistą możemy zrzutować na liczbę całkowitą, w wyniku czego wartość po przecinku liczby rzeczywistej zostanie ucięta. Z drugiej jednak strony, zamiana stringu `"Witajcie!"` na liczbę rzeczywistą nie ma sensu i nie jest możliwa – kompilator nie zezwoli na taką próbę rzutowania.

Aby zrzutować wartość jednego typu na wartość innego typu, piszemy nazwę typu docelowego w nawiasach przed wartością, którą chcemy zrzutować:

```
// zrzutuj liczbe 3.14 na liczbe calkowita
// wartosc x bedzie wynosic 3
int x = (int) 3.14;

// rzutowanie niewykonalne - blad kompilacji
// Error incompatible types: java.lang.String cannot be converted to double
double d = (double) "Witajcie!"; // BLAD!
```

W poprzednim podrozdziale wspomnieliśmy, że każdy obiekt klasy podrzędnej możemy traktować jako obiekt klasy-rodzica (każdy obiekt klasy `Pies` to `Zwierze`). Dzięki rzutowaniu, możemy także zadziałać w drugą stronę, to znaczy potraktować obiekt klasy nadrzędnej jako obiekt klasy podrzędnej – czyli powiedzieć kompilatorowi: ten obiekt to `Zwierze`, ale ja wiem, że w pamięci jest faktycznie obiekt klasy `Pies`. Dzięki temu, uzyskamy dostęp do pól i metod specyficznych dla klasy `Pies`, które nie występują w klasie `Zwierze`.

Skoro klasa `Object` jest klasą nadrzędną dla wszystkich innych klas, to możemy powiedzieć kompilatorowi, że chcemy przesłany jako argument obiekt `o` traktować jako obiekt klasy `Osoba`:

Kolejna wersja metody `equals` *w klasie* `Osoba`

```
public boolean equals(Object o) {
  Osoba innaOsoba = (Osoba) o;

  return this.imie.equals(innaOsoba.imie) &&
      this.nazwisko.equals(innaOsoba.nazwisko) &&
      this.wiek == innaOsoba.wiek;
}
```

Spróbujmy teraz uruchomić klasę `PorownywanieObiektow` i zobaczmy, co zostanie wypisane na ekran – przypomnijmy kod metody `main` z tej klasy:

Metoda main z klasy PorownywanieObiektow

```
public static void main(String[] args) {
  Osoba o1 = new Osoba("Jan", "Nowak", 25);
  Osoba o2 = o1;
  Osoba o3 = new Osoba("Jan", "Nowak", 25);

  if (o1.equals(o2)) {
    System.out.println("o1 i o2 sa rowne");
  } else {
    System.out.println("o1 i o2 nie sa rowne");
  }

  if (o1.equals(o3)) {
    System.out.println("o1 i o3 sa rowne");
  } else {
    System.out.println("o1 i o2 nie sa rowne");
  }
}
```

Aktualna wersja metody `equals`, którą zdefiniowaliśmy w klasie `Osoba`, spowoduje, że tym razem, w wyniku działania powyższego kodu, na ekranie zobaczymy:

```
o1 i o2 sa rowne
o1 i o3 sa rowne
```

Nasza metoda `equals` działa! Pomimo, że zmienne `o1` i `o3` wskazują na różne obiekty w pamięci, to wynik porównania obiektów `o1` i `o3` dał wynik "są równe" – porównane zostały wartości pól obu obiektów, a nie ich adresy w pamięci. Jest to jednak dopiero początek naszej implementacji metody `equals` – musi zadbać o jeszcze kilka rzeczy.

Zauważmy, że typem argumentu metody `equals` jest `Object`, a jako argument przesyłamy do niej obiekt typu `Osoba` – jak już wiemy, nie ma z tym problemu, ponieważ obiekty typu `Osoba` mogą być także traktowane jako obiekty klasy `Object`.

Ale skoro tak, to czy możemy przesłać jako argument do metody `equals` obiekt innej klasy, na przykład `String`? Spróbujmy:

```
Osoba o1 = new Osoba("Jan", "Nowak", 25);
String powitanie = "Witajcie!";

if (o1.equals(powitanie)) {
  System.out.println("o1 i powitanie sa rowne");
}
```

Powyższy kod skompiluje się bez problemów – w końcu klasa `String` także może być traktowana

jako obiekt klasy `Object`. Kompilacja, co prawda nie zakończy się błędem, ale wykonanie programu już tak:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to Osoba
     at Osoba.equals(Osoba.java:13)
     at PorownywanieObiektow.main(PorownywanieObiektow.java:6)
```

Błąd wykonania programu wynika z poniższej linii kodu metody `equals` z klasy `Osoba`:

```
Osoba innaOsoba = (Osoba) o;
```

W tej linii zakładamy, że każdy obiekt przesłany do metody `equals` będziemy mogli zrzutować do obiektu klasy `Osoba`. Jest to założenie zdecydowanie zbyt optymistyczne – gdy typem argumentu jest `Object`, do metody możemy przesłać obiekt dowolnej klasy. Jak, w takim razie, uchronić się przed nieprawidłowym rzutowaniem?

Każda klasa dziedziczy po klasie `Object` jeszcze inną, przydatną metodę, z której możemy teraz skorzystać. Ta metoda to `getClass`, która zwraca obiekt służący do identyfikacji klasy obiektu.

Korzystając z wyniku metody `getClass` możemy sprawdzić, czy dwa obiekty są obiektami tego samego typu:

Nazwa pliku: PrzykladGetClass.java

```
public class PrzykladGetClass {
  public static void main(String[] args) {
    Osoba osoba = new Osoba("Jan", "Kowalski", 35);
    String powitanie = "Witajcie!";
    Object pewienObiekt = new Osoba("Marek", "Nowak", 30); // 1

    // 2
    System.out.println(osoba.getClass());
    System.out.println(powitanie.getClass());
    System.out.println(pewienObiekt.getClass());

    if (pewienObiekt.getClass() == powitanie.getClass()) { // 3
      System.out.println("Klasy obiektu 'pewienObiekt' i 'powitanie' " +
          "sa takie same.");
    } else {
      System.out.println("Klasy obiektu 'pewienObiekt' i 'powitanie' " +
          "sa rozne.");
    }

    if (pewienObiekt.getClass() == osoba.getClass()) { // 4
      System.out.println("Klasy obiektu 'pewienObiekt' i 'osoba' " +
          "sa takie same.");
    } else {
      System.out.println("Klasy obiektu 'pewienObiekt' i 'osoba' " +
          "sa rozne.");
    }
  }
}
```

W powyższym przykładzie tworzymy trzy obiekty: dwa klasy `Osoba` oraz jeden klasy `String`. Zmiennej typu `Object` *(1)* przypisujemy obiekt klasy `Osoba` – możemy to zrobić, ponieważ każdy obiekt może być traktowany jak obiekt klasy `Object`. Następnie *(2)*, wypisujemy na ekran tekstową reprezentację wyników wywołania metody `getClass` na każdej ze zmiennych.

W wyniku działania powyższego programu, na ekranie zobaczymy:

```
class Osoba
class java.lang.String
class Osoba
Klasy obiektu 'pewienObiekt' i 'powitanie' sa rozne.
Klasy obiektu 'pewienObiekt' i 'osoba' sa takie same.
```

Pierwsze trzy linijki informują nas o typach każdego z obiektów wskazywanego przez trzy zdefiniowane zmienne. Jak widać, pomimo, że zmienna `pewienObiekt` została zdefiniowana jako `Object`, to na ekranie widzimy wypisany tekst `"class Osoba"`, ponieważ w rzeczywistości na obiekt właśnie tej klasy zmienna `pewienObiekt` wskazuje.

Możemy porównać wyniki wywołania metody `getClass` – jak widzimy powyżej, porównanie klas obiektów `pewienObiekt` i `powitanie` daje w wyniku **`false`**, co symbolizuje komunikat:

```
Klasy obiektu 'pewienObiekt' i 'powitanie' sa rozne.
```

Z drugiej jednak strony, porównanie klas obiektów `pewienObiekt` i `osoba` daje w wyniku **`true`**, ponieważ obie zmienne wskazują na obiekt klasy `Osoba`.

Z powyższego faktu możemy skorzystać w naszej metodzie `equals` – zanim zrzutujemy argument o nazwie `o` typu `Object` na obiekt typu `Osoba`, sprawdzimy, czy aby na pewno pod tym argumentem kryje się obiekt tej klasy – spójrzmy na nową wersję metody `equals` w klasie `Osoba`:

Metoda equals z klasy Osoba

```
public boolean equals(Object o) {
  if (this.getClass() != o.getClass()) { // 1
    return false;
  }

  Osoba innaOsoba = (Osoba) o; // 2

  return this.imie.equals(innaOsoba.imie) &&
      this.nazwisko.equals(innaOsoba.nazwisko) &&
      this.wiek == innaOsoba.wiek;
}
```

Na początek metody `equals` dodaliśmy sprawdzanie, czy klasa aktualnego obiektu (**`this`**) jest różna od klasy obiektu przesłanego jako argument – jeżeli klasy obiektów są od siebie różne, to oba obiekty nie mogą być sobie równe – w takim przypadku, zwracamy od razu **`false`**, dzięki czemu nie dojdzie sytuacji, w ktorej spróbujemy zrzutować obiekt innej klasy na obiekt klasy `Osoba` *(2)*.

Dzięki powyższej zmianie, poniższy kod nie spowoduje już błędu działania programu:

```
Osoba o1 = new Osoba("Jan", "Nowak", 25);
String powitanie = "Witajcie!";

if (o1.equals(powitanie)) { // 1
  System.out.println("o1 i powitanie sa rowne");
}
```

Wywołanie metody `equals` z argumentem `powitanie` typu `String` nie stanowi już dla nas problemu – przed potencjalnym problemem broni nas sprawdzenie z użyciem metody `getClass`.

To jednak nie koniec implementacji metody `equals`. Jaki będzie teraz wynik działania poniszego kodu?

Fragment metody main klasy PorownywanieObiektow

```
Osoba o1 = new Osoba("Jan", "Nowak", 25);

if (o1.equals(null)) {
  System.out.println("o1 to null");
} else {
  System.out.println("o1 nie jest nullem");
}
```

Program zakończy się znanym nam już błędem `Null Pointer Exception`:

```
Exception in thread "main" java.lang.NullPointerException
     at Osoba.equals(Osoba.java:13)
     at PorownywanieObiektow.main(PorownywanieObiektow.java:10)
```

Dlaczego zobaczyliśmy taki komunikat?

Winna jest następująca linijka z metody `equals` klasy `Osoba`:

```
if (this.getClass() != o.getClass()) {
```

Próbujemy wywołać metodę `getClass` na obiekcie `o`, który jest nullem. Aby uchronić się przed potencjalnym działaniem na nullowym argumencie, musimy najpierw sprawdzić, czy nie jest nullem:

```
public boolean equals(Object o) {
  if (o == null || this.getClass() != o.getClass()) { // 1
    return false;
  }

  Osoba innaOsoba = (Osoba) o;

  return this.imie.equals(innaOsoba.imie) &&
      this.nazwisko.equals(innaOsoba.nazwisko) &&
      this.wiek == innaOsoba.wiek;
}
```

Przed jakimkolwiek operowaniem na obiekcie `o` dodaliśmy sprawdzenie *(1)*, czy jest on nullem – jeżeli tak, to nie ma szansy, aby obiekt, na rzecz którego wywołaliśmy metodę `equals`, był równy **`null`** – zwracamy od razu **`false`**.

Teraz uruchomienie wcześniejszego kodu podającego argument **`null`** do `equals` nie zakończy się błędem, a na ekranie zobaczymy:

```
o1 nie jest nullem
```

Nasza metoda `equals` jest już *prawie* gotowa. Spójrzmy na kolejny przypadek do rozważenia:

Fragment metody main z klasy PorownywanieObiektow

```
Osoba o3 = new Osoba("Jan", "Nowak", 25);
Osoba o4 = new Osoba(null, "Nowak", 40);

if (o4.equals(o3)) {
  System.out.println("o3 i o4 sa rowne");
} else {
  System.out.println("o3 i o4 nie sa rowne");
}
```

Jaki będzie efekt wykonania powyższego kodu?

Ponownie zobaczymy na ekranie błąd `Null Pointer Exception`:

```
Exception in thread "main" java.lang.NullPointerException
     at Osoba.equals(Osoba.java:19)
     at PorownywanieObiektow.main(PorownywanieObiektow.java:32)
```

Czym spowodowany jest powyższy błąd?

Ponownie próbowaliśmy wywołać metodę na nullowym obiekcie – tym razem podczas porównywanie do siebie pól obiektów:

Fragment metody equals z klasy Osoba

```
return this.imie.equals(innaOsoba.imie) && // 1
    this.nazwisko.equals(innaOsoba.nazwisko) &&
    this.wiek == innaOsoba.wiek;
```

Zawiniła linijka `(1)`. Obiekt `o4`, na rzecz którego wywołaliśmy metodę `equals`, został utworzony z polem `imie` zainicjalizowanym wartością **`null`** (taką wartość przesłaliśmy jako argument do konstruktora). W linijce `(1)` próbujemy, na rzecz pola `imie` aktualnego obiektu (**`this`**), wywołać metodę `equals`, by wartość tego pola porównać z wartością pola `imie` obiektu `innaOsoba`. Wynikiem, w takim przypadku, jest błąd działania programu `Null Pointer Exception`.

Aby naprawić naszą metodę `equals`, dodamy sprawdzanie nulla dla pól `imie` oraz `nazwisko`. Dodatkowo, jeżeli zarówno pole obiektu **`this`**, jak i obiektu obiektu przesłanego jako argument, będą wskazywać na **`null`**, to uznamy, że pole `imie` (lub `nazwisko`) są takie same. Innymi słowy, obiekty o np. takich samych polach `nazwisko` i `wiek` i polach `imie` ustawionych na **`null`** uznamy za takie same – spójrzmy na kolejną wersję metody `equals`:

```
public boolean equals(Object o) {
  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }

  Osoba innaOsoba = (Osoba) o;

  if ((this.imie == null && innaOsoba.imie != null) || // 1
      (this.imie != null && !this.imie.equals(innaOsoba.imie))) { // 2
    return false;
  }

  // 3
  if ((this.nazwisko == null && innaOsoba.nazwisko != null) ||
      (this.nazwisko != null && !this.nazwisko.equals(innaOsoba.nazwisko))) {
    return false;
  }

  return this.wiek == innaOsoba.wiek; // 4
}
```

Dodaliśmy następujące sprawdzenie:

- jeżeli pole `imie` aktualnego obiektu jest nullem, a pole `imie` przesłanego jako argumentu obiektu nie `(1)`

  lub

- pole `imie` aktualnego obiektu nie jestem nullem i nie jest ono równe polu `imie` obiektu przesłanego jako argument `(2)`

to zwracamy `false` – obiekty na pewno nie będą sobie równe. Ponowne sprawdzenie wykonujemy dla pola `nazwisko` *(3)*. Na końcu zwracamy wynik porównania pól `wiek` – jeżeli dotarliśmy w metodzie do tej tego miejsca, to znaczy, że wszystkie poprzednie warunki zostały spełnione i zostaje nam już do sprawdzenia tylko pole `wiek`.

Nasza metoda jest już w zasadzie gotowa, jednak możemy do niej dodać jeszcze jeden warunek. Załóżmy, że nasza klasa ma kilkadziesiąt pól – czasem porównywanie ich wszystkich może być czasochłonne – jeżeli używalibyśmy często metody `equals`, mogłoby się to odbić negatywnie na wydajności naszego programu – możemy jednak wprowadzić proste usprawnienie do naszej metody. Zauważmy, że każdy obiekt jest zawsze równy sobie – w końcu to ten sam obiekt! Możemy w takim razie dodać na początku metody `equals` warunek sprawdzający, czy obiekt przesłany jako argument jest tym samym obiektem, na rzecz którego `equals` zostało wywołane – w takim przypadku możemy od razu zwrócić wartość `true`.

Finalna wersja klasy `Osoba`, z opisanym powyżej usprawnieniem, prezentuje się następująco:

Nazwa pliku: Osoba.java

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private int wiek;

  public Osoba(String imie, String nazwisko, int wiek) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wiek = wiek;
  }

  public boolean equals(Object o) {
    if (this == o) { // 1
      return true;
    }

    if (o == null || this.getClass() != o.getClass()) {
      return false;
    }

    Osoba innaOsoba = (Osoba) o;

    if ((this.imie == null && innaOsoba.imie != null) ||
      (this.imie != null && !this.imie.equals(innaOsoba.imie))) {
      return false;
    }

    if ((this.nazwisko == null && innaOsoba.nazwisko != null) ||
      (this.nazwisko != null && !this.nazwisko.equals(innaOsoba.nazwisko))) {
      return false;
    }

    return this.wiek == innaOsoba.wiek;
  }
}
```

Na początek metody `equals` dodaliśmy sprawdzanie, czy `this` (aktualny obiekt) jest tym samym obiektem, co obiekt `o` przesłany jako argument – to znaczy, czy zarówno `this`, jaki i argument `o`, wskazują na ten sam obiekt w pamięci.

Spójrzmy teraz na kilka przykładów użycia powyższej metody `equals`:

Fragment pliku PorownywanieObiektow.java

```
Osoba x = new Osoba("Jan", "Kowalski", 20);
Osoba y = new Osoba("Jan", "Kowalski", 30);
Osoba z = new Osoba("Jan", "Kowalski", 20);

System.out.println("x rowne y? " + x.equals(y));
System.out.println("x rowne z? " + x.equals(z));
System.out.println("x rowne null? " + x.equals(null));
System.out.println("x rowne 'Witajcie!'? " + x.equals("Witajcie!"));


Osoba a = new Osoba(null, null, 30);
Osoba b = new Osoba(null, null, 30);

System.out.println("a rowne b? " + a.equals(b));
System.out.println("a rowne x? " + a.equals(x));
```

Wynikiem działania powyższego fragmentu kodu jest:

```
x rowne y? false
x rowne z? true
x rowne null? false
x rowne 'Witajcie!'? false
a rowne b? true
a rowne x? false
```

Jak widać po tym podrozdziale, pisanie metod `equals` nie należy do najprostszych – musimy wziąć pod uwagę wiele czynników. Dodatkowo, metody `equals` powinny spełniać pewien *kontrakt*, o którym zaraz sobie opowiemy. Poza tym, musieliśmy przyswoić dużo podstawowych informacji o dziedziczeniu, o którym będziemy się uczyć w jednym z kolejnych rozdziałów. Jeżeli coś jest w tej chwili niezrozumiałe – cierpliwości! Wkrótce wszystko sobie dokładnie wyjaśnimy.

Istnieje wiele bibliotek (na przykład *lombok*), których możemy użyć w naszych programach, które mogą wygenerować za nas ciało metody `equals` na podstawie pól, jakie zawiera nasza klasa, **ale**: wiedza, do czego służy i jak powinna być pisana metoda `equals` jest fundamentalna i bardzo ważna. Można stosować automatyczne generowanie metod `equals` wtedy, gdy wiemy jak działa ta metoda i jakie problemy mogą wyniknąć, gdy jest napisana w sposób nieprawidłowy. Do tych kwestii będziemy jeszcze wracać w rozdziale o dziedziczeniu oraz o kolekcjach.

W ostatnim podrozdziale o metodzie `equals` rozpisany został krok po kroku sposób pisania tej metody. Można tam zajrzeć w celu znalezienia zwięzłego, podstawowego algorymtu dotyczącego pisania metody `equals`.

9.7.2.4 Kontrakt equals

Poza wymaganiami zaadresowanymi w poprzednim podrozdziale, by nasza metoda `equals` działała w różnych przypadkach i nie powodowała kończenia się naszych programów błędem, metoda `equals` musi także spełniać tak zwany *kontrakt equals*. Kontrakt ten ma zapewnić spójne działanie metody `equals` w różnych przypadkach.

Kontrakt equals to zestaw pięciu reguł, które powinna spełniać każda metoda `equals`. To my, jako programiści, jesteśmy odpowiedzialni za napisanie metody `equals` w taki sposób, by te reguły były spełnione. Reguły te opisane są w oficjalnej dokumentacji Java klasy Object – wedle nich, każda metoda `equals` powinna:

1. być *zwrotna* – dla każdego obiektu `x`, jeżeli `x` nie jest nullem, to `x.equals(x)` powinno zwracać **`true`**,
2. być *symetryczna* – dla nienullowych obiektów `x` oraz `y`, jeżeli `x.equals(y)` zwraca **`true`**, to `y.equals(x)` także powinno zwracać **`true`**,
3. być *przechodnia* – dla nienullowych obiektów `x`, `y`, oraz `z`, jeżeli `x.equals(y)` zwraca **`true`**, oraz `y.equals(z)` zwraca **`true`**, to `x.equals(z)` także powinno zwrócić **`true`**,
4. być *spójna* – dla nienullowych obiektów `x` oraz `y`, jeżeli `x.equals(y)` zwraca **`true`** bądź **`false`**, to ponowne wywołanie `x.equals(y)` powinno zwrócić taką samą wartość jak poprzednio, o ile żadne z pól, które jest wykorzystywane do porównania obiektów w metodzie `equals`, nie zostało zmienione w obiektach wskazywanych przez `x` oraz `y`.
5. dla każdego nienullowego obiektu `x`, `x.equals(null)` powinno zawsze zwracać **`false`**.

Przekładając powyższe na język polski:

1. Każdy obiekt powinien być równy sobie.
2. Jeżeli obiekt `x` jest równy `y`, to powinno z tego wynikać, że `y` jest równy `x`.
3. Jeżeli obiekt `x` i `y` są równe oraz `y` i `z` są równe, to powinno z tego wynikać, że także `x` i `z` są sobie równe.
4. Jeżeli `x` i `y` są równe (bądź nierówne), i nie zmienimy tych obiektów, to powinny pozostać równe (bądź nierówne).
5. Żaden nienullowy obiekt nie powinien być uznany za równy nullowi.

Po co nam kontrakt `equals`? Nasze programy powinny działać deterministycznie i to mają zapewnić powyższe założenia. Jeżeli naruszylibyśmy np. regułę drugą, to nasz program mógłby zachowywać się w nieprzewidziany sposób – wynik porównania obiektów, które de facto powinny być uznawane za równe, zależałby od kolejności ich porównywania.

Na pierwszy rzut oka może się wydawać, że nasza implementacja metody `equals` spełnia wszystkie powyższe założenia – i tak jest w rzeczywistości. Łatwo jednak napisać metodę `equals` w taki sposób, żeby naruszyć jedną bądź więcej reguł kontraktu `equals`, szczególnie, gdy w grę wchodzi dziedziczenie. Wrócimy do kontraktu `equals` w rozdziale o dziedziczeniu, gdzie omówimy sobie potencjalne problemy z wynikające z naruszenia powyższych reguł.

Spójrzmy na przykłady użycia metody `equals` z klasy `Osoba`, które udowodnią, że nasza implementacja spełnia kontrakt `equals`:

Nazwa pliku: KontraktEquals.java

```
public class KontraktEquals {
  public static void main(String[] args) {
    Osoba x = new Osoba("Jan", "Nowak", 30);
    Osoba y = new Osoba("Jan", "Nowak", 30);
    Osoba z = new Osoba("Jan", "Nowak", 30);
    Osoba a = new Osoba("Marek", "Kowalski", 35);

    // 1. zwrotnosc - obiekt jest rowny samemu sobie
    System.out.println("1. Czy obiekt x jest rowny sobie? " + x.equals(x));

    // 2. symetrycznosc - jesli x jest rowne y, to y jest rowne x
    System.out.println("2. Czy obiekt x jest rowny obiektowi y? " + x.equals(y));
    System.out.println("2. Czy obiekt y jest rowny obiektowi x? " + y.equals(x));

    // 3. przechodniosc - jesli x jest rowne y i y jest rowne z, to x jest rowne z
    System.out.println("3. Czy obiekt x jest rowny obiektowi y? " + x.equals(y));
    System.out.println("3. Czy obiekt y jest rowny obiektowi z? " + y.equals(z));
    System.out.println("3. Czy obiekt x jest rowny obiektowi z? " + x.equals(z));

    // 4. spojnosc - wynik equals nie zmienia sie, jezeli obiekty sie nie zmienia
    System.out.println("4. Czy obiekt x jest rowny obiektowi y? " + x.equals(y));
    System.out.println("4. Czy obiekt x jest rowny obiektowi a? " + x.equals(a));
    System.out.println("4. Czy obiekt x jest rowny obiektowi y? " + x.equals(y));
    System.out.println("4. Czy obiekt x jest rowny obiektowi a? " + x.equals(a));

    // 5. zaden obiekt nie jest rowny null
    System.out.println("5. Czy obiekt x jest rowny null? " + x.equals(null));
    System.out.println("5. Czy obiekt y jest rowny null? " + y.equals(null));
  }
}
```

W wyniku działania powyższego programu na ekranie zobaczymy – każda z reguł kontraktu `equals` jest spełniona:

```
1. Czy obiekt x jest rowny sobie? true
2. Czy obiekt x jest rowny obiektowi y? true
2. Czy obiekt y jest rowny obiektowi x? true
3. Czy obiekt x jest rowny obiektowi y? true
3. Czy obiekt y jest rowny obiektowi z? true
3. Czy obiekt x jest rowny obiektowi z? true
4. Czy obiekt x jest rowny obiektowi y? true
4. Czy obiekt x jest rowny obiektowi a? false
4. Czy obiekt x jest rowny obiektowi y? true
4. Czy obiekt x jest rowny obiektowi a? false
5. Czy obiekt x jest rowny null? false
5. Czy obiekt y jest rowny null? false
```

9.7.2.5 Equals – przykład z tablicą

Nasze klasy mogą zawierać także tablice – często będziemy chcieli porównywać je w ramach wykonywania metody `equals`.

Jak ocenić, czy dwie tablice są równe? Dwie tablicy uznamy za równe, gdy:

- obie będę nullami

  lub

- obie będą miały tyle samo elementów i elementy te będą takie same parami, to znaczy obiekt spod indeksu `0` z tablicy pierwszego obiektu będzie taki sam, jak obiekt pod indeksem `0` z tablicy drugiego obiektu itd. dla każdego indeksu w tablicy.

Dwie uwagi do porównywanie tablic:

1. Jeżeli tablice przechowują elementy typów złożonych, to pamiętajmy, aby porównywać do siebie te obiekty za pomocą metody `equals` ich typu, np. mając tablicę stringów `String[]`, poszczególne elementy tych tablic będziemy do siebie porównywać za pomocą metody `equals` z klasy `String`.
2. Porównywanie dużych tablic może być czasochłonne i wpływać negatywnie na wydajność naszych programów – jeżeli mamy jeszcze inne pola w naszych klasach, to porównujmy je na początku, a porównywanie tablic i innych złożonych typów zostawmy na koniec – w ten sposób, porównując inne pola, istnieje szansa, że znajdziemy różnicę w porównywanych polach i nie będziemy musieli już wykonywać czasochłonnego porównywania tablic.

Jak zaraz zobaczymy, trzeba się trochę napisać, aby porównać w metodzie `equals` dwie tablice. Pod koniec tego podrodziału zobaczymy, jak możemy uprościć sobie życie wykorzystując do tego celu bibliotekę standardową Java.

Poniżej zaprezentowane zostały jeszcze dwa przykłady metody `equals` dla klas `Koszyk` oraz `Owoc`. W pierwszej kolejności, przyjrzymy się metodzie `equals` klasy `Owoc`:

Nazwa pliku: Owoc.java

```
public class Owoc {
  private String nazwa;
  private double cena;

  public Owoc(String nazwa, double cena) {
    this.nazwa = nazwa;
    this.cena = cena;
  }

  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }

    if (o == null || this.getClass() != o.getClass()) {
      return false;
    }

    Owoc other = (Owoc) o; // 1

    if ((this.nazwa == null && other.nazwa != null) ||
        (this.nazwa != null && !this.nazwa.equals(other.nazwa))) {
```

```
      return false;
    }

    return this.cena == other.cena;
  }
}
```

Metoda `equals` klasy `Owoc` jest bardzo podobna do metody `equals` klasy `Osoba` – mamy w niej o jedno pole do porównania mniej, a także rzutujemy argument `o` nie do klasy `Osoba`, lecz do klasy `Owoc` *(1)*.

Spójrzmy teraz na metodę `equals` klasy `Koszyk`, która jest zdecydowania ciekawsza – jednym z pól klasy `Koszyk` jest tablica obiektów typu `Owoc` – zauważmy, jak w metodzie `equals` porównujemy do siebie tablice `owoce` obiektów **`this`** oraz `other`:

Nazwa pliku: Koszyk.java

```
public class Koszyk {
  private Owoc[] owoce;
  private boolean oplacony;

  public Koszyk(Owoc[] owoce, boolean oplacony) {
    this.owoce = owoce;
    this.oplacony = oplacony;
  }

  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }

    if (o == null || this.getClass() != o.getClass()) {
      return false;
    }

    Koszyk other = (Koszyk) o;

    if (this.oplacony != other.oplacony) {
      return false;
    }

    // jezeli obie tablice sa ta sama tablica w pamieci
    // lub obie tablice sa nullem, to zwracamy true
    if (this.owoce == other.owoce) { // 1
      return true;
    }

    if (this.owoce == null || other.owoce == null) { // 2
      return false;
    }

    // tablice moga byc rowne, gdy obie maja tyle samo elementow
    // jezeli liczba elementow sie rozni - zwracamy false
    if (this.owoce.length != other.owoce.length) { // 3
      return false;
    }

    for (int i = 0; i < this.owoce.length; i++) { // 4
      // jezeli para obiektow z obu tablic pod danym indeksem
      // sie rozni, to zwracamy false
```

```
        Owoc o1 = this.owoce[i]; // 5
        Owoc o2 = other.owoce[i]; // 6

        if ((o1 == null && o2 != null) || // 7
          (o1 != null && !o1.equals(o2))) {
          return false; // 8
        }
      }

      // jezeli dotarlismy do tego miejsca, to znaczy, ze nie znalezlismy
      // zadnej roznicy pomiedzy obiekami this i other
      return true; // 9
    }
  }
```

Klasa `Koszyk` zawiera dwa pola: `oplacony` oraz `owoce`. Początek metody `equals` jest taki sam, jak w klasie `Owoc`.

Od linii *(1)* zaczynamy porównywanie tablic `owoce` z obu obiektów. Jedną instrukcją warunkową sprawdzamy dwa przypadki:

1. Czy tablica `owoce` w obiekcie **`this`** i `other` jest tą samą tablicą w pamięci?
2. Czy obie referencje: **`this`**`.owoce` oraz `other.owoce` są nullami?

W obu przypadkach uznajemy tablice za równe i zwracamy **`true`**.

Jeżeli docieramy do warunku z linii *(2)*, to jesteśmy pewni, że obie tablice nie są nullem – taką możliwość eliminuje warunek z lini *(1)*. Jednakże, nadal jedna z tablica może być nullem – sprawdzamy to w linii *(2)* – jeżeli tak jest, to tablice nie mogą być równe i zwracamy **`false`**.

Mając wyeliminowane przypadki, gdy jedna z tablic (bądź obie) jest nullem, możemy sprawdzić w lini *(3)*, czy obie zawierają tyle samo elementów – jeżeli nie, to na pewno nie uznamy je za takie same – zwracamy **`false`**.

Gdy już wiemy, że tablice nie są nullami i zawierają tyle samo elementów, przechodzimy do porównania ich elementów, jeden po drugim *(4)*. Przypisujemy do pomocniczych zmiennych elementy tablic znajdujące się pod aktualnym indeksem pętli *(5)* *(6)*. Następnie *(7)*, porównujemy do siebie oba elementy tablicy (do których odnosimy się poprzez zmienne `o1` i `o2`). Zauważmy, że skoro elementami tablicy `owoce` są obiekty typu `Owoc`, to musimy do ich porównania skorzystać z metody `equals` klasy `Owoc`. Bierzemy także pod uwagę, że elementy mogą być nullami. Jeżeli wykryjemy różnicę w którejś z par elementów obu tablic, zwracamy **`false`** *(8)*.

Jeżeli dotrzemy na koniec `equals` do linii *(9)*, będzie to oznaczało, iż nie znaleźliśmy żadnej różnicy w polach obiektów **`this`** oraz `other` i powinniśmy zwrócić **`true`**.

Jak widać, porównywanie tablic wymaga napisania kilkunastu linii kodu. Jest to na tyle częsta operacja, że biblioteka standardowa Java udostępnia metodę, która wykonuje właśnie to zadanie – możemy skorzystać z niej po zaimportowaniu klasy `Arrays` do naszego pliku źródłowego. O importowaniu klas opowiemy sobie w jednym z kolejnych podrozdziałów.

Aby uprościć nasz kod i skorzystać z istniejącej metody do porównywania tablicy, na początku naszego programu powinniśmy dodać następującą linię kodu, dzięki której będziemy mogli skorzystać z klasy `Arrays`:

```
import java.util.Arrays;
```

Następnie, cały kod zawarty między liniami *(1)* a *(9)* możemy zastąpić jedną, poniższą linią kodu:

```
return Arrays.equals(this.owoce, other.owoce);
```

Metoda `equals` z klasy `Arrays` nie powinna być mylona z metodami `equals`, które pisaliśmy do tej pory – przyjmuje ona dwie tablice obiektów i zwraca **`true`**, jeżeli:

- obie zmienne wskazują na tą samą tablicę

  lub

- obie tablice są nullami

  lub

- obie tablice mają taką samą liczbę elementów i każda para elementów pod kolejnymi indeksami jest sobie równa – wykorzystywana jest metoda `equals` obiektów będących elementami tablic (obiekty są też uznawane za równe, jeżeli oba są nullami).

W przeciwnym razie, metoda `Arrays.equals` zwraca **`false`**. Spójrzmy na prosty przykład użycia `Arrays.equals`:

Nazwa pliku: PrzykladUzyciaArraysEquals.java

```
import java.util.Arrays;

public class PrzykladUzyciaArraysEquals {
  public static void main(String[] args) {
    int[] tablica1 = { 1, 2, 3 };
    int[] tablica2 = { 1, 2, 3 };
    int[] tablica3 = { 4, 5 };

    if (Arrays.equals(tablica1, tablica2)) {
      System.out.println("tablica1 == tablica2");
    } else {
      System.out.println("tablica1 != tablica2");
    }

    if (Arrays.equals(tablica1, tablica3)) {
      System.out.println("tablica1 == tablica3");
    } else {
      System.out.println("tablica1 != tablica3");
    }
  }
}
```

Wynik działania powyższego przykładu:

```
tablica1 == tablica2
tablica1 != tablica3
```

Warto sprawdzić, czy dany problem nie został już zaadresowany i czy w bibliotece standardowej Java (i nie tylko) nie znajduje się klasa bądź metoda, którą możemy wykorzystać do rozwiązania danego problemu. Warto także, z drugiej strony, rozumieć z czym wiążę się rozwiązanie danego problemu i mieć to na uwadze korzystając z gotowego rozwiązania.

Na koniec, spójrzmy na kilka przykładów użycia metody `equals` z klasy `Koszyk`:

Fragment pliku Koszyk.java

```
public static void main(String[] args) {
    Owoc czeresnie = new Owoc("Czeresnie", 10.0);
    Owoc jablko = new Owoc("Jablko", 5.0);

    Koszyk koszyk1 = new Koszyk(new Owoc[] { czeresnie, jablko }, false);
    Koszyk koszyk2 = new Koszyk(new Owoc[] { czeresnie, jablko }, true);
    Koszyk koszyk3 = new Koszyk(new Owoc[] { czeresnie, jablko }, false);
    Koszyk koszyk4 = new Koszyk(null, false);
    Koszyk koszyk5 = new Koszyk(new Owoc[] { jablko, czeresnie }, false);
    Koszyk koszyk6 = new Koszyk(
        new Owoc[] { czeresnie, jablko, czeresnie }, false);

    System.out.println(koszyk1.equals(koszyk2));
    System.out.println(koszyk1.equals(koszyk3));
    System.out.println(koszyk1.equals(koszyk4));
    System.out.println(koszyk1.equals(koszyk5));
    System.out.println(koszyk1.equals(koszyk6));
    System.out.println(koszyk1.equals(null));
}
```

Tylko obiekty wskazywane przez zmienne `koszyk1` i `koszyk3` są takie same:

```
false
true
false
false
false
false
```

9.7.2.6 Krok po kroku – pisanie metody equals

Pisząc metodę `equals`, musimy pamiętać o kilku ważnych aspektach związanych z porównywaniem obiektów. Poniższa instrukcja opisuje krok po kroku jak dodać metodę `equals` do Twojej klasy:

1. Zacznij od napisania sygnatury metody `equals` (pamiętaj o modyfikatorze **`public`** i typie argumentu `Object`!):

```
public boolean equals(Object o) {

}
```

2. Dodaj warunek sprawdzający, czy przesłany jako argument obiekt `o` nie jest tym samym obiektem, co aktualny obiekt (który reprezentowany jest przez słowo kluczowe **`this`**) – jeżeli tak, to od razu zwróć **`true`**:

```
if (this == o) {
  return true;
}
```

3. Nastepnie sprawdź, czy przesłany jako argument obiekt `o` jest nullem lub jest innego typu, niż obiekt wskazywany przez **`this`** (aby wyeliminować próby porównywania obiektów niekompatybilnych typów do naszego obiektu) – jeżeli tak, zwróć od razu **`false`**:

```
if (o == null || this.getClass() != o.getClass()) {
  return false;
}
```

4. Zdefiniuj zmienną typu klasy, do której dodajesz `equals` i przypisz do niej zrzutowany obiekt `o` przesłany jako argument (dochodząc w kodzie do tego miejsca będziemy pewni, że rzutowanie się powiedzie, gdyż sprawdzamy typ obiektu w punkcie 3.):

```
Osoba other = (Osoba) o;
```

5. Na końcu porównaj do siebie wszystkie pola obu obiektów (**`this`** oraz `other`), które powinny być wzięte pod uwagę podczas sprawdzania równości dwóch obiektów. Jeżeli któreś z pól się różni, zwróć **`false`**. Jeżeli wszystkie pola są sobie równe – zwróć **`true`**:

```
if ((this.imie == null && innaOsoba.imie != null) ||
    (this.imie != null && !this.imie.equals(innaOsoba.imie))) {
  return false;
}

if ((this.nazwisko == null && innaOsoba.nazwisko != null) ||
    (this.nazwisko != null && !this.nazwisko.equals(innaOsoba.nazwisko))) {
  return false;
}

return this.wiek == innaOsoba.wiek;
```

Finalnie, przykładowa metoda `equals` prezentuje się następująco:

```
public boolean equals(Object o) {
  // sprawdz, czy przeslany obiekt jest tym samym obiektem,
  // na rzecz ktorego zostala wywolana metoda equals
  if (this == o) {
    return true;
  }

  // sprawdz, czy nie przeslano nulla lub obiektu innej,
  // niekompatybilnej klasy (np. String)
  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }

  // powyzsza instrukcja if zapewnia, ze ponizsza instrukcja
  // zakonczy sie sukcesem - rzutujemy referencje do obiektu typu Object,
  // (przeslana jako argument), do obiektu typu Osoba, aby uzyskac
  // dostep do pol imie, nazwisko, oraz wiek
  Osoba innaOsoba = (Osoba) o;

  // porownujemy kolejne pola do siebie - musimy uwzglednic,
  // ze pola typow zlozonych moga byc nullowe
  // jezeli pola aktualnego i porownywanego obiektu sa nullowe,
  // to uznajemy je za rowne
  if ((this.imie == null && innaOsoba.imie != null) ||
    (this.imie != null && !this.imie.equals(innaOsoba.imie))) {
    return false;
  }

  if ((this.nazwisko == null && innaOsoba.nazwisko != null) ||
    (this.nazwisko != null && !this.nazwisko.equals(innaOsoba.nazwisko))) {
    return false;
 }

  return this.wiek == innaOsoba.wiek;
}
```

Uwaga! Podczas porównywania pól musimy wziąć pod uwagę kilka bardzo istotnych reguł:

a) Wszystkie pola typów złożonych należy porównywać do siebie za pomocą ich metod `equals`.

b) Musimy wziąć pod uwagę, że pola typów złożonych mogą być nullami. Należy zadecydować także, gdy dane pole ma wartość **`null`** w obiekcie **`this`** oraz `other`, czy będą one traktowane jako równe, czy nie.

c) Jeżeli polem jest tablica, to należy porównać wszystkie elementy obu tablic. Jeżeli jest to tablica typu złożonego (np. `String[]` – tablica stringów), elementy należy porównywać do siebie za pomocą metody `equals` typu `String` (patrz punkt *b* powyżej).

d) Jeżeli mamy wiele pól do porównania, to warto zaczynać porównywanie od najprostszych pól, by potencjalnie jak najszybciej znaleźć różnicę i zwrócić **`false`**. Dla przykładu, gdyby obiekty naszych klas miały tablice z tysiącem elementów i pole `nazwa` typu `String`, to należy najpierw porównać do siebie pola `nazwa` – porównywanie tablic jako pierwszych może być czasochłonne i wpłynąć na wydajność naszego programu.

