# Nauka programowania w języku Java

## Odpowiedzi na pytania i zadania

*v. 0.1.2, 25-06-2020*



kontakt:

przemyslaw.kruglej@gmail.com

www:

https://kursjava.com

https://przemyslawkruglej.com

https://craftsmanshipof.software

# Spis Treści

Ten dokument zawiera odpowiedzi na pytania i zadania znajdujące się na końcach rozdziałów kursu języka Java, którego jestem autorem.

Kurs Java dostępny jest za darmo w internecie na stronie:

https://kursjava.com

Na powyższej stronie znajdziesz także inne darmowe kursy, takie jak *Podstawy linii poleceń*, *IntelliJ IDEA w akcji*, *Kurs tworzenia gier w języku Java*, *Podstawy Maven*, i inne.

Treść tego dokumentu znajduje się także na stronie:

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

Kody źródłowe rozwiązań do zadań dostępne są na stronie przykładów z kursu na GitHubie:

GitHub – repozytorium przykładów kursu Java i rozwiązań do zadań

**Będę wdzięczny za poinformowanie o znalezionych błędach oraz sugestiach** – mój e-mail kontaktowy to:

przemyslaw.kruglej@gmail.com

Miłego czytania! :)

*Przemysław Kruglej*

# 1 Rozdział I – Wstęp

## 1.1 Pytania

1. Czym jest proces kompilacji?

> Jest to proces, w którym program, zwany *kompilatorem*, zamienia kod źródłowy zrozumiały dla człowieka, na kod zrozumiały dla komputera bądź na *kod pośredni*, który może zostać *zinterpretowany* przez inny program.

2. Jak nazywa się kompilator Java i jak się go używa?

> Kompilator języka Java nazywa się *javac*. Aby go użyć, w linii poleceń wywołujemy komendę `javac`, której, jako argument, przekazujemy nazwę pliku z kodem źródłowym, który ma zostać skompilowany, np.:

```
javac HelloWorld.java
```

3. Co powstaje w wyniku kompilacji kodu Java?

> W wyniku kompilacji kodu Java powstaje tzw. *bytecode*, który może zostać uruchomiony w *Maszynie Wirtualnej Java* i w niej zinterpretowany.

4. Jak uruchomić kod Java?

> Należy z linii komend wywołać program *java*, któremu przekażemy jako argument nazwę skompilowanej klasy (bez podawania rozszerzenia `.class`), na przykład:

```
java HelloWorld
```

5. Czym różni się kompilator języka Java od Maszyny Wirtualnej Java?

> Kompilator ma za zadanie skompilować kod Java do bytecode'u, natomiast Maszyna Wirtualna Java ma ten skompilowany bytecode zinterpretować i wykonać.

6. Skąd wziąć kompilator Java i Maszynę Wirtualną Java?

> Należy zainstalować JDK – *Java Development Kit*, który jest zestawem aplikacji wymaganych do tworzenia i uruchamiania aplikacji napisanych w języku 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?");
  }
}
```

> Plik z tym kodem źródłowym powinien nazywać się `Zagadka.java`

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

```
javac TestowyProgram
```

> Nie jest to poprawna komenda, ponieważ, korzystając z kompilatora języka Java, powinniśmy jako argument przekazać nazwę pliku wraz z rozszerzeniem.

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

```
java TestowyProgram.class
```

> Nie jest to poprawna komenda, ponieważ wywołując Maszynę Wirtualną Java powinniśmy podać argument bez rozszerzenia `.class`.

10. Jakie jest specjalne znaczenie metody `main`?

> Jest to specjalna metoda, ponieważ to od niej rozpoczyna się wykonanie każdego programu napisanego w języku Java.

11. Jak wypisać na ekran tekst "Witajcie!"?

> Należy skorzystać z instrukcji `System.out.println`. W nawiasach, zawarty w cudzysłowach, powinien znaleźć się komunikat do wyświetlenia, na przykład:

```
System.out.println("Pewien komunikat.");
```

12. Jaki będzie efekt próby kompilacji każdego z poniższych programów?

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

> Kompilacja zakończy się błędem, ponieważ na końcu drugiej linii brakuje nawiasu otwierającego ciało metody `main`.

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

Kompilacja zakończy się błędem, ponieważ na końcu trzeciej linii brakuje średnika kończącego instrukcję wypisującą na ekran komunikat.

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

Kompilacja zakończy się sukcesem. Ciała metod mogą być puste.

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

Kompilacja zakończy się błędem, ponieważ komunikat do wyświetlenia jest ujęty w apostrofy, zamiast w cudzysłowy.

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

Kompilacja zakończy się błędem, ponieważ komunikat do wyświetlenia jest ujęty w specjalne znaki cudzysłowów, używane w edytorach dokumentów, zamiast w zwykłe cudzysłowy.

13. Które z poniższych stwierdzeń jest prawdziwe?
    1. System Windows rozumie bytecode.

Nieprawda, to Maszyna Wirtualna Java rozumie bytecode.

    2. Program `javac` jest potrzebny do uruchomienia skompilowanego kodu Java.

Nieprawda, program `javac` służy do kompilacji kodu Java, a nie do jego uruchamiania.

    3. Stringi (łańcuchy tekstowe) powinny być zawarte w apostrofach.

Nieprawda, łańcuchy tekstowe powinny być ujęte w cudzysłowy.

4. Ciało metody zawarte jest pomiędzy nawiasami: ( )

> Nieprawda, ciało metody powinno być zawarte w nawiasach klamrowych { }

5. Jeżeli kompilator napotka problemy w naszym kodzie, to nie wygeneruje pliku z rozszerzeniem `.class` z naszym skompilowanym kodem.

> Prawda.

## 1.2 Zadania

### 1.2.1 Wypisz imię

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

Rozwiązanie jest podobne do pierwszego programu, jaki napisaliśmy w języku Java – programu `HelloWorld`.

Możemy, wzorując się na tamtym przykładzie, utworzyć plik o nazwie np. `WypiszImie.java`, następnie skopiować kod klasy `HelloWorld`, zmienić nazwę klasy na `WypiszImie`, a na końcu zmienić wypisywany na ekran komunikat:

```
public class WypiszImie {
  public static void main (String[] args) {
    System.out.println("Czesc, Przemek");
  }
}
```

### 1.2.2 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?*

Zacznijmy od programu, który jest poprawny:

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

Ten program kompiluje się poprawnie, a w wyniku jego uruchomienia na ekranie zobaczymy komunikat `Witam!`.

Zmieńmy plik zgodnie z wymaganiami opisanymi w zadaniu – usuwamy ostatni znak }:

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

Próba kompilacji tej wersji programu kończy się następującym błędem:

```
Powitanie.java:4: error: reached end of file while parsing
}
^
1 error
```

Kompilator nie jest w stanie skompilować kodu, ponieważ jest on niepoprawny – brakuje klamry

kończącej klasę `Powitanie`.

**Pozostało ostatnie pytanie z zadania** – jaki będzie efekt próby uruchomienia programu `Powitanie`?

W wyniku wywołania komendy `java Powitanie` ponownie zobaczymy na ekranie komunikat `Witaj!`.

**Dlaczego tak się stało?** Przecież nasz program się nie kompiluje! Podczas próby kompilacji niedziałającego programu, wskutek błędnego kodu, plik z bytecodem `.class` naszego programu nie został wygenerowany. Gdy więc wywołaliśmy komendę `java Powitanie`, wykonana została poprzednia wersja naszego programu zapisana w bytecodzie w pliku `Powitanie.class`, który utworzony został po kompilacji pierwszej, poprawnej wersji programu `Powitanie`.

Jeżeli jednak usunąłeś/usunęłaś wcześniej wygenerowany plik `.class`, to zobaczysz następujący komunikat:

```
Error: Could not find or load main class Powitanie
Caused by: java.lang.ClassNotFoundException: Powitanie
```

# 2 Rozdział II – Komentarze i formatowanie kodu

## 2.1 Pytania

1. Jak zapisujemy każdy rodzaj komentarza w Javie?

> Komentarze jednolinijkowe zaczynamy od dwóch znaków slash //
>
> Komentarze wielolinijkowe zaczynamy od znaków /* a kończymy znakami */
>
> Komentarze dokumentacyjne zaczynamy od znaków /** a kończymy znakami */

2. Które komentarze mogą być zagnieżdżone?

> Komentarze jednolinijkowe mogą być w sobie zagnieżdżone.
>
> Komentarze wielolinijkowe/dokumentacyjne nie mogą być w sobie zagnieżdżone.
>
> Komentarze jednolinijkowe mogą być zagnieżdżone w komentarzach wielolinijkowych i na odwrót.

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!");
  }
}
```

> Nazwa klasy powinna być zapisana CamelCasem. Pierwsza litera nazwy klasy także powinna być wielka.
>
> Zakomentowany kod powinien zostać usunięty.
>
> Wcięcie przed instrukcją wypisująca na ekran tekst powinno być o jeden poziom większe.
>
> Kod poprawkach, mógłby wyglądać następująco:

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

4. Która z poniższych nazw zapisana jest Camel-Casem?
    1. nazwa
    2. MojaNazwa
    3. mojanazwa
    4. Wiadomosc

> Tylko pierwsza – chociaż zawiera ona tylko jedno słowo, to jest zgodna z konwencją. Nazwa `MojaNazwa` powinna mieć pierwszą małą literę, tak samo jak nazwa `Wiadomosc`. Nazwa `mojanazwa` powinna mieć wielką pierwszą literę drugiego słowa.

5. Która z nazw klas jest poprawna?
    1. HelloWorld
    2. helloworld
    3. helloWorld
    4. Helloworld

> Jest to podchwytliwe pytanie – wszystkie powyższe nazwy klas są poprawne, chociaż tylko jedna z nich – `HelloWorld` – jest zgodna z konwencją nazewnictwa klas w języku Java (czyli Camel-Case + wielka pierwsza litera pierwszego wyrazu).

## 2.2 Zadania

### 2.2.1 Dopisz komentarze

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

Przykład rozwiązania z komentarzami:

```
// poczatek klasy
public class HelloWorld {
  public static void main(String[] args) {
    /*
      wypisujemy na ekran wiadomosc
     */
    System.out.println("Witaj Swiecie!");
  }
} // koniec klasy
```

### 2.2.2 Brak main

*W tym samym programie zakomentuj całą metodę `main`.*

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

Program z zakomentowaną metodą `main`:

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

Klasa skompiluje się z zakomentowaną metodą `main`, ale po uruchomieniu, wykonanie programu zakończy się następującym błędem:

```
Error: Main method not found in class HelloWorldBezMain, please define the
main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Wynika to z faktu, że Maszyna Wirtualna Java próbowała wywołać metodę `main` w naszej klasie, która to metoda jest punktem startowym wykonywania naszych programów. Metoda nie została znaleziona, więc Maszyna Wirtualna Java zgłosiła błąd.

# 3 Rozdział III – Zmienne

## *3.1 Podstawy zmiennych – zadania*

### 3.1.1 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.*

Przykładowe rozwiązanie do tego zadania:

```
public class DodawanieLiczb {
  public static void main(String[] args) {
    int x, y, z;

    x = 5;
    y = 10;
    z = x + y;

    System.out.println("Wynik dodawania: " + z);
  }
}
```

### 3.1.2 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.*

Przykładowe rozwiązanie do tego zadania:

```
public class ObwodTrojkata {
  public static void main(String[] args) {
    int a = 5;
    int b = 6;
    int c = 7;

    int obwodTrojkata = a + b + c;

    System.out.println("Obwod trojkata wynosi: " + obwodTrojkata);
  }
}
```

### 3.1.3 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.*

Przykładowe rozwiązanie do tego zadania:

```
public class AktualnaData {
  public static void main(String[] args) {
    int dzien = 7;
    int miesiac = 4;
    int rok = 2019;

    System.out.println(
        "Dzisiaj jest " + dzien + "-" + miesiac + "-" + rok
    );
  }
}
```

### 3.1.4 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.*

Przykładowe rozwiązanie do tego zadania:

```
public class LiczbaMiesiecyWRoku {
  public static void main(String[] args) {
    final int LICZBA_MIESIECY_W_ROKU = 12;

    System.out.println(
        "Liczba miesiecy w roku to " + LICZBA_MIESIECY_W_ROKU
    );
  }
}
```

### 3.1.5 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.*

Przykładowe rozwiązanie do tego zadania:

```
public class Inicjaly {
  public static void main(String[] args) {
    char p = 'P';
    char k = 'K';

    System.out.println(p + "." + k + ".");
  }
}
```

Pojedyncze znaki, które przechowuje typ **`char`**, ujmujemy w cudzysłowy. Podczas wypisywania na ekran, znaki powinniśmy połączyć z łańcuchami tekstowymi, które, w tym przypadku, są kropkami po każdej literze inicjału.

## 3.2 Pytania

1. Jak definiujemy zmienne?

> Zmienne definiujemy podając, kolejno, ich typ, nazwę, oraz ewentualną wartość początkową. Możemy zdefiniować więcej, niż jedną zmienną na raz, oddzielając ich nazwy przecinkami:

```
int promienKola = 8;
char a;
double x, y;
```

2. Czy zmiennej trzeba nadać wartość początkową w momencie definicji?

> Nie trzeba, ale przed użyciem takiej zmiennej należy jej przypisać wartość. Jeżeli spróbujemy użyć zmiennej zdefiniowanej w metodzie zanim nadamy tej zmiennej wartość, to nasz program w ogóle się nie skompiluje:

```
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);
  }
}
```

Błąd kompilacji, jaki zobaczymy na ekranie:

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

3. Jakie są zasady nazewnictwa zmiennych (i innych obiektów) w Javie?

> Nazwy w Javie:

- muszą zaczynać się od litery, podkreślenia _ bądź znaku dolara $
- mogą zawierać cyfry, ale nie mogą się od cyfr zaczynać,
- nie mogą być takie same, jak nazwy zastrzeżone w języku Java (np. `class`, `public`, `void` itd.).

4. Czy wielkie litery są rozróżniane w nazwach w Javie?

> Tak – dla przykładu, nazwy `pi` oraz `PI` to dwie różne nazwy w języku Java.

5. Jakie typy podstawowe są dostępne w Javie? Czym się różnią?

Java oferuje 8 typów podstawowych: **`boolean`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, oraz **`char`**. Różnią się one rodzajem danych, jakie mogą przechowywać, a także ich zakresem.

6. Jak nazywamy zwykłe wartości zapisane w kodzie, takie jak `3.14`, `25`, `'a'`?

Wartości te nazywamy *literałami*.

7. Do czego służą stałe, jak się je definiuje i nazywa?

Stałe służą do przechowywania wartości, które nigdy nie powinny się zmienić w trakcie wykonywania programu, np. stała o nazwie `PI` powinna mieć wartość `3.14`.

Stałe definiujemy podobnie, jak zmienne, z tym, że dodajemy na początek słowo kluczowe **`final`**. Stałe nazywamy używając wielkich liter i oddzielając od siebie słowa znakiem podkreślenia:

```
final int LICZBA_DNI_W_TYGODNIU = 7;
```

8. Czy wszystkie operatory mają takie same priorytety?

Nie, różne operatory mają różne priorytety. Dla przykładu, operator * (mnożenie) ma wyższy priorytet, niż operator + (dodawanie), więc w wyrażeniu `2 + 2 * 2`, w pierwszej kolejności wykonane zostanie mnożenie.

9. Czy można zmieniać priorytet operatorów?

Można, korzystając z nawiasów. Aby w wyrażeniu `2 + 2 * 2` najpierw wykonać dodawanie, możemy zapisać je z nawiasami w następujący sposób: `(2 + 2) * 2`.

10. Jaki wynik daje użycie operatora dzielenia / gdy jego argumenty (operandy) są liczbami całkowitymi?

Operatora dzielenia / daje w wyniku liczbę całkowitą, zaokrągloną w dół, gdy oba jego argumenty są liczbami całkowitymi. Wartość wyrażenia `10 / 4` będzie wynosiła `2`.

11. Co to jest rzutowanie typów?

Rzutowanie typów to traktowanie wartości jednego typu jako wartości innego typu. Aby wykonywać rzutowanie, zapisujemy oczekiwany typ w nawiasach przed wartością, np. `(double)zmiennaTypuInt` spowoduje, że wartość zmiennej `zmiennaTypuInt` będzie traktowana jako liczba rzeczywista **`double`**.

12. Jak otrzymać w wyniku dzielenia liczbę rzeczywistą?

Możemy albo zapisać którąś z liczb jako liczba rzeczywista, albo skorzystać z rzutowania – w drugiej linii poniżej korzystamy z rzutowania, a w trzeciej – pierwszy argument operatora / to liczba rzeczywista, więc wynikiem dzielenia także będzie liczba rzeczywista:

```
System.out.println(10 / 4); // wypisze 2 - dzielenie calkowite
System.out.println((double)10 / 4); // wypisze 2.5
System.out.println(10.0 / 4); // wypisze 2.5
```

13. Do czego służą operatory +=, -= itp.?

Są to skrótowe operatory przypisania. Dla przykładu, operator += powoduje dodanie do zmiennej po lewej stronie operatora wartości wyrażenia po jego prawej stronie:

```
int a = 5;
a += 10; // a bedzie mialo wartosc 5 + 10, czyli 15
```

14. Do czego służą operatory ++ i -- oraz czym się różnią ich post- i pre-fixowe wersje?

Operatory te służą do zwiększania oraz zmniejszania wartości zmiennej o 1. Post-fixowe wersje tych operatorów różnią się tym od pre-fixowych, że najpierw wracają wartość zmienianej zmiennej, a dopiero potem ją zmieniają. Pre-fixowe wersje najpierw zmieniają wartość zmiennej, a potem zwracają jej wartość:

```
int x = 2;
int y = 2;
System.out.println(x++); // wypisze 2
System.out.println(++y); // wypisze 3
```

15. Jaką wartość będzie miała zmienna `x`, a jaką zmienna `y`, w poniższym przykładzie?

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

Kod w ogóle się nie skompiluje, ponieważ operator ++ oczekuje zmiennej jako argumentu, a nie literału liczbowego.

16. Do czego służy typ `String`?

Typ String służy do przechowywania stringów, czyli łańcuchów tekstowych.

17. Jak połączyć ze sobą dwa łańcuchy tekstowe?

Należy skorzystać z operatora + jak w przykładzie poniżej:

```
String komunikat = "Witaj " + "Swiecie!";
```

18. Co zostanie wypisane w wyniku wykonania poniższego programu?

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

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

Wypisany zostanie komunikat `"Witajcie!"`. Co prawda użyliśmy na zmiennej `powitanie` metody, która zamienia znaki w stringu z małych na duże, ale nie przypisaliśmy wyniku działania tej metody do żadnej zmiennej. Metoda `toUpperCase` nie modyfikuje oryginalnej zmiennej, lecz zwraca nową wartość z małymi literami zamienionymi na wielkie.

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);
  }
}
```

Kod w ogóle się nie skompiluje, ponieważ nie zainicjalizowaliśmy zmiennej `x` żadną wartością. Kompilator jest to w stanie wykryć i zgłosi błąd już na etapie kompilacji.

20. Jaką wartość będzie miała zmienna `liczba`?

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

Kod w ogóle się nie skompiluje. Wynik działania `2.5 * 20` to liczba rzeczywista (ponieważ jeden z argumentów operatora * to liczba rzeczywista), a zmienna, do której ten wynik próbujemy przypisać, to zmienna typu **`int`**. Zmienne typu **`int`** mogą przechowywać jedynie liczby całkowite.

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

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

Niepoprawna jest nazwa `class`, ponieważ jest to słowo kluczowe w języku Java i nie może być używane jako nazwa zmiennej. `liczbaPrzedmiotow#` także jest niepoprawną nazwą, ponieważ zawiera niedozwolony znak #. `60godzin` także jest nieprawidłową nazwą,

ponieważ nazwy nie mogą zaczynać się od cyfr.

Nazwa `ZNAK` jest co prawda poprawna, ale powinna zostać zapisana małymi literami. Tylko stałe zapisujemy wielkimi literami. Nazwa `$saldo` jest poprawna – nazwy mogą zaczynać się od znaku dolara.

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);
  }
}
```

Kod w ogóle się nie skompiluje – kompilator zgłosi błąd, że nie wie, czym są `X` oraz `Y`. Zmienne, które wcześniej zdefiniowaliśmy, zapisaliśmy małymi literami – w języku Java wielkość znaków ma znaczenia.

23. Czy poniższy kod skompiluje się poprawnie?

```
char z = "Z";

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

Nie, ponieważ do zmiennej typu `char` próbujemy przypisać łańcuchów znaków. Chociaż złożony tylko z jednego znaku, nadal jest to łańcuch znaków, a nie pojedynczy znak. Aby kod był poprawny, do zmiennej powinien zostać przypisany znak ujęty w apostrofy `'Z'`.

24. Jaka wartość zostanie wypisana?

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

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

Na ekranie zostanie wypisana wartość `1.5`. Skorzystaliśmy z rzutowania, by liczba `15` została potraktowana jako liczba rzeczywista, więc jeden z argumentów operatora / jest liczbą rzeczywistą i wynik też będzie liczbą rzeczywistą.

25. Jaką wartość będzie miała zmienna `y`?

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

Kod się nie skompiluje. Do zmiennej `y` próbujemy przypisać wartość sumy zmiennych `x` oraz `y`. Zmienna `y` nie ma jeszcze przypisanej żadnej wartości, więc nie może być użyta do wyznaczenia swojej własnej wartości.

## 3.3 Zadania

### 3.3.1 Obwód trójkąta z pobranych danych

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

Aby pobrać od użytkownika liczby całkowite, należy skorzystać z funkcjonalności przedstawionej pod koniec trzeciego rozdziału – dodać do kodu programu instrukcję `import` oraz metodę `getInt`.

Przykładowe rozwiązania korzystające z metody `getInt`:

```
import java.util.Scanner;

public class ObwodTrojkataPobraneDane {
  public static void main(String[] args) {
    int x, y, z;
    int obwodTrojkata;

    System.out.println("Podaj pierwszy bok trojkata:");
    x = getInt();

    System.out.println("Podaj drugi bok trojkata:");
    y = getInt();

    System.out.println("Podaj trzeci bok trojkata:");
    z = getInt();

    obwodTrojkata = x + y + z;

    System.out.println(
        "Obwod trojkata o tych bokach wynosi " + obwodTrojkata
    );
  }

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

Przykładowe wykonanie powyższego programu (na biało zaznaczone zostały dane wprowadzone przez użytkownika z klawiatury):

```
Podaj pierwszy bok trojkata:
7
Podaj drugi bok trojkata:
8
Podaj trzeci bok trojkata:
9
Obwod trojkata o tych bokach wynosi 24
```

### 3.3.2 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*

Tym razem musimy skorzystać z metody `getString`, wprowadzonej pod koniec trzeciego rozdziału, dzięki której będziemy mogli pobierać od użytkownika słowa (musimy także skorzystać z instrukcji `import`):

```
import java.util.Scanner;

public class OdwrotnaKolejnoscPobraneSlowa {
  public static void main(String[] args) {
    String slowo1, slowo2, slowo3;

    System.out.println("Podaj pierwsze slowo:");
    slowo1 = getString();

    System.out.println("Podaj drugie slowo:");
    slowo2 = getString();

    System.out.println("Podaj trzecie slowo:");
    slowo3 = getString();

    System.out.println(slowo3 + ", " + slowo2 + ", " + slowo1);
  }

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

Przykładowe wykonanie powyższego programu (na biało zaznaczone są słowa podane przez użytkownika):

```
Podaj pierwsze slowo:
Ala
Podaj drugie slowo:
ma
Podaj trzecie slowo:
kota
kota, ma, Ala
```

### 3.3.3 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/en/java/javase/12/docs/api/java.base/java/lang/String.html*

Aby dowiedzieć się, z ilu znaków składa się wartość zmiennej typu `String`, należy skorzystać z metody `length` udostępnianej przez typ `String`, która zwraca liczbę znaków w stringu. W poniższym przykładzie korzystamy także z metody `getString`, poznanej pod koniec trzeciego rozdziału, do pobrania słowa od użytkownika:

```
import java.util.Scanner;

public class LiczbaZnakowWSlowie {
  public static void main(String[] args) {
    String slowo;

    System.out.println("Podaj slowo:");
    slowo = getString();

    System.out.println("To slowo ma " + slowo.length() + " znakow.");
  }

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

Przykładowe wykonanie powyższego programu (na biało zaznaczono słowo podane przez użytkownika):

```
Podaj slowo:
nauka
To slowo ma 5 znakow.
```

### 3.3.4 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);
  }
}
```

Aby otrzymać w wyniku dzielenia liczbę rzeczywiste w powyższym przypadku, należy zrzutować wartość jednej ze zmiennych na typ `double` – gdy jeden z argumentów operatora / będzie liczbą rzeczywistą, to i wynik dzielenia będzie liczbą rzeczywistą:

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

    double wynik = (double)x / y;

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

Wynik działania powyższego programu:

```
2.5
```

### 3.3.5 Wielkie litery

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

Ponownie korzystamy z metody `getString` w celu pobrania słowa od użytkownika:

```
import java.util.Scanner;

public class WielkieLitery {
  public static void main(String[] args) {
    String slowo;

    System.out.println("Podaj slowo:");
    slowo = getString();

    System.out.println(slowo.toUpperCase());
  }

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

Przykładowe wykonanie tego programu (na biało zaznaczono słowo podane przez użytkownika):

```
Podaj slowo:
nauka
NAUKA
```

### 3.3.6 Pole koła o podanym promieniu

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

W tym programie skorzystamy z poznanej metody `getInt` w celu pobrania od użytkownika promienia koła:

```
import java.util.Scanner;

public class PoleKolaOPodanymPromieniu {
  public static void main(String[] args) {
    int promienKola;
    double poleKola;

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

    poleKola = 3.14 * promienKola * promienKola;

    System.out.println(
        "Pole kola o tym promieniu wynosi: " + poleKola
    );
  }

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

Przykładowe wykonanie powyższego programu (na biało zaznaczono promień koła podany przez użytkownika):

```
Podaj promien kola:
5
Pole kola o tym promieniu wynosi: 78.5
```

# 4 Rozdział IV – Instrukcje warunkowe

## 4.1 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");
}
```

> Kod jest niepoprawny, ponieważ warunek instrukcji `if` powinien być ujęty w nawiasy `(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");
}
```

> Kod jest niepoprawny, ponieważ sekcja `else` instrukcji warunkowej powinna być zawsze na końcu.

```
int a = 5;

if (a) {
  System.out.println("a = 5");
}
```

> Kod jest niepoprawny, ponieważ wartość wyrażenia używanego w instrukcji warunkowej powinna zawsze mieć jedną z dwóch wartości: `true` bądź `false`.

```
boolean b = true;

if (b) {
  System.out.println("Warunek spelniony.");
}
```

> Kod jest poprawny – możemy użyć zmiennej typu `boolean` jako warunku instrukcji warunkowej.

```
int y = 5;

if (y = 5) {
  System.out.println("y = 5");
}
```

> Kod jest niepoprawny, ponieważ skorzystaliśmy z nieprawidłowego operatora – zamiast użyć operatora relacyjnego ==, który porównuje do siebie swoje argumentu i zwraca wartość `true` bądź `false`, użyty został operator przypisania =. Powyższy kod w ogóle się nie skompiluje.

```
int c = 5;
boolean czyDodatnia = c > 0;

if (czyDodatnia) {
  System.out.println("Warunek spelniony.");
}
```

> Powyższy kod jest poprawny – możemy przypisać do zmiennej typu `boolean` wynik wyrażenia, które korzysta z operatora relacyjnego.

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.");
}
```

> Dla wartości `10` wypisane zostanie:

```
10
20
```

> Dla wartości `20` wypisane zostanie:

```
20
```

Dla wartości `30` wypisane zostanie:

```
30
Inna wartosc.
```

Dla innej wartości, na przykład `50`, wypisane zostanie:

```
Inna wartosc.
```

Dla wartości `10` i `30` wypisane zostały po dwie wartości, ponieważ niektóre bloki **`case`** instrukcji **`switch`** nie korzystają ze słowa kluczowego **`break`**. W takim przypadku, gdy wartość zostanie dopasowana, wykonywane zostają instrukcje z bloków **`case`** (oraz bloku **`default`**) znajdujących się poniżej. Dla wartości `10` wypisywanie komunikatów zakończyło się na `20`, ponieważ sekcja **`case`** dla wartości `20` zawiera instrukcję **`break`**.

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.");
}
```

Program w ogóle się nie skompiluje, ponieważ w instrukcji **`switch`** nie można sprawdzać wartości typu **`double`**.

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;
```

Nie, instrukcja **`if`** i użycie trój-argumentowego operatora logicznego w tym przypadku nie są sobie równoważne. Instrukcja **`if`** jest poprawnie użyta do wyznaczenia wartości absolutnej zmiennej `z`, natomiast w przypadku trój-argumentowego operatora logicznego

wyrażenia dla prawdy/fałszu warunku są odwrócone. Jeżeli `z` jest większe od zero, to wartością wyrażenia będzie odwrotność `z`. Trój-argumentowy operator logiczny zwraca pierwszą wartość, gdy warunek jest prawdziwy: `pewienWarunek ? wartoscGdyTrue : wartoscGdyFalse`, więc kod powinien być zapisany następująco: `z > 0 ? z : -z`

5. Jak porównać ze sobą dwa łańcuchy tekstowe (`String`)?

Należy skorzystać z metody `equals` typu `String`, np.:

```
zmiennaTypuString.equals("Pewien tekst");
```

6. W jaki sposób można skrócić poniższy kod?

```
int x = 5;
boolean czyDodatnia;

if (x > 0) {
  czyDodatnia = true;
} else {
  czyDodatnia = false;
}
```

Instrukcja **`if`** jest nadmiarowa – możemy przypisać wynik sprawdzenia `x > 0` bezpośrednio do zmiennej typu boolean:

```
int x = 5;
boolean czyDodatnia = x > 5;
```

7. Jaka będzie wartość zmiennej `pewnaZmienna`?