9.7.3 Podsumowanie

- Zmienne typów referencyjnych wskazują na obiekty w pamięci – nie są one obiektami, lecz referencjami do obiektów.
- W poniższym fragmencie kodu tworzone są trzy zmienne mogące pokazywać na obiekty typu `Wspolrzedne`, ale tylko dwa obiekty tego typu – zmienne `w1` i `w2` wskazują na ten sam obiekt w pamięci:

```
Wspolrzedne w1 = new Wspolrzedne(10, 20);
Wspolrzedne w2 = w1;
Wspolrzedne w3 = new Wspolrzedne(1, -5);
```

- Operator == użyty do porównania dwóch zmiennych typu referencyjnego odpowiada na pytanie: *"Czy te zmienne wskazują na ten sam obiekt w pamięci?"* – nie porównuje on wartości pól obu obiektów wskazywanych przez zmienne.
- W związku z powyższym, poniższy kod wypisze `"w1 jest rowne w2"` oraz `"w1 nie jest rowne w3"` – `w1` i `w2` są tym samym obiektem, a `w1` i `w3` – pomimo, że obiekty, na które wskazują, mają takie same wartości – są dwoma różnymi obiektami:

```
if (w1 == w2) {
  System.out.println("w1 jest rowne w2");
} else {
  System.out.println("w1 nie jest rowne w2");
}

if (w1 == w3) {
  System.out.println("w1 jest rowne w3");
} else {
  System.out.println("w1 nie jest rowne w3");
}
```

- Operatorow == oraz != (różne od) powinniśmy używać tylko wtedy, gdy chcemy sprawdzić, czy dwie zmienne wskazują (bądź nie) na ten sam obiekt w pamięci. Możemy je też stosować do sprawdzenia, czy dana zmienna wskazuje na jakiś obiekt, czy nie – poprzez przyrównanie zmiennej do wartości **`null`**.
- Aby porównać dwa obiekty na podstawie wartości, jakie mają ich pola, stosujemy specjalną metodę `equals`.
- Metoda `equals` musi mieć określoną sygnaturę. Metoda `equals` musi:
  a) przyjmować jako parametr jeden argument typu `Object`,
  b) być publiczna (posiadać modyfikator **`public`**),
  c) zwracać wartość typu **`boolean`**.

```
public boolean equals(Object o) {
  // implementacja
}
```

- W metodzie `equals` powinniśmy porównać obiekt, na rzecz którego metoda ta została wywołana, do obiektu przesłanego jako argument o nazwie `o`. Jeżeli, wedle naszych kryteriów, obiekty zostaną uznane za równe, to metoda powinna zwrócić wartość **`true`**. Jeżeli obiekty są od siebie różne, powinna zwrócić wartość **`false`**.

- Typ argumentu metody `equals` to `Object`. Jest to typ nadrzędny dla wszystkich typów złożonych (klas) w języku Java.
- W Javie istnieje mechanizm nazywany *dziedziczeniem*, który pozwala na rozszerzanie istniejących już klas, w wyniku czego nowa klasa posiada wszystkie pola i metody klasy, po której dziedziczy, i może dostarczyć własne pola i metody.
- Dziedziczenie pozwala na traktowanie obiektów klas nadrzędnych jak obiekty klasy-rodzica, np. obiekty klas `Pies` i `Kot`, dziedziczące po klasie `Zwierze`, można traktować jak obiekty klasy `Zwierze`.
- Klasa `Object` dostarcza domyślną wersję metody `equals`, która sprawdza, czy przesłany jako argument obiekt jest tym samym obiektem, co **`this`** (czyli obiekt, na rzecz którego `equals` zostało wywołane). Domyślna metoda `equals` sprawdza więc, czy dwa obiekty są tym samym obiektem w pamięci (tak jak operator ==).
- Jako argument `equals` powinniśmy podać `Object`, ponieważ tak zdefiniowana jest ta metoda w klasie `Object`. Dostarczając w naszej klasie własną implementację tej metody, powinniśmy zachować taką samą jej sygnaturę.
- Poza wymaganiami odnośnie sygnatury metody `equals`, powinna ona także spełanić tzw. *kontrakt equals*, który jest zestawem pięciu reguł.
- Wedle kontraktu `equals`:
  1. Każdy obiekt powinien być równy sobie (metoda `equals` jest wtedy *zwrotna*).
  2. Jeżeli obiekt `x` jest równy `y`, to powinno z tego wynikać, że `y` jest równy `x` (metoda `equals` jest wtedy *symetryczna*)
  3. Jeżeli obiekt `x` i `y` są równe oraz `y` i `z` są równe, to powinno z tego wynikać, że także `x` i `z` są sobie równe (metoda `equals` jest wtedy *przechodnia*).
  4. Jeżeli `x` i `y` są równe/nierówne, i nie zmienimy tych obiektów, to powinny pozostać równe/nierówne (metoda `equals` jest wtedy *spójna*).
  5. Żaden nienullowy obiekt nie powinien być uznany za równy nullowi.
- Nasze programy powinny działać deterministycznie i to mają zapewnić powyższe założenia. Jeżeli naruszylibyśmy np. regułę drugą, to nasz program mógłby zachowywać się w nieprzewidziany sposób – wynik porównania obiektów, które de facto powinny być uznawane za równe, zależałby od kolejności ich porównywania.
- Kontrakt `equals` opisany jest w oficjalnej dokumentacji języka Java.
- Tworzenie metody `equals` krok po kroku opisane zostało w podrozdziale 9.7.2.6. Krok po kroku – pisanie metody equals.

9.7.4 Pytania

1. Czym różni się porównywanie obiektów za pomocą operatora porównania == i metody `equals`?
2. Czy do przyrównywania zmiennych do wartości `null` możemy korzystać z operatorów == i !=, czy powinniśmy korzystać z metody `equals`?
3. Jak powinna wyglądać sygnatura metody `equals`?
4. Czy dla dwóch zmiennych `x` i `y`, wskazujących na ten sam obiekt w pamięci, metoda `x.equals(y)` może zwrócić `false`?
5. Na podstawie jakich warunków metoda `equals` powinna zwrócić `true` lub `false`?
6. Co to jest kontrakt `equals`?
7. Jak rzutuje się wartość danego typu na wartość innego typu?
8. Do czego służy metoda `getClass`?
9. Jeżeli klasa `PewnaKlasa` ma pola `int liczba`, `String tekst`, oraz `char[] znaki`, to jak, biorąc pod uwagę typy pól klasy `PewnaKlasa`, powinniśmy sprawdzić równość dwóch obiektów tej klasy?
10. Do czego służy metoda `Arrays.equals`, przyjmująca jako argumenty dwie tablice?
11. Czy metoda `equals`, gdy jej argumentem jest `null`, na przykład `x.equals(null)`, może zwrócić `true`?

```
// klasa na potrzeby zadania 12, 13, 14
public class A {
  private int liczba;

  public A(int liczba) {
    this.liczba = liczba;
  }
}
```

12. Biorąc pod uwagę klasę `A` zdefiniowaną powyżej, jaki będzie wynik uruchomienia poniższego fragmentu kodu? Czy kod w ogóle się skompiluje?

```
public static void main(String[] args) {
  A a1 = new A(10);
  A a2 = a1;
  A a3 = new A(10);

  System.out.println("a1 rowne a2? " + a1.equals(a2));
  System.out.println("a1 rowne a3? " + a1.equals(a3));
}
```

13. Czy poniższa metoda `equals` byłaby poprawna dla klasy `A`?

```
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }

  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }

  return this.liczba == o.liczba;
}
```

14. Jaki będzie wynik działania poniższej metody `main`, gdyby klasa `A` miała załączoną poniżej metodę `equals`?

```
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }

  if (this.getClass() != o.getClass() || o == null) {
    return false;
  }

  A other = (A) o;

  return this.liczba == other.liczba;
}
```

```
public static void main(String[] args) {
  A a1 = new A(10);
  A a2 = a1;
  A a3 = new A(10);

  System.out.println("a1 rowne a2? " + a1.equals(a2));
  System.out.println("a1 rowne a3? " + a1.equals(a3));
  System.out.println("a1 rowne null? " + a1.equals(null));
}
```

9.7.5 Zadania

9.7.5.1 Klasa Punkt z equals

Napisz klasę `Punkt`, która będzie zawierała punkt na płaszczyźnie opisany przez dwie wartości `x` oraz `y` (pola typu `int`). Napisz konstruktor inicjalizujący pola `x` i `y`, a także zaimplementuj metodę `equals`. Sprawdź, czy metoda działa zgodnie z założeniami.

9.7.5.2 Klasa Figura z equals

Napisz klasę `Figura`, która będzie zawierała tablicę obiektów typu `Punkt`. Pole z tablicą nazwij `wierzcholki`. Napisz konstruktor inicjalizujący pole `wierzcholki`, a także zaimplementuj metodę `equals` dla klasy `Figura`. Skorzystaj z metody `Arrays.equals` do porównania tablic.

9.8 Referencje do obiektów

Z poprzednich rozdziałów wiemy, że zmienne typów referencyjnch wskazują na obiekty w pamięci – są one *referencjami* do tych obiektów. Gdy definiujemy zmienną typu `Osoba`, zmienna ta będzie mogła wskazywać na obiekty typu `Osoba` utworzone wcześniej za pomocą operatora **`new`**.

Dla przykładu, spójrzmy na poniższy fragment kodu – ile obiektów jest w nim tworzonych?

```
Osoba o1 = new Osoba("Jan", "Nowak", 25);
Osoba o2 = o1;
```

W tym fragmencie definiujemy dwie zmienne mogące pokazywać na obiekty typu `Osoba`, ale tworzymy tylko jeden obiekt tego typu – obie zmienne wskazują na ten sam, jeden obiekt w pamięci.

Jeżeli zmienilibyśmy pole `imie` za pomocą zmiennej `o1`, to po wypisaniu imienia za pomocą zmiennej `o2` zobaczylibyśmy tą zmianę:

```
System.out.println("imie o1: " + o1.getImie());
System.out.println("imie o2: " + o2.getImie());

o1.setImie("Karol");

System.out.println("imie o1: " + o1.getImie());
System.out.println("imie o2: " + o2.getImie());
```

Wynikiem działania powyższego kodu jest:

```
imie o1: Jan
imie o2: Jan
imie o1: Karol
imie o2: Karol
```

Dzieje się tak dlatego, że zmienne `o1` i `o2` wskazują na ten sam obiekt – nieważne, za pomocą której referencji się do niego odnosimy – nadal jest to ten sam, jeden obiekt typu `Osoba`.

W tym rozdziale zobaczymy, że rozróżenienie referencji do obiektu, oraz obiektu, do którego ta referencja się odnosi, jest bardzo ważne i może przysporzyć problemów, jeżeli nie weźmiemy pod uwagę konsekwencji, jaką niesie ze sobą współdzielenie obiektów.

9.8.1 Przesyłanie i modyfikowanie obiektów w metodach

W rozdziale o metodach wspomnieliśmy, że modyfikowanie obiektów przesłanych do metody odbywa się na oryginalnym obiekcie, a nie na kopii obiektu. Spójrzmy na poniższy przykład – skorzystamy z klasy `Produkt` z jednego z poprzednich podrozdziałów:

Nazwa pliku: Produkt.java

```
public class Produkt {
  private double cena;
  private String nazwa;

  public Produkt(double cena, String nazwa) {
    this.cena = cena;
    this.nazwa = nazwa;
  }
  public void setCena(double cena) {
    this.cena = cena;
  }

  public void setNazwa(String nazwa) {
    this.nazwa = nazwa;
  }

  public String toString() {
    return "Produkt o nazwie " + nazwa + " kosztuje " + cena;
  }
}
```

Prześlemy obiekt klasy `Produkt` do metody, a także wartość typu `double`, zmodyfikujemy je w metodzie i zobaczymy, jakie wartości będą miały oryginalne zmienne:

Nazwa pliku: ModyfikacjeObiektowPrzyklady.java

```
public class ModyfikacjeObiektowPrzyklady {
  public static void main(String[] args) {
    Produkt ksiazka = new Produkt(49.0, "Hyperion");

    System.out.println("Ksiazka przed zmiana ceny: " + ksiazka);
    double nowaCena = 30.0;

    zmienCene(nowaCena, ksiazka); // 1

    System.out.println("nowaCena po wywolaniu metody: " + nowaCena);
    System.out.println("Ksiazka po zmianie ceny: " + ksiazka);
  }

  public static void zmienCene(double cena, Produkt produkt) {
    cena = cena * 1.23;
    produkt.setCena(cena); // 2
  }
}
```

Powyższa klasa spowoduje wypisanie na ekran:

```
Ksiazka przed zmiana ceny: Produkt o nazwie Hyperion kosztuje 49.0
nowaCena po wywolaniu metody: 30.0
Ksiazka po zmianie ceny: Produkt o nazwie Hyperion kosztuje 36.9
```

Jak widzimy, zmiana ceny w metodzie `zmienCene` *(2)* odbyła się na oryginalnym obiekcie

`Produkt`, utworzonym w metodzie `main`, a przesłanym do metody `zmienCene` w linii *(1)*, a nie na kopii tego obiektu. Z drugiej strony, wartość zmiennej `nowaCena`, po powrocie z metody `zmienCene`, nie zmieniła się – pozostała równa `30.0`.

Dzieję się tak dlatego, ponieważ do metody `zmienCene`, jako drugi argument, przesyłamy referencję do obiektu w pamięci. W związku z tym, w metodzie `zmienCene` uzyskujemy dostęp do oryginalnego obiektu typu `Produkt`, na który wskazuje zmienna o nazwie `ksiazka`. Ten oryginalny obiekt typu `Produkt` modyfikujemy w linii *(2)*, odnosząc się do niego za pomocą zmiennej (argumentu) `produkt`.

Z kolei pierwszy argument, `cena`, jest typu prymitywnego – zmienne typu prymitywnego wskazują na konkretne, "proste" wartości w pamięci – podczas przesyłania do metod są one kopiowane. Wszelkie zmiany w metodach nie są odzwierciedlane w oryginalnych zmiennych.

Zwróćmy jeszcze uwagę na jeden fakt. Argumenty typu złożonego (jak argument `produkt` typu `Produkt` powyżej) *także są przesyłane jako kopie*, ale kopiowany jest adres obiektu, a nie cały obiekt, ponieważ zmienne typu złożonego (referencyjnego) zawierają referencję (odniesienie) do obiektu do pamięci – i właśnie to odniesienie do obiektu jest kopiowane i przesyłane do metody. Za pomocą tej referencji mamy dostęp do obiektu w pamięci.

Przypomnijmy także, że podobnie dzieje się z tablicami, które także są typami złożonymi – gdy prześlemy tablicę do metody i zmodyfikujemy ją, to wszelkie zmiany będą wykonywane na oryginalnej tablicy.

Rozróżnienie pomiędzy obiektami w pamięci a referencjami do nich jest bardzo istotne podczas programowania w języku Java – możemy mieć wiele referencji (zmiennych) do tego samego obiektu w pamięci, co może prowadzić do niechcianych i trudnych do zdiagnozowania błędów, o czym zaraz opowiemy sobie dokładniej.

Spójrzmy na inny przykład związany z obiektami i referencjami – co tym razem zostanie wypisane na ekran?

```
public static void main(String[] args) {
  Produkt owoc = new Produkt(10.0, "Czeresnie"); // 1

  System.out.println("Owoc przed zmiana: " + owoc);

  zmienProdukt(owoc, 5.0, "Jablko");

  System.out.println("Owoc po zmianie: " + owoc);
}

public static void zmienProdukt(Produkt produkt, double nowaCena, String nowaNazwa){
  produkt = new Produkt(nowaCena, nowaNazwa); // 2
}
```

Na ekranie zobaczymy dwa razą taką samą informację o obiekcie `owoc`:

```
Owoc przed zmiana: Produkt o nazwie Czeresnie kosztuje 10.0
Owoc po zmianie: Produkt o nazwie Czeresnie kosztuje 10.0
```

Dlaczego przypisanie do argumentu `produkt` nowo utworzonego obiektu typu `Produkt` w metodzie `zmienProdukt` *(2)* nie spowodowało zmiany oryginalnego obiektu utworzonego w linii *(1)*?

Zrozumienie, co dzieje się w powyższym przykładzie jest bardzo istotne – przeanalizujmy, co robi poniższa linia kodu:

```
produkt = new Produkt(nowaCena, nowaNazwa);
```

W tej linii tworzymy nowy obiekt typu `Produkt` na podstawie wartości przesłanych jako argumenty. Następnie, obiekt ten przypisywany jest do argumentu `produkt`, który jest prostu zmienną mogącą pokazywać na obiekty typu `Produkt`.

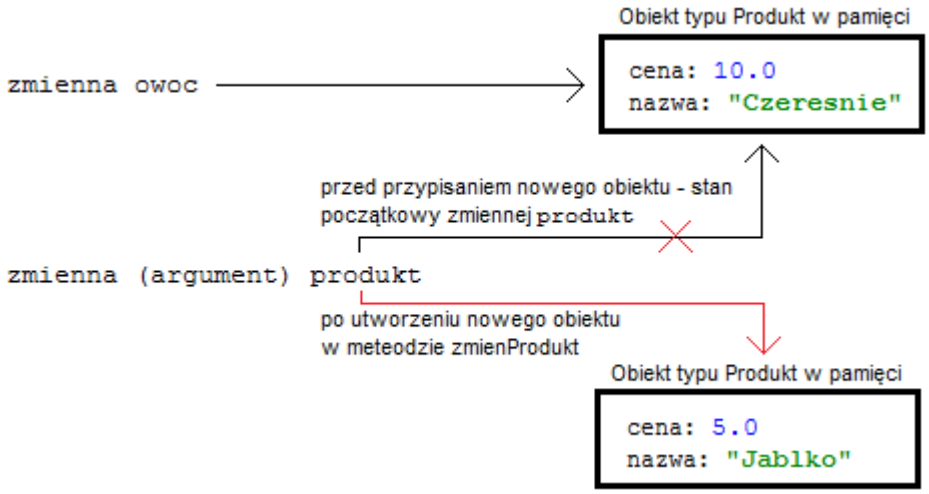
Na początku tej metody, argument `produkt` wskazuje na obiekt w pamięci, do którego referencję podaliśmy wywołując metodę `zmienProdukt`:

```
zmienProdukt(owoc, 5.0, "Jablko");
```

Dlatego, przed przypisaniem do argumentu `produkt` nowo utworzonego obiektu, pokazuje on na obiekt utworzony w linii *(1)*, a poźniej na nowy obiekt w pamięci o cenie i nazwie `nowaCena` i `nowaNazwa`.

Metoda `zmienProdukt` w żaden sposób nie zmienia ani obiektu, na który pokazuje zmienna `owoc`, ani nie powoduje, że zmienna `owoc` zacznie pokazywać na obiekt utworzony wewnątrz metody `zmienProdukt`. Jedyna zmiana, jaka zostaje wykonana, to zmiana, na jaki obiekt pokazuje argument `produkt` – z obiektu utworzonego w linii *(1)* na nowy obiekt utworzony w metodzie *(2)*.

Zagadnienie to zostało przedstawione poniżej:



Jedynym efektem przypisania nowego obiektu do argumentu `produkt` jest to, że od momentu wykonania tej instrukcji, nie mamy już w metodzie `zmienProdukt` dostępu do obiektu, na który wcześniej wskazywał argument `produkt`. Oryginalny obiekt, jak i zmienna `owoc`, która na niego wskazuje, nie zmieniają się.

Takie samo zachowanie moglibyśmy zauważyć w metodzie `main` – spójrzmy na poniższy fragment kodu – co zobaczymy na ekranie?

```
Produkt rower = new Produkt(999.0, "Rower"); // 1
Produkt pojazd = rower;

pojazd = new Produkt(50000.0, "Samochod"); // 2

System.out.println(rower);
System.out.println(pojazd);
```

Na ekranie zobaczymy:

```
Produkt o nazwie Rower kosztuje 999.0
Produkt o nazwie Samochod kosztuje 50000.0
```

Mamy tutaj podobną sytuację, jak z metodą `zmienProdukt` – odpowiednikiem argumentu `produkt` jest tutaj zmienna `pojazd` – przypisujemy jej odniesienie do tego samego obiektu, na który wskazuje zmienna `rower`.

Następnie, przypisujemy do zmiennej `pojazd` nowy obiekt typu `Produkt`. To przypisanie w żaden sposób nie wpływa na obiekt utworzony w linii *(1)*, ani też nie ma wpływu na to, na co pokazuje zmienna `rower`. Jedyne, co się zmienia, to zmienna `pojazd` – od linii *(2)* zaczyna ona pokazywać na inny obiekt w pamięci.

9.8.2 Współdzielenie obiektów

To, że możemy mieć wiele zmiennych pokazujących na ten sam obiekt w pamięci, ma swoje wady i zalety.

Spójrzmy na przykład z klasami `Odcinek` i `Punkt` – obiekty klasy `Odcinek` opisane są poprzez dwa punkty na płaszczyźnie, do czego wykorzystywane są obiekty klasy `Punkt`:

Nazwa pliku: wspoldzielenieobiektow/Punkt.java

```java
public class Punkt {
  private int x;
  private int y;

  public Punkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    this.y = y;
  }

  public String toString() {
    return "Punkt(x: " + x + ", y: " + y + ")";
  }
}
```

Nazwa pliku: wspoldzielenieobiektow/Odcinek.java

```
public class Odcinek {
  private Punkt poczatek;
  private Punkt koniec;

  public Odcinek(Punkt poczatek, Punkt koniec) {
    this.poczatek = poczatek;
    this.koniec = koniec;
  }

  public void przesunPoczatek(int x, int y) {
    this.poczatek.setX(x);
    this.poczatek.setY(y);
  }

  public String toString() {
    return "Odcinek zawarty pomiedzy " + poczatek + " i " + koniec;
  }
}
```

Obie klasy mają konstruktory, inicjalizujące ich pola, oraz metody `toString`, pomocne przy opisywaniu obiektów obu klas. Klasa `Odcinek` korzysta z klasy `Punkt` – ma ona dwa prywatne pola typu `Punkt`, które wyznaczają, odpowiednio, początek i koniec odcinka na płaszczyźnie.

Klasa `Odcinek` udostępnia także publiczną metodę `przesunPoczatek`, która pozwala na zmianę położenia początkowego punktu – wywołuje ona po prostu settery na obiekcie `poczatek`, z odpowiednimi argumentami.

Dopiszemy do klasy `Odcinek` metodę `main`, w której utworzymy dwa obiekty typu `Odcinek`. Oba odcinki będą zaczynały się w tym samym punkcie, więc użyjemy tego samego obiektu `Punkt` dwukrotnie:

Fragment pliku Odcinek.java z katalogu wspoldzielenieobiektow

```
public static void main(String[] args) {
  Punkt poczatekOdcinkow = new Punkt(10, 20);
  Punkt koniecPierwszego = new Punkt(15, 25);
  Punkt koniecDrugiego = new Punkt(10, 30);

  Odcinek odcinek1 = new Odcinek(poczatekOdcinkow, koniecPierwszego); // 1
  Odcinek odcinek2 = new Odcinek(poczatekOdcinkow, koniecDrugiego); // 2

  System.out.println(odcinek1);
  System.out.println(odcinek2);
}
```

Dzięki metodom `toString`, które dodaliśmy do naszych, klas, na ekranie zobaczymy czytelną, tekstową reprezentację obu obiektów typu `Odcinek`:

```
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 10, y: 30)
```

W powyższym przykładzie utworzyliśmy trzy obiekty typu `Punkt` i dwa obiekty typu `Odcinek`. Pierwszy z utworzonych punktów, `poczatekOdcinkow`, użyliśmy dwukrotnie – przekazaliśmy go jako pierwszy argument do konstruktorów dwóch obiektów typu `Odcinek` *(1)* *(2)*.

Załóżmy teraz, że chcielibyśmy przesunąć pierwszy odcinek – zmienić położenie jego punktu początkowego – możemy w tym celu skorzystać z metody `przesunPoczatek` klasy `Odcinek`.

Pytanie: co zostanie wypisane na ekran w wyniku działania poniższego fragmentu kodu?

```
odcinek1.przesunPoczatek(0, 5);

System.out.println(odcinek1);
System.out.println(odcinek2);
```

Tym razem na ekranie zobaczymy:

```
Odcinek zawarty pomiedzy Punkt(x: 0, y: 5) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 0, y: 5) i Punkt(x: 10, y: 30)
```

Jak widzimy, po zmianie pierwszego odcinka, zmiany zaszły także w drugim obiekcie – oba odcinki mają zmienione początkowe położenie – dlaczego tak się stało?

Tworząc obiekty `odcinek1` i `odcinek2`, przesłaliśmy jako argument do ich konstruktorów referencję do tego samego obiektu `Punkt` o nazwie `poczatekOdcinkow`:

```
Odcinek odcinek1 = new Odcinek(poczatekOdcinkow, koniecPierwszego);
Odcinek odcinek2 = new Odcinek(poczatekOdcinkow, koniecDrugiego);
```

W konstruktorze klasy `Odcinek`, do pola `poczatek`, przypisujemy referencję do obiektu przesłanego jako pierwszy argument:

```
public Odcinek(Punkt poczatek, Punkt koniec) {
  this.poczatek = poczatek;
  this.koniec = koniec;
}
```

Oba obiekty `odcinek1` i `odcinek2` przechowują w polu `poczatek` odniesienie do tego samego obiektu w pamięci – obiektu `poczatekOdcinkow`.

Gdy wywołujemy poniższą linię kodu, zmieniamy położenie obiektu `odcinek1`:

```
odcinek1.przesunPoczatek(0, 5);
```

Jak działa ta metoda? Przypisuje ona do obiektu wskazywanego przez pole `poczatek` nowe wartości za pomocą setterów:

```
public void przesunPoczatek(int x, int y) {
  this.poczatek.setX(x);
  this.poczatek.setY(y);
}
```

Modyfikując obiekt wskazywany przez pole `poczatek`, zmieniamy de facto obiekt, który jest współdzielony przez oba obiekty: `odcinek1` i `odcinek2`.

Mało tego – do obiektu opisującego początek obu odcinków mamy jeszcze dostęp w inny sposób – poprzez zmienną `poczatekOdcinkow`:

```
poczatekOdcinkow.setX(-20);
poczatekOdcinkow.setY(50);

System.out.println(odcinek1);
System.out.println(odcinek2);
```

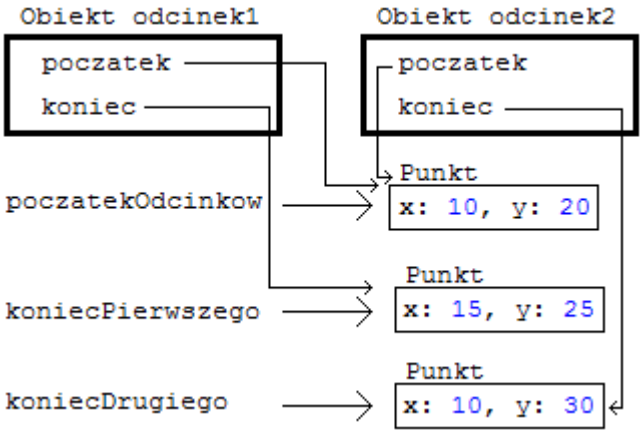
Powyższy kod spowoduje wypisanie na ekran:

```
Odcinek zawarty pomiedzy Punkt(x: -20, y: 50) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: -20, y: 50) i Punkt(x: 10, y: 30)
```

Modyfikując obiekt `poczatekOdcinkow` sprawiliśmy, że zmieniło się położenie obu odcinków! Wynika to z faktu, że wszystkie trzy zmienne:

- `poczatekOdcinkow`,
- `poczatek` obiektu `odcinek1`,
- `poczatek` obiektu `odcinek2`,

wskazują na ten sam obiekt w pamięci:



W tym przypadku, takie zachowanie jest nieporządane, ponieważ chcemy móc niezależnie modyfikować pola obiektów klasy `Odcinek`, nie wpływając na inne, istniejące obiekty tej klasy.

Co możemy zrobić, aby zopobiec takiemu zachowaniu obiektów w naszych programach?

Mamy trzy opcje:

1. Możemy utworzyć osobne obiekty dla punktów początkowych każdego z odcinków.

   lub

2. Możemy w konstruktorze klasy `Odcinek` zrobić kopię przesłanych obiektów typu `Punkt`.

   lub

3. Możemy korzystać z obiektów niemutowalnych (*immutable objects*), o których opowiemy sobie za chwilę w jednym z kolejnych rozdziałów.

Powyższe propozycje mają wady i zalety – spójrzmy na implementację dwóch pierwszych opcji.

9.8.2.1 Osobne obiekty typu Punkt

Przyjrzyjmy się najpierw implementacji, w której korzystamy z osobnych obiektów typu `Punkt` dla każdego z obiektów klasy `Odcinek`:

```
Punkt poczatekPierwszego = new Punkt(10, 20);
Punkt koniecPierwszego = new Punkt(15, 25);

Punkt poczatekDrugiego = new Punkt(10, 20);
Punkt koniecDrugiego = new Punkt(10, 30);

Odcinek odcinek1 = new Odcinek(poczatekPierwszego, koniecPierwszego);
Odcinek odcinek2 = new Odcinek(poczatekDrugiego, koniecDrugiego);

odcinek1.przesunPoczatek(0, 5); // 1

System.out.println(odcinek1);
System.out.println(odcinek2);
```

Gdy teraz zmienimy położenie pierwszego odcinka `(1)`, to położenie drugiego odcinka się nie zmieni, ponieważ każdy z odcinków ma osobny obiekt typu `Punkt`, który opisuje początek:

```
Odcinek zawarty pomiedzy Punkt(x: 0, y: 5) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 10, y: 30)
```

Zaletą tego rozwiązania jest to, że jest proste – tworzymy dedykowane obiekty typu `Punkt` dla każdego z obiektów typu `Odcinek` i przesyłamy je jako argumenty do konstruktora.

Wadą tego rozwiązania jest to, że nadal mamy dostęp do utworzonych obiektów typu `Punkt` spoza wnętrza klasy `Odcinek`. Spójrzmy na poniższy przykład – za pomocą zmiennych `poczatekPierwszego` i `poczatekDrugiego` możemy zmodyfikować położenie obu odcinków:

```
poczatekPierwszego.setX(-100);
poczatekPierwszego.setY(-100);

poczatekDrugiego.setX(200);
poczatekDrugiego.setY(200);

System.out.println(odcinek1);
System.out.println(odcinek2);
```

Na ekranie zobaczymy:

```
Odcinek zawarty pomiedzy Punkt(x: -100, y: -100) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 200, y: 200) i Punkt(x: 10, y: 30)
```

Za pomocą zmiennych zdefiniowanych w metodzie `main` mamy dostęp do obiektów, na które wskazują prywatne pola klasy `Odcinek` – zazwyczaj nie jest dobra praktyka, ponieważ możliwe są do wykonania zmiany naszych obiektów bez naszej wiedzy, co może prowadzić do potencjalnych błędów. Wszystko zależy od kontekstu, w jakim tworzymy nasze klasy, to znaczy: kto i do czego będzie ich używał.

Często można w prosty sposób uchronić się przed potencjalnymi problemami pisząc kod w taki sposób, by nie współdzielić obiektów, które powinny być prywatne i wszelkie ich modyfikacje powinny odbywać się jedynie wewnątrz metod klasy, która z tych obiektów korzysta.

9.8.2.2 Tworzenie kopii obiektów

Możemy w konstruktorze klasy `Odcinek` nie przypisywać do pól `poczatek` i `koniec` referencji do obiektów przesłanych jako argumenty, lecz utworzyć nowe obiekty typu `Punkt` na podstawie tych argumentów:

```
// poprzednia wersja
public Odcinek(Punkt poczatek, Punkt koniec) {
  this.poczatek = poczatek;
  this.koniec = koniec;
}

// nowa wersja konstruktora
public Odcinek(Punkt poczatek, Punkt koniec) { // 1
  this.poczatek = new Punkt(poczatek.getX(), poczatek.getY());
  this.koniec = new Punkt(koniec.getX(), koniec.getY());
}
```

Nowa wersja konstruktora `(1)` nie przypisuje już do pól `poczatek` i `koniec` argumentów konstruktora, lecz tworzy zupełnie nowe obiekty klasy `Punkt` z takimi samymi wartościami pól `x` i `y`, jakie mają obiekty przesłane jako argumenty. Nowo utworzone obiekty są przypisywane do pól `poczatek` i `koniec`.

W ten sposób, obiekty klasy `Odcinek` zupełnie odcinają się od zewnętrznego świata i są od niego uniezależnione – wszelkie zmiany obiektów przesłanych jako argumenty nie mają wpływu na pola obiektów klasy `Odcinek`:

```
Punkt poczatekOdcinkow = new Punkt(10, 20);
Punkt koniecPierwszego = new Punkt(15, 25);
Punkt koniecDrugiego = new Punkt(10, 30);

Odcinek odcinek1 = new Odcinek(poczatekOdcinkow, koniecPierwszego);
Odcinek odcinek2 = new Odcinek(poczatekOdcinkow, koniecDrugiego);

System.out.println(odcinek1);
System.out.println(odcinek2);

poczatekOdcinkow.setX(-20);
poczatekOdcinkow.setY(50);

System.out.println(odcinek1);
System.out.println(odcinek2);
```

W pierwszej wersji powyższego kodu, zmiana obiektu `poczatekOdcinkow` powodowała, że zmieniał się początkowy punkt obu odcinków – zobaczmy, co tym razem zostanie wypisane na ekran:

```
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 10, y: 30)
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 10, y: 30)
```

Tym razem zmiana obiektu `poczatekOdcinkow` nie wpływa na obiekty `odcinek1` i `odcinek2`, ponieważ każdy z tych obiektów ma własne egzemplarze obiektów typu `Punkt`, które zostały utworzone w konstrukorze. Jedyną możliwością, by zmienić położenie odcinka, jest teraz użycie metody `przesunPoczatek`.

To rozwiązanie zapewnia, że obiekty klasy `Odcinek` są chronione przed zmianami ich wewnętrznego stanu poza klasą `Odcinek`. Wadą tego rozwiązania jest to, że musimy tworzyć więcej obiektów – tworzenie obiektów zajmuje czas procesora i pamięć komputera. W naszym prostym przykładzie nie ma to żadnego znaczenia, ale gdy działamy z tysiącami/milionami obiektów, które posiadają kilkanaście złożonych pól, które z kolei składają się z wielu kolejnych pól, to może się to odbić negatywnie na wydajności naszych programów.

9.8.2.3 Kiedy współdzielić obiekty?

Czasem chcemy, by wewnętrzny stan obiektów był możliwy do zmiany jedynie z wnętrza klasy tych obiektów. Obiekty klasy `Uczen` powinny zawsze mieć dedykowaną, prywatną dla nich i ukrytą przed światem tablicę ocen ucznia – nie chcemy, by wiele obiektów klasy `Uczen` korzystało z tego samego egzemplarza tablicy w pamięci!

Z drugiej strony, obiekty często oczekują, że zostaną im przesłane obiekty, z których będą mogły korzystać i nie ma znaczenia, czy obiekty te są dedykowane dla konkretnego obiektu, czy współdzielone pomiędzy wieloma obiektami – w rzeczywistości, jest to częsta praktyka[5].

Jeżeli jesteśmy w stanie zapewnić, że stan współdzielonych obiektów nie zostanie zmieniony w nieporządany sposób przez któryś z używających go obiektów, albo możemy wręcz zapewnić, że obiekty takie są niemodyfikowalne (o takich obiektach zaraz sobie opowiemy), to współdzielenie obiektów powoduje zmniejszenie zasobów, jakie są potrzebene dla programu – zamiast tworzyć wiele obiektów danego typu, możemy utworzyć jeden i pozwalać na korzystanie z niego przez zależne od niego obiekty.

Przykładem współdzielenia obiektów są np. obiekty odpowiedzialne za łączenie się z bazą danych i zarządzanie takimi połączeniami – są to kosztowne operacje z punkty widzenia wykonywania programu, które możemy zoptymalizować, jeżeli zarządzanie połączeniami oddelegujemy do jednego obiektu, z którego inne klasy będą mogły korzystać do zapisywania i odczytywania danych z bazy danych.

9.8.3 Stałe referencje

W rozdziale o konstruktorach zobaczyliśmy, że polom typu referencyjnego, zdefiniowanych z modyfikatorem `final`, nie możemy przypisać innej wartości po ich zainicjalizowaniu. Podobnie ze zmiennymi lokalnymi:

```
final Punkt punkt = new Punkt(10, 20);
punkt = new Punkt(25, 30);
```

Próba kompilacji powyższego fragmentu kodu kończy się następującymi błędem:

```
error: cannot assign a value to final variable punkt
        punkt = new Punkt(25, 30);
        ^
1 error
```

Zmienna `punkt` jest `final`, więc po jej inicjalizacji nie możemy już przypisać do niej innego obiektu, o czym informuje nas kompilator.

A co by się stało, gdybyśmy na powyższej zmiennej `punkt` spróbowali wywołać setter i zmienić np. pole `x` obiektu `punkt`? Jaki będzie wynik kompilacji poniższego kodu i uruchomienia go?

5 Mowa tutaj o *Dependency Injection*, popularnym w ostatnich latach sposobie zarządzania zależnościami pomiędzy obiektami w języku Java. Jest to jedna z podstawowych funkcjonalności udostępnianych przez framework *Spring*.

```
final Punkt punkt = new Punkt(10, 20);
System.out.println(punkt);

punkt.setX(5);
System.out.println(punkt);
```

Tym razem nie zobaczymy na ekranie błędu:

```
Punkt(x: 10, y: 20)
Punkt(x: 5, y: 20)
```

Dlaczego kod zadziałał i udało nam się zmienić obiekt zdefiniowany z modyfikatorem **final**?

Musimy pamiętać, że w tym przypadku to nie obiekt, któremu zmieniamy wartość pola `x`, lecz zmienna o nazwie `punkt`, jest **final**.

Nic nie stoi nam na przeszkodzie, by zmienić obiekt typu `Punkt`, znajdujący się gdzieś w pamięci, odnosząc się do niego za pomocą zmiennej `punkt`. To, czego nie możemy zrobić, to zmienić, na co zmienna `punkt` pokazuje – to właśnie zapewnia nam słowo kluczowe **final**.

W powyższym kodzie zmienna `punkt` jest *stałą referencją* do obiektu typu `Punkt`, ale obiekt ten nie musi być stały – możemy zmieniać wartości jego pól. Należy to zawsze mieć na uwadze pracując z obiektami.

9.8.4 Obiekty niemutowalne (immutable objects)

W poprzednim podrozdziale wspomnieliśmy o *obiektach niemutowalnych* (*immutable objects*).

Obiekty niemutowalne to takie obiekty, których wartości nie mogą zmienić się po ich utworzeniu.

Aby zapewnić niemożliwość zmiany, klasy, których obiekty mają być niemutowalne, nie posiadają setterów oraz nie udostępniają żadnego sposobu na modyfikację wewnętrznego stanu obiektów tej klasy. Wartości pól raz utworzonego, niemodyfikowalnego obiektu, pozostają takie same przez całe "życie" takiego obiektu.

Wróćmy do klasy `Punkt` z poprzedniego podrozdziału:

Nazwa pliku: wspoldzielenieobiektow/Punkt.java

```
public class Punkt {
  private int x;
  private int y;

  public Punkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    this.y = y;
  }

  public String toString() {
    return "Punkt(x: " + x + ", y: " + y + ")";
  }
}
```

Czy obiekty tej klasy są niemutowalne?

Po utworzeniu obiektu klasy `Punkt`, możemy zmienić wartości jego pól `x` oraz `y` za pomocą setterów `setX` oraz `setY`, więc **obiekty tej klasy nie są obiektami niemutowalnymi**.

Moglibyśmy usunąć settery z powyższej klasy – wtedy w żaden sposób nie można byłoby zmienić wartości pól `x` oraz `y`. Wartości nadane tym polom w konstruktorze byłyby wartościami, które obiekty klasy `Punkt` miałyby już na stałe – w ten sposób, otrzymalibyśmy obiekty niemutowalne. Spójrzmy na drugą wersję klasy `Punkt`, którą nazwiemy `NiemutowalnyPunkt`:

Nazwa pliku: NiemutowalnyPunkt.java

```
public class NiemutowalnyPunkt {
  private final int x;
  private final int y;

  public NiemutowalnyPunkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public String toString() {
    return "NiemutowalnyPunkt(x: " + x + ", y: " + y + ")";
  }
}
```

Obiekty powyższej klasy są niemutowalne – zauważmy, że gdy utworzymy obiekt klasy `NiemutowalnyPunkt`, to nie będziemy już mogli zmienić wartości pól `x` oraz `y` w żaden sposób. Zauważmy także, że dodaliśmy modyfikator **`final`** do pól `x` oraz `y`, aby mieć pewność, że nigdzie wewnątrz klasy `NiemutowalnyPunkt` nie spróbujemy zmodyfikować wartości tych pól.

Moglibyśmy teraz napisać klasę `NiemutowalnyOdcinek`, która zawierałaby dwa obiekty typu `NiemutowalnyPunkt`:

Nazwa pliku: NiemutowalnyOdcinek.java

```
public class NiemutowalnyOdcinek {
  private final NiemutowalnyPunkt poczatek;
  private final NiemutowalnyPunkt koniec;

  public NiemutowalnyOdcinek(
      NiemutowalnyPunkt poczatek, NiemutowalnyPunkt koniec) {
    this.poczatek = poczatek;
    this.koniec = koniec;
  }

  public String toString() {
    return "NiemutowalnyOdcinek zawarty pomiedzy " +
        poczatek + " i " + koniec;
  }
}
```

W poprzednim podrozdziale, w jednej z wersji klasy `Odcinek`, natknęliśmy się na sytuację, w której konstruktor klasy `Odcinek` przypisywał do pól `poczatek` oraz `koniec` obiekty typu `Punkt` przesłane jako argument:

fragment pliku wspoldzielenieobiektow/Odcinek.java

```
public Odcinek(Punkt poczatek, Punkt koniec) {
  this.poczatek = poczatek;
  this.koniec = koniec;
}
```

Powodowało to, że mogliśmy modyfikować stan obiektów typu `Odcinek` za pomocą referencji do obiektów klasy `Punkt`, które przekazaliśmy do konstruktora klasy `Odcinek`:

```
Punkt punktPierwszy = new Punkt(10, 20);
Punkt punktDrugi = new Punkt(15, 25);

Odcinek odcinek1 = new Odcinek(punktPierwszy, punktDrugi);

punktPierwszy.setX(-100); // zmieniamy stan obieku odcinek1!
punktPierwszy.setY(-100);
```

W powyższym fragmencie, zmiana obiektu `punktPierwszy` powoduje, że zmienia się także obiekt `odcinek1`, ponieważ pole `poczatek` obiektu `odcinek1` wskazuje na ten sam obiekt, co zmienna `punktPierwszy`.