```
boolean pewnaZmienna;

pewnaZmienna = !pewnaZmienna;
```

Powyższy kod w ogóle się nie skompiluje! Próbujemy przypisać do zmiennej `pewnaZmienna` jej przeciwieństwo – jednakże, zmiennej `pewnaZmienna` nie została nadana jeszcze żadna wartość – kompilator zaprotestuje.

8. Czy operatora = można używać do porównywania wartości?

Nie, operator = to operator przypisania – do porównywania wartości używamy operatora relacyjnego ==.

9. Jaka będzie wartość poniższego wyrażenia, jeżeli `x = -5`, `y = -10`, `z = 20` ?

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

Wartość tego wyrażenia to **`true`**. Operator && ma większy priorytet, niż operator ||. Gdyby było na odwrót, to wartością byłaby wartość **`false`**.

10. Jakie wartości mogą mieć zmienne typu **boolean**?

Zmienne typu **boolean** mogą przyjmować jedną z dwóch wartości: **true** bądź **false**.

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?

Wartość wyrażenia `y > 0` nie musi być obliczona, jeżeli wartością wyrażenia `x > 0` będzie **false**, ponieważ w takim przypadku wartość całego wyrażenia `x > 0 && y > 0` nie ma szansy mieć wartości **true** – operator && zwraca **true** tylko, gdy oba argumenty mają wartość **true**. Funkcjonalność, która powoduje, że niektóre wyrażenia nie są obliczane, nazywa się *short-circuit evaluation* i jest dla nas dostępna automatycznie w Maszynie Wirtualnej Java.

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);
```

Kod się nie skompiluje, ponieważ zmienna `poleKwadratu` "żyje" jedynie w bloku instrukcji `if` – tam została zdefiniowana. Po wyjściu z bloku instrukcji `if`, zmienna `poleKwadratu` przestaje istnieć.

13. Dla jakich wartości argumentów operatory && oraz || zwracają **true**? A dla jakiego argumentu zwraca wartość **true** operator ! (zaprzeczenie logiczne)?

Operator && zwraca **true** tylko w przypadku, gdy oba jego argumenty mają wartość **true**.

Operator || zwraca **true**, gdy chociaż jeden z jego argumentów ma wartość **true**.

Operator ! zwraca **true** w przypadku, gdy jego argument ma wartość **false**.

14. Który z operatorów ma wyższy priorytet: || czy && ?

Operator && ma wyższy priorytet od operatora ||.

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);
```

> Program zawsze wypisze wartość przeciwną podaną przez użytkownika, na przykład, dla `5` wypisze `-5`, a dla `-5` wypisze `5`. Instrukcja `x = -x` powinna być objęta blokiem w instrukcji `if`. W powyższym programie, do instrukcji `if` przypisana jest tylko jedna instrukcja – wypisująca na ekran komunikat `"x jest mniejsze od 0"`. Poprawiony kod:

```
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.");
}
```

> Wypisane zostanie komunikat `"Sloneczna pogoda."`, ponieważ wielkość znaków podczas porównywania stringów ma znacznie – `"Bedzie padac"` i `"bedzie padac"` to dwa różne łańcuchy tekstowe.

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);
```

> Kod się nie skompiluje, ponieważ zmiennej `poleKwadratu` wartość nadajemy w instrukcji `if` – istnieje szansa na takie wykonanie naszego programu, w którym `poleKwadratu` nie otrzyma wartości. Kompilator zaprotestuje, ponieważ zmienna `poleKwadratu` może nie mieć nadanej wartości zanim zostanie użyta w ostatniej linii.

## 4.2 Zadania

### 4.2.1 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`.*

Przykładowe rozwiązanie tego zadania:

```
import java.util.Scanner;

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

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

    if (x % 3 == 0) {
      System.out.println("Liczba " + x + " jest podzielna przez 3.");
    } else {
      System.out.println("Liczba " + x + " nie jest podzielna przez 3.");
    }
  }

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

Korzystamy z operatora reszty z dzielenia %. Jeżeli wynik wyrażenia `x % 3` wynosi `0`, oznacza to, że liczba zawarta w zmiennej `x` dzieli się przez `3`.

### 4.2.2 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).*

W naszym programie musimy sprawdzić, czy suma każdego z dwóch boków jest większa od trzeciego – przykładowe rozwiązanie:

```
import java.util.Scanner;

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

    System.out.println("Podaj pierwszy bok trojkata:");
    a = getInt();

    System.out.println("Podaj drugi bok trojkata:");
    b = getInt();

    System.out.println("Podaj trzeci bok trojkata:");
    c = getInt();

    if (a + b > c && a + c > b && b + c > a) {
      System.out.println("Z tych bokow mozna zbudowac trojkat.");
    } else {
      System.out.println("Z tych bokow nie mozna zbudowac trojkata.");
    }
  }

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

W instrukcji `if` korzystamy z operatora warunkowego && do złączenia trzech wyrażeń, które sprawdzają, czy każde dwa z podanych boków są większe od trzeciego.

### 4.2.3 Wypisz największą z dwóch liczb

*Napisz program, który pobierze od użytkownika dwie liczby i wypisze największą z nich.*

Przykładowe rozwiązanie tego zadania:

```
import java.util.Scanner;

public class WypiszNajwiekszaZDwochLiczb {
  public static void main(String[] args) {
    int x, y ;

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

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

    if (x > y) {
      System.out.println("Najwieksza liczba to " + x);
    } else {
      System.out.println("Najwieksza liczba to " + y);
    }
  }

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

### 4.2.4 Wypisz największą z trzech liczb

*Napisz program, który pobierze od użytkownika trzy liczby i wypisze największą z nich.*

Przykładowe rozwiązanie tego zadania:

```
import java.util.Scanner;

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

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

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

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

    if (a >= b) {
      if (a >= c) {
        System.out.println("Najwieksza liczba to " + a);
      } else {
        System.out.println("Najwieksza liczba to " + c);
      }
    } else {
      if (b >= c) {
        System.out.println("Najwieksza liczba to " + b);
      } else {
        System.out.println("Najwieksza liczba to " + c);
      }
    }
  }

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

W tym rozwiązaniu skorzystaliśmy z zagnieżdżonej instrukcji `if`. Najpierw sprawdzamy wartość `a` względem `b`, by wyeliminować jedną z nich, dzięki czemu zagnieżdżonych instrukcjach `if` mamy już do porównania tylko dwie liczby.

### 4.2.5 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`.*

W rozwiązaniu musimy pamiętać, by w każdej sekcji `case` użyć słowa kluczowego `break` – inaczej nasz program będzie wypisywał wiele nazw miesięcy:

```
import java.util.Scanner;

public class ZamienLiczbeNaNazweMiesiaca {
  public static void main(String[] args) {
    int numerMiesiaca;

    System.out.println("Podaj numer miesiaca:");
    numerMiesiaca = getInt();

    switch (numerMiesiaca) {
      case 1:
        System.out.println("Styczen");
        break;
      case 2:
        System.out.println("Luty");
        break;
      case 3:
        System.out.println("Marzec");
        break;
      case 4:
        System.out.println("Kwiecien");
        break;
      case 5:
        System.out.println("Maj");
        break;
      case 6:
        System.out.println("Czerwiec");
        break;
      case 7:
        System.out.println("Lipiec");
        break;
      case 8:
        System.out.println("Sierpien");
        break;
      case 9:
        System.out.println("Wrzesien");
        break;
      case 10:
        System.out.println("Pazdziernik");
        break;
      case 11:
        System.out.println("Listopad");
        break;
      case 12:
        System.out.println("Grudzien");
        break;
      default:
        System.out.println("Nieprawidlowy numer miesiaca!");
    }
  }
```

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

### 4.2.6 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 == !*

Przykładowe rozwiązanie tego zadania – zwróć uwagę, że pobrane imię porównujemy za pomocą metody `equals`, a nie operatora ==:

```
import java.util.Scanner;

public class SprawdzImie {
  public static void main(String[] args) {
    String imie;

    System.out.println("Podaj swoje imie:");
    imie = getString();

    if (imie.equals("Przemek")) {
      System.out.println("Mamy takie samie imiona.");
    } else {
      System.out.println("Mamy rozne imiona.");
    }
  }

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

### 4.2.7 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.*

Przykładowe rozwiązanie tego zadania:

```
import java.util.Scanner;

public class CzyPelnoletni {
  public static void main(String[] args) {
    int wiek;
    boolean czyPelnoletni;

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

    czyPelnoletni = wiek >= 18 ? true : false;

    System.out.println("Czy pelnoletni? " + czyPelnoletni);
  }

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

Moglibyśmy też po prostu przypisać do zmiennej `czyPelnoletni` wynik wyrażenia `wiek >= 18`:

```
czyPelnoletni = wiek >= 18;
```

### 4.2.8 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`.*

Przykładowe rozwiązanie do tego zadania:

```
import java.util.Scanner;

public class CzyRokPrzestepny {
  public static void main(String[] args) {
    int rok;

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

    if ((rok % 4 == 0 && rok % 100 != 0) || rok % 400 == 0) {
      System.out.println("Ten rok jest przestepny.");
    } else {
      System.out.println("Ten rok nie jest przestepny.");
    }
  }

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

W tym programie korzystamy ze złożonego warunku w instrukcji `if` – jeżeli `rok` dzieli się bez reszty przez `4` i nie dzieli się bez reszty przez `100`, to jest to rok przestępny. Jeżeli ten warunek nie jest spełniony, to sprawdzone będzie, czy `rok` dzieli się bez reszty przez `400` – wtedy także `rok` jest rokiem przestępnym. W przeciwnym razie, nie jest to rok przestępny.

# 5 Rozdział V – Pętle

## 5.1 Pytania

1. Do czego służą pętle?

> Pętle służą do wielokrotnego wykonywania zestawu instrukcji, który nazywamy *ciałem* pętli. Pętla wykonuje się tak długo, jak jej warunek jest spełniony, tzn. ma wartość **`true`**. Język Java ma cztery rodzaje pętli – do tej pory poznaliśmy pętle **`while`**, **`do`**..**`while`**, oraz **`for`**. Ponadto, istnieje także pętla **`for`**-each.

2. Czym różnią się od siebie poznane dotąd pętle?

> W przeciwieństwie do pętli **`do`**..**`while`**, ciała pętli **`while`** oraz **`for`** mogą nie wykonać się ani razu, jeżeli warunek pętli nie będzie spełniony na początku działania pętli. Pętla **`for`** jako jedyna posiada trzy człony – instrukcję inicjalizującą, warunek, oraz inicjalizację kroku. Pozostałe dwie pętle posiadają jedynie warunek pętli.

3. Jak nazywamy obieg pętli?

> Obieg pętli nazywamy *iteracją*.

4. Czy ciało pętli zawsze wykona się chociaż raz?

> Zależy to od rodzaju pętli, którego użyjemy. Ciało pętli **`do`**..**`while`** zawsze wykona się przynajmniej raz, ponieważ jej warunek sprawdzany jest na końcu iteracji. Z kolei, pętle **`for`** i **`while`** sprawdzają swój warunek na początku, więc ich ciała mogą się nie wykonać ani razu.

5. Co się stanie, gdy warunek pętli będzie zawsze spełniony i nie zmieni się w trakcie działania programu?

> Program będzie działał "w nieskończoność". Program taki, jeżeli jest to program konsolowy (uruchamiany z linii poleceń), możemy spróbować zakończyć skrótem **`Ctrl + c`**.

6. Z jakich części składa się pętla **`for`**? Czy są one wymagane?

> Pętla **`for`** posiada: instrukcję inicjalizującą, warunek, oraz instrukcję kroku, a także ciało pętli. Żadna z nich nie jest wymagana.

7. Jak sprawdzić znak na danej pozycji w zmiennej typu `String`? Jak sprawdzić pierwszy znak, a jak ostatni?

> Aby otrzymać znak na danej pozycji w zmiennej typu `String`, korzystamy z metody `charAt`, której przekazujemy jako argument indeks znaku, który chcemy pobrać. Indeks pierwszego znaku to `0`, a nie `1`! Ostatni znak ma indeks o jeden mniejszy, niż liczba znaków w stringu. Możemy się do niego odnieść korzystając z metody `length`, która zwraca liczbę znaków w stringu: `tekst.charAt(tekst.length() - 1);`

8. Jak sprawdzić z ilu znaków składa się zmienna typu `String`?

> Aby sprawdzić liczbę znaków w stringu, korzystamy z metody `length`: `tekst.length();`

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 + " ");
}
```

> **Nie, te fragmentu kodu nie są sobie równoważne**, ponieważ, w przypadku pętli `for`, zmienna `i` nie jest dostępna po zakończeniu działania pętli. W przypadku pierwszego fragmentu (z pętlą `while`) moglibyśmy nadal korzystać ze zmiennej `i`, ponieważ została zdefiniowana jeszcze przed pętlą, a nie wewnątrz niej, jak w przypadku fragmentu z pętlą `for`.

10. Do czego służą instrukcje `break` oraz `continue`?

> Instrukcja `break` służy do natychmiastowego przerwania działania pętli, w której została użyta, natomiast instrukcja `continue` przerywa jedynie aktualną iterację tejże pętli.

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;
      }
  }
}
```

> Na ekranie zobaczymy poniższy tekst, ponieważ instrukcja `break` przerywa jedynie wykonanie wewnętrznej pętli – nie ma ona wpływu na pętlę zewnętrzną:

```
j = 0
j = 1
j = 0
j = 1
j = 0
j = 1
```

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) + " ");
}
```

Na ekranie zobaczymy komunikat `W i t a j` (ze spacjami po każdym znaku), a także komunikat błędu spowodowanym tym, że wyszliśmy poza zakres indeksów zmiennej `komunikat` – w pętli powinniśmy użyć operatora < zamiast <=, ponieważ indeks ostatniego znaku w stringach to `length() - 1`, a nie `length()`:

```
W i t a j Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: String index out of range: 5
    at java.lang.String.charAt(String.java:658)
```

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) + " ");
}
```

Powyższa pętla, przepisana na pętlę **`while`**, mogłaby wyglądać następująco:

```
int i = 1, j = 1;

while (i * j < 100) {
  System.out.print((i * j) + " ");
  i++;
  j += 2;
}
```

## 5.2 Zadania

### 5.2.1 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.*

W ciele pętli musimy sprawdzić wartość zmiennej `i` – gdy będzie równa `10`, nie będziemy chcieli wypisywać na ekran przecinka:

```
public class WhileILiczbyOd1Do10 {
  public static void main(String[] args) {
    int i = 1;

    while (i <= 10) {
      if (i != 10) {
        System.out.print(i++ + ", ");
      } else {
        System.out.print(i++);
      }
    }
  }
}
```

Efekt uruchomienia programu:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

### 5.2.2 Policz silnię

*Napisz program, który policzy i wypisze silnię liczby, którą poda użytkownik. Silnia to iloczyn kolejnych liczb od `1` do danej liczby, np. silnia `5` to `1 * 2 * 3 * 4 * 5`, czyli `120`. Silnia liczby `0` to `1`.*

W poniższym przykładowym rozwiązaniu sprawdzamy najpierw pobraną liczbę – silnię powinniśmy liczyć jedynie dla liczb nieujemnych.

Jeżeli liczba jest nieujemna, to w pętli `for` przemnażamy zmienną `wynik` przez kolejne liczby całkowite.

Zauważmy, że podanie przez użytkownika liczby `0` spowoduje, że ciało pętli nie wykona się ani razu, ponieważ warunek pętli nie będzie spełniony. Wtedy zmienna `wynik` będzie miała po prostu swoją wartość nadaną jej podczas inicjalizacji, czyli `1` – a silnia `0` wynosi właśnie `1`.