Zauważmy, że tego problemu nie napotkamy w przypadku klas `NiemutowalnyPunkt` oraz `NiemutowalnyOdcinek`, ponieważ, nawet jeżeli będziemy mieli referencję do obiektów typu `NiemutowalnyPunkt`, które prześlemy jako argumenty do konstruktora klasy `NiemutowalnyOdcinek`, to i tak nie będziemy mogli ich w żaden sposób zmienić, ponieważ są niemutowalne:

fragment pliku NiemutowalnyOdcinek.java

```
public static void main(String[] args) {
  NiemutowalnyPunkt poczatek = new NiemutowalnyPunkt(10, 20);
  NiemutowalnyPunkt koniec = new NiemutowalnyPunkt(15, 25);

  NiemutowalnyOdcinek odcinek =
      new NiemutowalnyOdcinek(poczatek, koniec);
}
```

Pomimo, że mamy dostęp do obiektów `poczatek` i `koniec`, z których będzie korzystać obiekt `odcinek`, to nie mamy możliwości zmiany wartości przechowywanych przez te obiekty, ponieważ są niemutowalne.

Spójrzmy na inny przykład – czy obiekty tej klasy są niemutowalne?

Nazwa pliku: Figura.java

```
public class Figura {
  private final NiemutowalnyPunkt[] punkty;

  public Figura(NiemutowalnyPunkt[] punkty) {
    this.punkty = punkty;
  }

  public String toString() {
    String punktyFigury = "";

    for (NiemutowalnyPunkt punkt : punkty) {
      punktyFigury += "\t" + punkt + "\n";
    }

    return "Figura sklada sie z punktow:\n" + punktyFigury;
  }
}
```

Na pierwszy rzut oka może się wydawać, że obiekty klasy `Figura` są niemutowalne – nie dajemy dostępu do pola `punkty` – nie ma setterów oraz pole `punkty` jest prywatne i finalne, a jego wartość

jest nadawana jednorazowo w konstruktorze klasy `Figura`. Ponadto, korzystamy z klasy `NiemutowalnyPunkt`, której obiekty, jak już wiemy, sa niemutowalne, więc nie ma opcji, aby zmodyfikować którykolwiek z nich, nawet, gdybyśmy mieli do niego referencję.

Niestety, zapomnieliśmy o jeszcze jednym obiekcie – tablicy, którą przesyłamy jako argument do konstruktora klasy `Figura`! Spójrzmy na przykład, który udowadnia, że klasa `Figura` **nie jest** niemutowalna:

fragment pliku Figura.java

```
public static void main(String[] args) {
  NiemutowalnyPunkt[] punktyKwadratu = {
      new NiemutowalnyPunkt(0, 0),
      new NiemutowalnyPunkt(10, 0),
      new NiemutowalnyPunkt(10, 10),
      new NiemutowalnyPunkt(0, 10),
  };

  Figura figura = new Figura(punktyKwadratu);

  System.out.println(figura);

  punktyKwadratu[0] = new NiemutowalnyPunkt(-10, -5);

  System.out.println(figura);
}
```

W powyższym przykładzie tworzymy tablicę `punktyKwadratu`, która przechowuje wierzchołki kwadratu, a następnie przekazujemy ją do konstruktora klasy `Figura`. Wypisujemy na ekran obiekt `figura`, a następnie modyfikujemy tablicę `punktyKwadratu` – do pierwszego elementu tablicy zapisujemy zupełnie nowy obiekt typu `NiemutowalnyPunkt` i ponownie wypisujemy na ekran obiekt `figura`. To, co widzimy na ekranie po wykonaniu powyższej metody `main`, świadczy o tym, że jesteśmy w stanie zmienić obiekt typu `Figura`:

```
Figura sklada sie z punktow:
     NiemutowalnyPunkt(x: 0, y: 0)
     NiemutowalnyPunkt(x: 10, y: 0)
     NiemutowalnyPunkt(x: 10, y: 10)
     NiemutowalnyPunkt(x: 0, y: 10)

Figura sklada sie z punktow:
     NiemutowalnyPunkt(x: -10, y: -5)
     NiemutowalnyPunkt(x: 10, y: 0)
     NiemutowalnyPunkt(x: 10, y: 10)
     NiemutowalnyPunkt(x: 0, y: 10)
```

W powyższej metodzie `toString` skorzystaliśmy z dwóch "znaków kontrolnych": `\t` oraz `\n`. Są to specjalne znaki, które powodują zmianę wyświetlania tekstu na ekranie. Znak `\t`, czyli znak tabulacji, powoduje, że tekst będzie wcięty, natomiast znak `\n` (newline), spowoduje, że na ekranie dodany będzie znak nowej linii, a to, co nastepuje za nim, zostanie wypisane linijkę niżej na ekranie. Dzięki temu, tekst jest ładnie sformatowany na ekranie.

Nie zadbaliśmy w naszej klasie, by wziąć pod uwagę, że użytkownik naszej klasy może zmodyfikować zawartość tablicy przesłanej jako argument do konstruktora klasy `Figura`. Może to być skutkiem potencjalnych błędów w naszym programie, jednak możemy w prosty sposób uchronić się przed powyższą sytuacją, jeżeli zamiast przypisywać przesłaną tablicą do pola `punkty`, utworzymy nową tablicę:

```java
public class NiemutowalnaFigura {
  private final NiemutowalnyPunkt[] punkty;

  public NiemutowalnaFigura(NiemutowalnyPunkt[] punkty) {
    // utworz nowa tablice
    this.punkty = new NiemutowalnyPunkt[punkty.length];

    // i przepisz zawartosc z tablicy-argumentu do nowej tablicy
    for (int i = 0; i < punkty.length; i++) {
      this.punkty[i] = punkty[i];
    }
  }

  public String toString() {
    String punktyFigury = "";

    for (NiemutowalnyPunkt punkt : punkty) {
      punktyFigury += "\t" + punkt + "\n";
    }

    return "Figura sklada sie z punktow:\n" + punktyFigury;
  }

  public static void main(String[] args) {
    NiemutowalnyPunkt[] punktyKwadratu = {
        new NiemutowalnyPunkt(0, 0),
        new NiemutowalnyPunkt(10, 0),
        new NiemutowalnyPunkt(10, 10),
        new NiemutowalnyPunkt(0, 10),
    };

    NiemutowalnaFigura figura = new NiemutowalnaFigura(punktyKwadratu);

    System.out.println(figura);

    punktyKwadratu[0] = new NiemutowalnyPunkt(-10, -5);

    System.out.println(figura);
  }
}
```

W konstruktorze klasy `NiemutowalnaFigura` zamiast przypisać do pola `punkty` wartość przesłanego argumentu, tworzymy nową tablicę o takim samym rozmiarze, jak rozmiar przesłanej tablicy. Następnie, przepisujemy do nowej tablicy zawartość tablicy-argumentu. Tym razem, gdy uruchomimy program, zobaczymy na ekranie dwa razy ten sam komunikat:

```
Figura sklada sie z punktow:
     NiemutowalnyPunkt(x: 0, y: 0)
     NiemutowalnyPunkt(x: 10, y: 0)
     NiemutowalnyPunkt(x: 10, y: 10)
     NiemutowalnyPunkt(x: 0, y: 10)

Figura sklada sie z punktow:
     NiemutowalnyPunkt(x: 0, y: 0)
     NiemutowalnyPunkt(x: 10, y: 0)
     NiemutowalnyPunkt(x: 10, y: 10)
     NiemutowalnyPunkt(x: 0, y: 10)
```

Zmiana tablicy `punktyKwadratu` nie miała wpływu na obiekt `figura`, ponieważ obiekt `figura`

posiada własną, osobną tablicę obiektów typu `NiemutowalnyPunkt`. Dzięki temu zabiegowi, obiekty klasy `NiemutowalnaFigura` są niemutowalne.

9.8.4.1 Zalety i wady obiektów niemutowalnych

Obiekty niemutowalne mają kilka zalet:

- mogą być współdzielone przez różne obiekty – będziemy mieli pewność, że ich stan się nie zmieni,
- mając raz utworzony obiekt niemutowalnej klasy mamy pewność, że nie zmienił on swojej wartości w trakcie działania programu,
- powyższe cechy są bardzo ważne podczas programowania wielowątkowego, ponieważ nie musimy się przejmować potencjalną modyfikacją obiektu przez różne wątki,
- niezmienialne obiekty są dobrymi kluczami dla map, które są często wykorzystywanymi strukturami danych (o mapach opowiemy sobie w jednym z dalszych rozdziałów kursu).

Wadą obiektów niemutowalnych jest to, że potrzeba zmiany wartości takiego obiektu wiąże się z wymogiem utworzenia nowego obiektu z nowymi wartościami. Tworzenie nowego obiektu wymaga czasu oraz pamięci komputera, jednak, o ile nie będziemy tworzyli tysięcy "ciężkich" obiektów (mających wiele pól, z których wiele będzie obiektami mającymi kolejne obiekty itd.), to nie powinno być to problemem.

9.8.4.2 Jak zapewnić niemutowalność obiektów

Aby obiekty były niemutowalne, muszą spełniać kilka wymagań, jak opisano w oficjalnej dokumentacji języka Java na stronie:

https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

1. Klasa nie powinna posiadać setterów.
2. Pola klasy powinny być prywatne i finalne (`private final`).
3. Klasa powinna być *finalna*, tzn. nie powinno się móc tworzyć podklas, których ta klasa byłaby rodzicem. O klasach finalnych opowiemy sobie w rozdziale o dziedziczeniu. Pozwalając na tworzenie podklas klasy, która z założenia miała być niemutowalna, moglibyśmy dać użytkownikom szansę na zniesienie niemutowalności. Zobaczymy taki przypadek w rozdziale o dziedziczeniu.
4. Jeżeli klasa zawiera pola, które nie są niemutowalne (np. tablica – możemy przypisać inny obiekt do któregoś z elementów tablicy), to nie powinniśmy zwracać referencji do takich obiektów – w przeciwnym razie, użytkownicy uzyskają możliwość zmiany wewnętrznego stanu obiektu.

Spójrzmy na problem opisany w punkcie 4 – wróćmy do klasy `NiemutowalnaFigura` – załóżmy, że chcielibyśmy mieć metodę w tej klasie, która zwraca tablicę punktów, z których ta figura się składa:

fragment pliku NiemutowalnaFigura.java

```
public NiemutowalnyPunkt[] getPunkty() {
  return punkty;
}
```

Czy po dodaniu takiej metody do klasy `NiemutowalnaFigura` obiekty tej klasy nadal będą niemutowalne? Nie! Zwracamy dostęp do prywatnego pola `punkty`, które jest tablicą – możemy

zmienić teraz elementy tej tablicy. Aby naprawić ten problem, powinniśmy zwracać kopię tablicy `punkty`:

fragment pliku NiemutowalnaFigura.java

```
public NiemutowalnyPunkt[] getPunkty() {
  NiemutowalnyPunkt[] rezultat =
      new NiemutowalnyPunkt[punkty.length];

  for (int i = 0; i < rezultat.length; i++) {
    rezultat[i] = punkty[i];
  }

  return rezultat;
}
```

W tej wersji metody `getPunkty` tworzymy kopię tablicy `punkty` i to ją zwracamy. Jakiekolwiek zmiany wykonane na tablicy zwróconej przez powyższą metodę nie będą miały wpływu na obiekt klasy `NiemutowalnaFigura`.

9.8.4.3 Niemutowalny typ String

Typ `String` był pierwszym typem złożonym, z jakiego zaczęliśmy korzystać. Wielokrotnie używaliśmy metod typu `String` do wykonywania różnych działań na łańcuchach tekstowych, jak na przykład `toUpperCase`, która zwraca łańcuch tekstowy z małymi literami zamienionymi na wielkie litery.

Wszystkie metody typu `String` mają wspólną cechę – nie zmieniają one obiektu, na rzecz którego są wywoływane, lecz za każdym razem zwracają nowy string, który jest wynikiem wywołania metody. Dla przykładu, poniższy fragment kodu:

```
String powitanie = "Witajcie!";

powitanie.toUpperCase();

System.out.println(powitanie);
```

spowoduje wypisanie na ekran komunikatu `"Witajcie!"`, a nie `"WITAJCIE!"` – oryginalna wartość przechowywana w zmiennej `powitanie` nie ulega zmianie – zamiast tego, zwracany jest nowy string.

Powodem jest to, że wartości typu `String` są z założenia twórców języka Java wartościami niemutowalnymi – jedyny sposób na zmianę wartości typu `String` to przypisanie do niej zupełnie nowej wartości.

Jest kilka powodów, dla których wartości typu `String` zostały zaprojektowane jako niemutowalne, wśród których jest możliwość ich cache'owanie przez Maszynę Wirtualną Java, co przekłada się na wydajność aplikacji. Ponadto, zapewnienie, że wartości typu `String` są niemutowalne sprawia, że są dobrymi kandydatami jako klucze map. Mapy to struktury danych o których opowiemy sobie w dalszej części kursu.

9.8.5 Pamięć programu – stos i sterta

Bardzo ważnym aspektem w programowaniu jest użycie pamięci komputera. Każdy program ma przydzieloną pamięć, z której może korzystać z trakcie swojego działania – jeżeli program przekroczy przydzieloną mu pamięć, zakończy się on błędem.

W wielu językach programiści muszą sami zatroszczyć się o alokację pamięci dla tworzonych obiektów, a także o jej zwolnienie, gdy obiekty nie są już potrzebne – przykładem takiego języka jest język C++.

Język Java, jak wiele innych nowoczesnych języków programowania, wykonuje, dla naszej wygody, obie powyższe operacje bez potrzeby naszej pomocy. Maszyna Wirtualna Java zarówno alokuje (czyli przydziela) pamięć dla obiektów, które tworzymy używając słowa kluczowego **`new`**, jak i zwalnia pamięć zajętą przez utworzone wcześniej obiekty, które przestały być potrzebne. Zwalnianiem pamięci zajmuje się specjalny mechanizm nazywany *Garbage Collector*.

Garbage Collector śledzi referencje do utworzonych przez nas obiektów. Jeżeli do obiektu nie ma już żadnych referencji (bo np. zmienna, która na obiekt wskazywała, została utworzona w metodzie, która zakończyła działanie), to Garbage Collector zwalnia miejsce, które było zarezerowane dla tego obiektu, dzięki czemu może ono zostać wykorzystane przez kolejne obiekty, które będziemy potrzebowali utworzyć w naszym programie.
Algorytm działania Garbage Collectora jest dość skomplikowany. Warto, prędzej czy później, się z nim zaznajomić. Więcej informacji o działaniu Garbage Collectora można znaleźć w Internecie.

Automatyzacja alokacji i zwalniania pamięci nie zmienia jednak faktu, że trzeba mieć na uwadze pamięć systemu operacyjnego, która jest przydzielona do naszego programu. Gdy piszemy proste programy, to raczej nie napotkamy na problem z pamięcią, ale tworząc produkcyjne aplikacje, z których potencjalnie będą korzystały tysiące użytkowników, powinniśmy zastanowić się i przetestować nasz program pod dużym obciążeniem.

Do tej pory poznaliśmy kilka różnic pomiędzy typami referencyjnymi (złożonymi), a typami prymitywnymi. Kolejną z nich jest to, że zmienne typów prymitywnych trzymane są na tzw. *stosie* (*stack*), natomiast obiekty, do których odnoszą się zmienne typów referencyjnych, tworzone są na *stercie* (*heap*).

Stos i sterta to obszary pamięci, które są zarezerowane na czas działania naszego programu, którymi zarządza dla nas Maszyna Wirtualna Java. Zasadniczą różnicą pomiędzy stosem i stertą jest to, że stos ma dużo mniej pamięci do wykorzystania, niż sterta.

Gdy uruchamiamy Maszynę Wirtualną Java (program java), która ma wykonać kod naszego programu, to możemy podać jako argument maksymalny rozmiar sterty, jaki ma przysługiwać naszemu programowi, tzn. ile pamięci maksymalnie nasz program może zużyć. Ten argument to `-Xmx`, po którym od razu powinien nastąpić maksymalny rozmiar pamięci. Zapisujemy go jako liczbę, po której od razu następuje M bądź G – od Megabyte i Gigabyte (jeden Gigabyte to 1024 Megabyte'y).

Spróbujmy napisać program, który celowo zużyje całą dostępną dla niego pamięć, którą ustawimy na 2 Megabyte'y. Spróbujemy w programie utworzyć milion obiektów, które będą zawierały pewien ciąg znaków:

Nazwa pliku HeapOveflow.java

```
public class PrzepelnienieSterty {
  private String tekst;

  public PrzepelnienieSterty(String tekst) {
    this.tekst = tekst;
  }

  public static void main(String[] args) {
    PrzepelnienieSterty[] tab = new PrzepelnienieSterty[1000000];

    for (int i = 0; i < tab.length; i++) {
      tab[i] = new PrzepelnienieSterty("Witaj po raz" + i + "!");
    }
  }
}
```

Kompilacja i uruchomienie powyższego programu, bez ustawiania domyślnego rozmiaru pamięci sterty, kończy się bez błędu:

```
> javac PrzepelnienieSterty.java
> java PrzepelnienieSterty
```

Domyślna pamięć dla sterty jest w tym przypadku wystarczająca, jednak jeżeli bardzo ją ograniczymy, na przykład, do dwóch megabyte'ów, to zobaczymy na ekranie błąd po uruchomieniu:

```
> java -Xmx2M PrzepelnienieSterty
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at PrzepelnienieSterty.main(PrzepelnienieSterty.java:9)
```

Argumentem `-Xmx2M` powiedzieliśmy Maszynie Wirtualnej Java: "*Gdy będziesz wykonywać kod programu PrzepelnienieSterty, to nie zużyj więcej, niż dwa megabyte'y pamięci*". Tym razem, działanie programu zakończyło się błędem `OutOfMemory`, co świadczy o tym, że program potrzebował do swojego wykonania więcej, niż zapewniła mu Maszyna Wirtualna Java (za naszym przykazem poprzez użycie argumentu `-Xmx`).

Co należy zapamiętać:

1. My, jako programiści, nie musimy sami alokować i zwalniać pamięci – robi to za nas Java.
2. Automatyzacja alokacji i zwalniania pamięci nie zwalnia nas z obowiązku pisania kodu w taki sposób, by nie zużywał nadmiernie pamięci, tzn. nie tworzył nadmiernej liczby obiektów.
3. Stos służy jako pamięć podręczna naszego programu, gdzie m. in. trzymane są zmienne typów w prymitywnych.
4. Sterta to miejsce, gdzie przechowywane są obiekty typów złożonych.

Ten rozdział jest o tyle istotny, że programiści, którzy zaczynają naukę programowania bezpośrednio od takich języków jak Java czy C# nie mają wiedzy o alokacji i zwalnianiu pamięci, ponieważ jest to robione za nich. W warunkach produkcyjnych, tzn. pisząc programy, który będą używane przez użytkowników, nierzadko natknąć się można na problemy z brakiem pamięci.

9.8.6 Czas życia obiektów utworzonych w metodach

W rozdziale o metodach dowiedzieliśmy, że zmienne tworzone w metodach to zmienne lokalne, które kończą swoje "życie" wraz z zakończeniem działania metody.

Poniższy przykład korzysta z klasy `NiemutowalnyPunkt` z poprzedniego podrozdziału – po zakończeniu metody `przesunPunkt`, wszystkie trzy zmienne: `noweX`, `noweY`, oraz `przesunietyPunkt`, przestają istnieć:

Nazwa pliku: ZwracaniaObiektuZMetody.java

```
public class ZwracanieObiektuZMetody {
  public static void main(String[] args) {
    NiemutowalnyPunkt pewienPunkt = new NiemutowalnyPunkt(10, 20);

    przesunPunkt(pewienPunkt, 5);
  }

  public static void przesunPunkt(NiemutowalnyPunkt punkt, int wartosc) {
    int noweX = punkt.getX() + wartosc;
    int noweY = punkt.getY() + wartosc;

    NiemutowalnyPunkt przesunietyPunkt =
        new NiemutowalnyPunkt(noweX, noweY);
  }
}
```

Metoda `przesunPunkt` tworzy nowy `NiemutowalnyPunkt` na podstawie wartości argumentu `punkt`, którego współrzędne `x` oraz `y` przesunięte są o `wartosc`. Tworzony obiekt na razie nie jest zwracany, ani w żaden sposób używany. Gdy metoda `przesunPunkt` kończy działanie, to wszystkie zmienne lokalne przestają istnieć.

Garbage Collector, o którym wspomnieliśmy w poprzednim podrozdziale, zauważa, że do utworzonego w zaznaczonej powyżej linii obiektu typu `NiemutowalnyPunkt`, po zakończeniu metody `przesunPunkt`, nie odnosi się już żadna zmienna, więc Garbage Collector będzie mógł zwolnić zajmowaną przez ten obiekt pamięć.

A co by się stało, gdybyśmy zwrócili z metody `przesunPunkt` utworzony obiekt typu `NiemutowalnyPunkt`? Spróbujmy:

Nazwa pliku: ZwracaniaObiektuZMetody.java

```
public class ZwracanieObiektuZMetody {
  public static void main(String[] args) {
    NiemutowalnyPunkt pewienPunkt = new NiemutowalnyPunkt(10, 20);
    System.out.println("Poczatkowy punkt: " + pewienPunkt);

    NiemutowalnyPunkt nowyPunkt = przesunPunkt(pewienPunkt, 5);
    System.out.println("Przesuniety punkt: " + nowyPunkt);
  }

  public static NiemutowalnyPunkt przesunPunkt(
      NiemutowalnyPunkt punkt, int wartosc) {

    int noweX = punkt.getX() + wartosc;
    int noweY = punkt.getY() + wartosc;

    return new NiemutowalnyPunkt(noweX, noweY);
  }
}
```

Zmieniliśmy typ zwracany przez metodę `przesunPunkt` z **`void`** na `NiemutowalnyPunkt`. W zaznaczonej linii zwracamy nowo utworzony obiekt. W metodzie `main`, wynik działania metody `przesunPunkt` przypisujemy do zmiennej `nowyPunkt`, a następnie wypisujemy na ekran tekstową reprezentację tego obiektu. Na ekranie zobaczymy:

```
Poczatkowy punkt: NiemutowalnyPunkt(x: 10, y: 20)
Przesuniety punkt: NiemutowalnyPunkt(x: 15, y: 25)
```

Jak widzimy, obiekt utworzony w metodzie `przesunPunkt` "żyje" po zakończeniu metody, w której został utworzony – świadczy o tym to, że jesteśmy w stanie wypisać jego zawartość w metodzie `main`. Tym razem obiekt typu `NiemutowalnyPunkt`, który tworzymy w metodzie `przesunPunkt`, nie przestaje istnieć wraz z końcem metody `przesunPunkt`, ponieważ w kodzie programu nadal istnieje zmienna, która na niego wskazuje – jest to zmienna `nowyPunkt` zdefiniowana w metodzie `main`. Dopóki istnieje zmienna, która odnosi się do obiektu, to obiekt także będzie istniał. Jest to bardzo ważna cecha obiektów typów referencyjnych.

9.8.7 Podsumowanie różnic typów prymitywnych i referencyjnych

W kilku ostatnich rozdziałach tego kursu poznaliśmy wiele różnic pomiędzy typami referencyjnymi (złożonymi) oraz typami prymitywnymi. Poniższa tabela podsumowuje te różnice:

	Typy Prymitywne	**Typy Referencyjne**
Wartości	konkretne, np. `5`, **`true`**, `'a'`	adresy obiektów
Tworzenie	zdefiniowanie zmiennej	operator **`new`**
Domyślne wartości	zmienne lokalne – brak, pola klas – zależne od typu	zmienne lokalne – brak, pola klas – **`null`**
Porównywanie	porównywane są wartości	porównywane są wartości, którymi są adresy obiektów, a nie obiekty, więc użycie operatora == zwróci **`true`** tylko wtedy, gdy dwie zmienne będą wskazywały na dokładnie ten sam obiekt w pamięci – do porównywania obiektów złożonych powinniśmy stosować metodę `equals`
Pamięć	tworzone są na stosie	tworzone są na stercie
Przesyłanie do metod	przez wartość	przez wartość – wysyłana jest referencja do obiektu, a nie kopia obiektu – obiekt źródłowy może zostać zmieniony, jeżeli wykonujemy na nim operacje

9.8.8 Podsumowanie

- Zmienne typów złożonych (referencyjnych) wskazują na obiekty utworzone w pamięci. Dwie różne zmienne mogą wskazywać na ten sam obiekt w pamięci – będziemy wtedy mogli zmodyfikować ten sam obiekt za pomocą dwóch różnych zmiennych:

```
Osoba o1 = new Osoba("Jan", "Nowak", 25);
Osoba o2 = o1;

System.out.println("imie o1: " + o1.getImie());
System.out.println("imie o2: " + o2.getImie());

o1.setImie("Karol");

System.out.println("imie o1: " + o1.getImie());
System.out.println("imie o2: " + o2.getImie());
```

Zmieniając pole `imie` za pomocą settera `setImie`, wywołanego na rzecz zmiennej `o2` w zaznaczonej linii, modyfikujemy ten sam obiekt, na który wskazuje zmienna `o1` – wynikiem działania tego programu jest:

```
imie o1: Jan
imie o2: Jan
imie o1: Karol
imie o2: Karol
```

- Przesłanie do metody obiektu typu złożonego powoduje, że wszelkie zmiany wykonane na tym obiekcie będą wykonywane na oryginalnym obiekcie – w poniższym programie przesyłamy jako argument obiekt `ksiazka` utworzony w pierwszej linii metody `main` – obiekt ten modyfikujemy w metodzie `zmienCene`. Zmiana odbywa się na oryginalnym obiekcie przesłanym do tej metody:

```
public static void main(String[] args) {
  Produkt ksiazka = new Produkt(49.0, "Hyperion");

  System.out.println("Ksiazka przed zmiana ceny: " + ksiazka);

  zmienCene(30, ksiazka);

  System.out.println("Ksiazka po zmianie ceny: " + ksiazka);
}

public static void zmienCene(double cena, Produkt produkt) {
  cena = cena * 1.23;
  produkt.setCena(cena);
}
```

Wynik działania tego programu:

```
Ksiazka przed zmiana ceny: Produkt o nazwie Hyperion kosztuje 49.0
Ksiazka po zmianie ceny: Produkt o nazwie Hyperion kosztuje 36.9
```

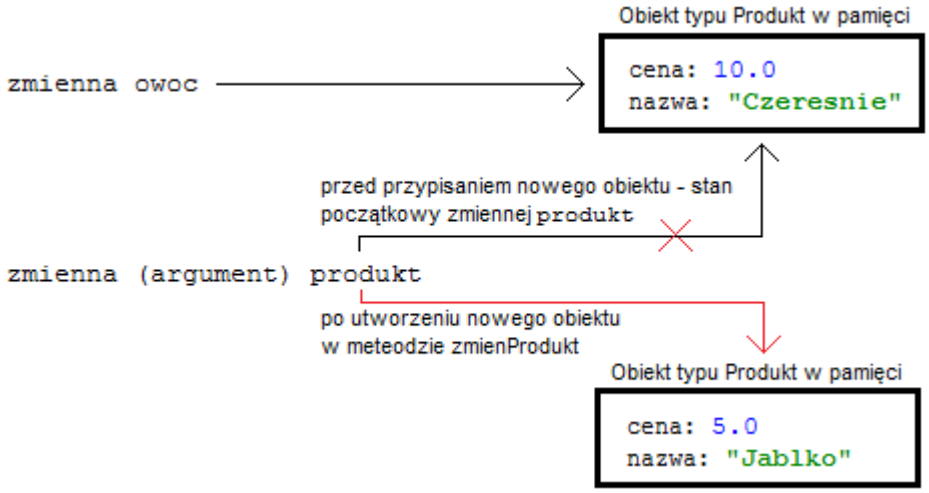
- Jeżeli w metodzie do jej argumentu typu złożonego przypiszemy nowy obiekt, to oryginalny obiekt, przesłany jako argument, nie zmieni się – zmieni się jedynie to, na co wskazuje argument metody:

```
public static void main(String[] args) {
  Produkt owoc = new Produkt(10.0, "Czeresnie");

  System.out.println("Owoc przed zmiana: " + owoc);

  zmienProdukt(owoc, 5.0, "Jablko");

  System.out.println("Owoc po zmianie: " + owoc);
}

public static void zmienProdukt(Produkt produkt, double nowaCena, String nowaNazwa){
  produkt = new Produkt(nowaCena, nowaNazwa);
}
```

W zaznaczonej linii przypisujemy do argumentu `produkt` nowo utworzony obiekt typu `Produkt`. Nie powoduje to zmiany obiektu utworzonego w pierwszej linii metody `main` o nazwie `owoc` – zmienia się jedynie obiekt, na który pokazuje argument `produkt` metody `zmienProdukt`. Wynik działania tego programu jest następujący:

```
Owoc przed zmiana: Produkt o nazwie Czeresnie kosztuje 10.0
Owoc po zmianie: Produkt o nazwie Czeresnie kosztuje 10.0
```

Powyższe zagadnienia przedstawia poniższy rysunek:



- Zmienne typów złożonych mogą być zdefiniowane z modyfikatorem `final` – spowoduje to, że zmiennej nie będzie można przypisać innego obiektu (pierwszy fragment kodu poniżej), **ale obiekt, na który zmienna ta wskazuje, będzie mógł być zmodyfikowany** (drugi fragment kodu):

```
final Punkt punkt = new Punkt(10, 20);
punkt = new Punkt(25, 30); // blad kompilacji!
```

```
final Punkt punkt = new Punkt(10, 20);
System.out.println(punkt);

punkt.setX(5);
System.out.println(punkt);
```

Pierwszy fragment kodu spowodowałby błąd kompilacji, natomiast drugi jest poprawny i spowoduje wypisanie na ekran:

```
Punkt(x: 10, y: 20)
Punkt(x: 5, y: 20)
```

Musimy pamiętać, że w tym przypadku to nie obiekt, któremu zmieniamy wartość pola `x`, lecz zmienna o nazwie `punkt`, jest **`final`**.

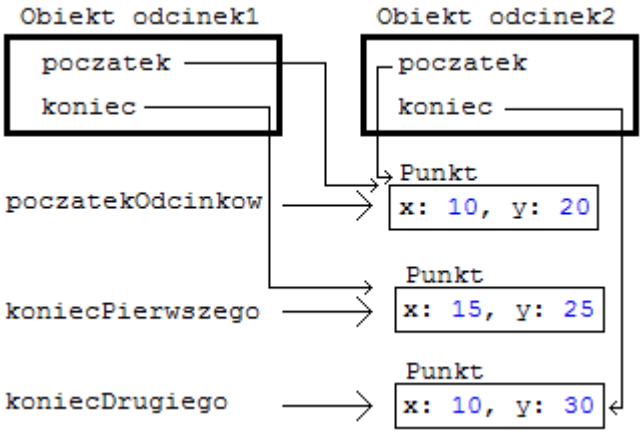
- Obiekty, które tworzymy, mogą być współdzielone przez wiele innych obiektów. Powoduje to, że zmiana współdzielonego obiektu pociągnie za sobą de facto zmianę także obiektów, które z nich korzystają:

```
Punkt poczatekOdcinkow = new Punkt(10, 20);
Punkt koniecPierwszego = new Punkt(15, 25);
Punkt koniecDrugiego = new Punkt(10, 30);

Odcinek odcinek1 = new Odcinek(poczatekOdcinkow, koniecPierwszego);
Odcinek odcinek2 = new Odcinek(poczatekOdcinkow, koniecDrugiego);

System.out.println(odcinek1);
System.out.println(odcinek2);

odcinek1.przesunPoczatek(0, 5);

System.out.println(odcinek1);
System.out.println(odcinek2);
```

Oba obiekty `odcinek1` oraz `odcinek2` korzystają z tego samego obiektu typu `Punkt` o nazwie `poczatekOdcinkow`. Gdy w zaznaczonej linii zmieniamy początkowe położenie pierwszego odcinka, to zmiany widoczne będą także w obiekcie `odcinek2`, ponieważ korzysta on z tego samego obiektu typu `Punkt` – na ekranie zobaczymy:

```
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: 10, y: 20) i Punkt(x: 10, y: 30)
Odcinek zawarty pomiedzy Punkt(x: -20, y: 50) i Punkt(x: 15, y: 25)
Odcinek zawarty pomiedzy Punkt(x: -20, y: 50) i Punkt(x: 10, y: 30)
```

Na poniższym obrazku widać, że pola `poczatek` obu obiektów `odcinek1` i `odcinek2` wskazują na ten sam obiekt w pamięci, na który wskazuje także zmienna `poczatekOdcinkow`:



- Aby uchronić się przed potencjalnymi problemami związanymi ze współdzieleniem obiektów, możemy:
  - w konstruktorze robić kopie przesyłanych argumentów

  lub

  - korzystać z obiektów niemutowalnych (immutable objects).
- Obiekty niemutowalne to takie obiekty, których wartości nie mogą zmienić się po ich utworzeniu.
- Aby zapewnić niemożliwość zmiany, klasy, których obiekty mają być niemutowalne, nie posiadają setterów oraz nie udostępniają żadnego sposobu na modyfikację wewnętrznego stanu obiektów tej klasy. Wartości pól raz utworzonego, niemodyfikowalnego obiektu, pozostają takie same przez całe "życie" takiego obiektu.
- Przykład klasy, której obiekty są niemutowalne – po utworzeniu obiektu tej klasy, nie ma możliwości na zmianę jej pól:

```
public class NiemutowalnyPunkt {
  private final int x;
  private final int y;

  public NiemutowalnyPunkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public String toString() {
    return "NiemutowalnyPunkt(x: " + x + ", y: " + y + ")";
  }
}
```

- Obiekty niemutowalne mogą być współdzielone przez różne obiekty – będziemy mieli pewność, że ich stan się nie zmieni.
- Wadą obiektów niemutowalnych jest to, że potrzeba zmiany wartości takiego obiektu wiąże się z wymogiem utworzenia nowego obiektu z nowymi wartościami. Tworzenie nowego obiektu wymaga czasu oraz pamięci komputera, jednak, o ile nie będziemy tworzyli tysięcy "ciężkich" obiektów (mających wiele pól, z których wiele będzie obiektami mającymi kolejne obiekty itd.), to nie powinno być to problemem.
- Aby obiekty były niemutowalne, muszą spełniać kilka wymagań, jak opisano w oficjalnej dokumentacji języka Java na stronie:

  https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

  - Klasa nie powinna posiadać setterów.
  - Pola klasy powinny być prywatne i finalne (`private final`).

- Klasa powinna być finalna, tzn. nie powinno się móc tworzyć podklas, których ta klasa byłaby rodzicem. Pozwalając na tworzenie podklas klasy, która z założenia miała być niemutowalna, moglibyśmy dać użytkownikom szansę na zniesienie niemutowalności.
- Jeżeli klasa zawiera pola, które nie są niemutowalne (np. tablica – możemy przypisać inny obiekt do któregoś z elementów tablicy), to nie powinniśmy zwracać referencji do takich obiektów – w przeciwnym razie, użytkownicy uzyskają możliwość zmiany wewnętrznego stanu obiektu.

- Typ `String` jest przykładem typu niemutowalnego.
- Programy napisane w języku Java, które uruchamiamy w Maszynie Wirtualnej Java, mają dostęp do dwóch rodzajów pamięci – *sterty* (heap) oraz *stosu* (stack).
- Obiekty typu złożonego tworzone są na stercie, a prymitywnego – na stosie.
- Więcej pamięci dla sterty możemy ustawić podczas uruchamiania Maszyny Wirtualnej Java, np.: `java -Xmx2G NazwaKlasy`
- W języku Java nie musimy sami alokować pamięci na tworzone obiekty, ani go zwalniać – robi to za nas Maszyna Wirtualna Java. Mechanizm, które monitoruje, które obiekty można usunąć, a tym samym zwolnić zajmowane przez nie pamięć, nazywa się *Garbage Collector.*
- Poniżej znajduje się podsumowanie różnic pomiędzy typami złożonymi (referencyjnymi) a prymitywnymi:

	Typy Prymitywne	**Typy Referencyjne**
Wartości	konkretne, np. `5`, `true`, `'a'`	adresy obiektów
Tworzenie	zdefiniowanie zmiennej	operator `new`
Domyślne wartości	zmienne lokalne – brak, pola klas – zależne od typu	zmienne lokalne – brak, pola klas – `null`
Porównywanie	porównywane są wartości	porównywane są wartości, którymi są adresy obiektów, a nie obiekty, więc użycie operatora == zwróci `true` tylko wtedy, gdy dwie zmienne będą wskazywały na dokładnie ten sam obiekt w pamięci – do porównywania obiektów złożonych powinniśmy stosować metodę `equals`
Pamięć	tworzone są na stosie	tworzone są na stercie
Przesyłanie do metod	przez wartość	przez wartość – wysyłana jest referencja do obiektu, a nie kopia obiektu – obiekt źródłowy może zostać zmieniony, jeżeli wykonujemy na nim operacje

9.8.9 Pytania

1. Co zostanie wypisane w wyniku działania poniższego programu?

```
public class ZagadkaReferencje {
  private int x;

  public ZagadkaReferencje(int x) {
    this.x = x;
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getX() {
    return x;
  }

  public static void main(String[] args) {
    ZagadkaReferencje z1 = new ZagadkaReferencje(5);
    ZagadkaReferencje z2 = z1;

    z2.setX(100);

    System.out.println(z1.getX());
  }
}
```

2. Jak wartość zostanie wypisana na ekran?

```
public class ZagadkaArgument {
  public static void main(String[] args) {
    double wartosc = 5.0;

    ustawWartosc(wartosc, 10.0);

    System.out.println(wartosc);
  }

  public static void ustawWartosc(
      double wartoscDoZmiany, double nowaWartosc) {

    wartoscDoZmiany = nowaWartosc;
  }
}
```

3. Co zostanie wypisane na ekran?

```
public class NowyObiektZagadka {
  private String wiadomosc;

  public NowyObiektZagadka(String wiadomosc) {
    this.wiadomosc = wiadomosc;
  }

  public String getWiadomosc() {
    return wiadomosc;
  }
```

```
  public static void main(String[] args) {
    NowyObiektZagadka o = new NowyObiektZagadka("Witaj!");

    zmienObiekt(o, "Halo!");

    System.out.println(o.getWiadomosc());
  }

  public static void zmienObiekt(
      NowyObiektZagadka obiekt, String wiadomosc) {

    obiekt = new NowyObiektZagadka(wiadomosc);
  }
}
```

4. Co zostanie wypisane na ekran?

```
public class ZagadkaReferencje {
  private final int[] liczby;

  public ZagadkaReferencje(int[] liczby) {
    this.liczby = liczby;
  }

  public int sumaLiczb() {
    int suma = 0;

    for (int x : liczby) {
      suma += x;
    }

    return suma;
  }

  public static void main(String[] args) {
    int[] liczby = { 1, 10, 100 };

    ZagadkaReferencje o1 = new ZagadkaReferencje(liczby);
    ZagadkaReferencje o2 = new ZagadkaReferencje(liczby);

    liczby[0] = -100;

    System.out.println(o1.sumaLiczb());
    System.out.println(o2.sumaLiczb());
  }
}
```

5. Co zostanie wypisane na ekran?

```
public class StaleReferencje {
  public static void main(String[] args) {
    final int[] liczby = { 1, 2 , 3 };

    liczby = new int[] { 3, 2, 1, 0 };

    System.out.println(liczby[0]);
  }
}
```

6. Co zostanie wypisane na ekran?

```
public class StaleReferencje2 {
  public static void main(String[] args) {
    final int[] liczby = { 1, 2 , 3 };

    liczby[0] = 5;

    System.out.println(liczby[0]);
  }
}
```

7. Co charakteryzuje obiekty niemutowalne?
8. Czy, i dlaczego, obiekty poniższej klasy są, lub nie są, niemutowalne?

```
public class ZagadkaMutowalne {
  public final int x;

  public ZagadkaMutowalne(int x) {
    this.x = x;
  }
}
```

9. Czy, i dlaczego, obiekty poniższej klasy są, lub nie są, niemutowalne?

```
public class ZagadkaMutowalne2 {
  private String komunikat;

  public void setKomunikat(String komunikat) {
    this.komunikat = komunikat;
  }

  public String getKomunikat() {
    return komunikat;
  }
}
```

10. Czy, i dlaczego, obiekty poniższej klasy są, lub nie są, niemutowalne?

```
public class ZagadkaMutowalne3 {
  private final String[] slowa;

  public ZagadkaMutowalne3(String[] slowa) {
    this.slowa = slowa;
  }
}
```

11. Co to jest sterta i stos?
12. Czym różnią się typy prymitywne od typów referencyjnych (złożonych)?

9.8.10 Zadania

9.8.10.1 Immutowalna Ksiazka i Biblioteka

Napisz immutowalną klasę `Ksiazka` z polami `tytul`, `autor`, oraz `cena`, oraz metodą `toString`. Następnie, napisz immutowalną klasę `Biblioteka`, która będzie zawierała tablicę obiektów typu `Ksiazka`. W klasie `Biblioteka` zawrzyj metodę `getKsiazki`, która będzie zwracałą tablicę z obiektami typu `Ksiazka`, ktore przechowuje obiekt typu `Biblioteka`.