```
import java.util.Scanner;

public class PoliczSilnie {
  public static void main(String[] args) {
    System.out.print("Podaj liczbe nieujemna: ");
    int liczba = getInt();

    if (liczba < 0) {
      System.out.println("Nastepnym razem podaj liczbe nieujemna.");
    } else {
      int silnia = 1;

      for (int i = 1; i <= liczba; i++) {
        silnia = silnia * i;
      }

      System.out.println("Silnia " + liczba + " to " + silnia);
    }
  }

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

Trzy przykładowe wykonanie powyższego programu:

```
Podaj liczbe nieujemna: -2
Nastepnym razem podaj liczbe nieujemna.
```

```
Podaj liczbe nieujemna: 0
Silnia 0 to 1
```

```
Podaj liczbe nieujemna: 6
Silnia 6 to 720
```

### 5.2.3 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.*

Zauważmy, że aby słowo było palindromem, jego pierwsza litera musi być taka sama, jak jego ostatnia litera, oraz jego druga litera musi być taka sama, jak jego przedostatnia litera itd.

Zauważmy także, że w przypadku słów o nieparzystej liczbie znaków, środkowa litera nie ma dla nas znaczenia – nie ma ona pary, więc nie musimy jej sprawdzać.

Aby odpowiedzieć na pytanie, czy słowo jest palindromem, powinniśmy w takim razie porównać kolejne pary liter: pierwszą i ostatnią, drugą i przedostatnią itd. – jeżeli którakolwiek z par znaków nie będzie się zgadzać, to słowo nie będzie palindromem. Możemy w takim razie w pętli przejść przez połowę długości stringu, ponieważ w każdym obiegu będziemy porównywać ze sobą dwa znaki:

```
import java.util.Scanner;

public class Palindrom {
  public static void main(String[] args) {
    System.out.print("Podaj slowo: ");
    String slowo = getString();

    boolean czyPalindrom = true;
    int dlugoscSlowa = slowo.length();

    for (int i = 0; i < dlugoscSlowa / 2; i++) {
      if (slowo.charAt(i) != slowo.charAt(dlugoscSlowa - 1 - i)) {
        czyPalindrom = false;
        break;
      }
    }

    if (czyPalindrom) {
      System.out.println("Slowo " + slowo + " jest palindromem.");
    } else {
      System.out.println("Slowo " + slowo + " nie jest palindromem.");
    }
  }

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

W programie:

1. Wczytujemy słowo.
2. Ustawiamy wartość zmiennej `czyPalindrom` na `true` – jeżeli znajdziemy w dalszej części programu parę znaków, które się różnią na swoich pozycjach, to ustawimy tą zmienną na `false`.
3. Zapisujemy w pomocniczej zmiennej `dlugoscSlowa` długość podanego przez użytkownika słowa.
4. W pętli `for` przechodzimy przez połowę indeksów znaków pobranego słowa – ta połowa indeksów to indeksy od `0` do `dlugoscSlowa / 2` (bez uwzględniania końcowego indeksu

dzięki użyciu operatora < w warunku pętli).

5. W ciele pętli porównujemy kolejne pary znaków – jeżeli sobie nie odpowiadają, to wiemy, iż słowo nie będzie palindromem – ustawiamy zmienną `czyPalindrom` na **`false`** i korzystamy z instrukcji **`break`** do przerwania działania pętli – nie ma potrzeby sprawdzać kolejnych par znaków – wiemy już, że słowo nie jest palindromem. Zauważmy, że aby odnieść się do n-tego znaku od końca, używamy formuły `dlugoscSlowa - 1 - i`. W ten sposób, w pierwszym obiegu pętli odniesiemy się do ostatniego znaku (`dlugoscSlowa - 1 - 0` to indeks ostatniego znaku). W drugim obiegu pętli – do przedostatniego (`dlugoscSlowa - 1 - 1`) itd.
6. Wypisujemy na ekran informację, czy podane słowo jest palindromem, czy nie.

Kilka przykładowych uruchomień tego programu:

```
Podaj slowo: kajak
Slowo kajak jest palindromem.
```

```
Podaj slowo: programowanie
Slowo programowanie nie jest palindromem.
```

```
Podaj slowo: anna
Slowo anna jest palindromem.
```

```
Podaj slowo: kotek
Slowo kotek nie jest palindromem.
```

### 5.2.4 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.*

Zauważmy, że nie potrzebujemy mieć wszystkich liczb na raz, aby odpowiedzieć na pytanie, która z nich jest największa – wystarczy, że po pobraniu każdej liczby sprawdzimy, czy jest ona większa od poprzednio zapamiętanej, największej liczby:

- jeżeli tak, to wczytana właśnie liczba staje się największą (dotychczasowo) liczbą,
- jeżeli nie, to nie robimy nic z wczytaną liczbą.

Musimy w naszym programie wziąć pod uwagę jednak jeden przypadek – pobieranie pierwszej liczby od użytkownika – tej liczby nie mamy jeszcze z czym porównać, więc powinniśmy od razu potraktować ją jako największą (bo nie ma jeszcze żadnych innych kandydatów).

Możemy obejść się z tą sytuacją na wiele sposobów – w poniższym przykładowym rozwiązaniu najpierw pobieramy pierwszą liczbę i od razu przypisujemy ją do zmiennej `najwiekszaLiczba`, a dopiero w ciele pętli, po pobraniu kolejnej liczby, porównujemy nową liczbę z poprzednio zapisaną.

Skoro jeszcze przed działaniem pętli pobraliśmy liczbę, musimy już na samym początku pętli zapytać użytkownika, czy chce kontynuować – jeżeli nie, przerywamy pętlę instrukcją **`break`**:

```
import java.util.Scanner;

public class WypiszNajwiekszaZPodanychLiczb {
  public static void main(String[] args) {
    System.out.print("Podaj liczbe: ");
    int najwiekszaLiczba = getInt();

    while (true) {
      System.out.print("Czy chcesz zakonczyc program? [t/n] ");
      String czyKoniec = getString();

      if (czyKoniec.equals("t")) {
        break;
      }

      System.out.print("Podaj kolejna liczbe: ");
      int nowaLiczba = getInt();

      if (nowaLiczba > najwiekszaLiczba) {
        najwiekszaLiczba = nowaLiczba;
      }
    }

    System.out.println("Najwieksza liczba z podanych to " + najwiekszaLiczba);
  }

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

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

Przykładowe wykonania tego programu:

```
Podaj liczbe: 10
Czy chcesz zakonczyc program? [t/n] n
Podaj kolejna liczbe: 15
Czy chcesz zakonczyc program? [t/n] n
Podaj kolejna liczbe: 1
Czy chcesz zakonczyc program? [t/n] n
Podaj kolejna liczbe: 20
Czy chcesz zakonczyc program? [t/n] t
Najwieksza liczba z podanych to 20
```

```
Podaj liczbe: 20
Czy chcesz zakonczyc program? [t/n] t
Najwieksza liczba z podanych to 20
```

```
Podaj liczbe: 100
Czy chcesz zakonczyc program? [t/n] n
Podaj kolejna liczbe: -5
Czy chcesz zakonczyc program? [t/n] n
Podaj kolejna liczbe: 99
Czy chcesz zakonczyc program? [t/n] t
Najwieksza liczba z podanych to 100
```

### 5.2.5 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`**).*

Przykładowe rozwiązanie tego programu – pętla główna ma etykietę `glowna_petla`, do której odnosimy się podczas użycia instrukcji **`continue`** w pętli zagnieżdżonej – dzięki temu, przerywana jest aktualna iteracja nie pętli zagnieżdżonej, lecz pętli głównej:

```
public class ZagniezdzonePetle {
  public static void main(String[] args) {
    glowna_petla:
    for (int i = 1; i <= 10; i++) {
      // wymaganie 1: petla glowna pomija swoje
      // iteracje, gdy jej zmienna jest nieparzysta
      if (i % 2 == 1) {
        continue;
      }

      for (int j = 1; j <= 10; j++) {
        System.out.print(j + " ");

        // wymganie 2: petla zagniezdzona
        // powoduje zakonczenie iteracji glownej petli
        // gdy wartosc zmiennej petli zagniezdzonej
        // jest wieksza od wartosci petli glownej
        if (j > i) {
          continue glowna_petla;
        }
      }
    }
  }
}
```

Wynik uruchomienia tego programu:

```
1 2 3 1 2 3 4 5 1 2 3 4 5 6 7 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 10
```

### 5.2.6 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.*

Ten program jest bardziej skomplikowany, niż te, które do tej pory pisaliśmy:

- musimy wziąć pod uwagę szczególny przypadek w naszym programie – na samym jego początku potrzebujemy dwóch liczb, a nie jednej – po pierwszym obliczeniu działania będziemy zawsze mieli już jedną liczbę – poprzedni wynik,
- musimy wziąć pod uwagę, że użytkownik może podać drugą liczbę jako `0` podczas dzielenia lub wpisać znak działania, którego nie obsługujemy,
- chcemy także wypisać na ekran wykonywane działanie, ale tylko w przypadku, gdy obliczenie się udało (tzn. użytkownik nie podał `0` jako mianownik podczas dzielenia bądź nieznanego działania).

Przykładowe rozwiązanie tego zadania jest następujące:

```
import java.util.Scanner;

public class Kalkulator {
  public static void main(String[] args) {
    // poczatek dzialania programu jest szczegolny,
    // bo potrzebujemy dwoch liczb od uzytkownika,
    // wiec pobierzemy pierwsza z nich od razu i zapiszemy
    // w zmiennej poprzedniWynik
    System.out.print("Podaj liczbe: ");
    int poprzedniWynik = getInt();

    String czyKoniec;
```

```
    boolean czyBlednaOperacja;

    do {
      int nowyWynik = 0;
      // ustaw ta zmienna na false - jezeli cos bedzie nie tak,
      // ustawimy wartosc na true (patrz instrukcja switch ponizej)
      czyBlednaOperacja = false;

      System.out.print("Podaj dzialanie (* / - +): ");
      String dzialanie = getString();

      System.out.print("Podaj kolejna liczbe: ");
      int drugaLiczba = getInt();

      // wykonaj dzialanie i zapisz wynik w zmiennej nowyWynik
      switch (dzialanie) {
        case "+":
          nowyWynik = poprzedniWynik + drugaLiczba;
          break;
        case "-":
          nowyWynik = poprzedniWynik - drugaLiczba;
          break;
        case "*":
          nowyWynik = poprzedniWynik * drugaLiczba;
          break;
        case "/":
          if (drugaLiczba == 0) {
            czyBlednaOperacja = true;
            System.out.println("Nie moge podzielic przez 0.");
          } else {
            nowyWynik = poprzedniWynik / drugaLiczba;
          }
          break;
        default:
          czyBlednaOperacja = true;
          System.out.println("Nieprawidlowa operacja.");
      }

      // jezeli podano 0 przy dzieleniu lub nieznany symbol dzialania,
      // nie chcemy nic wypisywac na ekranie
      if (!czyBlednaOperacja) {
        // wypisz wykonywane dzialanie i jego wynik
        System.out.println(
            poprzedniWynik + " " + dzialanie + " " + drugaLiczba + " = " + nowyWynik
        );

        // nowy wynik staje sie poprzednim wynikiem
        poprzedniWynik = nowyWynik;
      }

      System.out.println("Czy chcesz zakonczyc program? [t/n]");
      czyKoniec = getString();
    } 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 programie pobieramy pierwszą liczbę, a następnie, w pętli, pobieramy działanie i kolejną liczbę. W instrukcji `switch` sprawdzamy podane działanie i wykonujemy je, przypisując wynik do pomocniczej zmiennej `nowyWynik`, ale tylko wtedy, gdy podane działanie jest poprawne (* / + -) lub nie podano `0` jako dzielnik przy dzieleniu.

Jeżeli jednak działanie jest niepoprawne lub podano `0` podczas dzielenia, ustawiamy wartość pomocniczej zmiennej `czyBlednaOperacja` na `true`. Dzięki temu, po instrukcji `switch`, będziemy wiedzieli, czy powinniśmy wypisywać na ekran wynik działania, czy nie.

Jeżeli nie było problemów, to wypisujemy ładnie sformatowane działanie i jego wynik, a następnie przepisujemy do zmiennej `poprzedniWynik` wyliczony wynik ze zmiennej `nowyWynik`, aby w kolejnym obiegu pętli to ta wartość została użyta w kolejnym działaniu.

Przykładowe uruchomienie tego programu:

```
Podaj liczbe: 10
Podaj dzialanie (* / - +): +
Podaj kolejna liczbe: 2
10 + 2 = 12
Czy chcesz zakonczyc program? [t/n]
n
Podaj dzialanie (* / - +): /
Podaj kolejna liczbe: 3
12 / 3 = 4
Czy chcesz zakonczyc program? [t/n]
n
Podaj dzialanie (* / - +): !
Podaj kolejna liczbe: 10
Nieprawidlowa operacja.
Czy chcesz zakonczyc program? [t/n]
n
Podaj dzialanie (* / - +): -
Podaj kolejna liczbe: 3
4 - 3 = 1
Czy chcesz zakonczyc program? [t/n]
t
```

### 5.2.7 Choinka

*Napisz program, który pobierze od użytkownika jedną liczbę całkowitą. Następnie, program powinien wypisać na ekran choinkę 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 – użytkownik podał liczbę 5, efekt wyświetlony na ekranie:*

```
  *
 ***
*****
```

*Przykład drugi – użytkownik podał liczbę 6, efekt na ekranie:*

```
  **
 ****
******
```

Zauważmy, że niezależnie od liczby gwiazdek, jaką podamy, na szczycie choinki (tzn. w pierwszym rzędzie) zawsze będzie albo jedna gwiazdka, albo dwie. Ich liczba zależy od tego, czy w podstawie choinki (która jest liczbą podaną przez użytkownika) jest parzysta liczba gwiazdek, czy nieparzysta:

- jeżeli podstawa ma nieparzystą liczbę gwiazdek, to na szczycie zawsze będzie 1 gwiazdka,
- jeżeli podstawa ma parzystą liczbę gwiazdek, to na szczycie zawsze będą 2 gwiazdki.

Wiemy już zatem, od wyświetlania ilu gwiazdek musimy zacząć.

A ile gwiazdek powinniśmy wyświetlić w każdym kolejnym rzędzie? Zauważmy, że każdy kolejny rząd ma o 2 gwiazdki więcej od poprzedniego – będziemy więc zwiększać liczbę gwiazdek do wyświetlania w każdym kolejnym rzędzie o dwie.

Kiedy mamy przestać wyświetlać gwiazdki? Gdy wyświetlimy tyle, ile miało być w podstawie choinki.

Wiemy już prawie wszystko – pozostaje jeszcze jedna kwestia – wcięcia na początku każdego rzędu, by gwiazdki na każdym poziomie były ładnie wyśrodkowane. Skąd mamy wiedzieć, ile spacji powinniśmy wyświetlić na początku w każdym rzędzie?

Ponownie należy zauważyć pewien wzór – spacje, które należy wyświetlić w każdym rzędzie, można wyliczyć następującym wzorem (zaokrąglając wynik w dół):