9.9 Metody i pola statyczne

Od początku naszej przygody z językiem Java korzystaliśmy ze słowa kluczowego **static** – każda metoda `main`, którą do tej pory napisaliśmy, była *statyczna*, tzn. jej sygnatura zawierała słowo kluczowe **static**:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

W klasach możemy definiować dwa rodzaje metod i pól:

- statyczne (korzystając z modyfikatora **static**),
- niestatyczne, nazywane także *polami i metodami instancji* (z ang. *non-static fields and methods*, czy też, *instance fields and methods*) – definiujemy je pomijając modyfikator **static**.

Pola statyczne różnią się od pól instancji (niestatycznych) tym, że są one współdzielone przez wszystkie obiekty tej klasy, tzn. przynależą one do całej klasy, a nie konkretnie utworzonego obiektu. Metody statyczne, natomiast, różnią się od metod niestatycznych tym, że nie mogą korzystać z pól i metod niestatycznych.

Wszystkie obiekty klasy mają dostęp do pól i metody statycznych. Mało tego – do pól i metod statycznych mamy dostęp nawet wtedy, gdy nie utworzymy żadnego obiektu klasy. Wartości pól statycznych są współdzielone przez wszystkie obiekty klasy – w przeciwieństwie do pól niestatycznych (instancji), których własne egzemplarze ma każdy obiekt klasy, pola statyczne są tworzone jako pojedyncze wartości/obiekty.

Metody statyczne mają dostęp jedynie do metod i pól, które także są statyczne. Nie mogą one odnosić się do niestatycznych pól – nie operują one na konkretnych obiektach klasy, lecz w kontekście całej klasy – nie mają one dostępu do obiektu **this**, który wskazuje na obiekt, na rzecz którego metoda została wywołana.

Zobaczymy teraz przykłady każdej z powyżej opisanych cech pól i metod statycznych.

9.9.1 Pola statyczne

Poniższa klasa zawiera dwa pola – jedno statyczne, a drugie nie:

Nazwa pliku: PrzykladStatic.java

```
public class PrzykladStatic {
  private int poleInstancji;

  private static int poleStatyczne = 5;

  public PrzykladStatic(int poleInstancji) {
    this.poleInstancji = poleInstancji;
  }

  public static void main(String[] args) {
    System.out.println("Pole statyczne (przez klase): " +
        PrzykladStatic.poleStatyczne // 1
    );

    PrzykladStatic obiekt1 = new PrzykladStatic(10);

    System.out.println("Pole statyczne (przez obiekt1): " +
        obiekt1.poleStatyczne // 2
    );
  }
}
```

Zauważmy, że w tym przykładzie odnosimy się do pola `poleStatyczne` *(1)*, zanim w ogóle utworzymy obiekt klasy `PrzykladStatic`, Aby to osiągnąć, piszemy nazwę klasy, po której następuje kropka oraz nazwa pola statycznego:

```
PrzykladStatic.poleStatyczne // 1
```

Do pola statycznego możemy odnieść się także za pomocą obiektu klasy, w której to pole jest zdefiniowane. W powyższym programie wypisujemy drugi raz na ekran wartość pola `poleStatyczne`, ale tym razem odnosimy się do niego za pomocą zmiennej `obiekt1` *(2)*:

```
obiekt1.poleStatyczne // 2
```

W wyniku działania powyższego programu, na ekranie zobaczymy:

```
Pole statyczne (przez klase): 5
Pole statyczne (przez obiekt1): 5
```

Do pól statycznych możemy odnosić się za pomocą zarówno nazwy klasy, jak i obiektów tej klasy, ale tego samego nie można powiedzieć o zmiennych niestatycznych – do takich zmiennych możemy odnosić się jedynie przez obiekty klasy – poniższa linijka kodu spowodowałaby błąd kompilacji, ponieważ pole `poleInstancji` nie jest statyczne:

```
System.out.println(PrzykladStatic.poleInstancji);
```

```
PrzykladStatic.java:14: error: non-static variable poleInstancji cannot be
referenced from a static context
    System.out.println(PrzykladStatic.poleInstancji);
                                      ^
1 error
```

Na początku tego podrozdziału wspomnieliśmy, że pola statyczne są współdzielone przez wszystkie obiekty klasy – dodamy do metody `main` z powyższego przykładu jeszcze jeden obiekt klasy `PrzykladStatic`:

metoda main z pliku PrzykladStatic.java

```
public static void main(String[] args) {
  System.out.println("Pole statyczne (przez klase): " +
      PrzykladStatic.poleStatyczne // 1
  );

  PrzykladStatic obiekt1 = new PrzykladStatic(10);

  System.out.println("Pole statyczne (przez obiekt1): " +
      obiekt1.poleStatyczne // 2
  );

  PrzykladStatic obiekt2 = new PrzykladStatic(15);

  obiekt2.poleStatyczne = -20; // 3

  System.out.println("Pole statyczne po zmianie (przez klase): " +
      PrzykladStatic.poleStatyczne // 4
  );

  System.out.println("Pole statyczne po zmianie (przez obiekt1): " +
      PrzykladStatic.poleStatyczne // 5
  );

  System.out.println("Pole statyczne po zmianie (przez obiekt2): " +
      PrzykladStatic.poleStatyczne // 6
  );
}
```

W tej wersji metody `main`, tworzymy drugi obiekt klasy `PrzykladStatic` o nazwie `obiekt2`, i za jego pomocą zmieniamy wartość pola statycznego `poleStatyczne` *(3)*. Następnie, ponownie wypisujemy wartość tego pola za pomocą nazwy klasy *(4)*, a także za pomocą obu obiektów *(5)* *(6)*. Na ekranie zobaczymy:

```
Pole statyczne (przez klase): 5
Pole statyczne (przez obiekt1): 5
Pole statyczne po zmianie (przez klase): -20
Pole statyczne po zmianie (przez obiekt1): -20
Pole statyczne po zmianie (przez obiekt2): -20
```

Zmieniając wartość pola statycznego `poleStatyczne` w linii:

```
obiekt2.poleStatyczne = -20; // 3
```

zmieniamy pole, które jest wspólne dla wszystkich obiektów klasy `PrzykladStatic` – dlatego, gdy wypisujemy wartość tego pola na ekran za pomocą zarówno `obiekt1` *(5)*, jak i `obiekt2` *(6)*, widzimy na ekranie tą samą wartość.

9.9.2 Metody statyczne

Gdy wywołujemy metodę statyczną, nie mamy w niej dostępu do obiektu, na rzecz którego metoda jest wywołana – w końcu możemy wywołać taką metodę nie mając w ogóle jeszcze utworzonego żadnego obiektu tej klasy. Powoduje to, że metody statyczne:

- nie mogą odnosić się do pól niestatycznych,
- nie mogą wywoływać metod niestatycznych.

Spójrzmy na poniższy przykład:

Nazwa pliku: PrzykladMetodyStatycznej.java

```
public class PrzykladMetodyStatycznej {
  private String poleInstancji;

  private static int poleStatyczne = 20;

  public void metodaInstancji() {
    // ?
  }

  public static void metodaStatyczna() {
    // ?
  }
}
```

Do jakich pól i metod mamy dostęp w metodach `metodaInstancji` i `metodaStatyczna`?

1. Pierwsza metoda, `metodaInstancji`, to metoda niestatyczna. Możemy się w niej odnieść do, zarówno, pól i metod statycznych, jak i niestatycznych, więc mamy dostęp do:
   a) pola `poleInstancji`,
   b) pola `poleStatyczne`,
   c) metody `metodaStatyczna`.
2. Metoda `metodaStatyczna` ma dostęp jedynie do innych metod statycznych i pól statycznych, więc w tej metodzie możemy się jedynie odnieść do pola statycznego `poleStatyczne`.

Spójrzmy na przykład wykorzystanie możliwych pól w metodzie `metodaInstancji`:

metoda z pliku PrzykladMetodyStatycznej.java

```
public void metodaInstancji() {
  System.out.println("Wywolales metode instancji.");

  // odwolanie do pola niestatycznego
  System.out.println(
      "poleInstancji (w metodzieInstancji): " + poleInstancji
  );

  // odwolanie do pola statycznego
  System.out.println(
      "poleStatyczne (w metodzieInstancji): " + poleStatyczne
  );

  System.out.println("Wywoluje metode statyczna z metody instancji:");

  // wywolanie metody statycznej
  metodaStatyczna();
}
```

Wypisujemy na ekran wartości pola statycznego i niestatycznego, a także, na końcu, wywołujemy metodę statyczną.

Z kolei metoda `metodaStatyczna` mogłaby wyglądać następująco:

fragment pliku PrzykladMetodyStatycznej.java

```
public static void metodaStatyczna() {
  System.out.println("Wywolales metode statyczna.");
  // blad!
  // kod zakomentowany - nie mozemy w metodzie statycznej
  // korzystac z pola niestatycznego
  //System.out.println(
  //     "poleInstancji (w metodzieStatycznej):" + poleInstancji
  //);

  // ok - mozemy w metodzie statycznej
  // korzystac z pol statycznych
  System.out.println(
      "poleStatyczne (w metodzieStatycznej): " + poleStatyczne
  );

  // blad!
  // nie mozemy tutaj wywolac metody niestatycznej
  // metodaInstancji();
}
```

Zauważmy, że w metodzie statycznej nie mamy dostępu ani do metody `metodaInstancji`, ani do pola `poleInstancji`. Spójrzmy na wywołanie obu powyższych metoda w metodzie `main`:

```
public static void main(String[] args) {
  System.out.println("Wywoluje metode statyczne za pomoca klasy");
  PrzykladMetodyStatycznej.metodaStatyczna(); // 1

  System.out.println("Tworze obiekt klasy PrzykladMetodyStatycznej");
  PrzykladMetodyStatycznej obiekt = new PrzykladMetodyStatycznej();
  obiekt.metodaInstancji(); // 2
  obiekt.metodaStatyczna(); // 3
}
```

W metodzie `main` najpierw wywołujemy metodę statyczną `(1)`, zanim w ogóle utworzymy obiekt klasy. Następnie, tworzymy obiekt klasy `PrzykladMetodyStatycznej` i wywołujemy zarówno metodę niestatyczną `(2)`, jak i statyczną `(3)`. Na ekranie zobaczymy:

```
Wywoluje metode statyczne za pomoca klasy
Wywolales metode statyczna.
poleStatyczne (w metodzieStatycznej): 20

Tworze obiekt klasy PrzykladMetodyStatycznej
Wywolales metode instancji.
poleInstancji (w metodzieInstancji): null
poleStatyczne (w metodzieInstancji): 20
Wywoluje metode statyczna z metody instancji:
Wywolales metode statyczna.
poleStatyczne (w metodzieStatycznej): 20

Wywolales metode statyczna.
poleStatyczne (w metodzieStatycznej): 20
```

Spójrzmy na jeszcze jeden przykład użycia metody i pola statycznego – w poniższej klasie `Komunikat` korzystamy z jednego pola statycznego i jednej metody statycznej w celu zliczania liczby obiektów tej klasy, które zostały utworzone:

Nazwa pliku: Komunikat.java

```
public class Komunikat {
  private final String komunikat;

  private static int liczbaObiektowTejKlasy;

  public Komunikat(String komunikat) {
    this.komunikat = komunikat;
    liczbaObiektowTejKlasy++; // 1
  }

  public static int ileObiektowUtworzono() {
    return liczbaObiektowTejKlasy;
  }

  public static void main(String[] args) {
    System.out.println("Liczba obiektow klasy Komunikat: " +
        Komunikat.ileObiektowUtworzono()
    ); // 2

    Komunikat k1 = new Komunikat("Witaj");
    Komunikat k2 = new Komunikat("Witam!");
    Komunikat k3 = new Komunikat("Halo?");

    System.out.println("Liczba obiektow klasy Komunikat: " +
        Komunikat.ileObiektowUtworzono()
    ); // 3
  }
}
```

W konstruktorze klasy `Komunikat` poza tym, że ustawiamy wartość pola instancji o nazwie `komunikat`, to zwiększamy także o `1` liczbę przechowywaną w statycznym polu `liczbaObiektowTejKlasy` *(1)*.

W metodzie `main` najpierw wypisujemy liczbę przechowywaną w statycznym polu, która zwracana jest przez statyczną metodę `ileObiektowUtworzono` *(2)*. Następnie tworzymy trzy obiekty klasy `Komunikat` i ponownie wypisujemy na ekran liczbę obiektów *(3)*:

```
Liczba obiektow klasy Komunikat: 0
Liczba obiektow klasy Komunikat: 3
```

Konstruktory nie mogą być statyczne – nie miałoby to sensu – służą one do zainicjalizowania tworzonego obiektu, a metody statyczne działają poza konteksem wszelkich obiektów klas.

9.9.3 Dlaczego metoda main jest statyczna?

Wszystkie metody `main`, jakie do tej pory pisaliśmy, były statyczne – zawsze dodawaliśmy w ich sygnaturze słowo kluczowe **`static`**. Takie jest wymaganie odnośnie naszych programów, które mają być uruchamiane, ale z czego to wynika?

Twórcy języka Java postanowili, że podobnie, jak w programach pisanych w językach C i C++, na których język Java jest wzorowany, programy napisane w języku Java będą rozpoczynały swoje działanie od metody o nazwie `main`.

Dzięki temu, że `main` jest metodą statyczną, Maszyna Wirtualna Java, która uruchamia i wykonuje kod naszego programu, nie musi tworzyć obiektu klasy, w której metoda `main` jest zawarta. Klasa

główna naszego programu (tzn. ta, która zawiera metodę `main`) mogłaby mieć wiele konstruktorów, bądź mieć konstruktor prywatny i nie pozwalać na tworzenie obiektów swojej klasy (w rozdziale o dziedziczeniu zobaczymy przykład prywatnych konstruktorów). Gdyby było wiele dostępnych konstruktorów, to który z nich powinnien zostać użyty przez Maszynę Wirtualną Java do utworzenia obiektu uruchamianej klasy?

Statyczność metody `main` upraszcza ten proces – Maszyna Wirtualna Java uruchamiając nasz program szuka w klasie, którą uruchamiamy, statycznej metody `main` i ją wykonuje, rozpoczynając tym samym działanie naszego programu.

9.9.4 Kiedy stosować pola i metody statyczne?

Pola statyczne przydają się, gdy potrzebujemy przechować w klasie pewne dane, które nie powinny przynależeć do każdego obiektu w klasie osobno, lecz są wspólną cechą wszystkich obiektów tej klasy, lub dane te mogą być przydatne przez klasy, które będą z niej korzystały.

Metody statyczne są często wykorzystywane jako metody pomocnicze, które nie mają stanu i nie potrzebują zapisywać nic w polach klasy, w których zostały zdefiniowane.

Przykładem klasy pomocniczej z biblioteki standardowej Java jest klasa `Math`, które posiada jedynie pola i metody statyczne – jej pola to różne stałe matematyczne, jak liczba Pi, a zdefiniowane w niej metody służą do różnych obliczeń matematycznych, jak na przykład `sin` (funkcja sinus), `round` (zaokrąglanie liczby rzeczywistej), czy też `sqrt` (*square root* – pierwiastek).

Metody klasy `Math` działają na przesłanych do nich argumentach i nie potrzebują zapisywać nic w polach klasy `Math`, ani nic z nich później odczytywać – klasa `Math` nie ma w ogóle żadnych pól niestatycznych. Wszystkie jej metody są metodami pomocniczymi, które możemy wykorzystywać w naszych programach do różnych obliczeń matematycznych.

Dzięki temu, że wszystkie metody i pola tej klasy są statyczne, nie musimy za każdym razem, gdy chcemy z niej skorzystać, tworzyć nowego obiektu tej klasy – wystarczy po prostu wywołać daną metodę za pomocą nazwy klasy.

Spójrzmy na przykład użycia klasy `Math`:

Nazwa pliku: KorzystanieZMath.java

```
public class KorzystanieZMath {
  public static void main(String[] args) {
    System.out.println("Liczba PI wynosi: " + Math.PI);
    System.out.println("Liczba E wynosi: " + Math.E);

    System.out.println(
        "Sinus 90 stopni wynosi: " + Math.sin(Math.toRadians(90))
    );
    System.out.println(
        "Zaokraglona liczba PI: " + Math.round(Math.PI)
    );
    System.out.println(
        "Pierwiastek liczby 100 to " + Math.sqrt(100)
    );
  }
}
```

W powyższym programie korzystamy ze statycznych pól `PI` oraz `E` klasy `Math`, a także kilku metod statycznych: `sin`, `toRadians`, `round`, oraz `sqrt`. Zauważmy, jak wygodne jest używanie klasy `Math` – nie musieliśmy utworzyć obiekt tej klasy, aby z niej korzystać – wywołujemy po prostu metody, które są nam potrzebne i odnosimy się do pól klasy

za pomocą nazwy tej klasy. Wygoda ta wnika z faktu, że te pola i metody są statyczne.

W wyniku działania powyższego programu, na ekranie zobaczymy:

```
Liczba PI wynosi: 3.141592653589793
Liczba E wynosi: 2.718281828459045
Sinus 90 stopni wynosi: 1.0
Zaokraglona liczba PI: 3
Pierwiastek liczby 100 to 10.0
```

Więcej informacji o klasie `Math` można znaleźć w oficjalnej dokumentacji języka Java:

https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Math.html

9.9.5 Podsumowanie

- W klasach możemy definiować dwa rodzaje metod i pól:
  - statyczne (korzystając z modyfikatora **`static`**),
  - niestatyczne, nazywane także *polami i metodami instancji* (z ang. *non-static fields and methods*, czy też, *instance fields and methods*) – definiujemy je pomijając modyfikator **`static`**.
- Pola statyczne różnią się od pól instancji (niestatycznych) tym, że są ona współdzielone przez wszystkie obiekty tej klasy, tzn. przynależą one do całej klasy, a nie konkretnie utworzonego obiektu. W przeciwieństwie do pól niestatycznych (instancji), których własne egzemplarze ma każdy obiekt klasy, pola statyczne są tworzone jako pojedyncze wartości/obiekty.
- Metody statyczne różnią się od metod niestatycznych tym, że nie mogą korzystać z pól i metod niestatycznych.
- Do pól i metod statycznych mamy dostęp nawet wtedy, gdy nie utworzymy żadnego obiektu klasy – w poniższym przykładzie odwołujemy się pola `poleStatyczne` za pomocą nazwy klasy w pierwszej z zaznaczonych linii. Do pola statycznego możemy także odnieść się za pomocą obiektu klasy, do której pole (bądź metoda) przynależy (druga podświetlona linia):

```
public class PrzykladStatic {
  private int poleInstancji;

  private static int poleStatyczne = 5;

  public PrzykladStatic(int poleInstancji) {
    this.poleInstancji = poleInstancji;
  }

  public static void main(String[] args) {
    System.out.println("Pole statyczne (przez klase): " +
        PrzykladStatic.poleStatyczne
    );

    PrzykladStatic obiekt1 = new PrzykladStatic(10);

    System.out.println("Pole statyczne (przez obiekt1): " +
        obiekt1.poleStatyczne
    );
  }
}
```

- Metody statyczne mają dostęp jedynie do metod i pól, które także są statyczne. Nie mogą one odnosić się do niestatycznych pól – nie operują one na konkretnych obiektach klasy, lecz w kontekście całej klasy – nie mają one dostępu do obiektu **`this`**, który wskazuje na obiekt, na rzecz którego metoda została wywołana.

- W poniższym przykładzie:
  - `metodaInstancji` ma dostęp do, zarówno, pól i metod statycznych, jak i niestatycznych, więc mamy dostęp do pola `poleInstancji`, pola `poleStatyczne`, oraz metody `metodaStatyczna`,
  - `metodaStatyczna` ma dostęp jedynie do innych metod statycznych i pól statycznych, więc w tej metodzie możemy się jedynie odnieść do pola statycznego `poleStatyczne`.

```
public class PrzykladMetodyStatycznej {
  private String poleInstancji;

  private static int poleStatyczne = 20;

  public void metodaInstancji() {
    // ?
  }

  public static void metodaStatyczna() {
    // ?
  }
}
```

- Metoda `main` jest statyczna, ponieważ dzięki temu Maszyna Wirtualna Java nie musi tworzyć obiektu klasy, którą chcemy uruchomić – metodę statyczną można wywołać nie mając utworzonego żadnego obiektu klasy.
- Pola statyczne przydają się, gdy chcemy zawrzeć w klasie dane, które nie powinny przynależeć do każdego obiektu w klasie osobno, lecz są wspólną cechą wszystkich obiektów tej klasy.
- Metody statyczne są często wykorzystywane jako metody pomocnicze, które nie mają stanu i nie potrzebują zapisywać nic w polach klasy, w których zostały zdefiniowane.
- Przykładem klasy pomocniczej z biblioteki standardowej Java jest klasa `Math`, które posiada jedynie pola i metody statyczne – jej pola to różne stałe matematyczne, jak liczba Pi, a zdefiniowane w niej metody służą do różnych obliczeń matematycznych, jak na przykład `sin` (funkcja sinus), `round` (zaokrąglanie liczby rzeczywistej), czy też `sqrt` (square root – pierwiastek).

9.9.6 Zadania

9.9.6.1 Klasa użyteczna Obliczenia

Napisz klasę `Obliczenia`, która będzie zawierała dwie metody **statyczne**:

- `silnia` – metoda powinna zwracać silnię podanej jako argument liczby,
- `sumaLiczb` – metoda powinna przyjmować tablicę liczby typu `int` i zwracać ich sumę.

Podobne metody pisaliśmy już w zadaniach do poprzednich rozdziałów – możemy je skopiować z tamtych programów.

Napisz kolejną klasę, o nazwie `WykonywanieObliczen`, która użyje w metodzie `main` obie metody z klasy `Obliczenia`.

9.10 Pakiety i importowanie klas

9.10.1 Pakiety klas

Do tej pory umieszczaliśmy nasze programy w tym samym katalogu na dysku. Prędzej czy później natrafimy jednak na sytuację, w której będziemy potrzebowali dwóch różnych klas, które będą miały taką samą nazwę – zdarzyło się to już w rozdziale o klasach – dwie przykładowe klasy używane do wyjaśnienia różnych zagadnień miały nazwę `Punkt`.

Biorąc pod uwagę, że tysiące osób programujących w języku Java może tworzyć własne klasy i udostępniać je do użycia przez inne osoby, potrzebny jest mechanizm, który pozwalałby na odróżnianie klas o takich samych nazwach. Jest to na tyle powszechny wymóg, że różne języki programowania dostaraczają rozwiązania dające tę funkcjonalność.

W języku Java możemy określić w jakim *pakiecie* (package) znajduje się klasa. Pakiet ten stanie się integralną częścią nazwy naszej klasy, dzięki czemu potencjalny problem z powtórzeniem tej samej nazwy będzie mało prawdopodobny.

Nazwy klas można by porównać do imion i nazwisko osób – często zdarza się, że dwie osoby (bądź więcej) mają takie samo imię i nazwisko. Pakiety z kolei można by porównać do numerów PESEL – pomimo, że dwie osoby mogą nazywać się tak samo, to numer PESEL pozwala na unikalną identyfikację konkretnej osoby.

Spójrzmy na poniższy przykład klasy `Powitanie`, która zdefiniowana jest w pakiecie `przykladowypakiet.pl`:

Nazwa pliku: przykladowypakiet/pl/Powitanie.java

```
package przykladowypakiet.pl;

public class Powitanie {
  public static void main(String[] args) {
    wypiszKomunikat();
  }

  public static void wypiszKomunikat() {
    System.out.println("Witaj Swiecie!");
  }
}
```

Ta prosta klasa zdefiniowana jest w pakiecie `przykladowypakiet.pl` – w tym celu użyte zostało nowe słowo kluczowe **`package`**, po którym następuje nazwa pakietu.

Załóżmy teraz, że chcielibyśmy mieć drugą klasę o nazwie `Powitanie`, która wypisywałaby komunikat po angielsku – umieścimy ją w pakiecie `przykladowypakiet.eng`, dzięki czemu nie wystąpi kolizja nazw:

Nazwa pliku: przykladowypakiet/eng/Powitanie.java

```
package przykladowypakiet.eng;

public class Powitanie {
  public static void main(String[] args) {
    wypiszKomunikat();
  }

  public static void wypiszKomunikat() {
    System.out.println("Hello World!");
  }
}
```

Obie powyższe klasy nazywają się `Powitanie`, jednak pełna nazwa każdej z klas to:

- `przykladowypakiet.pl.Powitanie`
- `przykladowypakiet.eng.Powitanie`

Zanim zobaczymy te klasy w akcji, musimy jeszcze zwrócić uwagę na trzy istotne cechy klas pakietowych:

- Przynależność klasy do pakietu (definiowana za pomocą słowa kluczowego **`package`**) musi być na początku pliku – jedyne, co może poprzedzać użycie **`package`**, to komentarze – w przeciwnym razie, nasza klasa w ogóle się nie skompiluje.
- Najlepiej, by nazwy pakietów zawierały tylko litery i liczby. Dodatkowo, człony nazwy pakietu rozdzialne są kropkami oraz nie stosujemy camelCase – używamy zawsze małych liter.
- Nazwa pakietu, w którym umieszczamy klasy, powinna odpowiadać strukturze katalogów na dysku, w których ta klasa się znajduje. Jest to bardzo ważne, ponieważ w przeciwnym razie Maszyna Wirtualna Java nie będzie w stanie znaleźć klasy o danej nazwie. Dla przykładu, pierwsza z powyższych klas `Powitanie` powinna być umieszczona w katalogu `pl`, który z kolei powinien być zawarty w katalogu o nazwie `przykladowypakiet`.

Ostatni podpunkt jest bardzo istotny – jeżeli definiujemy, że klasa ma być w pewnym pakiecie, to powinniśmy utworzyć na dysku komputera odpowiednią strukturę katalogów, odpowiadających temu pakietowi. Powyższe klasy `Powitanie` znajdują się w katalogach `pl` oraz `eng`, które to katalogi zawarte są w katalogu `przykladowypakiet`:

```
programowanie
│
└──przykladowypakiet
   ├───eng
   │       Powitanie.java
   │
   └───pl
           Powitanie.java
```

Powyżej zaprezentowana jest struktura katalogów, w których umieszczone są klasy `Powitanie` – `programowanie`, `przykladowypakiet`, `eng`, oraz `pl`, to foldery na dysku. Pierwszy folder to `programowanie`, ponieważ jest to nasz folder na przykłady z tego kursu, który utworzyliśmy w rozdziale pierwszym. Folder `programowanie` nie wchodzi w skład pakietu, w którym klasy są zdefiniowane – jest to po prostu folder nadrzędny dla naszego pakietu klas.

Mając już odpowiednią strukturę katalogów i dwie klasy pakietowe, spróbujmy teraz uruchomić jedną z nich – przejdziemy do katalogu `przykladowypakiet/pl`, a następnie skompilujemy klasę i spróbujemy ją uruchomić:

```
C:\programowanie> cd przykladowypakiet

C:\programowanie\przykladowypakiet> cd pl

C:\programowanie\przykladowypakiet\pl> javac Powitanie.java

C:\programowanie\przykladowypakiet\pl> java Powitanie

Error: Could not find or load main class Powitanie
Caused by: java.lang.NoClassDefFoundError: przykladowypakiet/pl/Powitanie
(wrong name: Powitanie)
```

Korzystają z komendy **cd** (change directory), przeszliśmy do katalogu, w którym zdefiniowana jest jedna z klas `Powitanie`, po czym bez problemu ją skompilowaliśmy. Niestety, próba uruchomienia zakończyła się błędem – Maszyna Wirtualna Java nie może znaleźć szukanej klasy.

Problem wynika z faktu, że pełna nazwa naszej klasy to teraz `przykladowypakiet.pl.Powitanie`, a nie `Powitanie`. Jeżeli spróbowalibyśmy jednak podać taką nazwę, to ponownie zobaczymy błąd:

```
C:\programowanie\przykladowypakiet\pl> java przykladowypakiet/pl/Powitanie

Error: Could not find or load main class przykladowypakiet.pl.Powitanie
Caused by: java.lang.ClassNotFoundException: przykladowypakiet.pl.Powitanie
```

Maszyna Wirtualna Java nadal nie widzi klasy, którą skompilowaliśmy. Dlaczego tak się dzieje?

Kilka paragrafów wcześniej dowiedzieliśmy się, że struktura katalogów, w której klasa jest umieszczona, musi odpowiadać pakietowi, do którego klasa należy – wymóg ten jest spowodowany tym, że gdy Maszyna Wirtualna Java otrzymuje nazwę klasy pakietowej, to szuka klasy do uruchomienia w katalogach definiowanych przez pakiet tej klasy. Dlatego powyższe uruchomienie klasy się nie powiodło – Maszyna Wirtualna Java szukała klasy `Powitanie` w katalogu `pl` zawartym w katalog `przykladowypakiet` – poniżej przedstawione jest, czego oczekiwała Maszyna Wirtualna Java:

```
programowanie
│
└──przykladowypakiet
   ├──eng
   │      Powitanie.java
   │
   └──pl
      │   Powitanie.java
      │
      └──przykladowypakiet
         │
         └──pl
            │
            └──Powitanie.class
```

Maszyna Wirtualna Java potraktowała katalog `pl`, w którym aktualnie się znajdujemy, jako "główny" katalog. Otrzymując klasę o nazwie `przykladowypakiet.pl.Powitanie`, zaczęła szukać podkatalogu `przykladowypakiet`, a w nim kolejnego katalogu, o nazwie `pl`, w którym miała się znajdować skompilowana klasa `Powitanie`. Nie o to nam chodziło.

Jeżeli chcemy uruchomić klasę pakietową (tzn. zdefiniowaną w pewnym pakiecie przy użyciu słowa kluczowego **package**), to musimy znajdować się w katalogu, z którego Maszyna Wirtualna Java będzie mogła odnieść się do klasy, którą ma uruchomić, zgodnie z opisanym powyżej sposobem lokalizowania klas.

Jeżeli więc chcemy uruchomić klasę `przykladowypakiet.pl.Powitanie`, to musimy w lini poleceń znajdować się w katalogu, w którym znajduje się pierwszy katalog z pakietu klasy – w tym przypadku jest to katalog `przykladowypakiet` – ten katalog zawarty jest w katalog `programowanie` – i to jest właśnie miejsce, z którego musimy uruchomić naszą klasę:

```
C:\programowanie\przykladowypakiet\pl> cd ..

C:\programowanie\przykladowypakiet> cd ..

C:\programowanie> java przykladowypakiet.pl.Powitanie
Witaj Swiecie!
```

Korzystając dwukrotnie z komendy **cd** (change directory) z parametrem .. (dwie kropki), przeszliśmy do katalogu nadrzędnego – `programowanie`. W tym katalogu wywołujemy Maszynę Wirtualną Java, podając jako argument pełną nazwę klasy `przykladowypakiet.pl.Powitanie`. Tym razem widzimy, że kod naszej klasy został wykonany przez Maszynę Wirtualną Java – na ekranie pojawił się komunikat `Witaj Swiecie!`.

Jeżeli chcemy skompilować klasę pakietową, to nie musimy przechodzić do katalogu, w którym jest zdefiniowana – możemy zrobić to z poziomu katalog nadrzędnego dla danego pakietu (w naszym przypadku jest to katalog `programowanie`). Aby to osiągnąć, podajemy ścieżkę do klasy, którą chcemy skompilować – spójrzmy na przykład kompilacji drugiej klasy `Powitanie`:

```
C:\programowanie> javac przykladowypakiet/eng/Powitanie.java
```

Kompilatorowi `javac` przekazaliśmy jako argument klasę do skompilowania, gdzie ścieżka do tej klasy to pakiet, w którym klasa się znajduje – zauważmy, że tym razem nie korzystamy z kropek do rozdzielenia członów pakietu i klasy, lecz znaków slash /. Klasa została skompilowana bez błędów, a wynikowy plik `Powitanie.class` został umieszczony w lokalizacji `przykladowypakiet/eng`.

Możemy teraz uruchomić tę klasę:

```
C:\programowanie> java przykladowypakiet.eng.Powitanie
Hello World!
```

Druga klasa `Powitanie` wypisuje komunikat po angielsku.

Pełna nazwa klasy to pakiet i nazwa klasy, rozdzielone kropkami, na przykład `przykladowypakiet.pl.Powitanie` – takiej nazwy klasy oczekuje Maszyna Wirtualna Java jako argumentu.

Ścieżka do pliku z klasą do skompilowania, jakiej oczekuje kompilator języka Java (program javac), to nazwa klasy poprzedzona nazwami katalogów, w których klasa ta się znajduje (które mogą być częścią pakietu tej klasy) – w tym przypadku, nazwy katalogów oddzielamy znakiem slash / od nazwy klasy, np.:

`javac przykladowypakiet/eng/Powitanie.java`

Klasy nie muszą być zawarte w pakietach – do tej pory wszystkie klasy, które pisaliśmy, nie zawierały słowa kluczowego **`package`**. *Takie klasy są wtedy po prostu w ogólnym, "domyślnym" pakiecie.*

9.10.1.1 Konwencja nazewnicza pakietów klas

Istnieje konwencja nazewnicza pakietów klas, wedle której pakiety odzwierciedlają odwróconą domenę Internetową firm bądź osób, które są autorami klas.

Dla przykładu, pakiety klas napisane dla tego kursu mógłbym umieścić w pakiecie, którego nazwa zaczynałaby się od `com.kursjava` lub `com.przemyslawkruglej`. Kolejnym członem pakietu mogłyby być numery rozdziałów, z których pochodzą przykłady do tego kursu. Dla przykładu, programy z rozdziału o klasach mógłbym umieścić w pakiecie o nazwie `com.kursjava.rozdzial9`. W takim przypadku, kody źródłowe tych klas miałyby na początku następującą instrukcję **`package`**:

```
package com.kursjava.rozdzial9;
```

Po odwróconej domenie, firmy często umieszczają nazwę programu, do którego klasy należą, po której mogą wystąpić kolejne człony pakietu, w zależności od tego, jakie jest przeznaczenie klas. Jeżeli napisałbym grę kółko i krzyżyk, to "główna" część pakietu wszystkich klas mogłaby się nazywać `com.przemyslawkruglej.kolko_i_krzyzyk`.

Pakiety klas zdefiniowanych przez twórców języka Java znajdują się w pakietach zaczynających się od `java` oraz `javax`.

9.10.2 Importowanie klas

Aby skorzystać z klas, które zdefiniowane są w innych pakietach, musimy je *zaimportować* do naszych programów. Na przestrzeni ostatnich rozdziałów robiliśmy to już wielokrotnie – na przykład, gdy chcieliśmy pobrać od użytkownika dane – korzystaliśmy wtedy z następującej instrukcji importu:

```
import java.util.Scanner;
```

Aby pobrać dane od użytkownika, korzystaliśmy z klasy `Scanner` – jest to jedna z klas zdefiniowanych w jednym z pakietów Biblioteki Standardowej Java, czyli zestawu pakietów klas, które oddane są do użycia dla nas, programistów języka Java, przez twórców języka Java.

Jeżeli w programie, które chciałby korzystać z klasy `Scanner`, zabrakłoby instrukcji importu, to kompilacja programu zakończyłaby się błędem:

Nazwa pliku: WczytywanieDanychBezImport.java

```
public class WczytywanieDanychBezImport {
  public static int getInt() {
    // blad! braku import klasy Scanner
    return new Scanner(System.in).nextInt();
  }
}
```

Błąd kompilacji powyższego programu jest następujący:

```
WczytywanieDanychBezImport.java:4: error: cannot find symbol
    return new Scanner(System.in).nextInt();
               ^
  symbol:   class Scanner
  location: class WczytywanieDanychBezImport
1 error
```

Aby zaimportować klasę, korzystamy ze słowa kluczowego **`import`**, po którym następuje nazwa klasy, którą chcemy zaimportować. Możemy też zaimportować wszystkie klasy w pakiecie – zamiast nazwy klasy na końcu pakietu, korzystamy w takim przypadku z * (gwiazdka), która oznacza "wszystkie klasy".

> Instrukcje **`import`** muszą występować po (ewentualnej) instrukcji **`package`**, a przed definicją klasy – w przeciwnym razie kod klasy się nie skompiluje.

Wróćmy do klas `Powitanie` z poprzedniego rozdziału. Załóżmy, że w katalogu nadrzędnym `programowanie` istnieć będzie klasa `PrzykladImportu`, w której chcielibyśmy skorzystać z klasy `Powitanie` z pakietu `przykladowypakiet.pl` – hierarchia plików będzie następująca:

```
programowanie
│
│  PrzykladImportu.java
│
└──przykladowypakiet
   ├───eng
   │       Powitanie.java
   │
   └───pl
           Powitanie.java
```

Klasa `PrzykladImportu` nie znajduje się w żadnym pakiecie – nie korzystamy w niej z instrukcji **`package`**, jednak użyjemy w niej instrukcji **`import`** do zaimportowania klasy `przykladowypakiet.pl.Powitanie`:

Nazwa pliku: PrzykladImportu.java

```
import przykladowypakiet.pl.Powitanie;

public class PrzykladImportu {
  public static void main(String[] args) {
    // korzystamy z zaimportowanej klasy Powitanie
    // wywolujemy statyczna metoda tej klasy
    Powitanie.wypiszKomunikat();
  }
}
```

Na początku programu znajduje się instrukcja importu klasy `Powitanie` z pakietu `przykladowypakiet.pl`. Dzięki tej instrukcji uzyskujemy dostęp do klasy `Powitanie`, którą używamy do wypisania na ekran powitania, korzystając z jej metody statycznej `wypiszKomunikat`.

Zamiast podawać nazwę klasy, moglibyśmy skorzystać z * (gwiazdki), aby zaimportować wszystkie klasy z danego pakietu (ale nie z podpakietów!):

Nazwa pliku: PrzykladImportuZGwiazdka.java

```
import przykladowypakiet.pl.*;

public class PrzykladImportuZGwiazdka {
  public static void main(String[] args) {
    // korzystamy z zaimportowanej klasy Powitanie
    // wywolujemy statyczna metoda tej klasy
    Powitanie.wypiszKomunikat();
  }
}
```

Tym razem, zamiast podać nazwę klasy, którą chcemy zaimportować, podajemy gwiazdkę – zaimportowane zostaną wszystkie klasy z pakietu `przykladowypakiet.pl`, w tym – klasa `Powitanie`, która, w tym przypadku, jest jedyną klasą w tym pakiecie.

Należy tutaj zwrócić uwagę, że importowane są jedynie klasy z danego pakietu – jeżeli pakiet zawiera kolejne podkatalogi z klasami, to nie zostaną one zaimportowane.

Spójrzmy na poniższy, błędny przykład – importujemy wszystkie klasy z nadrzędnego pakietu `przykladowypakiet` – ten pakiet nie zawiera żadnych klas, a użycie gwiazdki **nie spowoduje**, że zaimportowane zostaną obie klasy `Powitanie` z pakietów podrzędnych `pl` oraz `eng`:

```
// uzycie * nie powoduje, ze zaimportowane zostana klasy
// z pakietow podrzednych - wystapi blad kompilacji,
// poniewaz kompilator nie wie, czym jest 'Powitanie'
import przykladowypakiet.*;

public class BlednyImport {
  public static void main(String[] args) {
    Powitanie.wypiszKomunikat();
  }
}
```

Próba kompilacji tej klasy zakończy się błędem.

Wiemy już, jak importować klasy z pakietów. Załóżmy teraz, że potrzebujemy w naszym programie

skorzystać z obu klas `Powitanie` – tej z pakietu `przykladowypakiet.pl` i tej z pakietu `przykladowypakiet.eng`. Spróbujmy zaimportować je obie:

Nazwa pliku: ImportDwochKlasOTejSamejNazwie.java

```
import przykladowypakiet.pl.Powitanie;
import przykladowypakiet.eng.Powitanie;

public class ImportDwochKlasOTejSamejNazwie {
  public static void main(String[] args) {
    Powitanie.wypiszKomunikat();
  }
}
```

Niestety, próba kompilacji tego programu kończy się dwoma błędami:

```
ImportDwochKlasOTejSamejNazwie.java:2: error: a type with the same simple
name is already defined by the single-type-import of Powitanie
import przykladowypakiet.eng.Powitanie;
^
ImportDwochKlasOTejSamejNazwie.java:6: error: reference to Powitanie is
ambiguous
    Powitanie.wypiszKomunikat();
    ^
  both class przykladowypakiet.pl.Powitanie in przykladowypakiet.pl and
class pr
zykladowypakiet.eng.Powitanie in przykladowypakiet.eng match
2 errors
```

Kompilator protestuje, ponieważ próbujemy zaimportować dwie klasy o tej samej nazwie i kompilator nie wie, czym jest `Powitanie` – czy chodzi nam o klasę z pakietu `przykladowypakiet.pl`, czy `przykladowypakiet.eng`? Jest jednak sposób, aby skorzystać z obu klas – zaimportujemy jedną z nich, a do drugiej odniesiemy się korzystając z jej pełnej nazwy, to znaczy dodamy przed nazwą klasy nazwę pakietu, w którym jest zawarta:

Nazwa pliku: KorzystanieZDwochKlasOTejSamejNazwie.java

```
import przykladowypakiet.pl.Powitanie;

public class KorzystanieZDwochKlasOTejSamejNazwie {
  public static void main(String[] args) {
    // klasa z pakietu przykladowypakiet.pl.Powitanie
    Powitanie.wypiszKomunikat();

    // inna klasa o nazwie Powitanie zdefiniowana w innym pakiecie
    // korzystamy z pelnej nazwy tej klasy, aby sie do niej odniesc
    przykladowypakiet.eng.Powitanie.wypiszKomunikat();
  }
}
```

Tym razem, program kompiluje się błędów, a na ekranie zobaczymy:

```
Witaj Swiecie!
Hello World!
```

Jedną z klas zaimportowaliśmy do naszego programu, dzięki czemu możemy korzystać z jej skróconej nazwy – jest to klasa `Powitanie` z pakietu `przykladowypakiet.pl`. Do drugiej klasy odnosimy się podając jej pełną nazwę, tzn. nazwę poprzedzoną pakietem, w którym się znajduje. Dzięki temu, kompilator nie ma problemu z rozróżnieniem, o które klasy nam chodzi. Dodatkowo,

pomimo, iż druga klasa `Powitanie` nie jest zaimportowana za pomocą instrukcji **`import`**, to kompilator i tak wie, gdzie tej klasy szukać – należy ona do pakietu `przykladowypakiet.eng`, więc będzie jej szukał w katalogu `przykladowypakiet/eng`.