```
(liczbaGwiazdekWPodstawie - liczbaGwiazdekWAktualnymRzedzie) / 2
```

Dzięki powyższym, możemy napisać rozwiązanie do tego zadania:

```java
import java.util.Scanner;

public class Choinka {
  public static void main(String[] args) {
    System.out.print("Podaj liczbe gwiazdek w podstawie: ");
    int liczbaGwiazdekWPodstawie = getInt();

    // liczba gwiazdek w pierwszym rzedzie, czyli
    // na szczycie choinki, zalezy od tego, czy
    // liczba gwiazdek w podstawie jest parzysta, czy nie
    // w kazdym rzedzie sa o 2 gwiazdki mniej,
    // niz w rzedzie ponizej, wiec na szczycie beda
    // 2 gwiazdki, gdy w podstawie jest parzysta liczba gwiazdek,
    // albo 1 gwiazdka, gdy podstawa ma nieparzysta liczbe gwiazdek
    int liczbaGwiazdekNaSzczycie = liczbaGwiazdekWPodstawie % 2 == 0 ? 2 : 1;

    // petla rozpoczyna dzialania od wypisania tylu gwiazdek, ile jest
    //  na szczycie
    // petla dziala dopoki, liczba gwiazdek do wypisania w rzedzie
    //  nie przekroczy liczby gwiazdek w podstawie
    // na koncu kazdego obiegu, zwiekszamy liczbe gwiazdek o 2,
    //  bo w kazdym kolejnym rzedzie sa o 2 gwiazdki wiecej,
    //  niz w poprzednim
    for (int gwiazdkiWRzedzie = liczbaGwiazdekNaSzczycie;
         gwiazdkiWRzedzie <= liczbaGwiazdekWPodstawie;
         gwiazdkiWRzedzie += 2) {

      // przed gwiazdkami musimy wypisac odpowiednia liczbe spacji,
      // aby gwiazdki byly ladnie wysrodkowane
      // liczba spacji do wypisania rowna jest polowie roznicy gwiazdek
      // w podstawie i w aktualnym rzedzie
      int liczbaSpacji = (liczbaGwiazdekWPodstawie - gwiazdkiWRzedzie) / 2;

      for (int i = 0; i < liczbaSpacji + gwiazdkiWRzedzie; i++) {
        // najpierw wypisujemy odpowiednia liczbe spacji, a potem gwiazdki,
        // wiec sprawdzamy wartosc zmiennej i - na jej podstawie wyznaczamy,
        // czy wypisac spacje, czy juz gwiazdke
        System.out.print(i < liczbaSpacji ? " " : "*");
      }

      // wypisujemy znak nowej linii, by kolejny
      // rzad gwiazdek byl wypisywany w nastepnej linii
      System.out.println();
    }
  }

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

Program wydaje się długi ze względu na dużą ilość komentarzy. Dwa przykładowe działania powyższego programu:

```
Podaj liczbe gwiazdek w podstawie: 8
   **
  ****
 ******
********
```

```
Podaj liczbe gwiazdek w podstawie: 7
   *
  ***
 *****
*******
```

# 6 Rozdział VI – Tablice

## 6.1 Pytania

1. Do czego służą tablice?

> Tablice służą do przechowywania kolekcji elementów pewnego typu. Elementy te są przechowywane w ustalonej kolejności.

2. Jak definiuje się tablice?

> Tablice definiuje się tak, jak zwykłe zmienne, z tym, że po nazwie typu należy dodać nawiasy kwadratowe. Dozwolone jest także umieszczenie nawiasów kwadratowych nie po nazwie typu, lecz po nazwie zmiennej:

```
int[] tablica;
int tablica[];
```

3. Jak utworzyć tablicę?

> Aby utworzyć tablicę, możemy albo skorzystać ze słowa kluczowego new wraz z podaniem rozmiaru tablicy, albo, podczas definiowania tablicy, w nawiasach klamrowych { } wyspecyfikować elementy, z których tablica ma się składać. Można też skorzystać ze sposobu łączącego dwie pierwsze opcje, gdzie korzystamy ze słowa kluczowego new i elementów tablicy w nawiasach klamrowych:

```
int[] calkowite = new int[5]; // pierwsza opcja
double[] rzeczywiste = { 3.14, 5, -20.5 }; // druga opcja

String[] slowa;
slowa = new String[] { "Ala", "ma", "kota" }; // trzecia opcja
```

4. Czy rozmiar tablicy można zmienić?

> Nie, rozmiaru raz utworzonej tablicy nie można zmienić.

5. Jak odnieść się do danego elementu tablicy?

> Aby odnieść się do danego elementu tablicy, korzystamy z nawiasów kwadratowych po nazwie zmiennej tablicowej. W nawiasach kwadratowych należy umieścić indeks elementu, do którego chcemy się odnieść. Indeksy elementów tablic zaczynają się w języku Java od `0`, a nie `1`. Przykład – wypisujemy *drugi* element tablicy o nazwie `tablica`:

```
System.out.println(tablica[1]);
```

6. Jak odnieść się do pierwszego, a jak do ostatniego elementu tablicy?

> Pierwszy element tablicy ma zawsze indeks `0`.
>
> Ostatni element tablicy ma indeks równy liczbę elementów w tablicy pomniejszony o `1` – możemy więc skorzystać z pola `length` tablicy, aby wyliczyć indeks ostatniego znaku:

```
System.out.println(tablica[0]); // pierwszy element
System.out.println(tablica[tablica.length - 1]); // ostatni element
```

7. Czy poniższy kod jest poprawny?

```
int[] tablica = { 1, 2, 3 };

System.out.println(tablica[3]);
```

> Ten kod jest poprawny pod kątem składniowym – program z powyższym fragmentem kodu skompiluje się bez błędów. Z drugiej jednak strony, ten program zawiera błąd, ponieważ próbujemy odnieść się do czwartego elementu tablicy, która ma tylko *trzy* elementy – ten program zakończy się błędem wykonania `ArrayIndexOutOfBoundsException`.

8. Jak sprawdzić ile elementów znajduje się w tablicy?

> Aby dowiedzieć się, z ilu elementów składa się tablica, należy skorzystać z pola `length`, które ma każda tablica, np.:

```
int[] liczby = { 1, 5, -20 };
System.out.println(liczby.length); // wypisze 3
```

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.");
  }
}
```

W przypadku, gdy będziemy szukać liczby `0`, na ekranie zobaczymy komunikat `"Liczba 0 zostala znaleziona!"`. W przypadku liczby `500`, jednakże, na ekranie zobaczymy komunikat:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
```

Wykonanie programu zakończy się błędem. Dlaczego tak się stało? Wynika to z niepoprawnego warunku pętli. Liczby `500` nie ma w tablicy, więc pętla przechodzi przez wszystkie elementy tablicy, jednak w warunku pętli powinniśmy byli użyć operatora < zamiast <=. Użycie operatora <= spowodowało, że w ostatnim obiegu pętli wyszliśmy poza zakres tablicy o nazwie `tablica`.

10. Czy poniższy kod:
    a) Skompiluje się?
    b) Wykona się bez błędów?

```
int[][] tablica2d = new int[3][5];

tablica2d[3][1] = 1;
```

Ten kod skompiluje się bez błędów, jednak jego wykonanie zakończy się błędem `ArrayIndexOutOfBoundsException`, ponieważ tablica `tablica2d` ma tylko trzy elementy w "pierwszym wymiarze", a w powyższym fragmencie kodu próbujemy odnieść się do czwartego elementu pierwszego wymiaru tej tablicy (która ma indeks `3`).

11. Czy poniższy kod jest poprawny?

```
String powitanie = { "Witaj", "Swiecie" };
for (int i = 0; i < powitanie.length(); i++) {
  System.out.println(powitanie[i] + " ");
}
```

Ten kod się nie skompiluje, ponieważ odnosząc się do rozmiaru tablicy, nie powinniśmy dodawać nawiasów ( ) po atrybucie `length`. Nie jest to jednak jedyny problem – tablica `powitanie` nie jest poprawnie zdefiniowana – brakuje nawiasów [ ] po typie `String`.

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.");
}
```

Na ekranie zobaczymy komunikat `"Tablice nie sa takie same."`, ponieważ tablice powinniśmy porównywać poprzez sprawdzenie elementów na tych samych pozycjach, a nie poprzez użycie operatora ==.

13. Która z poniższych tablic jest zdefiniowana/utworzona niepoprawnie i dlaczego?

```
// blad: brakuje [] po int
int liczby = { 1, 2, 3 };

// blad: stringi zapisujemy w "" a nie ''
String[] litery = { 'a', 'b', 'c' };


// blad: brakuje rozmiaru tablicy w []
String[] slowa = new String[];
// blad: inicjalizacja za pomoca { } moze być uzyta jedynie podczas
// definicji tablicy
slowa = { "Ala", "ma", "kota" };

// poprawna tablica
double[] rzeczywiste = new double[] { 3.14, 2.44, 20 };

// blad: niespojny typ double <-> int
double[] innaTablica = new int[3];

// blad: podczas tworzenia tablicy za pomoca new i podawania
// elementow w {}, nie należy podawac w nawiasach [] rozmiaru tablicy
int[] tablica = new int[5] { 1, 10, 100 };

// poprawna tablica
double[] kolejnaTablica = new double[3];
// blad: ale już w ten sposob po utworzeniu tablicy nie mozemy
// jej przypisac wartosci
kolejnaTablica = { 5, 10, 15 };

// poprawna tablica z tylko jednym elementem
String[] tab = { "Ala ma kota" };
```

14. Do czego służy pętla `for`-each i jak się z niej korzysta?

> Pętla `for`-each służy do iterowania po tablicach (i innych kolekcjach). Jest to krótka i wygodna alternatywa dla iterowania po elementach tablicy za pomocą pętli `for`.

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] + ", ");
}
```

> Jest to podchwytliwe zadanie – pętla `for`-each jest w nim niepoprawnie użyta. Zamiast po prostu wypisać wartość zmiennej pętli `i`, która w każdym obiegu przyjmuje wartość kolejnego elementu tablicy `liczby`, korzystamy z tej wartości jako indeksu elementów tablicy `liczby`. W związku z tym, że akurat pierwsze wartości przechowywane w tablicy `liczby` to `0`, `1`, `2`, to odniesiemy się, kolejno, do pierwszego, drugiego, oraz trzeciego elementu tej tablicy i na ekranie zobaczymy liczby `0`, `1`, `2`. Jednakże, w ostatnim obiegu pętli, zmienna `i` będzie miała wartość `-1` – jest to nieprawidłowy indeks tablicy, co spowoduje wystąpienie błędu `ArrayIndexOutOfBoundsException`. Finalnie, na ekranie zobaczymy:

```
0,
1,
2,
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
```

> Powyższy kod poprawnie powinien zostać zapisany w następujący sposób (wtedy na ekranie zobaczylibyśmy `0`, `1`, `2`, `-1`):

```
int[] liczby = { 0, 1, 2, -1 };

for (int i : liczby) {
  System.out.println(i + ", ");
}
```

16. Co wypisze na ekran poniższy fragment kodu?

```
int[] liczby = new int[3];

System.out.println(liczby[1]);
```

> Na ekranie zobaczymy liczbę `0`. Co prawda nie ustawiliśmy w tablicy `liczby` żadnych wartości, ale elementy tablic są inicjalizowane domyślnymi wartościami typu, którego wartości tablica może przechowywać. W przypadku typu `int` jest to `0`.

17. Czy poniższy fragment kodu jest poprawny?

```
int[][] dwuwymiarowa =
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 };
```

> Nie, ta tablica nie jest zainicjalizowana w poprawny sposób – brakuje jeszcze jednej pary nawiasów klamrowych, ponieważ jest to tablica dwuwymiarowa. Poprawna inicjalizacja:

```
int[][] dwuwymiarowa = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

## 6.2 Zadania

### 6.2.1 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.*

Nie ma różnicy, czy wypisujemy co drugi element z tablicy o parzystej czy nieparzystej liczbie elementów. Musimy tylko odpowiednio zapisać instrukcję kroku pętli `for` – tak, aby zmienna pętli zwiększała się o `2`, a nie o `1`. Przykładowe rozwiązanie tego zadania:

```
public class CoDrugaWartoscTablicy {
  public static void main(String[] args) {
    // parzysta liczba elementow
    int[] parzysta = { 1, 10, 15, 0, 100, 20 };

    // nieparzysta liczba elementow
    int[] nieparzysta = { 5, 7, 9 };

    for (int i = 0; i < parzysta.length; i += 2) {
      System.out.print(parzysta[i] + ", ");
    }

    System.out.println(); // nowa linia

    for (int i = 0; i < nieparzysta.length; i += 2) {
      System.out.print(nieparzysta[i] + ", ");
    }
  }
}
```

Wynik działania programu:

```
1, 15, 100,
5, 9,
```

### 6.2.2 Największa liczba w tablicy

*Napisz program, który wypisze największą liczbę z tablicy. Tablicę zainicjalizuj przykładowymi wartościami.*

Aby znaleźć największy element, musimy porównać kolejne elementy tablicy do poprzedniego, największego elementu. Jeżeli kolejny sprawdzany element będzie większy od poprzednio znalezionego, największego elementu, to zapiszemy tą aktualną wartość jako największą.

Trzeba tylko rozwiązać jeden problem – **jaką wartość powinniśmy użyć jako pierwszą "największą" liczbę, do której na początku będziemy porównywać wartości z tablicy?** Czy użyć jakiejś bardzo małej liczby, mając nadzieję, że w tablicy będzie chociaż jedna większa od niej liczba? Nie jest to dobre rozwiązanie – tym bardziej, że rozwiązanie poprawne jest proste do implementacji.

**Na początku, jako największą liczbę, wystarczy ustawić pierwszy element tablicy.** Następnie, w pętli przejdziemy przez *pozostałe* elementy tablicy i będziemy je porównywać z zapisaną, największą liczbą – jeżeli aktualnie sprawdzana wartość z tablicy będzie większa, niż poprzednio zapamiętana największa liczba, to zapamiętamy ten aktualny element jako największa liczba itd., aż nie przejdziemy przez całą tablicę:

```
public class NajwiekszaWartoscWTablicy {
  public static void main(String[] args) {
    int[] liczby = { 1, -20, 100, 40, -15 };

    // najwieksza liczbe inicjalizujemy wartoscia
    // pierwszego elementu tablicy
    int najwiekszaLiczba = liczby[0];

    // przechodzimy przez cala tablice, z pominieciem
    // pierwszego elementu - nie ma sensu porownywac go
    // do samego siebie - dlatego zmienna i zaczyna od 1,
    // czyli od indeksu drugiego elementu w tablicy
    for (int i = 1; i < liczby.length; i++) {
      // jezeli aktualnie sprawdzana wartosc z tablicy
      // jest wieksza, niz poprzednio zapamietana najwieksza
      // liczba, to zapisujemy ten aktualny element jako najwiekszy
      if (liczby[i] > najwiekszaLiczba) {
        najwiekszaLiczba = liczby[i];
      }
    }
    System.out.println("Najwieksza liczba to: " + najwiekszaLiczba);
  }
}
```

Wynik działania tego programu:

```
Najwieksza liczba to: 100
```

### 6.2.3 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.*

Korzystając z funkcjonalności `toUpperCase` typu `String`, możemy wypisać na ekran elementy tablicy ze wszystkimi literami zamienionymi na wielkie. Skorzystamy z pętli `for`-each, aby przejść przez wszystkie elementy tablicy:

```
public class SlowaTablicyWielkimiLiterami {
  public static void main(String[] args) {
    String[] slowa = { "Ala", "ma", "kota", "i", "psa" };

    for (String slowo : slowa) {
      System.out.print(slowo.toUpperCase() + " ");
    }
  }
}
```

Wynik działania:

```
ALA MA KOTA I PSA
```

### 6.2.4 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"`.*

Aby pobrać słowa od użytkownika, skorzystamy z poznanej już metody `getString`, którą dodamy do naszego programu.

Najpierw wypiszemy na ekran komunikat `"Prosze podac 5 slow:"`. Następnie, w pętli, pobierzemy tyle słów, ile może pomieścić tablica `slowa`. Zauważmy, że poniższy program napisany jest w taki sposób, że nieważne jest, z ilu elementów składa się tablica `slowa` – jeżeli zmienilibyśmy jej maksymalny rozmiar na `3` bądź `20` elementów – program nadal działałby poprawnie. Wynika to z faktu użycia atrybutu `length` tablicy, zamiast zaszycia w kodzie na stałe liczby `5` jako liczby elementów podczas pobierania danych od użytkownika.

Po pobraniu od użytkownika słów, korzystamy z zagnieżdżonej pętli `for`. W pętli zewnętrznej iterujemy od końca tablicy do jej początku. Zwróćmy uwagę, że w warunku pętli korzystamy z `length`, a nie `length()`, gdyż odnosimy się do rozmiaru tablicy.

W pętli wewnętrznej iterujemy od ostatniego do pierwszego znaku w aktualnym słowie – tym razem w warunku pętli korzystamy z `length()`, ponieważ sprawdzamy liczbę znaków wartości typu `String`, którą pobieramy z tablicy `slowa`. Wypisujemy na ekran kolejne znaki za pomocą poznanej już metody `charAt`, której podajemy jako argument indeks znaku zapisany w zmiennej `j`. Zmienna `j`, dzięki odpowiednio zapisanej wewnętrznej pętli `for`, będzie przybierała indeksy znaków przetwarzanego aktualnie słowa – od ostatniego znaku, do pierwszego.

Przykładowe rozwiązanie:

```
import java.util.Scanner;

public class OdwrotnosciSlowWTablicy {
  public static void main(String[] args) {
    String[] slowa = new String[5];

    System.out.println("Podaj " + slowa.length + " slow:");

    for (int i = 0; i < slowa.length; i++) {
      System.out.print((i + 1) + " slowo: ");
      slowa[i] = getString();
    }

    for (int i = slowa.length - 1; i >= 0; i--) {
      for (int j = slowa[i].length() - 1; j >= 0; j--) {
        System.out.print(slowa[i].charAt(j));
      }
      System.out.println(); // nowa linia
    }
  }

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

Przykładowy wynik działania powyższego programu:

```
Podaj 5 slow:
1 slowo: Ala
2 slowo: ma
3 slowo: kota
4 slowo: i
5 slowo: psa
asp
i
atok
am
alA
```

### 6.2.5 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.*

Sortowanie jest bardzo często potrzebne w programowaniu. Ma ono bardzo wiele zastosowań, chociażby podczas wyświetlania danych dla użytkownika czy też wyszukiwania elementów (szybciej szuka się w posortowanych danych).

Istnieje wiele sposobów na sortowanie danych, m. in.:

- sortowanie bąbelkowe,
- sortowanie przez wstawianie,
- quicksort,
- mergesort,
- heapsort,
- i inne.

Różne algorytmy sortowania mają różne cechy oraz różne *złożoności obliczeniowe i pamięciowe*, czyli (w uproszczeniu), jak dużo czasu i pamięci wymagane jest, aby posortować zbiór elementów. Porównując algorytmy oceniamy również, jaki wpływ ma rozmiar danych na czas działania algorytmu.

Poniższe, przykładowe rozwiązanie zadania, korzysta z algorytmu "*insertion sort*", czyli *sortowania przez wstawianie* – jest to jeden z prostszych algorytmów sortowania, który nadaje się dla niewielkich zestawów danych, ponieważ dla zbiorów wieloelementowych (tysiące/setki tysięcy/miliony) będzie niewydajny (będzie potrzebował zbyt dużo czasu na wykonanie).

Algorytm działa w następujący sposób: zaczynając od *drugiego* elementu, sprawdzamy poprzedzające go elementy i przesuwamy te poprzednie elementy w tablicy w prawo o jedno miejsce dopóki aktualnie przetwarzany element jest mniejszy niż poprzednie elementy. W pewnym momencie znajdziemy miejsce w (już przetworzonej) części tablicy, w której aktualna liczba powinna być ustawiona. Kontynuujemy w ten sposób dla wszystkich kolejnych elementów tablicy.

Spójrzmy na przykład działania tego algorytmu – posortowane zostaną liczby: `4`, `3`, `2`, `5`:

| 4 | 3 | 2 | 5 |
|---|---|---|---|

1. Algorytm zaczyna zawsze działanie od drugiego elementu tablicy – w tym przypadku, jest to liczba `3`. Sprawdzamy elementy ją poprzedzające – jest to liczba `4`, która jest większa od aktualnej liczby – przeniesiemy więc liczbę `4` na miejsce liczby `3`:

| 4 | 4 | 2 | 5 |
|---|---|---|---|

2. Więcej elementów wcześniej nie było, więc wpisujemy `3` na pierwszą pozycję w tablicy:

| 3 | 4 | 2 | 5 |
|---|---|---|---|

3. Przechodzimy do kolejnego, czyli trzeciego elementu – jest to liczba `2`. Sprawdzamy poprzedzające ją elementy. Liczba `4` jest większa od `2`, więc przesuwamy liczbę `4` o jedno miejsce w prawo:

| | | | |
|---|---|---|---|
| 3 | 4 | 4 | 5 |

4. Sprawdzamy kolejny poprzedni element – jest to liczba `3`, która także jest większa od `2`. Przesuwamy `3` o jedno miejsce w prawo:

| | | | |
|---|---|---|---|
| 3 | 3 | 4 | 5 |

5. Więcej elementów nie ma, więc wpisujemy dwójkę na pierwszą pozycję:

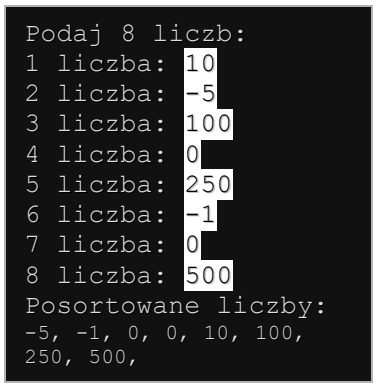
| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 5 |

6. Przechodzimy do kolejnego elementu tablicy – będzie to już ostatni element, liczba `5`. Sprawdzamy poprzedzające ją elementy – pierwszy to liczba `4`, która jest mniejsza od liczby `5`, więc przerywamy procesowanie liczby `5`, gdyż na pewno nie będzie już musiała być ustawiona wcześniej w tablicy (ponieważ przed nią będą na pewno mniejsze elementy).
7. Przeszliśmy przez wszystkie elementy – tablica jest teraz posortowana.

Przykładowe rozwiązanie:

```java
import java.util.Scanner;

public class SortowaniePobranychLiczb {
  public static void main(String[] args) {
    int[] liczby = new int[8];

    System.out.println("Podaj " + liczby.length + " liczb:");

    // pobieramy liczby
    for (int i = 0; i < liczby.length; i++) {
      System.out.print((i + 1) + " liczba: ");
      liczby[i] = getInt();
    }

    // zaczynamy od drugiego elementu tablicy
    // (ktorego indeks to 1, bo pierwszy element ma indeks 0)
    for (int i = 1; i < liczby.length; i++) {
      // liczba, dla ktorej aktualnie chcemy znalezc miejsce
      int aktualnaLiczba = liczby[i];

      // zaczynamy sprawdzanie elementow poprzednich
      // od elementu o indeksie o 1 mniejszym
      // niz ten, dla ktorego aktualnie szukamy miejsca
      int j = i - 1;

      // dopoki nie wyjdziemy poza zakres tablicy
      // lub poprzedni element jest wiekszy od tego,
      // dla ktorego aktualnie szukamy miejsca...
      while (j >= 0 && liczby[j] > aktualnaLiczba) {
        // ...przesuwamy poprzedni element o jedno miejsce w prawo
        liczby[j + 1] = liczby[j];
        j--;
      }

      // gdy petla sie konczy, j + 1 wyznacza indeks
      // dla wartosci, dla ktorej szukalismy miejsca
      liczby[j + 1] = aktualnaLiczba;
    }

    // na koncu petli jest posortowana
    System.out.println("Posortowane liczby:");

    for (int x : liczby) {
      System.out.print(x + ", ");
    }
  }

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

Trzy przykładowe uruchomienia tego programu:

| Pierwsze uruchomienie | Drugie uruchomienie | Trzecie uruchomienie |
|---|---|---|
|  |  |  |

### 6.2.6 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.*

Podobnie, jak w poprzednich dwóch zadaniach, pobierzemy w pętli od użytkownika dane, którymi zasilimy tablicę.

Następnie, korzystając z zagnieżdżonych pętli, przeiterujemy w pętli zewnętrznej przez tablicę pobranych liczb, używając do tego celu pętli `for`-each. W pętli wewnętrznej, dla aktualnej liczby z tablicy, wyliczymy silnię. Najpierw sprawdzimy jednak, czy liczba jest ujemna – jeżeli tak, to taką liczbę pominiemy – silnię powinniśmy liczyć tylko dla liczb nieujemnych. Po zakończeniu działania pętli wewnętrznej, wypiszemy na ekran policzoną silnię:

```
import java.util.Scanner;

public class SilniaLiczbWTablicy {
  public static void main(String[] args) {
    int[] liczby = new int[5];

    System.out.println("Podaj " + liczby.length + " liczb:");

    // pobieramy liczby
    for (int i = 0; i < liczby.length; i++) {
      System.out.print((i + 1) + " liczba: ");
      liczby[i] = getInt();
    }

    // iterujemy po pobranych liczbach
    for (int x : liczby) {
      // silnie powinnismy liczyc tylko dla liczb nieujemnych
      if (x < 0) {
        System.out.println(
            "Silnie mozna policzyc tylko dla liczb >= 0. " +
            "Pomijam liczbe " + x
        );
      } else {
        int silnia = 1;

        // liczymy silnie aktualnej liczby
        for (int i = 1; i <= x; i++) {
          silnia *= i;
        }

        System.out.println("Silnia liczby " + x + " wynosi " + silnia);
      }
    }
  }

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

Przykładowe wykonanie tego programu:

```
Podaj 5 liczb:
1 liczba: 1
2 liczba: 4
3 liczba: -2
4 liczba: 0
5 liczba: 5
Silnia liczby 1 wynosi 1
Silnia liczby 4 wynosi 24
Silnie mozna policzyc tylko dla liczb >= 0. Pomijam liczbe -2
Silnia liczby 0 wynosi 1
Silnia liczby 5 wynosi 120
```

### 6.2.7 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.*

To zadanie ma na celu sprawdzenie dwóch rzeczy:

- czy będziemy pamiętali o porównaniu tablic nie za pomocą operatora ==, lecz poprzez porównanie liczby elementów tablic oraz ich wartości parami, na odpowiadających sobie indeksach?
- czy będziemy pamiętali, że wartości typu `String` powinniśmy porównywać za pomocą `equals`, a nie operatora == ?

Przykładowe rozwiązanie tego zadania:

```
public class PorownajTabliceStringow {
  public static void main(String[] args) {
    String[] pierwsza = { "Ala", "ma", "kota" };
    String[] druga = { "Ala", "ma", "kota" };

    boolean roznicaZnaleziona = false;

    if (pierwsza.length != druga.length) {
      roznicaZnaleziona = true;
    } else {
      for (int i = 0; i < pierwsza.length; i++) {
        if (!pierwsza[i].equals(druga[i])) {
          roznicaZnaleziona = true;
          break;
        }
      }
    }

    if (roznicaZnaleziona) {
      System.out.println("Tablice nie sa takie same.");
    } else {
      System.out.println("Tablice sa takie same");
    }
  }
}
```

W programie definiujemy dwie przykładowe tablice przechowujące wartości typu `String`. Tablice te mają takie same elementy, w tej samej kolejności.

Zmienna `roznicaZnaleziona` będzie służyła za wyznacznik, czy tablice są takie same, czy nie. Na początku inicjalizujemy ją wartością `false`. Jeżeli znaleźlibyśmy różnicę w tablicach, to ustawilibyśmy jej wartość na `true`.

Sprawdzamy liczbę elementów obu tablic. Jeżeli się zgadza, to w pętli porównujemy pary elementów z obu tablic – co ważne, stosujemy metodę `equals`, zamiast porównywać wartości typu `String` za pomocą operatora ==. Jeżeli znajdziemy różnicę, ustawimy zmienną `roznicaZnaleziona` na `true` i zakończymy działanie pętli instrukcją `break` – wystarczy nam znalezienie jednej różnicy, by wiedzieć, że tablice nie są sobie równe.

Na końcu programu wypisujemy informację, czy tablice są takie same – w tym przypadku zobaczymy komunikat `"Tablice sa takie same"`, ponieważ nadaliśmy im takie same wartości.