Nasze programy mogą zawierać dowolną liczbę instrukcji import – możemy zaimportować tyle klas, ile nam potrzeba.

W programach pisanych w języku Java bardzo często korzysta się z klas z innych pakietów. Są różne konwencje dotyczące tego, czy powinniśmy korzystać z * (gwiazdki), czy używać konkretnych nazw klas, które chcemy zaimportować. Ja spotkałem się głównie z konwencją, w której zawsze podajemy nazwę klasy do zaimportowania, zamiast korzystać z gwiazdek. Dla przykładu, jeżeli chcemy skorzystać z klasy `Scanner`, to zamiast napisać:

```
import java.util.*;
```

powinniśmy użyć instrukcji:

```
import java.util.Scanner;
```

9.10.2.1 Importy statyczne – static import

Czasami w naszych programach potrzebujemy skorzystać z pewnej metody statycznej bądź stałej z innej klasy wiele razy – w takim przypadku, musimy za każdym razem przed nazwą metody (bądź stałej) umieścić nazwę klasy, z której pochodzi – spójrzmy na przykład z rozdziału o metodach i polach statycznych, w którym korzystaliśmy z klasy `Math`:

Nazwa pliku: KorzystanieZMath.java

```
public class KorzystanieZMath {
  public static void main(String[] args) {
    System.out.println("Liczba PI wynosi: " + Math.PI);
    System.out.println("Liczba E wynosi: " + Math.E);

    System.out.println(
        "Sinus 90 stopni wynosi: " + Math.sin(Math.toRadians(90))
    );
    System.out.println(
        "Zaokraglona liczba PI: " + Math.round(Math.PI)
    );
    System.out.println(
        "Pierwiastek liczby 100 to " + Math.sqrt(100)
    );
  }
}
```

Przed stałymi statycznymi `PI` oraz `E`, a także przed metodami `sin`, `round`, `toRadians`, oraz `sqrt`, musimy umieścić nazwę klasy, z której pochodzą, czyli `Math`.

Niekiedy korzystamy z pewnej metody bądź stałej z innej klasy tak często, że pisanie nazwy klasy za każdym razem staje się narzutem. Możemy w takim przypadku skorzystać ze *statycznego importu*.

Statyczny import to zaimportowanie metody bądź pola do naszej klasy w taki sposób, jakby ta metoda bądź stała była cześcią naszej klasy, a nie pochodziła z innej klasy – dzięki temu, nie musimy pisać nazwy klasy za każdym razem, gdy z tej metody bądź stałej będziemy korzystać – spójrzmy na przykład:

Nazwa pliku: StatycznyImportMath.java

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class StatycznyImportMath {
  public static void main(String[] args) {
    System.out.println("Liczba PI wynosi: " + PI);
    System.out.println("Liczba E wynosi: " + Math.E);

    System.out.println(
        "Sinus 90 stopni wynosi: " + Math.sin(Math.toRadians(90))
    );
    System.out.println(
        "Zaokraglona liczba PI: " + Math.round(PI)
    );
    System.out.println(
        "Pierwiastek liczby 100 to " + sqrt(100)
    );
  }
}
```

W tym przykładzie korzystamy ze statycznego importu do zaimportowania do klasy `StatycznyImportMath` statyczną stałą `PI` z klasy `Math`, oraz statyczną metodę `sqrt` z tej samej klasy. Dzięki temu, w zaznaczonych liniach nie musimy umieszczać nazwy klasy `Math` przed stałą `PI` oraz metodą `sqrt`.

Statyczny import różni się tym od zwykłego importowania klas, że po słowie kluczowym **`import`** dodajemy słowo kluczowe **`static`**:

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
```

Moglibyśmy użyć * (gwiazdki), aby zaimportować wszystkie pola i metody publiczne z klasy `Math` – wtedy nazwę klasy `Math` moglibyśmy pominąć przed stałą `E` oraz metodami `sin`, `toRadians`, oraz `round`:

Nazwa pliku: StatycznyImportMathZGwiazdka.java

```
import static java.lang.Math.*;

public class StatycznyImportMathZGwiazdka {
  public static void main(String[] args) {
    System.out.println("Liczba PI wynosi: " + PI);
    System.out.println("Liczba E wynosi: " + E);

    System.out.println(
        "Sinus 90 stopni wynosi: " + sin(toRadians(90))
    );
    System.out.println(
        "Zaokraglona liczba PI: " + round(PI)
    );
    System.out.println(
        "Pierwiastek liczby 100 to " + sqrt(100)
    );
  }
}
```

Dzięki użyciu gwiazdki, mogliśmy usunąć nazwę klasy `Math` sprzed jej pól i metod, z których korzystamy w powyższym przykładzie.

Statyczny import przydaje się od czasu do czasu, gdy wielokrotnie korzystamy z pewnych metod bądź pól innych klas, w szczególności z Biblioteki Standardowej Java. Nie powinniśmy jednak nadużywać tej funkcjonalności, ponieważ może to powodować, że nasz kod będzie trudniejszy w zrozumieniu, lub wystąpi kolizja nazwy metody bądź pola z naszej klasy i klasy, z której statycznie importujemy metodę bądź pole.

Nie powinniśmy także korzystać z opcji z * (gwiazdka), ponieważ powoduje to "hurtowy" import wszystkich publicznych pól i metod – najlepiej, tak jak w przypadku zwykłych importów, zawsze podawać nazwę obiektu, który chcemy zaimportować, zamiast korzystać z gwiazdki.

9.10.2.2 Lokalizacja klas – classpath

Czasem pisząc nasze programy będziemy chcieli korzystać z różnych klas, które niekoniecznie będą znajdować się w tym samym katalogu nadrzędnym, jak nasz program.

Załóżmy, że na innym dysku naszego komputera, np. na dysku D, znajduje się klasa `WykorzystanieClasspath`, w której chcielibyśmy skorzystać z klas `Powitanie`, zdefiniowanych wcześniej w tym rozdziale. Lokalizacja tych klas przedstawiona jest poniżej:

```
C:
│
└──programowanie
    │
    │ PrzykladImportu.java
    │
    └──przykladowypakiet
       ├───eng
       │       Powitanie.java
       │
       └───pl
               Powitanie.java

D:
│
└──programy
    │
    │
    └── WykorzystanieClasspath.java
```

Jeżeli w klasie `WykorzystanieClasspath` spróbowalibyśmy użyć którejś z klas `Powitanie`, to kompilator zaprotestuje – nie będzie on w stanie odnaleźć na dysku D klasy, której chcemy użyć:

Nazwa pliku: WykorzystanieClasspath.java

```
import przykladowypakiet.pl.Powitanie;

public class WykorzystanieClasspath {
  public static void main(String[] args) {
    Powitanie.wypiszKomunikat();
  }
}
```

Kompilator poinformuje nas o błędzie, że pakiet, w którym klasa `Powitanie` miałaby się znajdować, nie istnieje:

```
WykorzystanieClasspath.java:1: error: package przykladowypakiet.pl does not
exist
import przykladowypakiet.pl.Powitanie;
                           ^
WykorzystanieClasspath.java:5: error: cannot find symbol
    Powitanie.wypiszKomunikat();
    ^
  symbol:   variable Powitanie
  location: class WykorzystanieClasspath
2 errors
```

Moglibyśmy utworzyć w lokalizacji `D:\programy` katalog `przykladowypakiet`, w którym z kolei utworzylibysmy katalog `pl`, do którego skopiowalibyśmy klasę `Powitanie`, ale mamy lepsze wyjście – możemy wskazać kompilatrowi języka java (oraz Maszynie Wirtualnej Java) lokalizację, gdzie powinien szukać klas, których nasz program potrzebuje. Jest to tzw. *classpath*, czyli lista katalogów na dysku komputera, gdzie mogą znajdować się klasy napisane w języka Java.

Aby odpowiednio ustawić *classpath*, przekazujemy kompilatorowi języka Java parametr o nazwie `-classpath`, po którym następuje lista lokalizacji na dysku, gdzie ma szukać klas – możemy podać kilka katalogów, rozdzielonych przecinkami. Spójrzmy na ponowną próbę skompilowania klasy `WykorzystanieClasspath` z odpowiednio ustawioną wartością *classpath*:

```
D:\programy> javac -classpath C:\programowanie WykorzystanieClasspath.java
```

Tym razem klasa kompiluje się bez błędów, ponieważ kompilator znalazł w katalogu `C:\programowanie` klasę `przykladowypakiet.pl.Powitanie`, której chcemy użyć w klasie `WykorzystanieClasspath`.

W wyniku kompilacji, kompilator utworzył plik `WykorzystanieClasspath.class`. Spróbujmy teraz uruchomić tą klasę w Maszynie Wirtualnej Java:

```
D:\programy> java WykorzystanieClasspath
Exception in thread "main" java.lang.NoClassDefFoundError:
przykladowypakiet/pl/Powitanie
        at WykorzystanieClasspath.main(WykorzystanieClasspath.java:5)
Caused by: java.lang.ClassNotFoundException: przykladowypakiet.pl.Powitanie
```

Uruchomienie klasy się nie powiodło – tym razem to Maszyna Wirtualna Java nie wie, gdzie szukać skompilowanej klasy `przykladowypakiet.pl.Powitanie`. Ponownie skorzystamy z argumentu `-classpath`:

```
D:\programy> java -classpath C:\programowanie WykorzystanieClasspath

Error: Could not find or load main class WykorzystanieClasspath
Caused by: java.lang.ClassNotFoundException: WykorzystanieClasspath
```

Ku potencjalnemu zaskoczeniu, tym razem Maszyna Wirtualna Java nie może znaleźć klasy, którą chcemy uruchomić, czyli `WykorzystanieClasspath` – dlaczego tak się stało?

Ustawiając wartość *classpath* na `C:\programowanie`, powiedzieliśmy Maszynie Wirtualnej Java: "*Wszystkie klasy, jakie będą potrzebne do uruchomienia klasy `WykorzystanieClasspath`, **jak i klasa `WykorzystanieClasspath`**, znajdują się w lokalizacji wskazywanej przez wartość argumentu classpath*".

Nie do końca o to nam chodziło – chcieliśmy, by uruchomiona została klasa, która znajduje się w katalogu `D:\programy` o nazwie `WykorzystanieClasspath`, a wszelkie potrzebne jej klasy można było znaleźć w katalogu `C:\programowanie`. W naszym argumencie *classpath* zabrakło jednej

lokalizacji – aktualnego katalogu, w którym wywołujemy Maszynę Wirtualną Java – jeżeli umieścimy tam tą lokalizację, to Maszyna Wirtualna Java będzie w stanie znaleźć klasę `WykorzystanieClasspath`.

W lini komend, aktualny katalog, w którym się znajdujemy, oznaczany jest przez jedną kropkę. Możemy więc do argumentu `-classpath` dodać po średniku kropkę, którą będzie oznaczała "katalog, w którym aktualnie znajdujemy się w linii komend" – w tym przypadku, kropka będzie oznaczała katalog `D:\programowanie`. Spróbujmy:

```
D:\programy> java -classpath C:\programowanie;. WykorzystanieClasspath
Witaj Swiecie!
```

Tym razem udało nam się uruchomić naszą klasę – Maszyna Wirtualna Java otrzymała wartość classpath składającą się z dwóch lokalizacji:

- `C:\programowanie` – tutaj znajduje się klasa `przykladowypakiet.pl.Powitanie`, używana przez klasę `WykorzystanieClasspath`,
- `.` (kropka) oddzieloną od lokalizacji `C:\programowanie` znakiem średnika – ta kropka oznacza aktualny katalog, w którym znajdujemy się w linii komend, więc będzie to `D:\programy`.

W pierwszej lokalizacji z argumentu *classpath* Maszyna Wirtualna Java znalazła klasę `przykladowypakiet.pl.Powitanie`, a w drugiej – klasę, którą chcieliśmy uruchomić, czyli `WykorzystanieClasspath`.

9.10.2.3 Kiedy nie trzeba stosować instrukcji import

Wielokrotnie w poprzednich rozdziałach korzystaliśmy z jednej z utworzonych przez nas klas w innej, także utworzonej przez nas klasie, jednak wtedy nie musieliśmy korzystać z instrukcji importu.

Klasy musimy importować tylko w przypadku, gdy są one zawarte w innych pakietach – jeżeli klasy są w tym samym katalogu (lub w tym samym pakiecie), to nie musimy korzystać z instrukcji **`import`** – dlatego w poprzednich rozdziałach nie korzystaliśmy z instrukcji **`import`** do importowania klas, które sami pisaliśmy – zawsze przetrzymywaliśmy je w tym samym katalogu.

Z drugiej jednak strony, w rozdziale o metodach statycznych korzystaliśmy z klasy `Math`, ale nie musieliśmy korzystać z instrukcji **`import`**, chociaż ta klasa nie znajdowała się w tym samym katalogu, co nasza klasa. Podobnie z używaną od dawna klasą `String` – nigdy jej nie importowaliśmy. Dlaczego w tych przypadkach błąd braku importu nie występował?

Biblioteka Standardowa Java zawiera pakiet o nazwie `java.lang`, w którym zdefiniowane są m. in. klasy `String` oraz `Math`, oraz wiele innych klas, które są tak często wykorzystywane przez programistów, że kompilator języka Java dodaje **`import`** tego pakietu do każdego programu napisanego w języku Java automatycznie, dla naszej wygody, dzięki czemu możemy korzystać m. in. z klas `String` oraz `Math` bez potrzeby ich importowania własnoręcznie w naszych programach.

9.10.3 Dostęp domyślny (default access) i klasy niepubliczne

9.10.3.1 Dostęp domyślny

W jednym z poprzednich podrodziałów poznaliśmy *modyfikatory dostępu* **`private`** oraz **`public`**, oraz dowidzieliśmy się, że istnieją łącznie cztery różne modyfikatory dostępu w języku Java:

- **`private`**
- *dostęp domyślny*
- **`protected`**
- **`public`**

Dostęp domyślny (*default access*) nazywany jest po angielsku także *package-private*. Ten rodzaj dostępu do pól i metod jest o tyle wyjątkowy, że w jego przypadku po prostu nie stosujemy żadnego modyfikatora dostępu – jeżeli przed definicją metody bądź pola w klasie nie użyjemy ani modyfikatora **`private`**, ani **`public`**, ani **`protected`**, to takie pole (bądź metoda) będzie miało *dostęp domyślny*.

Dostęp domyślny charakteryzuje się tym, że jest prawie tak restrykcyjny, jak modyfikator **`private`**, ale z jednym wyjątkiem. W przypadku modyfikatora **`private`**, żadna inna klasa nie może korzystać z pola bądź metody prywatnej. *Dostęp domyślny*, natomiast, ogranicza dostęp do pól i metod klasy, pozwalając jednak na korzystanie z nich innym klasom, które znajdują się w tym samym pakiecie, tzn. klasom, które mają taki sam pakiet zdefiniowany za pomocą słowa kluczowego **`package`**.

Poniżej zaprezentowana jest struktura katalogów, w których umieszczonych zostało kilka klas, które posłużą do zobrazowania, czym *dostęp domyślny* różni się od modyfikatorów **`private`** oraz **`public`**:

```
programowanie
│
│   ObcaKlasa.java
│
└───przykladdomyslny
        Komunikaty.java
        WtajemniczonaKlasa.java
```

W katalogu `programowanie` znajduje się klasa `ObcaKlasa`, natomiast klasy `Komunikaty` oraz `WtajemniczonaKlasa` są klasami pakietowymi – zawarte są w pakiecie o nazwie `przykladdomyslny`. Spójrzmy teraz treść klasy `Komunikaty`:

Nazwa pliku: przykladdomyslny/Komunikaty.java

```
package przykladdomyslny;

public class Komunikaty {
  private static void tajnyKomunikat() {
    System.out.println("Cii! Tajny komunikat!");
  }

  static void komunikatDlaWtajemniczonych() {
    System.out.println("Komunikat dla wtajemniczonych!");
  }

  public static void komunikatOgolny() {
    System.out.println("Bedzie padac.");
  }
}
```

Klasa ta zawiera trzy metody statyczne:

- metoda `tajnyKomunikat` jest prywatna, ponieważ zdefiniowana jest z modyfikatorem **`private`**,
- metoda `komunikatOgolny` jest publiczna – użyty został modyfikator **`public`**,
- nowością jest sposób definicji metody `komunikatDlaWtajemniczonych` – jej sygnatura nie ma żadnego modyfikatora dostępu, więc jej dostęp to *dostęp domyślny*.

Z prywatnej metody `tajnyKomunikat` możemy korzystać jedynie we wnętrzu klasy `Komunikaty`. Z publicznej metody `komunikatOgolny` mogą korzystać wszystkie klasy, niezależnie od tego, w jakim pakiecie się znajdują. Najciekawsza z tych trzech metod, metoda `komunikatDlaWtajemniczonych`, może zostać użyta w klasie `WtajemniczonaKlasa`, ponieważ obie klasy są w tym samym pakiecie. Z kolei klasa `ObcaKlasa`, będąca poza pakietem `przykladdomyslny`, nie ma prawa z tej metody korzystać.

Spójrzmy co się stanie, jeżeli spróbujemy użyć metody o domyślnym dostępie z klasy `Komunikaty` w klasie `ObcaKlasa`:

Nazwa pliku: ObcaKlasa.java

```
import przykladdomyslny.Komunikaty;

public class ObcaKlasa {
  public static void main(String[] args) {
    // blad! metoda komunikatDlaWtajemniczonych
    // jest niedostepna spoza pakietu przykladdomyslny!
    Komunikaty.komunikatDlaWtajemniczonych();
  }
}
```

Zauważmy, że musimy skorzystać z instrukcji importu, ponieważ klasa `Komunikaty`, z której chcemy skorzystać, znajduje się w pakiecie `przykladdomyslny`. Próba kompilacji tego programu kończy się następującym błędem:

```
ObcaKlasa.java:7: error: komunikatDlaWtajemniczonych() is not public in
Komunikaty; cannot be accessed from outside package
    Komunikaty.komunikatDlaWtajemniczonych();
              ^
1 error
```

Kompilator informuje nas, że metoda `komunikatDlaWtajemniczonych` jest niedostępna spoza pakietu `pakietdomyslny`. Podobny błąd zobaczylibyśmy, gdybyśmy spróbowali użyć metody `tajnyKomunikat`, która jest prywatna w klasie `Komunikaty`. Jedyna metoda z klasy `Komunikaty`, z której możemy skorzystać w klasie `ObcaKlasa`, to publiczna metoda `komunikatOgolny`:

```
import przykladdomyslny.Komunikaty;

public class ObcaKlasa {
  public static void main(String[] args) {
    // blad! metoda komunikatDlaWtajemniczonych
    // jest niedostepna spoza pakietu przykladdomyslny!
    //Komunikaty.komunikatDlaWtajemniczonych();

    Komunikaty.komunikatOgolny();
  }
}
```

Tym razem program kompiluje się bez błędów – na ekranie zobaczymy komunikat:

```
Bedzie padac.
```

Przejdźmy teraz do klasy `WtajemniczonaKlasa`, która znajduje się w tym samym pakiecie, co klasa `Komunikaty` – spróbujmy skorzystać z metody o domyślnym dostępie, zdefiniowanej w klasie `Komunikaty`:

Nazwa pliku: przykladdomyslny/WtajemniczonaKlasa.java

```
public class WtajemniczonaKlasa {
  public static void main(String[] args) {
    Komunikaty.komunikatDlaWtajemniczonych();
    Komunikaty.komunikatOgolny();
  }
}
```

Zauważmy, że tym razem nie stosujemy instrukcji importu, ponieważ obie klasy: `WtajemniczonaKlasa` oraz `Komunikaty`, znajdują się w tym samym pakiecie. Próba kompilacji powyższej klasy kończy się bezbłędnie, a w wyniku jej uruchomienia, na ekranie zobaczymy:

```
Komunikat dla wtajemniczonych!
Bedzie padac.
```

Tym razem kompilator nie protestował – klasa `WtajemniczonaKlasa` może korzystać z metody o domyślnym dostępie `komunikatDlaWtajemniczonych`, ponieważ klasa ta znajduje się w tym samym pakiecie, co klasa, w której ta metoda jest zdefiniowana.

Dostęp domyślny mogą mieć zarówno pola, jak i metody.

Dostęp domyślny dotyczy zarówno pól i metod statycznych, jak i niestatycznych.

9.10.3.2 Kiedy stosować dostęp domyślny?

Dostęp domyślny przydaje się, gdy projektujemy klasy, które będą wspólnie działały na rzecz wykonania pewnego zadania. Dzięki dostępowi domyślnemu, możemy pozwolić klasom z tego samego pakietu na dostęp do przydatnych metod i pól z innych klas z tego pakietu, nie dającym tym samym tej możliwości jakimkolwiek innym klasom, zdefiniowanym w innych pakietach. Tylko "wtajemniczone" klasy z tego samego pakietu będą mogły odnosić się do tych pól i metod, które dla świata zewnętrznego powinny pozostać prywatne i niedostępne.

9.10.3.3 Modyfikator dostępu protected

Ostatni modyfikator, `protected`, poznamy w rozdziale o dziedziczeniu. W tej chwili ważną informacją na temat tego modyfikatora jest to, że tak samo jak *dostęp domyślny*, pozwala on na dostęp do pól i metod klasom, które zdefiniowane są w tym samym pakiecie – jest to cecha wspólna modyfikatora `protected` oraz dostępu domyślnego (czyli tego, który występuje, gdy nie skorzystamy z żadnego modyfikatora dostępu).

9.10.3.4 Niepubliczne klasy

Do tej pory wszystkie klasy, jakie pisaliśmy, były publiczne – zawsze stosowaliśmy modyfikator dostępu `public` podczas ich definiowania, na przykład:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Witaj Swiecie!");
  }
}
```

Klasy nie muszą być jednak publiczne – możemy pominąć modyfikator `public`. W takim przypadku, gdy brak jest modyfikatora dostępu, klasa będzie miała *domyślny dostęp* – jego znaczenie będzie takie samo, jak w przypadku pól i metod, które mają *domyślny dostęp*, jak zobaczyliśmy wcześniej w tym podrozdziale.

Klasy z *dostępem domyślnym* mogą być wykorzystywane jedynie przez klasy znajdujące się w tym samym pakiecie – wszystkie klasy znajdujące się w innych pakietach (bądź w pakiecie domyślnym, gdy nie podamy żadnego pakietu), nie będą mogły z takiej klasy korzystać.

Kiedy stosować klasy niepubliczne? Jeżeli tworzymy zestaw klas, które mają wspólnie rozwiązywać pewien problem, to dobrze jest zdefiniować wszystkie klasy, które nie są potrzebne "światu zewnętrznemu" jako klasy niepubliczne – tylko te klasy, których użytkownicy pakietu będą używać bezpośrednio powinny być publiczne. Dzięki temu ukrywamy przed użytkownikiem docelowym implementację rozwiązania, na które składa się wiele klas. Takie podejście powoduje także, że osoby, które będą analizowały klasy z naszego pakietu, będą musiały zajrzeć tylko do klas publicznych, aby dowiedzieć się, jak korzystać z funkcjonalności, które wszystkie klasy w pakiecie (kolektywnie) udostępniają, ponieważ tylko z tych publicznych klas użytkownicy będą mogli korzystać.

9.10.4 Podsumowanie

9.10.4.1 Pakiety klas

- Aby zmniejszyć prawdopodobieństwo kolizji nazw klas, możemy umieszczać je w *pakietach*.
- Do oznaczenia, w jakim pakiecie klasa się znajduje, używamy słowa kluczowego **package**, po którym następuje nazwa pakietu.
- Nazwa pakietu staje się integralną cześcią nazwy klasy – zdefiniowana poniżej klasa `Powitanie` znajduje się w pakiecie `przykladowypakiet.pl`. Pełna nazwa tej klasy to `przykladowypakiet.pl.Powitanie`:

```
package przykladowypakiet.pl;

public class Powitanie {
  public static void main(String[] args) {
    wypiszKomunikat();
  }

  public static void wypiszKomunikat() {
    System.out.println("Witaj Swiecie!");
  }
}
```

- Przynależność klasy do pakietu musi być na początku pliku – jedyne, co może poprzedzać użycie słowa kluczowego **package**, to komentarze – w przeciwnym razie, klasa w ogóle się nie skompiluje.
- Najlepiej, by nazwy pakietów zawierały tylko litery i liczby. Dodatkowo, człony nazwy pakietu rozdzialne są kropkami oraz nie stosujemy camelCase – używamy zawsze małych liter.
- Nazwa pakietu, w którym umieszczamy klasy, powinna odpowiadać strukturze katalogów na dysku, w których ta klasa się znajduje. W przeciwnym razie Maszyna Wirtualna Java nie będzie w stanie znaleźć klasy o danej nazwie. Dla przykładu, powyższa klasa `Powitanie` powinna być umieszczona w katalogu `pl`, który z kolei powinien być zawarty w katalogu o nazwie `przykladowypakiet`:

```
programowanie
│
└──przykladowypakiet
   │
   └──pl
          Powitanie.java
```

- Jeżeli chcemy uruchomić klasę pakietową, to musimy znajdować się w katalogu, z którego Maszyna Wirtualna Java będzie mogła odnieść się do klasy, którą ma uruchomić, zgodnie z opisanym powyżej sposobem lokalizowania klas.

- Jeżeli więc chcielibyśmy uruchomić klasę `przykladowypakiet.pl.Powitanie`, to w lini poleceń powinniśmy znajdować się w katalogu, w którym znajduje się pierwszy katalog z pakietu klasy – w tym przypadku jest to katalog `przykladowypakiet` – ten katalog zawarty jest w katalog `programowanie` – i to jest właśnie miejsce, z którego powinniśmy uruchamiać klasę `przykladowypakiet.pl.Powitanie`:

```
C:\programowanie> javac przykladowypakiet/pl/Powitanie

C:\programowanie> java przykladowypakiet.pl.Powitanie
Witaj Swiecie!
```

- Pełna nazwa klasy to pakiet i nazwa klasy, rozdzielone kropkami, na przykład `przykladowypakiet.pl.Powitanie` – takiej nazwy klasy oczekuje Maszyna Wirtualna Java jako argumentu.
- Ścieżka do pliku z klasą do skompilowania, jakiej oczekuje kompilator języka Java (program javac), to nazwa klasy poprzedzona nazwami katalogów, w których klasa ta się znajduje (które mogą być częścią pakietu tej klasy) – w tym przypadku, nazwy katalogów oddzielamy znakiem slash / od nazwy klasy, np.:

```
javac przykladowypakiet/eng/Powitanie.java
```

- Klasy nie muszą być zawarte w pakietach – do tej pory wszystkie klasy, które pisaliśmy, nie zawierały słowa kluczowego **`package`**. Takie klasy są wtedy po prostu w ogólnym, "domyślnym" pakiecie.
- Istnieje konwencja nazewnicza pakietów klas, wedle której pakiety odzwierciedlają odwróconą domenę Internetową firm bądź osób, które są autorami klas.
- Dla przykładu, pakiety klas napisane dla tego kursu mógłyby być umieszczone w pakiecie, którego nazwa zaczynałaby się od `com.kursjava`. Kolejnym członem pakietu mogłyby być numery rozdziałów, z których pochodzą przykłady do tego kursu, np. `com.kursjava.rozdzial9`.
- Pakiety klas zdefiniowanych przez twórców języka Java znajdują się w pakietach zaczynających się od `java` oraz `javax`.

9.10.4.2 Importowanie klas

- Aby skorzystać z klas, które zdefiniowane są w innych pakietach, musimy je *zaimportować* do naszych programów.
- Klasy musimy importować tylko w przypadku, gdy są one zawarte w innych pakietach.
- Jeżeli spróbujemy skorzystać z klasy, której nie zaimportujemy, oraz która znajduje się w innym pakiecie, to kompilacja klasy zakończy się błędem:

```
public class WczytywanieDanychBezImport {
  public static int getInt() {
    // blad! braku import klasy Scanner
    return new Scanner(System.in).nextInt();
  }
}
```

```
WczytywanieDanychBezImport.java:4: error: cannot find symbol
    return new Scanner(System.in).nextInt();
               ^
  symbol:   class Scanner
  location: class WczytywanieDanychBezImport
1 error
```

- Aby zaimportować klasę, korzystamy ze słowa kluczowego **import**, po którym następuje nazwa klasy, którą chcemy zaimportować:

```
import przykladowypakiet.pl.Powitanie;

public class PrzykladImportu {
  public static void main(String[] args) {
    // korzystamy z zaimportowanej klasy Powitanie
    // wywolujemy statyczna metoda tej klasy
    Powitanie.wypiszKomunikat();
  }
}
```

- Możemy zaimportować wszystkie klasy w pakiecie korzystając z * (gwiazdka), która oznacza "wszystkie klasy", zamiast podawać nazwę klasy z pakietu. **Uwaga**: gwiazdka * powoduje zaimportowanie wszystkich klas z konkretnego pakietu, ale nie z podpakietów!

```
import przykladowypakiet.pl.*;

public class PrzykladImportuZGwiazdka {
  public static void main(String[] args) {
    // korzystamy z zaimportowanej klasy Powitanie
    // wywolujemy statyczna metoda tej klasy
    Powitanie.wypiszKomunikat();
  }
}
```

- Instrukcje **import** muszą występować po (ewentualnej) instrukcji **package**, a przed definicją klasy – w przeciwnym razie kod klasy się nie skompiluje.
- Nie możemy importować dwóch klas o takiej samej nazwie – spowodowałoby to błąd kompilacji.

- Jest jednak sposób, aby skorzystać z klas o takich samych nazwach – importujemy jedną z nich, a do drugiej odnosimy się korzystając z jej pełnej nazwy, czyli wraz z nazwą pakietu:

```
import przykladowypakiet.pl.Powitanie;

public class KorzystanieZDwochKlasOTejSamejNazwie {
  public static void main(String[] args) {
    // klasa z pakietu przykladowypakiet.pl.Powitanie
    Powitanie.wypiszKomunikat();

    // inna klasa o nazwie Powitanie zdefiniowana w innym pakiecie
    // korzystamy z pelnej nazwy tej klasy, aby sie do niej odniesc
    przykladowypakiet.eng.Powitanie.wypiszKomunikat();
  }
}
```

- Nasze programy mogą zawierać dowolną liczbę instrukcji `import` – możemy zaimportować tyle klas, ile nam potrzeba.
- Jedną z konwencji odnośnie importowania klas jest by nie korzystać z gwiazdki, a zamiast tego używać konkretnych nazw klas, które chcemy zaimportować.
- W Javie istnieje także inny rodzaj importu – *import statyczny*.
- *Statyczny import* to zaimportowanie metody bądź pola do naszej klasy w taki sposób, jakby ta metoda bądź stała była cześcią naszej klasy, a nie pochodziła z innej klasy – dzięki temu, nie musimy pisać nazwy klasy za każdym razem, gdy z tej metody bądź stałej będziemy korzystać:

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class StatycznyImportMath {
  public static void main(String[] args) {
    System.out.println("Liczba PI wynosi: " + PI);
    System.out.println("Liczba E wynosi: " + Math.E);

    System.out.println(
        "Sinus 90 stopni wynosi: " + Math.sin(Math.toRadians(90))
    );
    System.out.println(
        "Zaokraglona liczba PI: " + Math.round(PI)
    );
    System.out.println(
        "Pierwiastek liczby 100 to " + sqrt(100)
    );
  }
}
```

W tym przykładzie korzystamy ze statycznego importu do zaimportowania do klasy `StatycznyImportMath` statyczną stałą `PI` z klasy `Math`, oraz statyczną metodę `sqrt` z tej samej klasy. Dzięki temu, w zaznaczonych liniach nie musimy umieszczać nazwy klasy `Math` przed stałą `PI` oraz metodą `sqrt`.

- Statyczny import różni się tym od zwykłego importowania klas, że po słowie kluczowym **import** dodajemy słowo kluczowe **static**:

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
```

- Korzystając ze statycznego importu, także możemy użyć gwiazdki * zamiast nazwy pola bądź metody, aby zaimportować statycznie wszystkie publiczne pola i metody klasy.
- Statyczny import przydaje się od czasu do czasu, gdy wielokrotnie korzystamy z pewnych metod bądź pól innych klas, w szczególności z Biblioteki Standardowej Java. Nie powinniśmy jednak nadużywać tej funkcjonalności, ponieważ może to powodować, że nasz kod będzie trudniejszy w zrozumieniu, lub wystąpi kolizja nazwy metody bądź pola z naszej klasy i klasy, z której statycznie importujemy metodą bądź pole.
- *classpath* to lista katalogów na dysku komputera, gdzie mogą znajdować się klasy napisane w języka Java, których kompilator języka Java może potrzebować do skompilowania naszego programu, oraz Maszyna Wirtualna Java do jego uruchomienia.
- Aby odpowiednio ustawić *classpath*, przekazujemy kompilatorowi języka Java parametr o nazwie `-classpath`, po którym następuje lista lokalizacji na dysku, gdzie ma szukać klas – możemy podać kilka katalogów, rozdzielonych przecinkami:

```
D:\programy> javac -classpath C:\programowanie WykorzystanieClasspath.java
```

- W lini komend, aktualny katalog, w którym się znajdujemy, oznaczany jest przez jedną kropkę.
- Jeżeli uruchamiamy Maszynę Wirtualną Java z parametrem *classpath*, a klasa, którą chcemy wykonać znajduje się w aktualnym katalogu, to do *classpath* powinniśmy dodać aktualny katalog, czyli kropkę, aby Maszyna Wirtualna Java była w stanie odnaleźć klasę, którą chcemy uruchomić:

```
D:\programy> java -classpath C:\programowanie;. WykorzystanieClasspath
Witaj Swiecie!
```

- Biblioteka Standardowa Java zawiera pakiet o nazwie `java.lang`, w którym zdefiniowane są m. in. klasy `String` oraz `Math`, oraz wiele innych klas, które są tak często wykorzystywane przez programistów, że kompilator języka Java dodaje import tego pakietu do każdego programu napisanego w języku Java automatycznie, dla naszej wygody.

9.10.4.3 Dostęp domyślny

- W Javie, poza modyfikatorami dostępu **`public`** oraz **`private`**, istnieją jeszcze modyfikatory: *domyślny* oraz **`protected`**.
- *Dostęp domyślny* (*default access*) nazywany jest po angielsku także *package-private*. W przypadku tego rodzaju dostępu nie stosujemy żadnego modyfikatora dostępu – jeżeli przed definicją metody bądź pola w klasie nie użyjemy ani modyfikatora **`private`**, ani **`public`**, ani **`protected`**, to takie pole (bądź metoda) będzie miało *dostęp domyślny*.
- Dostęp domyślny jest prawie tak restrykcyjny, jak modyfikator **`private`**, z jednym wyjątkiem. W przypadku modyfikatora **`private`**, żadna inna klasa nie może korzystać z pola bądź metody prywatnej. Dostęp domyślny, natomiast, ogranicza dostęp do pól i metod klasy, pozwalając jednak na korzystanie z nich innym klasom, które znajdują się w tym samym pakiecie, tzn. klasom, które mają taki sam pakiet zdefiniowany za pomocą słowa kluczowego **`package`**.
- Przykład dostępu domyślnego – struktura pakietów klas:

```
programowanie
│
│   ObcaKlasa.java
│
└──przykladdomyslny
        Komunikaty.java
        WtajemniczonaKlasa.java
```

treść klasy `Komunikaty`:

```
package przykladdomyslny;

public class Komunikaty {
  private static void tajnyKomunikat() {
    System.out.println("Cii! Tajny komunikat!");
  }

  static void komunikatDlaWtajemniczonych() {
    System.out.println("Komunikat dla wtajemniczonych!");
  }

  public static void komunikatOgolny() {
    System.out.println("Bedzie padac.");
  }
}
```

- Klasa `Komunikaty` zawiera metodą `komunikatDlaWtajemniczonych`, która ma dostęp domyślny, ponieważ nie ma zdefiniowanego żadnego modyfikatora dostępu.
- Z prywatnej metody `tajnyKomunikat` możemy korzytać jedynie we wnętrzu klasy `Komunikaty`.
- Z metody `komunikatOgolny` mogą korzystać wszystkie klasy, niezależnie od tego, w jakim pakiecie się znajdują.
- Metoda o dostępie domyślnym, `komunikatDlaWtajemniczonych`, może zostać użyta tylko w klasie `WtajemniczonaKlasa`, ponieważ obie klasy są w tym samym pakiecie. Z kolei klasa `ObcaKlasa`, będą poza pakietem `przykladdomyslny`, nie ma prawa z tej metody korzystać.

- Dostęp domyślny przydaje się, gdy projektujemy klasy, które będą wspólnie działały na rzecz wykonania pewnego zadania. Dzięki dostępowi domyślnemu, możemy pozwolić klasom z tego samego pakietu na dostęp do przydatnych metod i pól z innych klas z tego pakietu, nie dającym tym samym tej możliwości jakimkolwiek innym klasom, zdefiniowanym w innych pakietach. Tylko "wtajemniczone" klasy z tego samego pakietu będą mogły odnosić się do tych pól i metod, które dla świata zewnętrznego powinny pozostać prywatne i niedostępne.
- Modyfikator **`protected`** tak samo, jak dostęp domyślny, pozwala na dostęp do pól i metod klasom, które zdefiniowane są w tym samym pakiecie – jest to cecha wspólna modyfikatora **`protected`** oraz dostępu domyślnego. Inne cechy modyfikatora **`protected`** poznamy w rozdziale o dziedziczeniu.
- Klasy nie muszą być publiczne – możemy pominąć modyfikator **`public`**. W takim przypadku, gdy brak jest modyfikatora dostępu, klasa będzie miała domyślny dostęp – jego znaczenie będzie takie samo, jak w przypadku pól i metod, które mają domyślny dostęp.
- Klasy z dostępem domyślnym mogą być wykorzystywane jedynie przez klasy znajdujące się w tym samym pakiecie – wszystkie klasy znajdujące się w innych pakietach (bądź w pakiecie domyślnym, gdy nie podamy żadnego pakietu), nie będą mogły z takiej klasy korzystać.

9.10.5 Pytania

1. Jaka jest pełna nazwa poniższej klasy?

```
package com.kursjava;

public class TestowaKlasa {
  public static void main(String[] args) {
    wypiszKomunikat();
  }

  public static void wypiszKomunikat() {
    System.out.println("Witam.");
  }
}
```

2. Jaką komendą należy skompilować, a jaką uruchomić, klasę z powyższego zadania?
3. Mając poniższą strukturę katalogów:

```
programy
│
└──com
   └───kursjava
           TestowaKlasa.java
```

Czy poniższa próba uruchomienia klasy `TestowaKlasa` się powiedzie?

```
C:\programy\com\kursjava> java com.kursjava.TestowaKlasa
```

4. Czy poniższa próba kompilacji klasy `TestowaKlasa` powiedzie się?

```
javac com.kursjava.TestowaKlasa.java
```

5. Czy poniższa klasa skompiluje się bez błędów?

```
import com.*;

public class WykorzystanieTestowejKlasy {
  public static void main(String[] args) {
    TestowaKlasa.wypiszKomunikat();
  }
}
```

6. Czy poniższa klasa skompiluje się i wykona się bez błędów?

```
public class WykorzystanieTestowejKlasy {
  public static void main(String[] args) {
    com.kursjava.TestowaKlasa.wypiszKomunikat();
  }
}
```

7. Czy poniższa klasa skompiluje się bez błędów?

```
import java.util.Scanner;

package pakiet;

public class PytanieImport {
  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

8. Czy poniższy kod skompiluje się i wykona bez błędów?

```
public class TestowaKlasa {
  public static void main(String[] args) {
    System.out.println("Pi wynosi: " + Math.PI);
  }
}
```

9. Co zobaczymy na ekranie w wyniku uruchomienia poniższego programu? Czy kod w ogóle się skompiluje?

```
import static java.lang.Math.PI;

public class TestowaKlasa {
  private static int PI = 3;

  public static void main(String[] args) {
    System.out.println("Pi wynosi: " + PI);
  }
}
```

10. Czym jest classpath? Jak ustawić wartość classpath?
11. Dlaczego w naszych programach nie musimy importować typu `String` z Biblioteki Standardowej Java?
12. Czym różni się *dostęp domyślny* od dostępów definiowanych za pomoc modyfikatorów **`private`** oraz **`public`**?
13. Jak zdefiniować, że pole bądź metoda klasy ma mieć *dostęp domyślny*?
14. Czy klasy mogą być niepubliczne? Jeżeli tak, to czym takie klasy różnią się od klas publicznych?
15. Do których pól i metod klasy `A` oraz klasy `B` mają klasy `C` i `D`, oraz dlaczego?

```
programy
│
└──com
   │   D.java
   │
   └───kursjava
           A.java
           B.java
           C.java
```

Klasy `A` oraz `B` zdefiniowane są następująco:

```java
package com.kursjava;

public class A {
  private static int x;
  public static int y;
  static int z;

  void pewnaMetoda() {
    System.out.println("Bedzie padac.");
  }
}
```

```java
package com.kursjava;

class B {
  public static int n;
  static int m;

  public void innaMetoda() {
    System.out.println("Witam");
  }
}
```

10 Rozdział X – Dziedziczenie i polimorfizm – rozdział w przygotowaniu

Rozdział w przygotowaniu.

11 Rozdział XI – Wyjątki

Ten rozdział dotyczy obsługi sytuacji wyjątkowych w programach napisanych w języku Java.

W tym rozdziale:

- dowiemy się, czym są wyjątki i kiedy je stosować,
- zobaczymy, jak *łapać* wyjątki i obsługiwać je za pomocą `try`..`catch`..`finally`,
- nauczymy się jak tworzyć i wykorzystywać własne klasy wyjątków,
- poznamy dwa rodzaje wyjątków: Checked oraz Unchecked, a także zobaczymy, jak sprawdzić, do którego rodzaju należy konkretna klasa wyjątków,
- opowiemy sobie o wadach i zaletach wyjątków,
- dowiemy się, jak deklarować potencjał rzucenia wyjątków przez nasze metody,
- użyjemy mechanizmu *try-with-resources* wprowadzonego do Java w wersji 1.7 w przykładzie korzystającym z klas Java, które służą do czytania plików.

11.1 Czym są wyjątki?

W rozdziale o wartościach zwracanych przez metody korzystaliśmy z następującego przykładu:

Nazwa pliku: Rozdzial_07__Metody/WypiszWynikDzielenia.java

```
public class WypiszWynikDzielenia {
  public static void main(String[] args) {
    wypiszWynikDzielenia(10, 0);
    wypiszWynikDzielenia(25, 5);
  }

  public static void wypiszWynikDzielenia(int x, int y) {
    if (y == 0) {
      System.out.println("Nie mozna dzielic przez 0!");
      return;
    }

    System.out.println("Wynik dzielenia: " + (x / y));
  }
}
```

W metodzie `wypiszWynikDzielenia` wykonujemy sprawdzenie, czy liczba przekazana w argumencie `y` nie jest przypadkiem równa `0` – w takim przypadku nie możemy wykonać dzielenia.

Załóżmy, że nasza metoda ma zwracać wartość dzielenia, a nie ją wypisywać – zmodyfikujmy nasz przykład:

Nazwa pliku: ZwrocWynikDzielenia.java

```
public class ZwrocWynikDzielenia {
  public static void main(String[] args) {
    System.out.println(podziel(10, 0));
    System.out.println(podziel(25, 5));
  }

  public static int podziel(int x, int y) {
    return x / y;
  }
}
```

Metodę `wypiszWynikDzielenia` zastąpiliśmy metodą `podziel`, która zamiast wypisywać wynik na ekran, zwraca go. Jaki będzie wynik działania powyższego kodu? Na ekranie zobaczymy komunikat o błędzie dzielenia przez zero:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ZwrocWynikDzielenia.podziel(ZwrocWynikDzielenia.java:8)
        at ZwrocWynikDzielenia.main(ZwrocWynikDzielenia.java:3)
```

Ten błąd to właśnie *wyjątek*, który został spowodowany nieprawidłowym działaniem naszego programu – w tym przypadku, Maszyna Wirtualna Java poinformowała nas, że wystąpił błąd dzielenia przez zero, a ten konkretny wyjątek nazywa się `ArithmeticException`.

Wyjątki (*exceptions*) to sytuacje, w których coś w programie poszło nie tak. Gdy zajdzie taka sytuacja, mówimy, że został *rzucony* wyjątek. Wyjątki mogą być rzucane zarówno przez Maszynę Wirtualna Java, jak i przez nas – programistów – co zobaczymy w dalszej części rozdziału.

Bardzo ważną cechą wyjątków jest to, że są to tak naprawdę klasy – mają one

swoją nazwę, konstruktory, pola i metody. Co cechuje klasę, że może być traktowana jako wyjątek? Klasa taka musi rozszerzać klasę `Throwable` lub jedną z jej pochodnych, o czym wkrótce dokładniej sobie opowiemy. Rzucanie wyjątków sprowadza się do utworzenia słowem kluczowym **`new`** obiektu konkretnej klasy wyjątku i "rzucenie" go za pomocą słowa kluczowego **`throw`**. Zajmiemy się tymi zagadnieniami w kolejnych rozdziałach.

Z wyjątkami spotkaliśmy się już w poprzednich rozdziałach – widzieliśmy m. in. wyjątki:

- `StringIndexOutOfBoundsException` – gdy próbowaliśmy odnieść się do znaku w zmiennej typu `String` za pomocą metody `charAt` przekazując indeks znaku wychodzący poza zakres stringu,
- `ArrayIndexOutOfBoundsException` – gdy odnosiliśmy się do nieistniejącego elementu tablicy,
- `NullPointerException` – gdy próbowaliśmy odnosić się do pól bądź metod niezainicjalizowanego obiektu, tzn. gdy zmienna typu złożonego wskazywała na **`null`**.

Inne sytuacje, w których moglibyśmy natknąć się na wyjątek, to na przykład:

- podanie ujemnego wieku podczas tworzenia obiektu typu `Osoba`,
- próba otwarcia pliku, który nie istnieje,
- zerwanie połączenia z internetem podczas próby wysłania danych na serwer,
- podanie nieprawidłowego hasła podczas łączenia się do serwera baz danych,
- i wiele innych.

Sytuacje wyjątkowe możemy *obsługiwać* dzięki *mechanizmowi łapania wyjątków*, który poznamy w tym rozdziale.

11.1.1 Stack trace

Zauważmy w przykładzie powyżej, jak Maszyna Wirtualna Java prezentuje wyjątek:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ZwrocWynikDzielenia.podziel(ZwrocWynikDzielenia.java:8)
        at ZwrocWynikDzielenia.main(ZwrocWynikDzielenia.java:3)
```

Po rodzaju wyjątku i skojarzonym z nim komunikatem w pierwszej linii, następują informacje o ścieżce wykonania programu, która doprowadziła do występienia tego wyjątku.

Jest to tzw. *stack trace* – ścieżka wykonań metod, które doprowadziły do błędu. Często będziemy analizować stack trace'y programując w Javie.

Stack trace powinno się śledzić się od dołu, ponieważ prezentowana w nim kolejność metod jest odwrotna do kolejności wykonywania tych metod – ostatnia metoda (ta na dole stack trace'a) została wywołana jako pierwsza, a ta na samej górze – jako ostatnia – i to w niej rzucony został wyjątek.

W praktyce jednak patrzymy zazwyczaj na kilka pierwszych linii stack trace'a, bo zazwyczaj wystarczają nam one do zrozumienia dlaczego, a przynajmniej gdzie, wystąpił wyjątek.

W naszym przypadku najpierw wywołana została metoda `main` z klasy `ZwrocWynikDzielenia`:

```
at ZwrocWynikDzielenia.main(ZwrocWynikDzielenia.java:3)
```

W nawiasach mamy dodatkowo podany plik, z którego klasa pochodzi, a także linię kodu, w której

wykonanie metody przeszło do kolejnej metody – ta metoda, jak widzimy patrząc dalej na stack trace, to `podziel`:

```
at ZwrocWynikDzielenia.podziel(ZwrocWynikDzielenia.java:8)
```

Więcej metod nie zostało wywołanych – oznacza to, że wyjątek został rzucony właśnie w metodzie `podziel`. Dodatkowo mamy także podany numer 8 w nawiasach – to numer linii programu (a nie linii metody), w której wyjątek został rzucony. Są to bardzo przydatne informacje dla nas, programistów, podczas analizy błędów zaistniałych w naszych programach – dzięki stack trace'om łatwiej znaleźć miejsce, gdzie program zadziałał nieprawidłowo.

Jeżeli spojrzymy ponownie na kod naszej klasy:

Nazwa pliku: ZwrocWynikDzielenia.java

```
public class ZwrocWynikDzielenia {
  public static void main(String[] args) {
    System.out.println(podziel(10, 0));
    System.out.println(podziel(25, 5));
  }

  public static int podziel(int x, int y) {
    return x / y;
  }
}
```

to zobaczymy, że linia nr 3 umieszczona w nawiasach w stack trace odnosi się do:

```
System.out.println(podziel(10, 0));
```

a linia 8 do:

```
return x / y;
```

Oznaczenia linii w stack trace zgadzają się z wykonaniem programu, które doprowadziło do zaistnienia wyjątku `ArithmeticException`:

- program zostaje uruchomiony – rozpoczyna się wykonywanie metody `main`,
- w pierwszej linii metody `main` (zauważmy, że jest to jednocześnie trzecia linia całego programu) następuje wywołanie metody `podziel` z argumentami `10` i `0`,
- wykonanie programu przechodzi do metody `podziel`,
- z racji tego, że podaliśmy `0` jako argument `y`, Maszyna Wirtualna Java rzuca wyjątek `ArithmeticException`, gdy próbujemy wykonać dzielenie przez `0` – dzieje się to w ósmej linii programu,
- program kończy działanie, a na ekran zostaje wypisany zaistniały błąd: typ wyjątku, jego komunikat, oraz omówiony już stack trace.

11.2 Przykład obługi sytuacji wyjątkowej

Zastanówmy się teraz, jak powinien działać nasz program, aby obsłużyć sytuację, gdy przesłany zostanie nieprawidłowy argument. Czy program powinien:

- konczyć się błędem tak jak do tej pory?
- zwracać `0`?
- zwracać jakąś inną wartość, np. najmniejszą z możliwych wartości, jakie może przechowywać zmienna typu **`int`**?

Żadne z powyższych rozwiązań nie jest dobre – nie chcemy, aby program kończył się błędem, a zwracanie pewnej wartości na sztywno nie jest dobrym rozwiązaniem, bo przecież zarówno zero, jak i najmniejsza liczba, jaką może przechowywać typ **`int`**, mogą potencjalnie być wynikiem działania metody `podziel`.

Co zatem możemy poradzić na taką sytuację wyjątkową? Wiedząc już, czym są wyjątki, możemy *obsłużyć* wyjątek dzielenia przez zero, by nasz program nie kończył się błędem. Spójrzmy na zmodyfikowany przykład aby zobaczyć mechanizm łapania wyjątków w akcji:

Nazwa pliku: ZwrocWynikDzieleniaWyjatek.java

```
public class ZwrocWynikDzieleniaWyjatek {
  public static void main(String[] args) {
    try {
      System.out.println(podziel(10, 0));
    } catch (ArithmeticException e) {
      System.out.println("Nie wolno dzielic przez 0!");
    }
  }

  public static int podziel(int a, int b) {
    return a / b;
  }
}
```

Tym razem na ekranie zobaczymy:

```
Nie wolno dzielic przez 0!
```

W metodzie `main` użyliśmy mechanizmu **`try`**..**`catch`** do łapania i obsługiwania wyjątków. W naszym przypadku, gdy w metodzie `podziel` próbujemy wykonać dzielenie przez `0`, Maszyna Wirtualna Java rzuci wyjątek `ArithmeticException`, który następnie możemy obsłużyć dzięki skorzystaniu z **`try`**..**`catch`**.

Używając instrukcji **`try`** spodziewamy się, że w instrukcjach objętych przez **`try`** coś *może* pójść nie tak (ale nie musi). To, co powinno się zadziać w przypadku napotkania konkretnego problemu (i tylko wtedy), umieszczamy w sekcji **`catch`**:

- najpierw w nawiasach podajemy typ wyjątku, którego się spodziewamy i który chcemy obsłużyć – w naszym przypadku jest to błąd `ArithmeticException`; po typie następuje nazwa zmiennej, za pomocą której do tego obiektu-wyjątku będziemy się mogli odnieść – zazwyczaj, tak jak powyżej, ta zmienna nazywana jest po prostu `e`,
- następnie, w nawiasach klamrowych umieszczamy instrukcje, które chcemy wykonać wyłącznie w przypadku, gdy dany wyjątek zostanie rzucony – w powyższym przykładzie jest to tylko jedna instrukcja informująca, że nie można dzielić przez `0`.

11.3 Korzystanie z try..catch..finally

Mechanizm łapania wyjątków ma następującą składnię:

```
try {
  // instrukcje ktore moga potencjalnie zakonczyc sie wyjatkiem
} catch (TypWyjatku dowolnaNazwa) {
  // instrukcje, gdy zajdzie wyjątek TypWyjatku
} catch (KolejnyTypWyjatku dowolnaNazwa2) {
  // instrukcje, gdy zajdzie wyjątek KolejnyTypWyjatku
} finally {
  // instrukcje, ktore maja byc wykonane niezaleznie od tego, czy wyjatek
zostal zlapany, czy nie
}
```

Jak widzimy, możemy zdefiniować obsłużenie większej liczby wyjątków poprzez dopisywanie kolejnych bloków **catch** – może ich być dowolnie wiele. W nawiasach dla każdego z bloków **catch** podajemy, jaki typ wyjątku chcemy obsłużyć oraz nadajemy mu nazwę, ponieważ sam wyjątek jest obiektem (zmienną), którą możemy odpytać o informacje związane z zaistniałym błędem. Dla przykładu, możemy wyświetlić komunikat błędu:

fragment pliku ZwrocWynikDzieleniaWyjatek.java

```
try {
  System.out.println(podziel(10, 0));
} catch (ArithmeticException e) {
  System.out.println("Nie wolno dzielic przez 0!");
  System.out.println("Wystapil blad: " + e.getMessage());
}
```

W wyniku czego zobaczymy na ekranie:

```
Nie wolno dzielic przez 0!
Wystapil blad: / by zero
```

Fragment wypisanego tekstu *"/ by zero"* to opis błędu, który wystąpił. Opis ten przechowywany jest w obiekcie-wyjątku, który nazwaliśmy `e`. Metoda `getMessage` ten opis zwraca.

Mechanizm łapania wyjątków oferuje jeszcze jedną funkcjonalność – możemy w opcjonalnym bloku **finally** umieścić instrukcje, które mają *zawsze* się wykonać, niezależnie od tego, czy wyjątek zostanie złapany, czy nie. Blok **finally** zazwyczaj używany jest do czyszczenia zasobów, np. zamykania połączeń z bazą danych czy zamykania otwartych plików.

Istotne jest zrozumienie kolejności wykonywania instrukcji w blokach **try**..**catch**..**finally**:

1. Najpierw wywoływane są instrukcje w bloku **try**. Jeżeli któraś z instrukcji spowoduje rzucenie wyjątku, to:
   a. Wykonanie bloku **try** zostaje przerwane – wszystkie instrukcje następujące po instrukcji, która spowodowała wyjątek, nie zostaną wykonane.
   b. Typ rzuconego wyjątku jest dopasowywany do wyjątków zdefiniowanych w sekcjach **catch**. Zostają wykonane instrukcje przyporządkowane do pierwszej sekcji **catch**, do której rzucony wyjątek został dopasowany. Jeżeli wyjątek, który wystąpił, nie jest obsługiwany w żadnym z bloków **catch**, to przechodzimy do kroku 2.
2. Wywoływane są instrukcje z bloku **finally**, jeżeli blok ten jest obecny, niezależnie od tego, czy wyjątek wystąpił, czy nie.

Spójrzmy na dwa poniższe przykłady:

fragment pliku ZwrocWynikDzieleniaWyjatek.java

```
try {
  System.out.println("Zaraz podzielimy 10 przez 2");
  System.out.println("Wynik dzielenia: " + podziel(10, 2));
  System.out.println("Podzielilismy 10 przez 2");
} catch (ArithmeticException e) {
  System.out.println("Wystapil blad podczas dzielenia przez 2!");
} finally {
  System.out.println("Blok try..catch..finally zakonczony!");
}
```

Ten przykład spowoduje wypisanie na ekran:

```
Zaraz podzielimy 10 przez 2
Wynik dzielenia: 5
Podzielilismy 10 przez 2
Blok try..catch..finally zakonczony!
```

Jak widzimy, komunikat z sekcji `catch` nie został wypisany, ponieważ żaden wyjątek nie wystąpił.

Spójrzmy na drugi, ciekawszy przykład:

fragment pliku ZwrocWynikDzieleniaWyjatek.java

```
try {
  System.out.println("Zaraz podzielimy 10 przez ZERO!");
  System.out.println("Wynik dzielenia: " + podziel(10, 0));
  System.out.println("Podzielilismy 10 przez ZERO!"); // 1
} catch (ArithmeticException e) {
  System.out.println("Wystapil blad podczas dzielenia przez ZERO!"); // 2
} finally {
  System.out.println("Blok try..catch..finally zakonczony!");
}
```

Tym razem na ekranie zobaczymy:

```
Zaraz podzielimy 10 przez ZERO!
Wystapil blad podczas dzielenia przez ZERO!
Blok try..catch..finally zakonczony!
```

Tym razem nie została wykonana instrukcja `(1)`, która następowała po instrukcji dzielenia. Nie zobaczyliśmy na ekranie napisu `"Podzielilismy 10 przez ZERO!"`, ponieważ w momencie rzucenia wyjątku wykonanie programu przeszło od razu do obsługi wyjątku. Z racji tego, że wyjątek wystąpił, zobaczyliśmy napis wypisywany w obsłudze wyjątku `(2)`.

Zwróćmy uwagę, że niezależnie od tego, czy wyjątek wystąpił, czy nie, w obu przykładach wykonany został kod z części `finally`.

> Kod z bloku `finally` nie zostanie wykonany w szczególnym przypadku – gdy użyjemy metody `exit` z klasy `System`, ponieważ natychmiast kończy ona nasz program.

11.3.1 Zakres zmiennych definiowanych w bloku try

Wielokrotnie przy okazji omawiania instrukcji warunkowych, pętli, metod itp. widzieliśmy, że zmienne definiowane wewnątrz bloków kodu są niewidoczne poza tymi blokami, jeżeli nie wskazuje na nie żadna referencja spoza tego bloku. Jeżeli zdefiniujemy zmienną w bloku `try`, to po zakończeniu tego bloku przestanie ona istnieć – nie będziemy mieli do niej dostępu nawet w

sekcjach **catch** i **finally** – spójrzmy na przykład:

fragment metody main z pliku ZwrocWynikDzieleniaWyjatek.java

```
try {
  int wynik = podziel(10, 2);
} catch (ArithmeticException e) {
  System.out.println("Blad dzielenia, zmienna wynik ma wartosc: " + wynik);
} finally {
  System.out.println("Sekcja finally: wynik wynosi " + wynik);
}
```

Ten fragment kodu powoduje następujące błędy kompilacji:

```
ZwrocWynikDzieleniaWyjatek.java:36: error: cannot find symbol
  System.out.println("Blad dzielenia, zmienna wynik ma wartosc: " + wynik);
                                                                        ^
  symbol:   variable wynik
  location: class ZwrocWynikDzieleniaWyjatek
ZwrocWynikDzieleniaWyjatek.java:38: error: cannot find symbol
      System.out.println("Sekcja finally: wynik wynosi " + wynik);
                                                            ^
  symbol:   variable wynik
  location: class ZwrocWynikDzieleniaWyjatek
2 errors
```

W sekcjach **catch** i **finally** próbujemy odnieść się do nieistniejącej zmiennej – zmienna `wynik` istnieje jedynie w bloku **try**, bo w nim została zdefiniowana.

W praktyce często zachodzi potrzeba korzystania "na zewnątrz" sekcji **try** z tworzonych w niej zmiennych, czy też w sekcji **finally**. W takich przypadkach należy zdefiniować zmienną przed sekcją **try**:

fragment metody main z pliku ZwrocWynikDzieleniaWyjatek.java

```
int wynik = 0;

try {
  wynik = podziel(10, 2);
} catch (ArithmeticException e) {
  System.out.println("Blad dzielenia, zmienna wynik ma wartosc: " + wynik);
} finally {
  System.out.println("Sekcja finally: wynik wynosi " + wynik);
}

System.out.println("Po try wynik wynosi " + wynik);
```

Wynik działania:

```
Sekcja finally: wynik wynosi 5
Po try wynik wynosi 5
```

Tym razem kod skompilował i wykonał się bez problemów.

Zauważmy jeszcze jedną istotną cechę powyższego kodu – zmienna `wynik` została zainicjalizowana wartością `0`. Jeżeli byśmy tego nie zrobili, to kod ponownie by się nie skompilował. Kompilator zgłosiłby błąd `variable wynik might not have been initialized`. Powodem błędu byłby fakt, że metoda `podziel` może rzucić wyjątek i nie zwrócić żadnej wartości – w takim przypadku zmiennej `wynik` nie zostałaby przypisana żadna wartość, a jak wiemy z rozdziału o zmiennych – zmienne lokalne *muszą* być zainicjalizowane wartością przed

użyciem. Dlatego przed blokiem `try`, w linii, w której definiujemy zmienną `wynik`, nadajemy jej od razu wstępną wartość. Jeżeli korzystalibyśmy z zmiennej typu złożonego, to jako wartość domyślną moglibyśmy przypisać np. `null`.

11.3.2 Metoda getInt i obsługa wyjątków

W poprzednich rozdziałach wielokrotnie korzystaliśmy z metody `getInt`, która miała za zadanie zwrócić pobraną od użytkownika liczbę. Co się jednak działo w sytuacjach, gdy przez przypadek podaliśmy nieprawidłową liczbę?

```
import java.util.Scanner;

public class GetIntObslugaWyjatkow {
  public static void main(String[] args) {
    System.out.print("Podaj liczbe: ");
    int x = getInt();

    int kwadrat = x * x;
    System.out.println("Kwadrat tej liczby wynosi " + kwadrat);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Jeżeli po uruchomieniu powyższego programu podamy np. 'x' jako liczbę, to program zakończy się następującym błędem:

```
Podaj liczbe: x
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at GetIntObslugaWyjatkow.getInt(GetIntObslugaWyjatkow.java:13)
        at GetIntObslugaWyjatkow.main(GetIntObslugaWyjatkow.java:6)
```

Teraz już wiemy, że ten błąd to rzucony wyjątek. W tym przypadku, nazywa się on `InputMismatchException`, które zdefiniowany jest w bibliotece standardowej w pakiecie `java.util`.

Wiedząc już, jak obsługiwać wyjątki, możemy zmodyfikować nasz program, by brał pod uwagę możliwość podania przez użytkownika nieprawidłowej liczby. Jak jednak powinniśmy taką sytuację obsłużyć?

Możemy w pętli próbować pobrać od użytkownika liczbę – jeżeli wyjątek zostanie rzucony, poinformujemy użytkownika o nieprawidłowo podanej wartości i spróbujemy pobrać ją kolejny raz w następnym obiegu pętli. Gdy użytkownik poda poprawną liczbę, zmienimy zmienną warunkującą wykonanie pętli, by pętla już więcej się nie wykonywała.

Spójrzmy, jak mogłaby wyglądać implementacja:

Nazwa pliku: GetIntObslugaWyjatkow.java

```
import java.util.InputMismatchException; // 1
import java.util.Scanner;

public class GetIntObslugaWyjatkow {
  public static void main(String[] args) {
    boolean wartoscPodana = false; // 2
    int x = 0;

    while (!wartoscPodana) { // 3
      try {
        System.out.print("Podaj liczbe: ");
        x = getInt(); // 4

        wartoscPodana = true; // 5
      } catch (InputMismatchException e) { // 6
        System.out.println("To nie jest liczba!");
      }
    }

    int kwadrat = x * x;
    System.out.println("Kwadrat tej liczby wynosi " + kwadrat);
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

Najpierw dodajemy importowanie typu wyjątku, który będziemy łapać *(1)*. Wyjątek ten nie znajduje się w pakiecie `java.lang`, lecz `java.util`, dlatego musimy sami dodać jego import do naszego programu.

Następnie definiujemy w metodzie `main` zmienne `wartoscPodana` i `x` *(2)*. Pierwsza będzie wskazywała, czy użytkownik podał już poprawną wartość, czy jeszcze nie. Druga zmienna przechowuje wartość od użytkownika.

W pętli **`while`** *(3)* wykonujemy próbę pobrania liczby od użytkownika *(4)* tak długo, aż nie poda on poprawnej liczby. Jeżeli użytkownik poda poprawną liczbę, to w linii *(5)* ustawimy `wartoscPodana` na **`true`**, dzięki czemu pętla nie wykona się więcej razy. Jeżeli jednak użytkownik poda np. literę zamiast liczby, to wywołanie metody `getInt` zakończy się rzuceniem wyjątku `InputMismatchException`, który obsługujemy w linii *(6)*. W tym przypadku informujemy użytkownika, że podał nieprawidłową liczbę. Instrukcja **`try`**..**`catch`** się kończy i przechodzimy do kolejnego obiegu pętli – w zależności od tego, jaką wartość podał użytkownik, wykona się ona ponownie lub zakończy działanie.

Na końcu programu liczymy kwadrat pobranej liczby i wypisujemy wynik.

Przykładowe wykonanie tego programu:

```
Podaj liczbe: x
To nie jest liczba!
Podaj liczbe: y
To nie jest liczba!
Podaj liczbe: 5
Kwadrat tej liczby wynosi 25
```

11.4 Definiowanie i rzucanie wyjątków

W poprzednich przykładach, wyjątek `ArithmeticException` rzucany był przez Maszynę Wirtualną Java, jednak nie jest to jedyna możliwość rzucania wyjątków – my, jako programiści, możemy sami rzucać wyjątki z naszych metod.

Rzucanie wyjątków odbywa się poprzez użycie słowa kluczowego **`throw`**, po którym następuje tworzenie obiektu wyjątku takiego typu, jaki chcemy rzucić. Spójrzmy na przykład obsługi sytuacji, gdy ktoś poda ujemny wiek podczas tworzenia obiektu typu `Osoba`:

Nazwa pliku: Osoba.java

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private int wiek;

  public Osoba(String imie, String nazwisko, int wiek) {
    this.imie = imie;
    this.nazwisko = nazwisko;

    if (wiek <= 0) {
      throw new IllegalArgumentException("Wiek nie moze byc ujemny."); // 1
    }

    this.wiek = wiek;
  }

  public static void main(String[] args) {
    try {
      Osoba o = new Osoba("Jan", "Nowak", -1); // 2
    } catch (IllegalArgumentException e) { // 3
      System.out.println("Wystapil blad! " + e.getMessage());
    }
  }
}
```

Wynik działania tego programu to:

```
Wystapil blad! Wiek nie moze byc ujemny.
```

W konstruktorze klasy `Osoba` sprawadzamy, czy wiek jest niepoprawny – jeżeli tak *(1)*, to rzucamy wyjątek za pomocą składni:

```
throw new IllegalArgumentException("Wiek nie moze byc ujemny.");
```

Zauważmy, że po słowie kluczowym **`throw`** umieszczamy wyjątek, jaki ma zostać rzucony – w tym przypadku tworzymy nowy wyjątek typu `IllegalArgumentException`, zdefiniowany w bibliotece standardowej Java. Jako parametr konstruktora możemy podać opcjonalny komunikat błędu.

Rzucać możemy wyjątki dowolnego typu – zarówno te zdefiniowane już w bibliotece standardowej Java jak i zdefiniowane przez nas. Wyjątek `IllegalArgumentException` to często stosowany wyjątek mający na celu wskazanie, że pewne dane są nieprawidłowe, tak jak w powyższym przypadku. Wiek nie może być ujemny, a taką wartość przekazujemy do konstruktora klasy `Osoba` *(2)*. Wyjątek łapiemy w sekcji **`catch`** *(3)* i obsługujemy, wypisując na ekran komunikat.

Wyjątek to nic innego jak obiekt konkretnej klasy – tej, której wyjątek chcemy zasygnalizować. Tworzymy go tak jak wszystkie obiekty do tej pory – za pomocą słowa kluczowego **`new`**:

```
throw new IllegalArgumentException("Dzielnik nie moze byc rowny 0.");
```

Równie dobrze moglibyśmy powyższy kod zapisać jako:

```
IllegalArgumentException exc =
    new IllegalArgumentException("Dzielnik nie moze byc rowny 0.");

throw exc;
```

Stosujemy jednak to pierwsze podejście, ponieważ jest krótsze.

11.4.1 Definiowanie własnych wyjątków

Możemy zdefiniować na własne potrzeby nowe typy wyjątków, ale najpierw wyjaśnijmy, czym w ogóle są wyjątki?

Wyjątki to pochodne klasy `Throwable` lub jednej z jej klas pochodnych, np. `Exception` lub `RuntimeException`. Klasy te są zdefiniowane w bibliotece standardowej Java. To właśnie jedna bądź druga z tych klas pochodnych jest używana jako klasa bazowa dla wyjątków definiowanych przez programistów. Kiedy stosować każdą z nich zobaczymy w jednym z kolejnych rozdziałów, a w tym skupimy się na dziedziczeniu po klasie `Exception`.

Gdy definiujemy w blokach **`catch`** wyjątki do obsłużenia, muszą być one pochodnymi klasy `Exception` lub `RuntimeException` (pośrednio bądź bezpośrednio) – inaczej kod się sie skompiluje. Spójrzmy na najprostszy przykład zdefiniowania własnego wyjątku:

Nazwa pliku: NieprawidlowyWiekException.java

```
class NieprawidlowyWiekException extends Exception {

}
```

Zdefiniowaliśmy tutaj nową klasę wyjątków – `NieprawidlowyWiekException` – od teraz możemy łapać wyjątki tego typu w blokach **`catch`**. Aby wskazać, że nasz wyjątek dziedziczy po (rozszerza) klasę `Exception`, użyliśmy słowa kluczowego **`extends`**, poznanego w poprzednim rozdziale.

> Zgodnie z konwencją nazewniczą klas wyjątków, na końcu nazwy wyjątku dodaliśmy słowo `Exception`.

Możemy także w prosty sposób umożliwić zapisywanie w wyjątku naszego typu komunikatu błędu. Utwórzmy jeszcze jedną klasę wyjątków:

Nazwa pliku: NieprawidlowaWartoscException.java

```
public class NieprawidlowaWartoscException extends Exception {
  public NieprawidlowaWartoscException(String message) {
    // wywolaj konstruktor z klasy bazowej (czyli z Exception)
    super(message);
  }
}
```

W tej klasie zdefiniowaliśmy konstruktor, który przyjmuje jako argument komunikat błędu – przesyłamy go do konstruktora klasy bazowej za pomocą słowa kluczowego **`super`** z poprzedniego rozdziału. Konstruktor z klasy bazowej, `Exception`, zapisze komunikat w polu, które będzie dostępne za pomocą metody `getMessage`. `getMessage` to metoda, którą nasza klasa wyjątku dziedziczy po klasie bazowej.

Spróbujmy dodać do konstruktora klasy `Osoba` walidację pól `imie` oraz `wiek` – sprawdzimy, czy mają wartość `null` – jeżeli tak, to rzucimy wyjątek `NieprawidlowaWartoscException` z odpowiednim komunikatem. Użyjemy także wyjątku `NieprawidlowyWiekException`:

Nazwa pliku: Osoba.java

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private int wiek;

  public Osoba(String imie, String nazwisko, int wiek) {
    if (imie == null) { // 1
      throw new NieprawidlowaWartoscException(
          "Imie nie moze byc puste."
      );
    }

    if (nazwisko == null) { // 2
      throw new NieprawidlowaWartoscException(
          "Nazwisko nie moze byc puste."
      );
    }

    if (wiek <= 0) {
      throw new NieprawidlowyWiekException();
    }

    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wiek = wiek;
  }

  public static void main(String[] args) {
    try {
      Osoba o = new Osoba("Jan", "Nowak", -1);
    } catch (IllegalArgumentException e) {
      System.out.println("Wystapil blad! " + e.getMessage());
    }
  }
}
```

Dodaliśmy do konstruktora sprawdzanie wartości `imie` *(1)* oraz `nazwisko` *(2)*. Jednakże podczas próby kompilacji klasy kompilator zaczyna zgłaszać problemy:

```
Osoba.java:8: error: unreported exception NieprawidlowaWartoscException;
must be caught or declared to be thrown
      throw new NieprawidlowaWartoscException(
      ^
Osoba.java:14: error: unreported exception NieprawidlowaWartoscException;
must be caught or declared to be thrown
      throw new NieprawidlowaWartoscException(
      ^
Osoba.java:20: error: unreported exception NieprawidlowyWiekException; must
be caught or declared to be thrown
      throw new NieprawidlowyWiekException();
      ^
3 errors
```

Co się stało? Używaliśmy już wcześniej wyjątków i nie napotkaliśmy takiego błędu. Otóż

w naszym kodzie brakuje jeszcze jednego elementu.

Wyjątki dzielą się na dwa rodzaje – *Checked* oraz *Unchecked Exceptions*. O różnicy opowiemy sobie w kolejnym rozdziale. To, co musimy teraz wiedzieć, to to, że w przeciwieństwie do wyjątków Unchecked, takich jak `IllegalArgumentException` łapanych wcześniej w tym rozdziale, potencjał rzucenia wyjątków rodzaju Checked musi zostać zdefiniowany w sygnaturze metody, która może go rzucić. Służy do tego słowo kluczowe **`throws`**.

W kolejnym rozdziale dokładnie sobie omówimy wyjątki Checked oraz Unchecked. Dla dociekliwych – wyjątki Unchecked to wyjątki, które dziedziczą po klasie `Runtime Exception`. *Pozostałe wyjątki to wyjątki Checked.*

Spójrzmy na sygnaturę poprawionego konstruktora:

```
public Osoba(String imie, String nazwisko, int wiek)
    throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
```

Po nawiasie zamykającym definicję argumentów konstruktora, a przed klamrą `{` otwierająca ciało metody, napisaliśmy słowo kluczowe **`throws`**, po którym wypisaliśmy, rozdzielone przecinkami, nazwy typów wyjątków, które nasza metoda może rzucić. Jest to wymagane, gdy istnieje potencjał rzucenia wyjątku rodzaju Checked – kompilator Java nie pozwoli skompilować kodu, jeżeli zabraknie klauzuli **`throws`**, gdy kompilator zauważy, że metoda może rzucić wyjątek/wyjątki.

Kompilator wie, że wyjątki mogą być rzucone, bo analizując podczas kompilacji kod konstruktora klasy `Osoba` widzi użycie słowa kluczowego **`throw`** do rzucenia wyjątków.

Nie pomyl słów kluczowych **`throw`** i **`throws`** – pierwsze rzuca wyjątek, a drugie służy do zdefiniowania, jakie wyjątki metoda może rzucić.

Czy teraz kod klasy `Osoba` się skompiluje? Jeszcze nie – zobaczymy następujący komunikat kompilatora:

```
Osoba.java:31: error: unreported exception NieprawidlowaWartoscException;
must be caught or declared to be thrown
      Osoba o = new Osoba("Jan", "Nowak", -1);
                ^
1 error
```

Tym razem kompilator wiedząc, że konstruktor klasy `Osoba` może rzucić wyjątki, oczekuje od nas, że korzystając z tego konstruktora weźmiemy te potencjalne wyjątki pod uwagę – innymi słowy, kompilator oczekuje użycia **`try..catch`** i obsługi wyjątków `NieprawidlowaWartoscException` oraz `NieprawidlowyWiekException` (chociaż w powyższym komunikacie widnieje nazwa tylko jednego z nich, to po dodaniu jego obsługi w **`catch`** kompilator przy kolejnej próbie kompilacji wskazałby, że wyjątek `NieprawidlowyWiekException` także należy obsłużyć).

Ponownie widzimy tutaj różnicę między wyjątkami Checked i Unchecked. Wcześniej w rozdziale, gdy metoda `podziel` *mogła rzucić wy jątek, nie musieliśmy tego wyjątku obsługiwać, bo* `IllegalArgumentException` *to wyjątek rodzaju Unchecked. W konstruktorze klasy* `Osoba` *korzystamy natomiast z wyjątków rodzaju Checked, które muszą zostać obsłużone – stąd powyższy błąd kompilacji.*

Spójrzmy na kompletny przykład wraz z dodaną obsługą wyjątków za pomocą **`try..catch`**:

Nazwa pliku: Osoba.java

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private int wiek;

  public Osoba(String imie, String nazwisko, int wiek)
      throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
    if (imie == null) {
      throw new NieprawidlowaWartoscException(
          "Imie nie moze byc puste."
      );
    }

    if (nazwisko == null) {
      throw new NieprawidlowaWartoscException(
          "Nazwisko nie moze byc puste."
      );
    }

    if (wiek <= 0) {
      throw new NieprawidlowyWiekException();
    }

    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wiek = wiek;
  }

  public static void main(String[] args) {
    try {
      Osoba o = new Osoba("Jan", "Nowak", -1);
    } catch (NieprawidlowaWartoscException e) { // 1
      System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
    } catch (NieprawidlowyWiekException e) { // 2
      System.out.println("Nieprawidlowy wiek!");
    }

    try {
      Osoba o = new Osoba(null, "Nowak", 30);
    } catch (NieprawidlowaWartoscException e) { // 1
      System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
    } catch (NieprawidlowyWiekException e) { // 2
      System.out.println("Nieprawidlowy wiek!");
    }
  }
}
```

Powyższy program używa dwóch nowych typów wyjątków, które rzucane są w konstruktorze klasy `Osoba`. Wyjątki te obsługiwane są następnie w ciele metody `main` *(1)* i *(2)*.

W przypadku obsługi wyjątku typu `NieprawidlowaWartoscException`, do wypisywanego na ekran komunikatu dodajemy treść błędu, która zawarta jest w wyjątku – umieściliśmy ją tam rzucając wyjątek w konstruktorze klasy `Osoba`. Wiadomość ta zawiera informację, która wartość jest nieprawidłowa. Ta wiadomość zwracana jest przez metodę `getMessage`.

11.4.2 Przerywanie wykonania bloku kodu przez wyjątki

Chociaż widzieliśmy w poprzednim rozdziale, jak rzucanie wyjątku wpływa na wykonanie bloku

kodu, w którym wyjątek wystąpił, to warto jeszcze raz omówić to zagadnienie.

W momencie rzucenia wyjątku przerywany jest aktualnie wykonywany blok kodu. W przypadku konstruktora klasy `Osoba`, gdy okaże się, że przesłane imię jest nullem, wykonanie konstruktora klasy `Osoba` natychmiast się kończy – kolejne pola nie będą już sprawdzane – konstruktor kończy działanie. Nasz program kontynuuje wykonanie od sekcji **`catch`**, która odpowiedzialna jest za obsłużenie tego konkretnego wyjątku.

Oznacza to, że jeżeli przekazalibyśmy do konstruktora klasy `Osoba` wartość **`null`** dla imienia, **`null`** dla nazwiska, oraz `-1` dla wieku, to rzucony zostałby tylko *jeden* wyjątek – ten, który zasygnalizowałby nieprawidłowe imię, ponieważ to to pole sprawdzamy jako pierwsze:

konstruktor klasy Osoba z pliku Osoba.java

```
public Osoba(String imie, String nazwisko, int wiek)
    throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
  if (imie == null) {
    throw new NieprawidlowaWartoscException( // 1
        "Imie nie moze byc puste."
    );
  }

  if (nazwisko == null) {
    throw new NieprawidlowaWartoscException( // 2
        "Nazwisko nie moze byc puste."
    );
  }

  if (wiek <= 0) {
    throw new NieprawidlowyWiekException(); // 3
  }

  this.imie = imie; // 4
  this.nazwisko = nazwisko;
  this.wiek = wiek;
}
```

Poniższe wywołanie konstruktora:

```
Osoba o = new Osoba(null, null, -1);
```

Spowoduje, że wykona się tylko fragment konstruktora do linii oznaczonej jako `(1)` – od tej linii wykonanie dalszej części ciała konstruktora zostanie przerwane, ponieważ rzucony zostanie wyjątek.

Następujące wywołanie:

```
Osoba o = new Osoba("Jan", null, -1);
```

Spowoduje, że wykona się tylko fragment konstruktora do linii oznaczonej jako `(2)`.

Idąc dalej tym tropem:

```
Osoba o = new Osoba("Jan", "Nowak", -1);
```

W tym przypadku, konstruktor zostanie przerwany w linii `(3)`.

W żadnym z powyższych przypadków nigdy nie dojdzie do wykonania kodu zaczynającego się od linii `(4)`, a co ważniejsze, obiekt typu `Osoba` nie zostanie utworzony.

Dopiero poniższe wywołanie konstruktora, które zawiera poprawne wartości dla imienia, nazwiska, oraz wieku, spowoduje wykonanie całego ciała konstruktora klasy `Osoba` oraz utworzenie i zwrócenie nowego obiektu klasy `Osoba`:

```
Osoba o = new Osoba("Jan", "Nowak", 30);
```

11.4.3 Rzucanie wyjątków i nieosiągalny kod

W związku z tym, że rzucenie wyjątku przerywa aktualnie wykonywany blok kodu, kompilator jest w stanie wykryć sytuacje, w których pewien fragment kodu nigdy nie miałby szansy się wykonać z powodu rzucania wyjątku.

Poniższy przykład w ogóle się nie kompiluje – kompilator zgłasza błąd, ponieważ instrukcja `System.out.println` nie ma szansy się wykonać – jest przed nią rzucany wyjątek `IllegalArgumentException`:

Nazwa pliku: NieosiagalnyKodRzucanyWyjatek.java

```
public class NieosiagalnyKodRzucanyWyjatek {
  public static void main(String[] args) {
    throw new IllegalArgumentException("Fajrant!");

    System.out.println("Witaj Swiecie?");
  }
}
```

Komunikat błędu:

```
NieosiagalnyKodRzucanyWyjatek.java:5: error: unreachable statement
    System.out.println("Witaj Swiecie?");
    ^
```

11.4.4 Rzucanie wyjątków a wartość zwracana z metody

Gdy poznawaliśmy metody, dowiedzieliśmy się, że metoda zawsze musi zwrócić wartość jeżeli definiuje typ zwracany inny niż **`void`**. W przeciwnym razie kompilacja naszego kodu zakończy się błędem:

Nazwa pliku: Rozdzial_07__Metody.BrakReturn.java