# 7 Rozdział VII – Metody

## 7.1 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;
}
```

> Powyższa metoda zwraca wynik dzielenia liczb przesłanych jej jako argumenty. Argumentami tej metody są `x` oraz `y` – dwie wartości typu `int`. Ta metoda powinna sprawdzać, czy liczba `y` nie jest równa `0` – w takim przypadku dzielenie nie powinno zostać wykonane. Pytanie tylko, co wtedy zwrócić? W takim przypadku powinniśmy *rzucić wyjątek*. Tematem wyjątków zajmiemy się w dalszej części kursu.

2. Co zrobić, aby użyć metody (wywołać ją)?

> Aby użyć metody, czyli ją wywołać, należy napisać jej nazwę oraz nawiasy ( ), w których należy umieścić argumenty metody (o ile jakieś przyjmuje).

3. Które z poniższych są poprawnymi nazwami metod?
   a) _mojaMetoda
   b) zapiszUstawienia
   c) zapiszKosztW$
   d) 5NajlepszychOfert
   e) pobierz5OstatnichZamowien

> Poprawne nazwy metod to `_mojaMetoda`, `zapiszUstawienia`, `zapiszKosztW$`, oraz `pobierz5OstatnichZamowien`. Jedyną nieprawidłową nazwą jest `5NajlepszychOfert`, ponieważ zaczyna się od liczby.

## 7.2 Zadania do podstaw metod

### 7.2.1 Metoda wypisująca Witajcie!

*Napisz metodę, która wypisuje na ekran tekst `Witajcie!` i użyj jej w metodzie `main`.*

Metoda wypisująca komunikat nie będzie nic zwracać, więc jej zwracany typ to **`void`**, czyli brak zwracania wartości. Przykładowe rozwiązanie tego zadania:

```
public class MetodaWypisujacaWitajcie {
  public static void main(String[] args) {
    wypiszPowitanie();
  }

  public static void wypiszPowitanie() {
    System.out.println("Witajcie!");
  }
}
```

Wynik działania tego programu:

```
Witajcie!
```

### 7.2.2 Metoda odejmująca dwie liczby

*Napisz metodę, która wypisuje na ekran wynik odejmowania dwóch przesłanych do niej liczb i użyj jej w metodzie `main`.*

Metoda wypisująca wynik odejmowania przesłanych do niej argumentów nie będzie zwracać wartości, więc jej zwracany typ zdefiniujemy jako **`void`**. Przykładowe rozwiązanie tego zadania:

```
public class MetodaOdejmujacaDwieLiczby {
  public static void main(String[] args) {
    wypiszWynikOdejmowania(10, 10);
    wypiszWynikOdejmowania(-5, 10);
    wypiszWynikOdejmowania(2, 5);
    wypiszWynikOdejmowania(20, -10);
  }

  public static void wypiszWynikOdejmowania(int x, int y) {
    System.out.println(x - y);
  }
}
```

Wynik działania tego programu:

```
0
-15
-3
30
```

## 7.3 Pytania do zakresu i wywoływania metod, zmiennych lokalnych

1. Czym są zmienne lokalne?

> Zmienne lokalne to zmienne zdefiniowane w metodach. Są one dostępne tylko w metodach, w których zostały zdefiniowane.

2. Czy możemy z metody `abc` odwołać się do zmiennej lokalnej zdefiniowanej w metodzie `xyz`?

> Nie, zmienne lokalne są niedostępne poza metodami, w których zostały zdefiniowane.

3. Jaki jest zakres życia zmiennych lokalnych?

> Zmienne lokalne "żyją" w metodach, w których zostały zdefiniowane – gdy metoda się kończy, zmienne przestają istnieć.

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;
  }
}
```

> Powyższy program się nie skompiluje, ponieważ próbujemy użyć zmiennej `kwadrat` zanim zostanie zdefiniowana. Aby program był poprawny, należałoby przenieść definicję zmiennej `kwadrat` przed instrukcję `System.out.println`:

```
public class ZakresIWywolywanieMetodPytanie1 {
  public static void main(String[] args) {
    int kwadrat = policzKwadrat(10);
    System.out.println("Wynik: " + kwadrat);
  }

  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;
  }
}
```

Program się nie skompiluje, ponieważ w metodzie `main` próbujemy skorzystać z zmiennej lokalnej utworzonej w metodzie `policzKwadrat`. Zmienna `wynik` jest niedostępna poza metodą `policzKwadrat` – kompilator zgłosi błąd.

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;
  }
}
```

W pierwszej linii metody `main` częścią argumentu `System.out.println` jest wartość zwrócona z metody `policzKwadrat`, więc najpierw wykonane zostanie ciało metody `policzKwadrat`, a dopiero potem w metodzie `main` wypisany zostanie pierwszy komunikat. Dlatego, na ekranie zobaczymy:

```
Liczymy kwadrat liczby 10
Witajcie! Kwadrat 10 to: 100
Policzone!
```

## 7.4 Pytania do zwracania wartości

1. Jak zwrócić z metody wartość?

> Należy użyć słowa kluczowego `return`, po którym powinna nastąpić wartość, którą chcemy zwrócić. Na końcu powinniśmy umieścić średnik.

2. Czy metoda, która zwraca liczbę, może także zwrócić tekst (`String`)?

> Nie, zwrócić możemy wartość tylko jednego typu.

3. Czy metoda może nic nie zwracać, a jeśli tak, to jak to osiągnąć?

> Tak, metoda może nic nie zwracać. Wtedy definiujemy zwracany przez metodę typ jako `void`.

4. Czy możemy użyć słowa kluczowego `return` więcej niż raz w metodzie?

> Tak, słowa kluczowego `return` możemy używać wielokrotnie w metodach, np. w blokach instrukcji `if`.

5. Czy możemy użyć słowa kluczowego `return` w metodzie, która nic nie zwraca?

> Tak, ale nie może po nim nastąpić żadna wartość – w przeciwnym razie, program się nie skompiluje.

6. Jeżeli metoda ma zwrócić wartość typu `double`, ale nie użyjemy w niej `return`, czy kod się skompiluje?

> Nie, program się nie skompiluje. Kompilator jest w stanie wykryć sytuację, w której metoda nie zwraca wartości i nie pozwoli na skompilowanie takiego kodu.

7. Gdzie możemy użyć wartości zwracanej przez metodę?

> Wartość zwracaną przez metodę możemy użyć wszędzie tam, gdzie jest spodziewana wartość bądź wyrażenie, którego metoda może być częścią. Możemy m. in. przypisać wynik działania metody do zmiennej, możemy użyć wyniku metody jako argumentu do innej metody lub w warunku instrukcji warunkowej itp.

8. Czy wartość zwrócona przez metodę musi zostać zawsze użyta?

> Nie musi. Możemy zignorować wartość zwracaną przez metodę.

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;
}
```

> Ten fragment kodu nie jest poprawny, ponieważ wynik metody, która zwraca wartość typu **int**, próbujemy przypisać do zmiennej `tekst`, która jest typu `String`.

10. Które z poniższych metod są nieprawidłowe i dlaczego?

```
public static wypiszKomunikat() {
  System.out.println("Witajcie!");
}
```

> Ta metoda jest nieprawidłowa, ponieważ przed jej nazwą brakuje zwracanego typu.

---

```
public static void x() {}
```

> Ta metoda jest poprawna. Nic nie zwraca, nie przyjmuje argumentów i nie wykonuje żadnych instrukcji.

---

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

> Ta metoda jest niepoprawna, ponieważ próbuje zwrócić wartość pomimo, że jej zwracany typ to **void**, czyli metoda nie powinna zwracać jakiejkolwiek wartości.

---

```
public static int doKwadratu(int c) {
  int wynik = c * c;
}
```

> Ta metoda jest niepoprawna, ponieważ nie zwraca żadnej wartości, a powinna, ponieważ typ **int** jest zdefiniowany jako typ wartości, który ta metoda powinna zwracać.

```
public static int podzielLiczby(int a, int b) {
  if (b == 0) {
    return;
  }
  return a / b;
}
```

Ta metoda jest niepoprawna, ponieważ istnieje taka ścieżka wykonania tego programu, w której metoda nie zwróci wartości. Jeżeli `b` będzie równe `0`, to metoda nie zwróci wartości – kompilator jest w stanie wykryć ten problem już na etapie kompilacji.

---

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

Ta metoda jest niepoprawna, ponieważ metoda próbujemy zwrócić wartość typu `int`, a definiuje, że będzie zwracać wartość typu `String`.

---

```
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;
    }
  }
}
```

Ta metoda jest niepoprawna, ponieważ istnieje szansa, że nie zostanie zwrócona z niej żadna wartość – jeżeli `a > b` oraz `a <= c`, to metoda nie zwróci wartości. Kompilator wykrywa potencjalny problem i nie pozwali na skompilowanie powyższego kodu. Aby metoda była poprawna, powinniśmy po instrukcji warunkowej, sprawdzającej, czy `a > c`, dodać `return c;`

```
public static void wypiszKwadrat(int a) {
  System.out.println("Kwadrat wynosi: " + a * a);
  return;
  System.out.println("Policzone!");
}
```

> Metoda nie jest poprawna, ponieważ zaznaczona linia nigdy nie ma szansy na wykonanie ze względu na wcześniejsze użycie `return`. Kompilator jest w stanie to wykryć i nie pozwoli na kompilację.

---

```
public static void wypiszPowitanie {
  System.out.println("Witajcie!");
}
```

> Metoda jest niepoprawna, ponieważ brakuje nawiasów ( ) po nazwie metody. Nawiasy ( ) są wymagane nawet w przypadku, gdy metoda nie przyjmuje żadnych argumentów.

## *7.5 Zadania do zwracania wartości*

### 7.5.1 Metoda podnosząca do sześcianu

*Napisz metodę, która zwróci liczbę przesłaną jako argument podniesioną do sześcianu.*

Przykładowe rozwiązanie tego zadania:

```
public class MetodaPodnoszacaDoSzescianu {
  public static void main(String[] args) {
    System.out.println("0 do 3 potegi to " + szescian(0));
    System.out.println("1 do 3 potegi to " + szescian(1));
    System.out.println("3 do 3 potegi to " + szescian(3));
    System.out.println("10 do 3 potegi to " + szescian(10));
  }

  public static int szescian(int liczba) {
    return liczba * liczba * liczba;
  }
}
```

Wynik działania powyższego programu:

```
0 do 3 potegi to 0
1 do 3 potegi to 1
3 do 3 potegi to 27
10 do 3 potegi to 1000
```

### 7.5.2 Metoda wypisująca gwiazdki

*Napisz metodę, która wypisze podaną liczbę gwiazdek (znak *) na ekran.*

W tym programie skorzystamy z pętli `for`, aby wypisać odpowiednią liczbę gwiazdek. Na końcu metody `wypiszGwiazdki` skorzystamy z metody `System.out.println` bez podawania argumentu, aby przejść na ekranie do nowej linii (aby gwiazdki wypisywane w kolejnych wywołaniach metody `wypiszGwiazdki` wypisywane były od nowej linii):

```
public class MetodaWypisujacaGwiazdki {
  public static void main(String[] args) {
    wypiszGwiazdki(10);
    wypiszGwiazdki(0);
    wypiszGwiazdki(1);
    wypiszGwiazdki(20);
  }

  public static void wypiszGwiazdki(int ileGwiazdek) {
    for (int i = 0; i < ileGwiazdek; i++) {
      System.out.print("*");
    }
    System.out.println(); // nowa linia
  }
}
```

Wynik działania tego programu:

```
**********

*
********************
```

## 7.6 Pytania do argumentów metod i metod typu String

1. Do czego służą argumenty metod?

> Argumenty metod to dane wejściowe, które przekazujemy do metody. Argumenty są wykorzystywane przez metody do wykonania określonej operacji.

2. Czy metody mogą przyjmować zero argumentów?

> Tak, metody mogą przyjmować zero argumentów.

3. Czy metodę, która przyjmuje jeden argument – liczbę typu `int`, możemy wywołać bez podania żadnego argumentu?

> Nie, wszystkie argumenty metody są wymagane i nie możemy pominąć żadnego z nich podczas wywoływania metody.

4. Czy kolejność argumentów ma znaczenie?

> Tak, kolejność argumentów ma znaczenie – metoda przyjmująca `String` oraz liczbę typu `int` to inna metoda, niż metoda przyjmująca liczbę typu `int` i `String`.

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?

> Po zakończeniu metody, zmienna użyta jako argument będzie miała swoją oryginalną wartość. Zmiany argumentów w metodach nie są propagowane w przypadku wartości typów prymitywnych.

6. Jak można udokumentować metodę, opisując jej działanie, parametry, zwracany typ itp.?

> Aby udokumentować metodę, stosujemy komentarze dokumentacyjne, które zaczynają się od znaków `/**` i kończą się znakami `*/`. W komentarzu dokumentacyjnym może użyć specjalnego zapisu `@param nazwaParametru opis` oraz `@return opis` zwracanej wartości, by opisać argumenty metody i wartość, którą zwraca.

7. Czy poniższy kod jest poprawny?

```
public class Pytanie {
  public static void main(String[] args) {
    wypiszKomunikat;
  }

  public static void wypiszKomunikat() {
    System.out.println("Witajcie!");
  }
}
```

Ten kod nie jest poprawny, ponieważ podczas wywoływania metody należy po jej nazwie umieścić nawiasy ( ). W metodzie `main`, linia `wypiszKomunikat;` powinna być zastąpiona linią `wypiszKomunikat();` aby kod był poprawny.

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];
    }
  }
}
```

Na ekranie zobaczymy wartość `49`. Dlaczego tak się stało? Zmiany wartości typów złożonych (do których tablice przynależą) są wykonywane na oryginalnym obiekcie, a nie na jego kopii. Dlaczego jednak tak się stało, skoro w zaznaczonej linii tworzymy nową zmienną `tablica` i przypisujemy do niej wartość argumentu `tab`? Powodem jest to, że nie tworzymy nowej tablicy, a jedynie zmienną, która na nią pokazuje. O takim przypadku rozmawialiśmy w rozdziale o tablicach – w metodzie `metoda`, zarówno zmienna `tablica`, jak i zmienna `tab`, wskazują na tę samą tablicę – tablicę, która została przekazana jako argument w metodzie `main`.

9. Które z poniższych sygnatur (pomijamy brak ciała metod) metod są nieprawidłowe i dlaczego?
   - `public static void metoda(wiadomosc String)`

Ta metoda jest nieprawidłowa, ponieważ typ argumentu i jego nazwa są zapisane w nieprawidłowej kolejności – najpierw powinien zostać zdefiniowany typ argumentu, a dopiero potem jego nazwa. Poprawnie zapisany argument to `String wiadomosc`.

- `public static void metoda`

Ta metoda jest nieprawidłowa, ponieważ nie ma nawiasów ( ) po swojej nazwie.

- `public static void metoda(int byte, char znak)`

Ta metoda jest nieprawidłowa, ponieważ jako nazwy pierwszego argumentu używa nazwy zastrzeżonej w języku Java – `byte` to słowo kluczowe, jeden z typów prymitywnych.

- `public static void metoda()`

Sygnatura tej metody jest prawidłowa. Metoda ta nie przyjmuje żadnych argumentów i nic nie zwraca.

- `public static void metoda(int #numerPracownika)`

Ta metoda jest nieprawidłowa, ponieważ jej argument ma nieprawidłową nazwę. Nazwy nie mogą zawierać znaku #.

- `public static void metoda(string wiadomosc)`

Ta metoda jest nieprawidłowa, ponieważ typ argumentu jest nieprawidłowy – typ `string` nie istnieje w języku Java – poprawnie zapisany argument to `String wiadomosc`.

- `public static void metoda(int)`

Ta metoda jest nieprawidłowa, ponieważ pierwszy argument nie ma nazwy.

- `public static void metoda(int liczba String wiadomosc)`

Ta metoda jest nieprawidłowa, ponieważ brakuje znaku przecinka, który oddzieliłby od siebie argumenty powyższej metody.

- `public static void metoda(int _numerPracownika)`

Ta metoda jest poprawna – nazwy w języku Java mogą zaczynać się od podkreślenia _

- `public static void metoda(double pi = 3.14)`

Ta metoda jest nieprawidłowa, ponieważ argumenty w języku Java nie mogą mieć wartości domyślnych.

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);
  }
}
```

Ten program w ogóle się nie skompiluje, ponieważ argumenty przesyłane do metody `metoda` są w złej kolejności. Metoda ta jako pierwszego argumentu spodziewa się liczby całkowitej, a jako drugiego – liczby rzeczywistej. W metodzie `main` podajemy te argumenty w odwrotnej kolejności.