```
public class BrakReturn {
  public static void main(String[] args) {
    int liczbaDoKwadratu = kwadratLiczby(5);
  }

  public static int kwadratLiczby(int x) {
    int wynik = x * x;
    // ups! zapomnielismy zwrocic wynik!
  }
}
```

Ten kod kończy się błędem kompilacji `missing return statement` – zapomnieliśmy zwrócić wartość z metody `kwadratLiczby`, a musimy to zrobić – sygnatura metody wskazuje, że metoda ta zwraca wartość **`int`**.

W rozdziale o metodach wspomniałem o wyjątku od tej reguły – metoda nie musi zwrócić wartości, jeżeli rzuci wyjątek. Ma to taki sens, że skoro coś w metodzie poszło nie tak, skoro wystąpił jakiś błąd, to metoda zamiast zwrócić wartość może rzucić wyjątek:

Nazwa pliku: WyjatekZamiastReturn.java

```
public class WyjatekZamiastReturn {
  public static void main(String[] args) {
    try {
      System.out.println(podziel(10, 0));
    } catch (IllegalArgumentException e) {
      System.out.println("Wystpil wyjatek " + e.getMessage());
    }
  }

  public static int podziel(int x, int y) {
    if (y == 0) {
      throw new IllegalArgumentException("Dzielnik nie moze byc rowny 0.");
    }

    return x / y;
  }
}
```

Ten kod kompiluje się bez błędów pomimo, że istnieje taka ścieżka wykonania metody `podziel`, w której nie zwróci ona wartości – jeżeli `y` będzie równe zero, to rzucony zostanie wyjątek `IllegalArgumentException`. Rzucenie wyjątku powoduje natychmiastowe zakończenie działania metody `podziel` bez zwracania jakiejkolwiek wartości z metody. Wykonanie programu powraca do metody `main`, gdzie działanie kontynuowane jest w sekcji **`catch`**, w której obsługujemy złapany wyjątek. Metoda `podziel` nie zwraca co prawda wartości, ale rzuca wyjątek, co stanowi wyjątek od reguły, że metoda zawsze musi zwracać wartość, jeżeli definiuje zwracany typ inny niż **`void`**.

Zauważ, że wcześniej w tym rozdziale mówiłem, że jeżeli metoda rzuca wyjątek, to musimy go zdefiniować dodając po nazwie i argumentach metody słowo kluczwe **`throws`** oraz nazwy wyjątków, które metoda może rzucić. W powyższym przykładzie jednak nie ma **`throws`**, a kod działa – jak już wspomniałem w tym rozdziale, istnieją dwa rodzaje wyjątków – takie, które trzeba definiować za pomocą **`throws`** i te, których nie trzeba – porozmawiamy o tym w jednym z kolejnych rozdziałów.

11.4.5 Rzucanie wyjątków w try, catch, i finally

Wyjątki możemy rzucać w dowolnych blokach kodu – także w sekcji **`try`**, **`catch`**, a także **`finally`**.

Dla przykładu, moglibyśmy pobrać od użytkownika wiek osoby do utworzenia. Jeżeli ten wiek już w momencie pobrania od użytkownika jest nieprawidłowy (ujemny), to możemy rzucić w sekcji **`try`** wyjątek, by przejść natychmiast do sekcji **`catch`** obsługującej taką sytuację:

Nazwa pliku PobierzWiekOdUzytkownika.java

```
import java.util.Scanner;

public class PobierzWiekOdUzytkownika {
  public static void main(String[] args) {
    try {
      System.out.print("Podaj wiek osoby: ");
      int wiek = getInt(); // 1

      if (wiek <= 0) {
        throw new NieprawidlowyWiekException(); // 2
      }

      Osoba osoba = new Osoba("Jan", "Nowak", wiek); // 3

      System.out.println("Obiekt utworzony!");
    } catch (NieprawidlowaWartoscException e) {
      System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
    } catch (NieprawidlowyWiekException e) { // 4
      System.out.println("Nieprawidlowy wiek!");
    }
  }

  public static int getInt() {
    return new Scanner(System.in).nextInt();
  }
}
```

W tym przykładzie korzystamy z metody `getInt`, której używamy już od kilku rodziałów. W metodzie `main` pobieramy od użytkownika wiek osoby do utworzenia `(1)`.

Zanim w ogóle przystąpimy do tworzenia obiektu typu `Osoba` w linii `(3)`, najpierw sprawdzamy pobraną od użytkownika liczbę. Jeżeli jest nieprawidłowym wiekiem, to w ogóle nie będziemy próbowali utworzyć obiektu typu `Osoba`. Zamiast tego, od razu rzucimy wyjątek `NieprawidlowyWiekException` `(2)`, a wykonanie programu przejdzie do sekcji **`catch`**, która ten wyjątek obsługuje `(4)`.

Przykład wykonania z podaniem poprawnego i niepoprawnego wieku:

```
Podaj wiek osoby: 35
Obiekt utworzony!
```

```
Podaj wiek osoby: -5
Nieprawidlowy wiek!
```

Wyjątki mogą być też rzucane w sekcji **`catch`**. Czasem chcemy wykonać pewną akcję związaną z obsługą wyjątku, a potem *rzucić go ponownie*, co zobaczymy w kolejnym rozdziale.

Wyjątki mogą równie dobrze być rzucone w sekcji **`finally`** – jeżeli wykonujemy w nich kod bądź wywołujemy metody, które potencjalnie mogą zakończyć się wyjątkiem, to musimy mieć to na uwadze, opakowując taki kod ponownie w **`try`**..**`catch`** lub sygnalizując metodom, które z naszego kodu korzystają, że to one powinny się tymi potencjalnymi wyjątkami zająć – o tym zagadnieniu także porozmawiamy w jednym z kolejnych rozdziałów.

11.4.6 Ponowne rzucanie wyjątku

Złapanie wyjątku nie musi oznaczać zakończenia obsługi sytuacji wyjątkowych – złapany wyjątek można rzucić ponownie (*exception rethrow*), by został obsłużony przez metodę "wyżej" (tzn. jedną z wcześniejszych metod, które doprowadziły do wykonania metody, w której wyjątek był obsługiwany).

Jaki jest sens takiego działania? Możemy chcieć obsłużyć pewien wyjątek w dany sposób, a także dać szansę na jego obsłużenie w jednej z wcześniejszych metod.

Dla przykładu – wywołując metodę, która rzuca wyjątek, łapiemy go i zapisujemy do pliku logu informację, że wystąpił błąd. Moglibyśmy też umieścić w obiekcie wyjątku dodatkowe informacje o okolicznościach błędu. Następnie rzucamy ten wyjątek "dalej", by został obsłużony ponownie, potencjalnie już bez ponownego rzucania.

Do ponownego rzucania wyjątku stosuje się po prostu słowo kluczowe **`throw`**, po którym następuje wyjątek, który chcemy "rzucić dalej".

```
public static void glownaMetoda() {
  try {
    pewnaMetoda();
  } catch (PewienWyjatek e) {
    // ponowna obsluga wyjatku PewienWyjatek
  }
}

public static void pewnaMetoda() throws PewienWyjatek {
  try {
    // ...
    // instrukcje ktora moga spowodowac PewienWyjatek
    // ...
  } catch (PewienWyjatek e) {
    // zapisz informacje o bledzie do pliku logu
    log.error("Wystapil blad " + e.getMessage());

    // rzuc wyjatek dalej
    throw e;
  }
}
```

W tym przykładzie `glownaMetoda` wywołuje metodę `pewnaMetoda` i spodziewa się potencjalnego wyjątku `PewienWyjatek`, ponieważ `pewnaMetoda` deklaruje za pomocą **`throws`**, że taki wyjątek może rzucić. Dlatego też `glownaMetoda` zawiera **`try`**..**`catch`** i obsługę `PewienWyjatek`.

`pewnaMetoda` wykonuje pewne instrukcje, które mogą skutkować rzuceniem wyjątku `PewienWyjatek`. Obsługujemy go w sekcji **`catch`**, po czym rzucamy go "dalej" – teraz obsłuży go metoda `glownaMetoda`.

11.4.7 Treść wyjątku, stack trace i inne pola i metody

Wszystkie klasy w tym rozdziale znajdują się w jednym pliku: `ZawartoscWyjatkowPrzyklady.java`.

Wyjątki to klasy jak wszystkie inne – mogą mieć własne konstruktory, metody, i pola. Ich cechą specjalną jest to, że rozszerzają klasę `Exception`.

Najprostszym wyjątkiem jest klasa, która nie definiuje żadnych pól ani metod (pamiętajmy, że klasa

ta otrzyma automatycznie domyślny konstruktor):

```
class WyjatekBezTresciException extends Exception {}
```

Ta klasa jest już gotowa do użycia – możemy rzucać wyjątki tego typu za pomocą `throw` i łapać za pomocą `catch`. Mogłoby się wydawać, że taka klasa nie jest specjalnie przydatna, ale jest całkiem odwrotnie – dobrze nazwana klasa wyjątku bez żadnej dodatkowej treści może tak samo spełniać swoje zadanie, jak klasy wyjątków ze szczegółowymi komunikatami o błędzie. W poprzednim rozdziale widzieliśmy taki właśnie przypadek – klasa `NieprawidlowyWiekException` była pusta – sama jej nazwa i rzucenie takiego wyjątku daje nam wystarczającą informację, co i dlaczego się stało.

Klasy wyjątków dziedziczą po klasie `Exception` kilka metod. Jedną z nich jest `getMessage`, która zwraca treść (komunikat) wyjątku. Ta treść może być podana podczas rzucania wyjątku – musimy wtedy udostępnić w klasie naszego wyjątku konstruktor, który przyjmie tą wiadomość i przekaże ją do konstruktora klasy bazowej. Spójrzmy na kolejny przykład wyjątku, który pozwala na zapisanie w wyjątku komunikatu:

```
class WyjatekZKomunikatemException extends Exception {
  public WyjatekZKomunikatemException(String message) {
    // przekaz tresc wyjatku do konstruktora klasy bazowej,
    // ktory umiesci ja w polu message, ktore bedzie dostepne
    // za pomoca metody getMessage
    super(message);
  }
}
```

Przykładowe utworzenie wyjątku z komunikatem:

```
try {
  throw new WyjatekZKomunikatemException("Co tu sie wyprawia?!");
} catch (WyjatekZKomunikatemException e) {
  System.out.println("Wyjatek zawiera komunikat: " + e.getMessage());
}
```

Wynik:

```
Wyjatek zawiera komunikat: Co tu sie wyprawia?!
```

Jeżeli utworzymy wyjątek bez komunikatu, to `getMessage` zwróci `null` – zobaczymy to na przykładzie wyjątku `WyjatekBezTresciException`:

```
try {
  throw new WyjatekBezTresciException();
} catch (WyjatekBezTresciException e) {
  System.out.println("Wyjatek zawiera komunikat: " + e.getMessage());
}
```

Wynik:

```
Wyjatek zawiera komunikat: null
```

Czasem tworząc nowy typ wyjątku chcemy w nim mieć możliwość zapisać dodatkowe informacje, ktorą mogą pomóc podczas próby jego obsługi, lub by móc zapisać je do pliku logu w celu późniejszej analizy, co poszło nie tak. Nic nie stoi na przeszkodzie, aby klasa wyjątku definiowała nowe pola, które takie dane będą przechowywać:

```
class WyjatekZDodatkowymiDanymiException extends Exception {
  private int pewnaWartosc;
  private String innaWartosc;

  public WyjatekZDodatkowymiDanymiException(
      int pewnaWartosc, String innaWartosc) {
    this.pewnaWartosc = pewnaWartosc;
    this.innaWartosc = innaWartosc;
  }

  public String getMessage() {
    return "Wartosci zapisane w tym wyjatku: " +
        pewnaWartosc + " " + innaWartosc;
  }
}
```

Ta klasa wyjątku pozwala na skojarzenie z nim wartości `int` oraz `String` poprzez przekazanie ich do konstruktora. Te wartości możemy potem zobaczyć korzystając z przeładowanej metody `getMessage` (o przeładowaniu metod mówiliśmy w rozdziale o dziedziczeniu). Przykładowe użycie mogłoby wyglądać następująco:

```
try {
  int pewnaWartosc = 10;
  String innaWartosc = "test";

  throw new WyjatekZDodatkowymiDanymiException(pewnaWartosc, innaWartosc);
} catch (WyjatekZDodatkowymiDanymiException e) {
  System.out.println("Wyjatek zawiera komunikat: " + e.getMessage());
}
```

Wynik:

```
Wyjatek zawiera komunikat: Wartosci zapisane w tym wyjatku: 10 test
```

Moglibyśmy też dodać do powyższej klasy gettery, które zwracałyby wartość pola liczbowego i typu `String`, jeżeli potrzebowalibyśmy mieć możliwość bezpośrednioego odniesienia się do nich.

Inną metodą, którą wyjątki dziedziczą z klasy bazowej, jest `printStackTrace`. Metoda ta wypisuje na standardowe wyjście hierarchię wywołań metod w programie do momentu wystąpienia wyjątku:

```
try {
  throw new WyjatekZKomunikatemException("Co tu sie wyprawia?!");
} catch (WyjatekZKomunikatemException e) {
  System.out.println("Wyjatek zawiera komunikat: " + e.getMessage());
  e.printStackTrace();
}
```

Wywołanie `e.printStackTrace` spowoduje pojawienie się na standardowym wyjściu następujących komunikatów:

```
WyjatekZKomunikatemException: Co tu sie wyprawia?!
     at ZawartoscWyjatkowPrzyklady.main(ZawartoscWyjatkowPrzyklady.java:32)
```

Stack trace opisywałem na początku rozdziału o wyjątkach. W pierwszej linii znajduje się nazwa klasy wyjątku, po której następuje komunikat wyjątku. Następnie podane są metody i numery linii, który były po kolei wywoływane do momentu, w którym wystąpił wyjątek (te metody posortowane są, patrząc od góry, od ostatniej wywołanej do pierwszej). Umożliwia to prześledzenia wykonania programu aż do zaistnienia błędu i ułatwia analizę okoliczności, w jakich napotkany został problem.

11.5 Wyjątki Checked & Unchecked

Jak już kilka razy wspominałem, istnieją dwa typy wyjątków:

- checked exceptions,
- unchecked exceptions.

Główną różnicą pomiędzy tymi dwoma rodzajami wyjątków jest to, że gdy zamierzamy rzucić wyjątek tego pierwszego rodzaju (checked), to musimy go umieścić w klauzuli **throws** w sygnaturze naszej metody, natomiast w przypadku Unchecked Exceptions nie musimy tego robić (chociaż możemy).

To jest właśnie powód tego, że w niektórych przykładach w tym rozdziale korzystaliśmy z **throws** do zaznaczenia wyjątków rzucanych przez metodę, a w innych nie.

W przykładzie z dzieleniem liczb:

Nazwa pliku: ZwrocWynikDzielenia.java

```
public class ZwrocWynikDzielenia {
  public static void main(String[] args) {
    System.out.println(podziel(10, 0));
    System.out.println(podziel(25, 5));
  }

  public static int podziel(int x, int y) {
    return x / y;
  }
}
```

Pomimo, że wywołanie metody `podziel` może zakończyć się wyjątkiem `ArithmeticException`, nie korzystamy ani z **throws** do zaznaczenia tego faktu, ani nie musimy tego wyjątku obsługiwać. To dlatego, że `ArithmeticException` jest przykładem wyjątku Unchecked.

Z drugiej jednak strony, w przykładach w klasie `Osoba`, w której konstruktorze rzucaliśmy wyjątki `NieprawidlowyWiekException` i `NieprawidlowaWartoscException`, zaznaczyliśmy ten fakt za pomocą **throws** w sygnaturze konstruktora, a w metodzie `main` tej klasy, gdzie korzystaliśmy z tego konstruktora, stosowaliśmy **try**..**catch**:

fragment pliku Osoba.java

```
public Osoba(String imie, String nazwisko, int wiek)
    throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
  if (imie == null) {
    throw new NieprawidlowaWartoscException(
        "Imie nie moze byc puste."
    );
  }

  // ... pozostaly fragment konstruktora został pominiety
}

public static void main(String[] args) {
  try {
    Osoba o = new Osoba("Jan", "Nowak", -1);
  } catch (NieprawidlowaWartoscException e) {
    System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
  } catch (NieprawidlowyWiekException e) {
    System.out.println("Nieprawidlowy wiek!");
  }
}
```

11.5.1 Jak rozpoznać wyjątki Checked i Unchecked?

O tym, czy wyjątek to wyjątek Checked czy Unchecked decyduje tylko jedna właściwość – czy klasa wyjątku dziedziczy po klasie `RuntimeException`.

Wyjątki Unchecked nazywane są także runtime exceptions.

Co oznacza powyższe rozróżnienie?

Wyjątki to pochodne klasy `Exception` – klasa `RuntimeException` także jest pochodną klasy `Exception`. Jeżeli wyjątek w swojej hierarchii dziedziczenia nie ma klasy `RuntimeException`, to jest wyjątkiem Checked i trzeba go definiować w klauzuli **`throws`** w sygnaturze metody i obsługiwać w **`try`**..**`catch`**. Jeżeli ma w hierarchii klasę `RuntimeException`, to jest rodzaju Unchecked. Spójrzmy na kilka przykładów zdefiniowania własnych wyjątków:

```
// checked exceptions
class MojWyjatek extends Exception {}
class MojKolejnyWyjatek extends MojWyjatek {}

// unchcked exceptions
class MojRuntimeWyjatek extends RuntimeException {}
class MojKolejnyRuntimeWyjatek extends MojRuntimeWyjatek {}
```

Pierwsze dwa wyjatki mają następującą hierarchię klas (począwszy od klasy `Exception`):

```
Exception
    MojWyjatek
        MojKolejnyWyjatek
```

`MojKolejnyWyjatek` jest wyjątkiem rodzaju Checked, ponieważ ma w swojej hierarchii typ `Exception` oraz nie ma typu `RuntimeException` (podobnie jak typ `MojWyjatek`).

Z kolei dwa ostatnie wyjątki mają następującą hierarchię klas (począwszy od klasy `Exception`):

```
Exception
    RuntimeException
        MojRuntimeWyjatek
            MojKolejnyRuntimeWyjatek
```

Wyjątki te mają w swojej hierarchii klasę `RuntimeException`, są więc wyjątkami rodzaju Unchecked i nie trzeba ich definiować w klauzuli **`throws`**.

Wyjątki Unchecked można łapać tak samo w **`try`**..**`catch`** jak wyjątki Checked – widzieliśmy już taki przykład podczas obsługi metody `podziel`, gdy występował potencjał dzielenia przez `0`, oraz używając wyjątku `IllegalArgumentException` w pierwszej wersji klasy `Osoba` – ten wyjątek także jest wyjątkiem Unchecked.

11.5.2 Dlaczego istnieją dwa rodzaje wyjątków?

Idea wyjątków Checked jest taka, że są to wyjątki, które zazwyczaj można obsłużyć, i pozwolić na dalsze wykonywanie programu, np. gdy użytkownik poda nieprawidłową nazwę pliku do otwarcia możemy mu pozwolić ponownie spróbować ją podać. Takie wyjątki chcemy obsługiwać i zaznaczamy potencjał ich rzucenia w sygnaturze metody za pomocą słowa kluczowego **`throws`**. Jest to informacja dla każdego użytkownika naszej metody:

"*Jeśli będziesz korzystał z tej metody, to mogą się pojawić takie a takie wyjątki – powinieneś wziąć je pod uwagę i obsłużyć wedle własnego uznania.*"

Wyjątki Unchecked są często spowodowane błędnym stanem naszego programu i mogłoby być ciężko zareagować na nie w odpowiedni sposób – zamiast tego pozwalamy im zostać przetworzonymi przez Maszynę Wirtualną Java, nie podejmując sie sami próby ich obsługi. Takie wyjątki powinny zostać zauważne, a kod, które je powoduje, naprawiony, zamiast próbować obsłużyć je w kodzie za pomocą **`try`**..**`catch`**. Nie oznacza to jednak, że nie możemy obsługiwać wyjątków Unchecked w **`try`**..**`catch`** – możemy, jeżeli mamy taką potrzebę.

11.5.3 Sprawdzanie wyjątków rzucanych przez metodę

Korzystając z różnych bibliotek, w tym ze Standardowej Biblioteki Java, możemy czasem mieć potrzebę sprawdzić, jakie wyjątki dana metoda może rzucić – aby to zrobić, wystarczy zajrzeć do dokumentacji klasy, do której dana metoda należy.

Dla przykładu – używana już przez nas metoda `charAt` z klasy `String` może rzucić wyjątek `IndexOutOfBoundsException`. Dokładny opis można znaleźć w Java Doc. Z racji tego, że wyjątek ten jest rodzaju Unchecked, to korzystając z metody `charAt` nie musieliśmy stosować **`try`**..**`catch`** do łapania i obsługi tego wyjątku w poprzednich rozdziałach.

Jeżeli korzystamy z IntelliJ IDEA, możemy także sprawdzić wyjątki rzucane przez metodę poprzez naciśnięcie i przytrzymanie przycisku **`Ctrl`** na klawiaturze i kliknięcie lewym przyciskiem myszy nazwy metody – spowoduje to przejście do definicji tej metody, gdzie będziemy mogli spojrzeć na jej sygnaturę i sprawdzić, czy korzysta ze słowa kluczowego **`throws`** do zadeklarowania potencjału rzucenia pewnych wyjątków.

Wykorzystując powyższy sposób, spójrzmy na metodę `charAt` z klasy `String`:

```
/**
 * Returns the {@code char} value at the
 * specified index. An index ranges from {@code 0} to
 * {@code length() - 1}. The first {@code char} value of the sequence
 * is at index {@code 0}, the next at index {@code 1},
 * and so on, as for array indexing.
 *
 * <p>If the {@code char} value specified by the index is a
 * <a href="Character.html#unicode">surrogate</a>, the surrogate
 * value is returned.
 *
 * @param      index   the index of the {@code char} value.
 * @return     the {@code char} value at the specified index of this
string.
 *             The first {@code char} value is at index {@code 0}.
 * @exception  IndexOutOfBoundsException  if the {@code index}
 *             argument is negative or not less than the length of this
 *             string.
 */
public char charAt(int index) {
    if (isLatin1()) {
        return StringLatin1.charAt(value, index);
    } else {
        return StringUTF16.charAt(value, index);
    }
}
```

Jest to fragment klasy `String` ze standardowej biblioteki Java. Jak widzimy, sygnatura metody `charAt` nie zawiera **`throws`**, czyli nie rzuca wyjątków rodzaju Checked – nie trzeba więc stosować **`try`**..**`catch`**, gdy z niej korzystamy.

Warto jednak zauważyć, iż autorzy języka Java zawarli w komentarzu dokumentacyjnym sekcję *`@exception`*, w której napisali, że metoda ta może rzucić wyjątek `IndexOutOfBoundsException`. Jest to co prawda wyjątek Unchecked i nie musimy go obsługiwać, jednak gdybyśmy chcieli to zrobić, to mamy tutaj informację, że taki wyjątek może zostać rzucony i jeżeli mamy taką potrzebę, to możemy go obsłużyć.

Jak widzisz, w kodzie tej metody nie ma bezpośredniego rzucania wyjątku `IndexOutOfBoundsException` *– skąd więc taki komentarz twórców języka Java? Otóż wyjątek ten może rzucić jedna z metod wywoływanych przez tą metodę, a dokładniej* `StringLatin1.charAt`.

11.5.4 Jak sprawdzić rodzaj wyjątku

Sprawdzanie rodzaju wyjątku sprowadza się do analizy jego hierarchii dziedziczenia – jeżeli jest w niej zawarta klasa `RuntimeException`, oznacza to, że jest to wyjątek Unchecked i nie trzeba go obsługiwać w **`try`**..**`catch`**.

W przeciwnym razie wyjątek jest rodzaju Checked i jeżeli metoda, z której korzystamy, rzuca go, to musimy go prędzej czy później obsłużyć. Jak zobaczymy w jednym z kolejnych rozdziałów nie zawsze musimy koniecznie *od razu* obsługiwać wyjątki rodzaju Checked.

Jak jednak sprawdzić hierarchię dziedziczenia wyjątku? Najłatwiej sprawdzić to w dokumentacji – jeżeli korzystamy z klasy ze Standardowej Biblioteki Java, to informację o klasach znajdziemy w Java Doc. Dla przykładu, dla wyjątku `IllegalArgumentException` hierarchia dziedziczenia wygląda następująco:

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```

źródło: [oficjalna dokumentacja Java Doc – klasa Illegal ArgumentException]

Widzimy w tej hierarchii klasę `RuntimeException`, więc klasa `IllegalArgumentException` jest przedstawicielką wyjątków rodzaju Unchecked.

Jeżeli korzystamy z wyjątków z innej biblioteki, to musimy sprawdzić to w dokumentacji tej biblioteki w internecie.

11.5.5 Błędy kompilacji podczas braku obsługi wyjątków Checked

Jeżeli skorzystamy z metody, która rzuca wyjątki rodzaju Checked, a nie obsłużymy ich w **`try..catch`**, to nasza klasa się nie skompiluje – będzie to dla nas informacja, że brakuje obsługi wyjątków – widzieliśmy taki przypadek w klasie `Osoba` w tym rozdziale. Definiując, że konstruktor klasy `Osoba` może rzucić dwa wyjątki rodzaju Checked:

```
public Osoba(String imie, String nazwisko, int wiek)
    throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
```

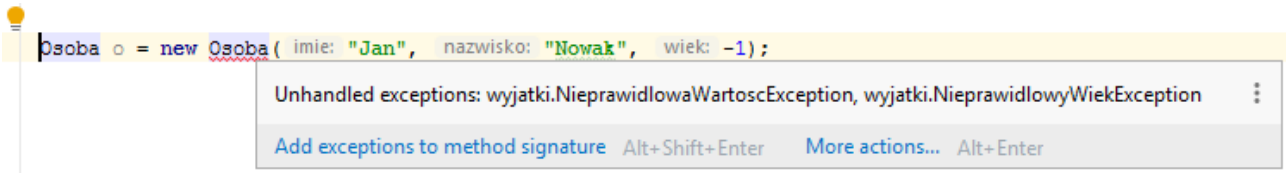
i korzystając z poniższego fragmentu kodu:

```
Osoba o = new Osoba("Jan", "Nowak", -1);
```

spowodujemy, że próba kompilacji zakończy się następującym błędem:

```
Osoba.java:31: error: unreported exception NieprawidlowaWartoscException;
must be caught or declared to be thrown
      Osoba o = new Osoba("Jan", "Nowak", -1);
                ^
1 error
```

Gdy korzystamy z IntelliJ IDEA to łatwo wykryć sytuację, gdy korzystamy z kodu, który powinien zostać opakowany w blok **`try..catch`** jeszcze zanim spróbujemy skompilować nasz kod. IntelliJ IDEA jako mądre narzędzie analizuje nasz kod na bieżąco i wykrywa, że zapomnieliśmy o złapaniu wyjątków – podkreśli wtedy fragment kodu, który może rzucić nieobsłużone wyjątki, dając nam znać, że powinniśmy to zrobić:



11.5.6 Błąd Error

Istnieje jeszcze trzeci rodzaj wyjątków, które są pochodnymi klasy `Error`. Podobnie jak klasa `Exception`, dziedziczy ona bezpośrednio po `Throwable`, czyli klasie, która jest klasa nadrzędną wszystkich wyjątków.

Chociaż `Throwable` to ta właściwa klasa nadrzędna dla wszystkich wyjątków, to w praktyce nigdy z niej bezpośrednio nie korzystamy – zamiast tego jako klas bazowych dla naszych wyjątków używamy klasy `Exception` bądź jej pochodnych.

Wyjątki dziedziczące po `Error` to błędy krytyczne, których za bardzo nie da się obsłużyć i nie powinniśmy próbować tego robić. Sami też takich wyjątków nie będziemy nigdy rzucać. Taki błąd to np. `OutOfMemoryError`, który występuje gdy skończy się pamieć komputera przeznaczona dla naszego programu. My, jako autorzy programu, nie możemy nic na wystąpienie takiego wyjątku poradzić. To, co możemy i powinniśmy zrobić, to przeanalizować dlaczego pamięć się skończyła:

- Czy po prostu za mało pamięci zostało przeznaczone na działanie naszego programu?
- Czy nasz program nie jest optymalnie napisany?
- Czy w programie występuje błąd, który powoduje niemożliwość zwalniania pamięci przez Garbage Collector (mechanizm odpowiedzialny za zwalnianie nieużywanej pamięci w naszych programach)?

Występienie wyjątku typu `Error` to poważniejszy problem, który trzeba przeanalizować – jego obsługa w kodzie albo nie ma sensu, albo wręcz nie jest możliwa.

11.6 Łapanie wyjątków

W poprzednim rozdziale rzucaliśmy wyjątki za pomocą słowa kluczowego `throw`, po którym następował wyjątek. Wyjątek to po prostu obiekt konkretnej klasy wyjątku. Gdy łapiemy wyjątek w sekcji `catch`, podajemy typ wyjątku, jaki chcemy obsłużyć, oraz nazwę zmiennej, za pomocą której będziemy się do tego obiektu-wyjątku odnosić.

We wszystkich przykładach do tej pory nazywaliśmy tą zmienną po prostu `e`, np.:

fragment metody main z pliku Osoba.java

```
try {
  Osoba o = new Osoba("Jan", "Nowak", -1);
} catch (NieprawidlowaWartoscException e) { // 1
  System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
} catch (NieprawidlowyWiekException e) { // 2
  System.out.println("Nieprawidlowy wiek!");
}
```

W tym przykładzie spodziewamy się dwóch różnych wyjątków, które chcemy obsłużyć. Do każdego z nich w odpowiedniej sekcji `catch` będziemy mogli odnieść się za pomocą zmiennej o nazwie `e` *(1)* *(2)*. Obiekt, na który ta zmienna będzie wskazywać w każdym z tych przypadków, to obiekt klasy wyjątku utworzonego w konstruktorze klasy `Osoba`:

fragment pliku Osoba.java - konstruktora klasy Osoba

```
public Osoba(String imie, String nazwisko, int wiek)
    throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
  if (imie == null) {
    throw new NieprawidlowaWartoscException( // 3
        "Imie nie moze byc puste."
    );
  }
  if (nazwisko == null) {
    throw new NieprawidlowaWartoscException( // 4
        "Nazwisko nie moze byc puste."
    );
  }
  if (wiek <= 0) {
    throw new NieprawidlowyWiekException(); // 5
  }

  this.imie = imie;
  this.nazwisko = nazwisko;
  this.wiek = wiek;
}
```

Gdy przekazane `imię` lub `nazwisko` będzie nullem *(3)* *(4)*, to zmienna `e` z pierwszej sekcji `catch` *(1)* będzie wskazywała na obiekt typu `NieprawidlowaWartoscException` utworzony w *(3)* bądź *(4)*. Analogicznie, jeżeli `wiek` bedzie nieprawidłowy, to zmienna `e` w drugiej sekcji `catch` *(2)* będzie odnosiła się do obiektu typu `NieprawidlowyWiekException` tworzonego i rzucanego w *(5)*.

W sekcji `catch` zmienna wyjątku może mieć dowolną nazwę, niekoniecznie musi to być `e`. Taka nazwa jest jednak używana dla wygody, tym bardziej, że nie ma ona specjalnie znaczenia – typ i treść wyjątku ma większe znaczenie, niż nazwa zmiennej, której chwilowo używamy do odnoszenia się do niego.

11.6.1 Łapanie wyjątków za pomocą klasy bazowej

Załóżmy, że mamy metodę, która może rzucić wiele wyjątków, ale my, jako osoby używające tej metody, nie chcemy obsługiwać osobno każdego z nich. Zamiast wypisywać poszczególne wyjątki, możemy zamiast tego złapać wyjątek nadrzędny dla naszych wyjątków (którym zawsze jest wyjątek typu `Exception`) i w jednym miejscu obsłużyć wszystkie błędy. Spójrzmy jak by to wyglądało w klasie `Osoba`:

fragment metody main z pliku Osoba.java

```
try {
  Osoba o = new Osoba("Joanna", "Strzelczyk", -1);
} catch (Exception e) {
  System.out.println("Wystapil blad! Komunikat bledu: " + e.getMessage());
}

try {
  Osoba o = new Osoba(null, "Strzelczyk", 30);
} catch (Exception e) {
  System.out.println("Wystapil blad! Komunikat bledu: " + e.getMessage());
}
```

Zamiast łapać konkretne wyjątki, złapaliśmy po prostu wszystkie wyjątki korzystając z klasy bazowej wyjątków – klasy `Exception`. Kod działa, ponieważ zarówno wyjątek `NieprawidlowaWartoscException`, jak i `NieprawidlowyWiekException`, bazują na typie `Exception`, więc kwalifikują się do złapania. Widzimy tutaj dziedziczenie i polimorfizm w akcji – korzystamy z klasy bazowej, a potencjalnie działamy na obiektach klas pochodnych. Klasa `Exception` jest "bardziej ogólna" niż dwa pozostałe wyjątki.

Zapisując kod w ten sposób mówimy kompilatorowi:

"*Nieważne czy to będzie* `NieprawidlowaWartoscException` *czy* `NieprawidlowyWiekException`*, każdy z nich to* `Exception` *i chcę je obsłużyc w tej jednej sekcji* **`catch`**".

W wyniku działania tego fragmentu kodu zobaczymy na ekranie:

```
Wystapil blad! Komunikat bledu: null
Wystapil blad! Komunikat bledu: Imie nie moze byc puste.
```

W pierwszej linii widzimy, że komunikat błędu jest nullem – wynika to z faktu, że tworząc wyjątek typu `NieprawidlowyWiekException` w konstruktorze klasy `Osoba` nie podajemy żadnego komunikatu błędu.

Możemy także złapać konkretne wyjątki, a na końcu podać `Exception`, by obsłużyć wyjątki pewnego rodzaju, a pozostałe obsłużyć w bloku obsługi ogólnego wyjątku `Exception`:

```
try {
  Osoba o = new Osoba("Jan", "Nowak", -1);
} catch (NieprawidlowyWiekException e) {
  System.out.println("Nieprawidlowy wiek!");
} catch (Exception e) {
  System.out.println("Wystapil blad! Komunikat bledu: " + e.getMessage());
}

try {
  Osoba o = new Osoba(null, "Nowak", 30);
} catch (NieprawidlowyWiekException e) {
  System.out.println("Nieprawidlowy wiek!");
} catch (Exception e) {
  System.out.println("Wystapil blad! Komunikat bledu: " + e.getMessage());
}
```

Ten fragment kodu spowodowalby wypisanie na ekran:

```
Nieprawidlowy wiek!
Wystapil blad! Komunikat bledu: Imie nie moze byc puste.
```

Używanym tutaj "wyjątkiem ogólnym" nie musi być `Exception`*, lecz dowolny typ wyjątku, który byłby w hierarchii dziedziczenia używanych przez nas klas wyjątków w klasie* `Osoba`*. Dla przykładu, jeżeli wyjątki* `NieprawidlowaWartoscException` *i* `NieprawidlowyWiekException` *dziedziczyłyby nie bezpośrednio po* `Exception`*, lecz po innym utworzonym przez nas typie wyjątków, np.* `BladWalidacjiDanychOsobyException`*, to moglibyśmy używać tego typu wyjątku w sekcji* **`catch`***, aby złapać oba rodzaje wyjątków pochodnych od tego nowego typu wyjątku.*

11.6.2 Łapanie kilku wyjątków za pomocą znaku |

Jest jeszcze inna składnia pozwalająca na łapanie wykluczających się wyjątków – możemy rozdzielić je w klauzuli **`catch`** za pomocą znaku | (pionowa kreska, ang. *pipe*):

fragment metody main z pliku Osoba.java

```
try {
  Osoba o = new Osoba("Adrian", "Sochacki", 30);
} catch (NieprawidlowaWartoscException | NieprawidlowyWiekException e) {
  System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
}
```

W jednej sekcji **`catch`** łapiemy dwa wyjątki, rozdzielając nazwy ich klas znakiem |. Wyjątki te nie mogą dziedziczyć po sobie – jeżeli spróbowalibyśmy w ten sposób umieścić w **`catch`** np. wyjątki `Exception | NieprawidlowaWartoscException`, to kod nie skompilowałby się, ponieważ typ wyjątku `Exception` jest typem bazowym wyjątku `NieprawidlowaWartoscException`.

Typ `Exception` jest "bardziej ogólny", więc już samo jego umieszczenie w sekcji **`catch`** powoduje, że i wyjątek `NieprawidlowyWiekException`, który jest klasą pochodną od `Exception`, zostanie złapany. Kompilator wykryłby taką sytuację i zgłosiłby błąd.

11.6.3 Kolejność łapania wyjątków ma znaczenie

Kolejność obsługiwania wyjątków w blokach **`catch`** ma znaczenie – bardziej ogólne wyjątki muszą zawsze następować po mniej ogólnych. Najbardziej ogólnymi wyjątkami są wyjątki typu `Exception` (ponieważ wszystkie wyjątki bazują na tym typie), a mniej ogólne to te,

które dziedziczą po klasie `Exception`. Możemy rozszerzać inne wyjątki zdefiniowane w bibliotece standardowej Java, a także nasze własne wyjątki, więc musimy pamiętać o hierarchii. Jeżeli kolejność będzie nieprawidłowa, kompilator zaprotestuje:

```
try {
  Osoba o = new Osoba(null, "Nowak", 30);
} catch (Exception e) {
  System.out.println("Wystapil blad! Komunikat bledu: " + e.getMessage());
} catch (NieprawidlowyWiekException e) {
  System.out.println("Nieprawidlowy wiek!");
}
```

Ten fragment kodu powoduje błąd kompilacji:

```
Error: java: exception NieprawidlowyWiekException has already been caught
```

Komunikat mówi o tym, że już obsłużyliśmy wyjątek typu `NieprawidlowyWiekException` – nastąpiło to w pierwszym bloku **`catch`** – wyjątek ten został dopasowany do pierwszego bloku **`catch`**, ponieważ wyjątek `NieprawidlowyWiekException` jest klasą pochodną klsay `Exception`.

Innymi słowy, sekcja **`catch`** z "bardziej ogólnym" wyjątkiem `Exception` obsługuje wszystkie wyjątki klasy `Exception` i jej klas pochodnych. W związku z tym, sekcja **`catch`** z wyjątkiem `NieprawidlowyWiekException` nigdy nie miałaby szansy zostać wykonana. Kompilator wykrywa ten problem i nie pozwala na skompilowanie takiego kodu.

Pamiętajmy, by sekcje **`catch`** zawsze zawierały najbardziej szczegółowe (najniżej w hierarchii dziedziczenia) wyjatki na początku, a najbardziej ogólne – na końcu. Obsługa wyjątków za pomocą typu `Exception` powinna zawsze być ostatnią sekcją **`catch`**, ponieważ załapią się do niej wszystkie rodzaje wyjątków w związku z tym, że klasa ta jest klasą bazową dla wszystkich wyjątków.

11.7 Pomijanie łapania wyjątków

Możemy zadać teraz pytanie: a co, jeżeli nie chcemy obsługiwać wyjątków (bądź obsłużyć tylko część z nich)? Czy zawsze musimy obsłużyć wszystkie możliwe wyjątki rzucane przez daną metodę (zadeklarowane za pomocą `throws`)?

Nie, ale w takim przypadku musimy w metodzie, która nie chce obsłużyć jakiegoś wyjątku (bądź wszystkich), zawrzeć informację, że może ona rzucić dany wyjątek za pomocą poznanego słowa kluczowego `throws`. Spójrzmy na przykład (korzystający z klasy `Osoba`):

fragment klasy Osoba.java

```
public static Osoba stworzPelnoletniaOsobe(String pierwszeImie, String
nazwisko) throws NieprawidlowaWartoscException { // 1

  Osoba result = null;

  try {
    result = new Osoba(pierwszeImie, nazwisko, 18);
  } catch (NieprawidlowyWiekException e) { // 2
    // nic nie robimy, bo podajemy poprawny wiek - nie zakladamy bledu
  }

  return result;
}
```

Zdefiniowaliśmy powyżej nową metodą – `stworzPelnoletniaOsobe`. Ma ona na celu stworzenie obiektu klasy `Osoba`, która ma wiek równy `18`. Nie chcemy w tej metodzie obsługiwać przypadku, gdy ktoś poda nieprawidłowe imię bądź nazwisko – w związku z tym, by kompilator nie protestował, że nie obsłużyliśmy wyjątku `NieprawidlowaWartoscException`, dodaliśmy klauzulę `throws` do metody `stworzPelnoletniaOsobe` *(1)* – oznacza to, że ten, kto wywoła metodę `stworzPelnoletniOsobea`, będzie musiał:

- obsłużyć wyjątek `NieprawidlowaWartoscException` lub
- także zdefiniować, że rzuca wyjątek `NieprawidlowaWartoscException`, jezeli nie będzie chciał go obsłużyć.

Możemy powiedzieć, że metoda `stworzPelnoletniaOsoba` oddelegowuje obsługę wyjątku `NieprawidlowaWartoscException` do metody, która będzie z niej korzystała.

Zauważmy, że sekcja `catch` w powyższej metodzie nie zawiera żadnych instrukcji *(2)* – zakładamy, że wyjątek `NieprawidlowyWiekException` nie zostanie rzucony, ponieważ podajemy poprawny wiek. Nie zmienia to faktu, że musimy obsługę tego wyjątku zawrzeć w tej metodzie, ponieważ kompilator języka Java nie jest w stanie wywnioskować, że w tym przypadku wyjątek na pewno nie będzie rzucony.

W metodzie `main` dodajemy poniższy fragment kodu, w którym korzystamy z metody `stworzPelnoletniaOsobe`:

fragment metody main z klasy Osoba.java

```
try {
  Osoba osobaPelnoletnia = stworzPelnoletniaOsobe("Jan", null);
} catch (NieprawidlowaWartoscException e) {
  System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
}
```

Ponieważ metoda `stworzPelnoletniaOsobe` deklaruje za pomocą `throws`, że może rzucić wyjątek

`NieprawidlowaWartoscException`, stosujemy **`try..catch`** w powyższym fragmencie kodu. Nie obsługujemy wyjątku `NieprawidlowyWiekException`, ponieważ obsługą tego wyjątku zajmuje się metoda `stworzPelnoletniaOsobe`.

Tak naprawdę to nie metoda `stworzPelnoletniaOsobe`*, lecz konstruktor klasy* `Osoba`*, rzuca wyjątek* `NieprawidlowaWartoscException`*, ale z racji tego, że metoda* `stworzPelnoletniaOsobe` *tego wyjątku nie obsługuje, to wywołując tą metodę może zajść sytuacja, która spowoduje pojawienie się takiego wyjątku.*

11.7.1 Silent catch

Obsługa wyjątków często nie jest łatwa – trzeba się zastanowić, jak program powinien zachować się, gdy wystąpi pewien wyjątek:

- Czy program powinien kontynuować działanie?
- Czy użytkownik powinien zostać powiadomiony o błędzie?
- Czy program powinien odczekać i spróbować ponownie wykonać kod, który spowodował wyjątek?

Obsługa wyjątków zawsze wiąże się z wymogiem napisania dodatkowego kodu, co nierzadko zajmuje sporo czasu.

Czasem możemy mieć chęć po prostu złapać wyjątki w sekcji **`catch`**, ale nie pisać żadnego kodu, który by je obsługiwał:

```
try {
  Osoba o = new Osoba("Adrian", "Sochacki", 30);
} catch (Exception e) {
  // zlap wszystkie wyjatki i sie nie przejmuj!
}
```

Tak zapisany kod zaoszczędza nam czas kosztem problemów, które ze sobą niesie. Powyższy sposób zapisu kodu to tzw. *silent catch*. Ja spotkałem się jeszcze z polskim określeniem "połykanie wyjątków".

Przez to, że nie obsługujemy wyjątków, nie mamy sposobu aby się dowiedzieć, że coś poszło w naszym programie nie tak, bo wszelkie błędy wynikłe z wykonania danej metody są ignorowane.

Może to powodować niestabilne działanie programu i bardzo trudne do wychwycenia błędy, których analiza, znalezienie, i naprawienie, zajmą dużo więcej czasu, niż napisanie dobrego kodu obsługi wyjątku na samym początku tworzenia programu.

Czasem napotkasz się na sytuacje, w których po prostu nie będziesz miał potrzeby obsługi wyjątków lub będziesz pewien, że nie będą rzucone – wtedy silent catch może być dobrym rozwiązaniem. Miej jednak na uwadze, że ignorowanie błędów ze względu na trudność ich obsługi to nie wymówka, aby tego nie robić.

11.7.2 Pomijanie try..catch w metodzie main

Może się zdarzyć sytuacja, w której będziemy w metodzie `main` korzystać z metody, która może rzucić wyjątek rodzaju Checked i z jakiegoś powodu nie będziemy chcieli tego wyjątku obsługiwać. W takim przypadku kompilator zgłosi błąd widząc, że wyjątek rodzaju Checked nie jest obsługiwany:

Nazwa pliku: MainUzywaThrows.java

```
class PewienWyjatekException extends Exception {

}

public class MainUzywaThrows {
  public static void main(String[] args) {
    pewnaMetoda(); // 1
  }

  public static void pewnaMetoda() throws PewienWyjatekException {
    // pewne instrukcje ktore powoduja rzucenie wyjatku
    throw new PewienWyjatekException();
  }
}
```

W metodzie `main` wywołujemy metodę `pewnaMetoda` *(1)*, która rzuca wyjątek `PewienWyjatekException`. Brak obsługi tego wyjątku powoduje błąd kompilacji tej klasy:

```
MainUzywaThrows.java:7: error: unreported exception PewienWyjatekException;
must be caught or declared to be thrown
    pewnaMetoda();
               ^
1 error
```

Nie musimy jednak tego wyjątku obsługiwać w metodzie `main` – jak wiemy z jednego z poprzednich rozdziałów, jeżeli metoda nie chce obsłużyć wyjątku, to musi zadeklarować potencjał jego rzucenia w swojej własnej sygnaturze:

```
public static void main(String[] args) throws PewienWyjatekException {
  pewnaMetoda();
}
```

Tak zapisana metoda w powyższym programie powoduje, że kod kompiluje się bez błędów.

Pytanie: kto w takim razie obsłuży ten wyjątek? Ten, kto wywołuje metodę `main` w naszym programie! A jest to nikt inny jak Maszyna Wirtualna Java – gdy rozpoczyna ona wykonanie naszego programu, wywołuje metodę `main`. Jeżeli rzucony zostanie nieobsłużony wyjątek, to "złapie" go Maszyna Wirtualna Java i wyświetli komunikat na standardowym wyjściu.

Wynik działania tego programu:

```
Exception in thread "main" PewienWyjatekException
        at MainUzywaThrows.pewnaMetoda(MainUzywaThrows.java:12)
        at MainUzywaThrows.main(MainUzywaThrows.java:7)
```

Wyjątek zostaje rzucony w metodzie `pewnaMetoda`. Jako, że metoda, która ją wywołała, czyli `main`, nie obsługuje tego wyjątku (nie korzysta z `try..catch`), a zamiast tego sama deklaruje za pomocą `throws`, że taki wyjętek może być rzucony, wędruje on dalej. Dalej jest już tylko Maszyna Wirtualna Java, która wywołała naszą metodę `main`. Wyjątek zostaje obsłużony przez Maszynę Wirtualną Java w ten sposób, że po prostu jego treść i stack trace zostają wypisane na standardowe wyjście.

11.8 Try with resources

Od wersji Java 1.7 dostępna jest nowa wersja instrukcji **try**..**catch**..**finally** nazywana *try-with-resources* (**try** z zasobami). Została ona wprowadzona dla wygody programistów, by zautomatyzować zamykanie różnego rodzaju zasobów takich jak obiekty klas odpowiedzialnych za np. odczytywanie danych z pliku.

Różnicę pomiędzy "standardowym" **try**..**catch**..**finally** oraz *try-with-resources* zobaczymy na przykładzie programu, którego jedynym zadaniem będzie wypisanie na ekran zawartości pliku znajdującego się na dysku komputera.

Aby odczytać z dysku plik w języku Java, musimy:

1. Utworzyć obiekt typu `File`, który będzie skojarzony z plikiem na dysku.
2. Utworzyć strumień danych, który będzie mógł czytać dane z pliku, na który wskazuje obiekt `File`. Strumieniem danych w naszych przykładach będzie obiekt klasy `FileReader` – klasa ta pozwala na czytanie z pliku znak po znaku.
3. Zamknąć strumień po zakończeniu pracy na nim wywołując jego metodę `close`.

Dodatkowo, gdy korzystamy z klas strumieni, takich jak `FileReader`, musimy obsłużyć wyjątki, które mogą wystąpić. Oznacza to, że w naszym programie będziemy musieli skorzystać z **try**..**catch** do obsługi potencjalnych sytuacji wyjątkowych. Wyjątkiem bazowym używanym w metodach klas, które mają coś wspólnego z działaniami na plikach, jest `IOException` (*Input/Output Exception*), i taki też wyjątek będziemy łapać.

Poniżej znajdują się dwa przykłady – pierwszy korzysta ze "starego" podejścia, natomiast drugi używa nowego mechanizmu *try-with-resources*. Oba programy mają za zadanie odczytać plik z dysku znak po znaku i wypisać go na ekran. Spójrzmy najpierw na przykład korzystający ze zwykłegeo **try**..**catch**..**finally**:

Nazwa pliku: CzytaniePlikuTryCatch.java

```
import java.io.File; // 1
import java.io.FileReader;
import java.io.IOException;

public class CzytaniePlikuTryCatch {
  public static void main(String[] args) {
    File f = new File("C:/programowanie/powitanie.txt"); // 2
    FileReader fileReader = null; // 3

    try {
      fileReader = new FileReader(f); // 4
      int odczytanyZnak;

      while ((odczytanyZnak = fileReader.read()) != -1) { // 5
        System.out.print((char) odczytanyZnak); // 6
      }
    } catch (IOException e) { // 7
      System.out.println(e.getMessage());
    } finally {
      try {
        if (fileReader != null) {
          fileReader.close(); // 8
        }
      } catch (IOException e) { // 9
        System.out.println(
            "Blad podczas zamykania strumienia: " + e.getMessage()
        );
      }
    }
  }
}
```

Ten program wypisuje na ekran zawartość pliku `powitanie.txt` znajdującego się w katalogu `C:\programowanie`:

```
Witaj
Swiecie
!
```

Krótka analiza tego programu:

1. Klasy do pracy z plikami, z których skorzystamy, znajdują się w pakiecie `java.io` w Bibliotece Standardowej Java.
2. Tworzymy obiekt klasy `File` przekazując jako argument konstruktora lokalizację pliku, z którym ten obiekt będzie skojarzony.
3. Definiujemy obiekt klasy `FileReader`, który będzie służył do odczytania pliku. Zmienna `fileReader` znajduje się przed blokiem **`try`**..**`catch`** ponieważ będziemy z niej chcieli skorzystać w bloku **`finally`**. Gdybyśmy umieścili definicję tej zmiennej wewnątrz bloku **`try`**, to zmienna ta byłaby dostępna jedynie w tym bloku.
4. Tworzymy obiekt typu `FileReader`, który skojarzony będzie z plikiem, na który wskazuje utworzony wcześniej obiekt typu `File`. Tworzenie tego obiektu jest w bloku **`try`**, ponieważ może zostać rzucony wyjątek `FileNotFoundException` – gdy plik, na który wskazuje przekazany do konstruktora obiekt typu `File`, nie zostanie znaleziony.
5. W warunku pętli wykonujemy dwie operacje: przypisujemy do zmiennej `odczytanyZnak`

wartość zwróconą z metody `fileReader.read()`, która zwraca kod liczbowy przeczytanego znaku, a nastepnie porównujemy całość tego wyrażenia do `-1`. Metoda `read` zwraca wartość `-1` w przypadku, gdy przeczytany został już cały plik – będzie to oznaczało koniec działania pętli.

6. Jeżeli przeczytany kod znaku nie był liczbą `-1`, to wypisujemy go na ekran, rzutując go najpierw do wartości typu **`char`**. Musimy to zrobić, ponieważ metoda `read` z poprzedniego punktu zwraca nie faktyczny znak w pliku, lecz jego liczbową reprezentację.
7. Jeżeli wystąpi wyjątek, to łapiemy go i wypisujemy na ekran komunikat błędu.
8. Na końcu programu powinniśmy zamknąć obiekt `fileReader`, który służył do czytania z pliku, za pomocą metody `close`. Najpierw sprawdzamy, czy `fileReader` nie jest nullem ponieważ w bloku **`try`** mógł wystąpić wyjątek podczas próby utworzenia tego obiektu i mógł on nie zostać zainicjalizowany inną wartością niż **`null`**, a na nullowym obiekcie nie chcemy wywołać metody `close`.
9. Sama metoda `close` także może rzucić wyjątek `IOException`, jeżeli z jakiegoś powodu nie powiedzie się próba zamknięcia zasobu, na której ją wywołujemy. Zauważmy, że w sekcji **`finally`** ponownie korzystamy z **`try`**..**`catch`**, by złapać ewentualny wyjątek związany z próbą zamknięcia obiektu `fileReader`.

Dlaczego instrukcja `fileReader.close();` nie jest częścią głównego bloku **`try`**..**`catch`**? Jeżeli instrukcja ta byłaby po pętli czytającej plik, a w trakcie czytania wystąpiłby wyjątek, to nigdy nie doszłoby do próby zamknięcia strumienia `fileReader`. Chcemy mieć pewność, że metoda `close` zostanie wywołana na rzecz obiektu `fileReader` niezależnie od tego, czy czytanie pliku się powiodło, czy nie – dlatego instrukcja zamykania jest w sekcji **`finally`**.

Porównaj powyższy program z jego drugą wersją, która korzysta z mechanizmu *try-with-resources*:

Nazwa pliku: CzytaniePlikuTryWithResources.java

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class CzytaniePlikuTryWithResources {
  public static void main(String[] args) {
    File f = new File("C:/programowanie/powitanie.txt");

    try (FileReader fileReader = new FileReader(f)) {
      int odczytanyZnak;

      while ((odczytanyZnak = fileReader.read()) != -1) {
        System.out.print((char) odczytanyZnak);
      }
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Ta wersja jest zdecydowanie krótsza – zauważ, że w ogóle nie ma w niej sekcji **`finally`**, a sekcja **`try`** ma dodatek w postaci tworzenia obiektu `fileReader` w nawiasach:

```
try (FileReader fileReader = new FileReader(f)) {
```

To właśnie ten fragment świadczy o tym, że jest to *try-with-resources*. Naszym zasobem w tym

przypadku jest strumień `fileReader` typu `FileReader`. Tak utworzony obiekt jest dostępny w bloku **`try`** – odczytywanie i wypisywanie znaku na ekran nie różni się pomiędzy obydwoma programami.

Ta wersja nie ma sekcji **`finally`**, ponieważ obiekt `fileReader` zostanie automatycznie zamknięty (zostanie na nim wywołana metoda `close`) po zakończeniu bloku **`try`**..**`catch`** dla naszej wygody i krótszego kodu.

Nie wszystkie klasy mają metodę `close` – skąd w takim razie wiemy, które klasy mogą być użyte jako "zasoby" w try-with-resources? Zależy to od tego, czy dana klasa implementuje interfejs `Closeable` lub `AutoCloseable`. O interfejsach opowiemy sobie w następnym rozdziale.

11.9 Wady i zalety wyjątków

Mechanizm wyjątków ma zarówno zalety, jak i wady.

11.9.1 Dlaczego używać mechanizmu wyjątków?

Mechanizm wyjątków ma dwie podstawowe zalety:

1. Możemy obsłużyć dowolne sytuacje, które uznamy za nieprawidłowe, bez potrzeby stosowania rozwiązań z np. zwracaniem specjalnej wartości (jak w przykładzie z dzieleniem).
2. Przenosi odpowiedzialność obsługi błędu do tego, kto używa kod, który potencjalnie rzuca wyjątek. Zamiast w funkcji `podziel` wypisywać na ekran, że nie można dzielić przez `0` bądź zwracać specjalną wartość w takim przypadku, pozwalamy temu, kto wywołuje metodę `podziel`, na odpowiednie obsłużenie takiego przypadku. Jest to o tyle ważne, że w bardziej skomplikowanych przypadkach może nie być jednego uniwersalnego sposobu na obsłużenie danego błędu – rzucenie wyjątku pozwoli, by w różnych sytucjach można było odpowiednio na dany błąd zareagować.

11.9.2 Wady wyjątków

W poprzednich rozdziałach widzieliśmy kilka cech wyjątków rodzaju Checked:

- jeżeli w wyniku wywołania metody może zostać rzucony wyjątek rodzaju Checked, a nie chcemy go obsługiwać, to musimy skorzystać ze słowa kluczowego **`throws`**, aby zaznaczyć, że dana metoda może taki wyjątek rzucić,
- jeżeli nie obsłużymy sytuacji, w której może być rzucony wyjątek rodzaju Checked i nie skorzystamy z **`throws`**, to nasz kod się nie skompiluje – kompilator zgłosi błąd.

Jeżeli wywołujemy metodę, która wywołuje kolejną metodę itd. i ostatnia z tych metod może rzucić wyjątek rodzaju Checked, a obsłużyć go chcemy dopiero na samej górzej tego stosu wywołań metod, to wszystkie metody po drodze muszą zawierać klauzulę **`throws`** – inaczej nasz kod się nie skompiluje. Powoduje to potencjalny narzut, ponieważ teraz w każdym miejscu naszego programu, w którym wywołamy którąś z tych metod, będziemy musieli korzystać z bloku **`try`**..**`catch`**.

Ponadto, wymóg deklaracji rzucanego wyjątku i obsługi go uniemożliwia łatwą modyfikację już napisanych metod. Załóżmy, że mamy metodę `pewnaMetoda`, z której korzystamy w wielu miejscach naszego programu. W pewnym momencie musimy ją zmodyfikować i wykorzystać metodę z pewnej biblioteki, którą dodaliśmy do naszego projektu. Jeżeli metoda z tej biblioteki może rzucić wyjątek rodzaju Checked, to musimy albo dodać jego obsługę w naszej metodzie `pewnaMetoda`, albo dodać do jej sygnatury **`throws`** – ale to spowoduje, że wszystkie miejsca w naszym programie, które wcześniej korzystały z metody `pewnaMetoda`, przestaną się kompilować, jeżeli nie korzystały z **`try`**..**`catch`**.

Częstym zarzutem odnośnie wyjątków jest ich nadużycie – niekiedy metody definiują rzucanie wielu wyjątków, które potem trzeba obsługiwać. Czasem takie wyjątki powinny po prostu być rodzaju Unchecked zamiast Checked. Czasem takie sytuacje rozwiązuje się po prostu poprzez złapanie wszystkich wyjątków za pomocą klasy `Exception`:

```
try {
  metodaMogacaRzucicWieleWyjatkow();
} catch (Exception e) {
  // .. zbiorcza obsluga wyjatkow
}
```

Jeżeli wyjątek ma być obsłużony wyżej w hierarchii wykonań, to zamiast go obsługiwać, jest on rzucany dalej, ale tym razem metody wyżej mają do obsługi jeden znormalizowany wyjątek.

Poza tym, poprzez *ciche łapanie wyjątków* (o którym wspominałem w jednym z wcześniejszych rozdziałów) możemy spowodować, że nasz program będzie zawierał ciężkie do wykrycia błędy. Chociaż takie przypadki to ewidentna wina programisty, a nie samego mechanizmu wyjątków.

Podsumowując – jeżeli w naszym programie chcemy zasygnalizować problem, który można rozwiązać i po którym program może dalej działać, to korzystajmy z wyjątków rodzaju Checked, a w przeciwnym razie rzucajmy wyjątki Unchecked. Unikajmy cichego łapania wyjątków i pamiętajmy, jakie konsekwencje niesie ze sobą dodanie rzucania wyjątków Checked przez już istniejące i używane metody.

11.10 Podsumowanie

11.10.1 Podstawy wyjątków

- Wyjątki (exceptions) to sytuacje, w których coś w programie poszło nie tak.
- Gdy wystąpi wyjątek, mówimy, że został on *rzucony*.
- Wyjątki to obiekty klas. Jak każda klasa mają one swoją nazwę, konstruktory, pola i metody.
- Dzięki wyjątkom możemy obsłużyć dowolne sytuacje, które uznamy za nieprawidłowe, bez potrzeby stosowania rozwiązań z np. zwracaniem specjalnej wartości.
- Spotkaliśmy już się z paroma wyjątkami, np. `ArrayIndexOutOfBoundsException` i `NullPointerException`.
- Klasy wyjątków to klasy rozszerzające klasę `Throwable` (lub jej pochodną np. `Exception`).
- Wyjątki mogą zawierać komunikat informujący o zaistniałym błędzie, który możemy pobrać za pomocą metody `getMessage`.
- *Stack trace* to ścieżka wykonań metod, które doprowadziły do błędu. Kolejność metod w stack trace jest odwrotna do kolejności ich wykonywania. Metoda na dole została wykonana jako pierwsza, a ta na górze jako ostatnia i w niej rzucony został wyjątek:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ZwrocWynikDzielenia.podziel(ZwrocWynikDzielenia.java:8)
        at ZwrocWynikDzielenia.main(ZwrocWynikDzielenia.java:3)
```

11.10.2 Łapanie wyjątków

- Wyjątki łapiemy (obsługujemy) za pomocą mechanizmu **`try`**..**`catch`**..**`finally`**:

```
try {
  // instrukcje ktore moga potencjalnie zakonczyc sie wyjatkiem
} catch (TypWyjatku dowolnaNazwa) {
  // instrukcje, gdy zajdzie wyjątek TypWyjatku
} catch (KolejnyTypWyjatku dowolnaNazwa2) {
  // instrukcje, gdy zajdzie wyjątek KolejnyTypWyjatku
} finally {
  // instrukcje, ktore maja byc wykonane niezaleznie od tego, czy wyjatek
zostal zlapany, czy nie
}
```

- Używając **`try`**..**`catch`** spodziewamy się, że w instrukcjach objętych przez **`try`** coś może pójść nie tak (ale nie musi). To, co powinno się zadziać w przypadku napotkania konkretnego problemu (i tylko wtedy), umieszczamy w sekcji **`catch`**.
- W sekcji **`catch`** podajemy typ wyjątku, jaki chcemy obsłużyć, oraz nazwę zmiennej, za pomocą której będziemy się do tego obiektu-wyjątku odnosić.
- Gdy występuje wyjątek, jego typ dopasowywany jest do listy wyjątkow z obecnych sekcji **`catch`**. Jeżeli wyjątek zostanie dopasowany, wykonywany jest kod powiązany z tą sekcją **`catch`**, która ten konkretny typ wyjątku obsługuje.
- W opcjonalnym bloku **`finally`** możemy umieścić instrukcje, które mają zawsze się wykonać, niezależnie od tego, czy wyjątek zostanie złapany, czy nie. Blok **`finally`**

zazwyczaj używany jest do czyszczenia zasobów, np. zamykania otwartych plików.

- Gdy wystąpi wyjątek, aktualnie wykonywany blok kodu zostaje przerwany. Dalsze wykonanie programu kontynuowane jest w sekcji **catch**, jeżeli jest obecna i dopasowany zostanie do niej typ wyjątku do obsłużenia.
- *Silent catch* to łapanie wyjątków bez ich obsługi – powinniśmy wystrzegać się takich sytuacji, ponieważ mogą prowadzić do trudnych do wykrycia i analizy błędów.
- Wyjątki do złapania definiowane w **catch** muszą być pochodnymi klasy `Throwable`, lub, jak to zazwyczaj ma miejsce, którejś z jej klas pochodnych: `Exception` lub `RuntimeException` (pośrednio bądź bezpośrednio) – inaczej kod się sie skompiluje.
- Zmienne definiowane wewnątrz bloku **try** po zakończeniu tego bloku przestają istnieć. Aby zmienna była dostępna poza **try**, należy ją zdefiniować i zainicjalizować przed **try**:

```
int wynik = 0;

try {
  wynik = podziel(10, 2);
} catch (ArithmeticException e) {
  System.out.println("Blad dzielenia!");
} finally {
  System.out.println("Sekcja finally: wynik wynosi " + wynik);
}

System.out.println("Po try wynik wynosi " + wynik);
```

- Zamiast łapać kilka wyjątków, które może rzucić metoda, możemy złapać wyjątek nadrzędny dla tych wyjątków (którym zawsze jest wyjątek typu `Exception`) i w jednym miejscu obsłużyć wszystkie błędy:

```
try {
  Osoba o = new Osoba("Joanna", "Strzelczyk", -1);
} catch (Exception e) {
  System.out.println(
      "Wystapil blad! Komunikat bledu: " + e.getMessage()
  );
}
```

- W powyższym przykładzie złapaliśmy wszystkie wyjątki korzystając z klasy bazowej wyjątków – `Exception`. Klasa ta jest "bardziej ogólna" niż inne klasy wyjątków, ponieważ jest ich rodzicem (dziedziczenie). Zapisując kod w ten sposób mówimy kompilatorowi:

*"Nieważne czy to będzie `NieprawidlowaWartoscException` czy `NieprawidlowyWiekException`, każdy z nich to `Exception` i chcę je obsłużyc w tej jednej sekcji **catch**".*

- "Wyjątkiem ogólnym" powyżej nie musi być `Exception`, lecz dowolny typ wyjątku, który byłby w hierarchii dziedziczenia używanych przez nas klas wyjątków, które chcemy złapać.
- Możemy także złapać kilka wykluczających się typów wyjątków za pomocą znaku | (pionowa kreska, ang. *pipe*):

```
try {
  Osoba o = new Osoba("Adrian", "Sochacki", 30);
} catch (NieprawidlowaWartoscException | NieprawidlowyWiekException e) {
  System.out.println("Nieprawidlowa wartosc: " + e.getMessage());
}
```

- Kolejność obsługiwania wyjątków w blokach **catch** ma znaczenie – bardziej ogólne wyjątki muszą zawsze następować po mniej ogólnych. Najbardziej ogólnymi wyjątkami są wyjątki typu `Exception` (ponieważ wszystkie wyjątki bazują na tym typie), a mniej ogólne to te, które dziedziczą po klasie `Exception`:

```
try {
  Osoba o = new Osoba(null, "Nowak", 30);
} catch (Exception e) {
  System.out.println(
      "Wystapil blad! Komunikat bledu: " + e.getMessage()
  );
} catch (NieprawidlowyWiekException e) { // blad kompilacji
  System.out.println("Nieprawidlowy wiek!");
}
```

- Ten fragment kodu powoduje błąd kompilacji:

```
Error: java: exception NieprawidlowyWiekException has already been caught
```

- Sekcje **catch** muszą zawierać najbardziej szczegółowe (najniżej w hierarchii dziedziczenia) wyjątki na początku, a najbardziej ogólne na końcu.
- Od wersja Java 1.7 możemy korzystać z nowego mechanizmu *try-with-resources*, który ułatwia pracę z zasobami poprzez ich automatyczne zamykanie po zakończeniu bloku **try**. Klasy, które możemy używać w *try-with-resources* to te klasy, które implementują interfejs `AutoCloseable` lub `Closeable`. Przykład użycia:

```
File f = new File("C:/programowanie/powitanie.txt");

try (FileReader fileReader = new FileReader(f)) {
  int odczytanyZnak;

  while ((odczytanyZnak = fileReader.read()) != -1) {
    System.out.print((char) odczytanyZnak);
  }
} catch (IOException e) {
  System.out.println(e.getMessage());
}
```

11.10.3 Rodzaje wyjątków

- Istnieją dwa typy wyjątków: *Checked exceptions* oraz *Unchecked exceptions*.
- Różnica pomiędzy tymi rodzajami wyjątków jest taka, że potencjał rzucenia przez metodę wyjątku typu Checked musi być umieszczony w klauzuli **throws** w sygnaturze metody. W przypadku wyjątków Unchecked nie musimy tego robić.
- O tym, czy wyjątek to jest rodzaju Checked czy Unchecked decyduje to, czy klasa wyjątku dziedziczy po klasie `RuntimeException`. `RuntimeException` to klasa pochodna od `Exception`.
- Jeżeli klasa wyjątku ma w hierarchii dziedziczenia klasę `RuntimeException`, to jest wyjątkiem typu Unchecked.

- Przykłady klas wyjątków rodzaju Checked:

```
class MojWyjatek extends Exception {}
class MojKolejnyWyjatek extends MojWyjatek {}
```

- Wyjątki te mają następującą hierarchię dziedziczenia (począwszy od klasy `Exception`):

```
Exception
    MojWyjatek
        MojKolejnyWyjatek
```

- `MojKolejnyWyjatek` jest wyjątkiem rodzaju Checked, ponieważ ma w swojej hierarchii typ `Exception` oraz nie ma typu `RuntimeException` (podobnie jak wyjątek `MojWyjatek`).
- Przykład wyjątków Unchecked:

```
class MojRuntimeWyjatek extends RuntimeException {}
class MojKolejnyRuntimeWyjatek extends MojRuntimeWyjatek {}
```

- Te klasy wyjątków mają następującą hierarchię dziedziczenia (począwszy od `Exception`):

```
Exception
    RuntimeException
        MojRuntimeWyjatek
            MojKolejnyRuntimeWyjatek
```

- Klasy te mają w swojej hierarchii klasę `RuntimeException`, są więc wyjątkami rodzaju Unchecked.
- Wyjątki Checked zazwyczaj chcemy obsłużyć, a potencjał ich rzucenia przez metodę zaznaczamy w sygnaturze metody za pomocą słowa kluczowego **`throws`**. Jest to informacja dla każdego użytkownika tej metody:

"Jeśli będziesz korzystał z tej metody, to mogą się pojawić takie a takie wyjątki – powinieneś wziąć je pod uwagę i obsłużyć wedle własnego uznania."

- Wyjątki Unchecked są często spowodowane błędnym stanem naszego programu i mogłoby być ciężko zareagować na nie w odpowiedni sposób.
- Istnieje jeszcze trzeci rodzaj wyjątków, które są pochodnymi klasy `Error`. Wyjątki te to błędy krytyczne, których za bardzo nie da się obsłużyć, np. `OutOfMemoryError`.

11.10.4 Definiowanie i rzucanie wyjątków

- Jeżeli nasza metoda może rzucić wyjątki rodzaju Checked, to musimy zaznaczyć to w sygnaturze tej metody za pomocą słowa kluczowego **`throws`**:

```
public Osoba(String imie, String nazwisko, int wiek)
    throws NieprawidlowaWartoscException, NieprawidlowyWiekException {
```

- Powyższy konstruktor klasy `Osoba` może rzucić wyjątek `NieprawidlowyWiekException` lub `NieprawidlowaWartoscException`.
- Korzystając z metody, która sygnalizuje, że może rzucić wyjątek, musimy:
  - umieścić wykonanie takiej metody w bloku **`try`**..**`catch`** i obsłużyć rzucane wyjątki lub
  - do sygnatury metody, która korzysta z metody, która rzuca wyjątki, także dodać **`throws`**,

oddelegowując w ten sposób obsługę wyjątków do kolejnej metody, która będzie z tej metody korzystać.

- Rzucanie wyjątków odbywa się poprzez użycie słowa kluczowego **throw**, po którym następuje tworzenie obiektu wyjątku takiego typu, jaki chcemy rzucić:

```
throw new IllegalArgumentException("Wiek nie moze byc ujemny.");
```

- **throw** i **throws** to dwa różne słowa kluczowe – pierwsze stosujemy do rzucania wyjątków, a drugie to sygnalizowania w sygnaturze metody, że może ona rzucić pewne wyjątki.
- W momencie rzucenia wyjątku przerywany jest aktualnie wykonywany blok kodu.
- Metoda nie musi zwrócić wartości za pomocą **return**, jeżeli rzuci wyjątek.
- Wyjątki możemy rzucić ponownie za pomocą **throw** (*exception rethrow*):

```
try {
  // ...
  // instrukcje ktora moga spowodowac PewienWyjatek
  // ...
} catch (PewienWyjatek e) {
  // zapisz informacje o bledzie do pliku logu
  log.error("Wystapil blad " + e.getMessage());

  // rzuc wyjatek dalej
  throw e;
}
```

- Wyjątki to obiekty klasy, więc mogą mieć własne konstruktory, metody, i pola.
- Cechą specjalną wyjątków jest jest to, że rozszerzają (pośrednio bądź bespośrednio) klasę `Exception`:

```
class NieprawidlowyWiekException extends Exception {

}
```

- Zgodnie z konwencją nazewniczą klas wyjątków, na końcu nazwy takiej klasy dodajemy słowo *Exception*.
- Możemy w prosty sposób umożliwić zapisywanie w wyjątku komunikatu błędu. Aby to osiągnąć, należy do klasy wyjątku dodać konstruktor, który przyjmie komunikat, a następnie przekaże go do konstruktora klasy nadrzędnej, gdzie zostanie zapamiętany w polu `message`. Komunikat wyjątku będzie później dostępny za pomocą metody `getMessage`.

```
public class NieprawidlowaWartoscException extends Exception {
  public NieprawidlowaWartoscException(String message) {
    // wywolaj konstruktor z klasy bazowej (czyli z Exception)
    super(message);
  }
}
```

11.10.5 Sprawdzanie rzucanych wyjątków i ich rodzaju

- Aby sprawdzić, jakie wyjątki może rzucić metoda, należy zajrzeć do dokumentacji tej metody w Java Doc jeżeli jest to metoda należąca do Biblioteki Standardowej Java, lub do odpowiedniej dokumentacji biblioteki, z której ta metoda pochodzi.
- W komentarzach dokumentacyjnych potencjał rzucenia przez metodę wyjątku opisywany jest za pomocą sekcji `@exception`.
- Sprawdzanie rodzaju wyjątku sprowadza się do analizy jego hierarchii dziedziczenia – jeżeli jest w niej zawarta klasa `RuntimeException`, oznacza to, że jest to wyjątek Unchecked i nie trzeba go obsługiwać w **`try`**..**`catch`**.
- Hierarchię dziedziczenia można sprawdzić w dokumentacji – jeżeli korzystamy z klasy ze Standardowej Biblioteki Java, to informację o klasach znajdziemy w Java Doc.
- Dla wyjątku `IllegalArgumentException` hierarchia dziedziczenia wygląda następująco:

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```

źródło: oficjalna dokumentacja Java Doc – klasa IllegalArgumentException

11.11 Pytania

1. Do czego służą wyjątki?
2. Co to jest *stack trace*?
3. W której z metod wymienionych w poniższym stack trace rzucony został wyjątek?

```
Exception in thread "main" java.lang.IllegalArgumentException
     at Pytania.innaMetoda(Pytania.java:13)
     at Pytania.pewnaMetoda(Pytania.java:9)
     at Pytania.main(Pytania.java:5)
```

4. Jak w języku Java obsługuje się wyjątki?
5. Czy musimy stosować obsługę wyjątków jeżeli metoda, z której chcemy skorzystać, może rzucić wyjątek?
6. Do czego służy sekcja `finally` i czy jest wymagana?
7. Jak rzuca się wyjątki?
8. Do czego służy słowo kluczowe `throws`?
9. Jaką regułę muszą spełniać klasy, aby były klasami wyjątków?
10. Co trzeba zrobić, aby pobrać komunikat skojarzony z wyjątkiem?
11. Czym różnią się wyjątki Checked oraz Unchecked?
12. Jakiego rodzaju (Unchecked / Checked) są poniższe wyjątki (musisz zajrzeć do dokumentacji Biblioteki Standardowej Java – Java Doc)?
    a) `EOFException`
    b) `ClassCastException`
    c) `DateTimeParseException`
    d) `SQLException`
13. Czy wyjątek `NullPointerException` to wyjątek rodzaju Checked czy Unchecked?
14. Co to jest *silent catch* (*połykanie wyjątków*)?
15. Do czego służy *try-with-resources* i jak się tego mechanizmu używa?
16. Czy poniższy kod jest poprawny?

```
class MojWyjatekException {

}

public class Pytania {
  public static void main(String[] args) {
    try {
      pewnaMetoda();
    } catch (MojWyjatekException e) {
      System.out.println("Wystapil blad.");
    }
  }

  public static void pewnaMetoda() {
    throw new MojWyjatekException();
```

```
  }
}
```

17. Czy poniższe metody skompilują się bez błędów?

```
public static void main(String[] args) {
  try {
    int x = getInt();
  } catch (InputMismatchException e) {
    System.out.println("Wystapil blad: " + e.getMessage());
  }

  if (x >= 0) {
    System.out.println("Podana liczba jest nieujemna.");
  } else {
    System.out.println("Podana liczba jest ujemna.");
  }
}

public static int getInt() {
  return new Scanner(System.in).nextInt();
}
```

18. Czy poniższe metody skompilują się bez błędów?

```
public static void pewnaMetoda() {
  try {
    innaMetoda();
  } catch (Exception e) {
    System.out.println(e.getMessage());
  } catch (MojWyjatekException e) {
    System.out.println(e.getMessage());
  }
}

public static void innaMetoda() throws MojWyjatekException {
  // pewne instrukcje mogace rzucic wyjatek
}
```

gdzie `MojWyjatekException` to:

```
class MojWyjatekException extends Exception {

}
```

19. Czy poniższa metoda skompiluje się bez błędów?

```
public static void pewnaMetoda() throw IllegalArgumentException {
  // pewne instrukcje moga rzucic wyjatek
}
```

20. Czy poniższe metody skompilują się bez błędów?

```
public static void main(String[] args) {
  int wynik;

  try {
    int x = getInt();
    wynik = x * x;
  } catch (InputMismatchException e) {
    System.out.println("Wystapil blad: " + e.getMessage());
  }

  System.out.println(wynik);
}

public static int getInt() {
  return new Scanner(System.in).nextInt();
}
```

21. Czy poniższy kod się skompiluje?

```
public class Pytania {
  public static void main(String[] args) {
    pewnaMetoda(null);
  }

  public static String pewnaMetoda(String str)
      throws NullPointerException {

    return str.toUpperCase();
  }
}
```

22. Czy poniższy kod się skompiluje?

```
class MojWyjatekException extends Exception {
}

public class Pytania {
  public static void main(String[] args) throws MojWyjatekException {
    throw new MojWyjatekException("Nic nie robie.");
  }
}
```

23. Czy poniższe metody skompilują się bez błędów?

```
public static void pewnaMetoda() {
  throw new IllegalArgumentException();
}

public static void innaMetoda() {
  throw new Exception();
}

public static void kolejnaMetoda() throws IOException {

}
```

24. Czy poniższy kod skompiluje się poprawnie?

```
class MojWyjatekException extends Exception {

}

public class Pytania {
  public static void main(String[] args) {
    try {
      pewnaMetoda();
    } catch (MojWyjatekException | Exception e) {
      System.out.println(e.getMessage());
    }
  }

  public static void pewnaMetoda() {
    // pewne instrukcje
  }
}
```

25. Czy poniższe metody skompilują się poprawnie?

```
public static void pewnaMetoda() {
  try {
    innaMetoda();
  } catch (Exception e) {
    System.out.println("Wystapil blad: " + e.getMessage());
    throw e;
  }
}

public static void innaMetoda() throws Exception {
  throw new Exception();
}
```

26. Czy poniższy kod skompiluje się bez błędów? Jeżeli tak, to co zobaczymy na ekranie?

```
public class Pytania {
  public static void main(String[] args) {
    try {
      pewnaMetoda(0);
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }

  public static int pewnaMetoda(int x) {
    if (x == 0) {
      throw new IllegalArgumentException();
    }

    return x * x;
  }
}
```

11.12 Zadania

11.12.1 Silnia z obsługą ujemnych liczb

Napisz metodę, która będzie zwracać silnię podanej jako argument liczby. Metoda powinna rzucać wyjątek rodzaju Checked zdefiniowanego przez Ciebie typu `BlednaWartoscDlaSilniException` w przypadku, gdy jej argument będzie ujemny. Skorzystaj z tej metody w `main`, obsługując potencjalny wyjątek.

11.12.2 Klasa Adres z walidacją danych

Napisz program z klasą `Adres`, która będzie miała podane poniżej pola, które będą ustawiane w konstruktorze klasy `Adres`. Konstruktor powinien sprawdzić wszystkie podane wartości i rzucić wyjątek `NieprawidlowyAdresException` rodzaju Checked, jeżeli któraś z wartości będzie nieprawidłowa. **Uwaga**: komunikat rzucanego wyjątku powinien zawierać informację o wszystkich nieprawidłowych wartościach przekazanych do konstruktora – dla przykładu, jeżeli `ulica` i `miasto` będą miały wartość `null`, to komunikat wyjątku powinien być następujący: *"Ulica nie może być nullem. Miasto nie może być nullem"*. Pola klasy:

1. `String ulica` – wartość nieprawidłowa to `null`,
2. `int numerDomu` – wartość nieprawidłowa to liczba <= 0,
3. `String kodPocztowy` – wartość nieprawidłowa to `null`,
4. `String miasto` – wartość nieprawidłowa to `null`.

11.12.3 Liczba znaków w pliku

Skorzystaj z *try-with-resources* w programie, które pobierze od użytkownika lokalizację pliku z rozszerzeniem `.txt` na dysku, a nastepnie wypisze na ekran liczbę znaków, z których składa się ten plik. Weź pod uwagę, że podany przez użytkownik plik może nie istnieć lub być plikiem o innym rozszerzeniu. Skorzystaj z klas `File` oraz `FileReader`.

Uwaga: wynik liczenia znaków nie będzie się zgadzał z liczbą *widocznych* w pliku znaków. Na końcu każdej linii, po której następuje kolejna linia, znajdują się (w systemie Windows) dwa dodatkowe znaki końca linii. Weź to pod uwagę testując swój program (w systemie Linux będzie to jeden dodatkowy znak na linię). Pamiętaj także, że spacje i tabulatory to także znaki!

11.12.4 Implementacja stosu

Stos to rodzaj kolekcji do przechowywania danych typu *FILO – First In, Last Out*. Gdy dodajemy na stos kilka elementów, to mamy jedynie dostęp do tego ostatniego. Jeżeli chcemy odnieść się do pierwszego dodanego na stos elementu, musimy najpierw "zdjąć" elementy dodane na stos po nim – stąd określenie *First In, Last Out*. Stos można porównać do kartek położonych jedna na drugiej – jeżeli chcemy kartkę ze spodu stosu, to musimy najpierw zdjąć kartki leżące na niej.

Napisz klasę `Stack` (*w opisie poniżej nazywaną stosem*). Ma ona za zadanie przechowywać liczby typu `int`. Klasa `Stack` powinna posiadać:

1. Konstruktor, który przyjmuje jako argument liczbę typu `int` – maksymalną liczbę elementów, które ten stos może przechowywać. Jeżeli podamy ujemną liczbę, powinien zostać rzucony wyjątek `IllegalArgumentException`.
2. Metody:

a) `push` – dodaje przekazaną jako argument liczbę typu **`int`** do stosu, jeżeli jest w nim jeszcze miejsce – jeżeli nie, rzuca nowy zdefiniowany wyjątek rodzaju Unchecked o nazwie `StackFullException`,

b) `pop` – usuwa ze stosu ostatnio dodany element i zwraca go – jeżeli stos był pusty, rzuca nowy zdefiniowany wyjątek rodzaju Unchecked o nazwie `StackEmptyException`,

c) `clear` – czyści stos,

d) `top` – zwraca ostatnio dodany do stosu element – jeżeli stos był pusty, rzuca wyjątek typu `StackEmptyException`,

e) `size` – zwraca liczbę elementów aktualnie przechowywanych w stosie.

Utwórz kilka obiektów typu `Stack` i przetestuj ich działanie. Obsłuż w **`try`**..**`catch`** potencjalne wyjątki.