11. Wymień kilka metod, które udostępnia typ `String`.

Typ `String` udostępnia, m. in, następujące metody: `length`, `charAt`, `toLowerCase`, `toUpperCase`, `endsWith`, `startsWith`, `contains`, `replace`, `split`, `substring`, `equals`, `equalsIgnoreCase`.

12. Jaki jest indeks pierwszego znaku w każdym stringu? A jaki ostatniego?

Indeks pierwszego znaku w stringu to `0`, a ostatniego – liczba znaków w stringu minus jeden.

13. Czy małe i wielkie litery są rozróżniane w stringach?

Tak, małe i wielkie litery w stringach traktowane są jako różne litery.

14. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";
tekst.replace("Wi", "Pamie");
System.out.println(tekst);
```

Na ekranie zobaczymy `"Witajcie!"`. Metoda `replace` (oraz pozostałe metody typu `String`) nie zmienia oryginalnej wartości zapisanej w zmiennej `tekst`, lecz zwraca nową wartość z zastąpioną częścią stringa. W tym przypadku, zwracana jest wartość `Pamietajcie!`, ale nie przypisujemy jej do żadnej zmiennej.

15. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";
System.out.println(tekst.charAt(tekst.length()));
```

Po uruchomieniu, program zakończy się błędem `StringIndexOutOfBoundsException`, ponieważ próbujemy odnieść się do znak o indeksie `9` w stringu. Ostatni indeks znaku w każdym stringu jest o jeden mniejszy, niż liczba znaków w tym stringu. W tym przypadku, ostatni znak ma indeks `8`. Indeks `9` wychodzi poza zakres i powoduje błąd działania programu.

16. Jaki będzie wynik działania poniższego kodu? Co zawiera tablica `tab`?

```
String tekst = "Witajcie!";
String[] tab = tekst.split(",");
```

Metoda `split` zwróci tablicę z jednym elementem – wartością `"Witajcie!"`. Wynika to z tego, że podanego separatora (przecinka) nie ma w stringu `"Witajcie!"`, więc string ten nie zostanie w ogóle podzielony. Zostanie on zwrócony w jednoelementowej tablicy.

17. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";

if (tekst.contains("witajcie")) {
  System.out.println("Zmienna zawiera slowo witajcie.");
}
```

Na ekran niezostanie wypisany komunikat, ponieważ wielkość znaków ma znaczenie podczas korzystania z łańcuchów tekstowych. Zmienna `tekst` ma wartość `"Witajcie!"`, natomiast do metody `contains` podajemy jako argument string `"witajcie"`, z pierwszą małą literą.

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!");
}
```

To, co próbuje zrobić ten fragment kodu, to podzielić string `"Ala ma kota"`, korzystając ze spacji `" "` jako separatora, a następnie "złożyć" z powrotem ten string i porównać go do oryginalnej wartości ze zmiennej `tekst`. Na ekranie nie zobaczymy jednak komunikatu `"Rowne!"`, ponieważ do porównywania stringów powinniśmy używać metody `equals` z typu `String`, a nie operatora porównania ==.

19. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";
String fragment = tekst.substring(0, 5);

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

Na ekranie zobaczymy komunikat `Witaj`. Metoda `substring` zwraca fragment stringu, zawartego pomiędzy pierwszym, a drugim indeksem podanym jako argument, z wyłączeniem znaku znajdującego się na końcu tego zakresu.

20. Jaki będzie wynik działania poniższego kodu?

```
String tekst = "Witajcie!";

if (tekst.equals("witajcie!") {
  System.out.println("Rowne!");
}
```

Na ekranie niezostanie wypisany komunikat, ponieważ stringi `"Witajcie!"` i `"witajcie!"` nie są takie same – wielkość liter ma znaczenie przy porównywaniu łańcuchów tekstowych.

## 7.7 Zadania do argumentów metod i metod typu String

### 7.7.1 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`.*

W tym programie musimy skorzystać z faktu, że ostatni indeks znaku w każdym stringu jest równy liczbie znaków w tym stringu minus jeden (ponieważ indeksy zaczynają się od `0`):

```
public class MetodaZwracajacaOstatniZnak {
  public static void main(String[] args) {
    System.out.println(zwrocOstatniZnak("Witaj!"));
    System.out.println(zwrocOstatniZnak("Ala ma kota"));
    System.out.println(zwrocOstatniZnak("?"));
  }

  public static char zwrocOstatniZnak(String s) {
    return s.charAt(s.length() - 1);
  }
}
```

Przykładowe wykonanie tego programu:

```
!
a
?
```

### 7.7.2 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`***.*

W rozdziale o pętlach rozwiązywaliśmy podobne zadanie, ale wtedy nie korzystaliśmy z pętli. Możemy skorzystać z napisanego wtedy fragmentu kodu, który był odpowiedzialny za sprawdzenie, czy słowo jest palindromem, czy nie.

Zwróćmy uwagę, że w zadaniu napisane jest, że zarówno słowo `kajak`, jak i `Kajak`, powinny być uznane za palindromy – będziemy więc musieli w jakiś sposób traktować małe i wielkie litery jako takie same litery – tutaj z pomocą przyjdzie nam metoda `toLowerCase` typu `String`, poznana w rozdziale o metodach.

Przykładowe rozwiązanie tego zadania:

```
public class MetodaCzyPalindrom {
  public static void main(String[] args) {
    System.out.println(
        "Czy kajak to palindrom? " + czyPalindrom("kajak")
    );

    System.out.println(
        "Czy Kajak to palindrom? " + czyPalindrom("Kajak")
    );

    System.out.println(
        "Czy kot to palindrom? " + czyPalindrom("kot")
    );
  }

  public static boolean czyPalindrom(String slowo) {
    String slowoMaleLitery = slowo.toLowerCase();
    int dlugoscSlowa = slowo.length();

    for (int i = 0; i < dlugoscSlowa / 2; i++) {
      if (slowoMaleLitery.charAt(i) != slowoMaleLitery.charAt(dlugoscSlowa - 1 - i)) {
        return false;
      }
    }

    return true;
  }
}
```

W metodzie `czyPalindrom` przypisujemy do zmiennej `slowoMaleLitery` wynik działania metody `toLowerCase` na przesłanym do metody argumencie, dzięki czemu wielkość liter nie będzie miała dla nas znaczenia podczas sprawdzania, czy słowo jest palindromem, czy nie, ponieważ wszystkie litery w `slowoMaleLitery` będą małymi literami.

W pętli sprawdzamy, czy słowo jest palindromem – jeżeli znajdziemy chociaż jedną różnicą w znakach na odpowiednich pozycjach, to możemy od razu zwrócić **`false`** i zakończyć tym samym metodą, ponieważ wystarczy jedna różnica, aby słowo nie było palindromem.

Jeżeli jednak pętla się zakończy, a różnica nie zostanie znaleziona, to zwracamy wartość **`true`**,

ponieważ w takim przypadku jesteśmy pewni, że słowo jest palindromem. Dokładniejszy opis tego algorytmu sprawdzania, czy słowo jest palindromem, znajduje się pod zadaniem Palindrom.

Przykładowe uruchomienie tego programu:

```
Czy kajak to palindrom? true
Czy Kajak to palindrom? true
Czy kot to palindrom? false
```

### 7.7.3 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.*

W tym programie musimy przeiterować przez całą tablicę i do zmiennej, która będzie przechowywała sumę liczb, kolejno dodawać elementy tablicy. Skorzystamy z pętli **for**-each:

```
public class MetodaSumujacaLiczbyWTablicy {
  public static void main(String[] args) {
    int[] tab1 = { 1, 7, 20, 100 };
    int[] tab2 = { }; // tablica z zeroma elementami
    int[] tab3 = { -1, 0, 1 };

    System.out.println("Suma liczb tab1: " + sumaLiczb(tab1));
    System.out.println("Suma liczb tab2: " + sumaLiczb(tab2));
    System.out.println("Suma liczb tab3: " + sumaLiczb(tab3));
  }

  public static int sumaLiczb(int[] tab) {
    int suma = 0;

    for (int i : tab) {
      suma += i;
    }

    return suma;
  }
}
```

Przykładowe wykonanie tego programu:

```
Suma liczb tab1: 128
Suma liczb tab2: 0
Suma liczb tab3: 0
```

### 7.7.4 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, którą należy napisać w tym zadaniu:*

```
int liczbaLiterA = zliczWystapienia("Ala ma kota", 'a');
```

W tym programie musimy zwrócić uwagę, że mamy do czynienia ze stringiem i ze znakami. W metodzie, którą napiszemy, przejdziemy w pętli przez wszystkie znaki przesłanego jako argumentu stringa. Każdy znak pobierzemy korzystając z metody `charAt`. Do porównywania znaków użyjemy operatora == zamiast metody `equals`, ponieważ znaki typu `char` to wartości prymitywne, a wartości prymitywne porównujemy za pomocą operatora ==:

```
public class MetodaZliczajacaZnakWStringu {
  public static void main(String[] args) {
    System.out.println(zliczZnaki("Ala ma kota", 'a'));
    System.out.println(zliczZnaki("Ala ma kota", 'A'));
    System.out.println(zliczZnaki("Ala ma kota", 'x'));
  }

  public static int zliczZnaki(String tekst, char znak) {
    int liczbaZnakow = 0;

    for (int i = 0; i < tekst.length(); i++) {
      if (tekst.charAt(i) == znak) {
        liczbaZnakow++;
      }
    }

    return liczbaZnakow;
  }
}
```

Zwróćmy uwagę, że znak pobrany za pomocą `charAt` porównujemy do wartości `znak`, przesłanej jako argument, za pomocą operatora ==. Wynika to z faktu, że typ `char` to typ prymitywny, a jego wartości przyrównujemy do siebie właśnie za pomocą operatora ==.

Przykładowe wykonanie tego programu:

```
3
1
0
```

## 7.8 Pytania do przeładowywania metod

1. Czym jest przeładowywanie metod?

> Przeładowywanie metod to pisanie metod o tych samych nazwach, ale różnych argumentach. Metody te muszą różnić się liczbą argumentów, kolejnością, lub typami.

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)
```

> Te metody nie są poprawnie przeładowane, ponieważ różnią się jedynie zwracanym typem, a zwracany typ nie ma znaczenia podczas przeładowywania metod.

   b)

```
public static void metoda(int liczba, int drugaLiczba)
public static void metoda(int liczba)
```

> Te metody są poprawnie przeładowane – mają różną liczbę argumentów.

   c)

```
public static void metoda(int liczba)
public static void metoda(int liczba)
```

> Te metody nie są poprawnie przeładowane, ponieważ są identyczne – w ogóle się nie różnią.

   d)

```
public static void metoda(double liczba)
public static void metoda(int liczba)
```

> Te metody są poprawnie przeładowane, ponieważ przyjmują argumenty różnych typów.

   e)

```
public static void metoda(double liczba, String tekst)
public static void metoda(String tekst, double liczba)
```

> Te metody są poprawnie przeładowane, ponieważ ich argumenty mają inną kolejność.

f)

```
public static void metoda(String tekst, double liczba)
public static void metoda(String komunikat, double wartosc)
```

Te metody nie są poprawnie przeładowane, ponieważ różnią się jedynie nazwami argumentów, a nazwy argumentów nie mają znaczenia podczas przeładowywania.

g)

```
public static void metoda()
public static void metoda(String komunikat)
```

Te metody są poprawnie przeładowane, ponieważ różnią się liczbą argumentów.

h)

```
public static void metoda(String komunikat)
public static void Metoda(String komunikat)
```

**Te metody są poprawne, ale nie są przeładowane**, ponieważ mają różne nazwy. Pierwsza z nich nazywa się `metoda`, a druga `Metoda` – wielkość znaków ma znaczenie w języku Java.

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.

Podczas przeładowywanie metod znaczenie mają: b) liczba argumentów, c) typy argumentów, oraz d) kolejność argumentów.

## 7.9 Zadania do przeładowywania metod

### 7.9.1 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[]`

W tym zadaniu musimy napisać kilka wersji metody o tej samej nazwie, korzystając z mechanizmu przeładowywania metod. Dla przykładu, nazwiemy tą metodę `equals` – tak samo, jak metoda typu `String` (nazwa naszej metody w żaden sposób nie koliduje z nazwą metody typu `String`, ponieważ są one zdefiniowane w różnych miejscach).

Pierwsze cztery metody są proste – wystarczy zwrócić wynik przyrównania argumentów metody.

Jednakże, w przypadku typu `String`, musimy pamiętać o użyciu metody `equals`.

W przypadku tablic wartości typu **`int`**, musimy sprawdzić rozmiary obu tablic i jeżeli są takie same, porównać wszystkie elementy i zwrócić **`false`**, jeżeli znajdziemy różnicę. Jeżeli różnicy nie znajdziemy, to zwrócimy **`true`**.

W przypadku tablicy wartości typu `String`, musimy zadziałać podobnie, jak w przypadku tablicy wartości typu **`int`** z tym, że porównując wartości musimy pamiętać o użyciu metody `equals`, a nie operatora ==.

Przykładowe rozwiązanie tego zadania:

```
public class MetodaPorownujacaSwojeArgumenty {
  public static void main(String[] args) {
    System.out.println("Czy 1 == 2? " + equals(1, 2));
    System.out.println("Czy 0 == 0? " + equals(0, 0));

    System.out.println(
        "Czy 3.14 == 3.14? " + equals(3.14, 3.14)
    );
    System.out.println(
        "Czy 5.55 == 0? " + equals(5.55, 0)
    );

    System.out.println(
        "Czy true == false? " + equals(true, false)
    );
    System.out.println(
        "Czy false == false? " + equals(false, false)
    );
```

```
    System.out.println(
        "Czy 'a' == 'A'? " + equals('a', 'A')
    );
    System.out.println(
        "Czy 'x' == 'x'? " + equals('x', 'x')
    );

    System.out.println(
        "Czy kot == Kot? " + equals("kot", "Kot")
    );
    System.out.println(
        "Czy kot == kot? " + equals("kot", "kot")
    );

    int[] tab1 = { 1, 2, 3 };
    int[] tab2 = { 1, 2, 3, 4 };
    int[] tab3 = { 1, 2, 3 };

    System.out.println("Czy tab1 == tab2? " + equals(tab1, tab2));
    System.out.println("Czy tab1 == tab3? " + equals(tab1, tab3));

    String[] str1 = { "Ala", "ma", "kota" };
    String[] str2 = { "ALA", "MA", "KOTA" };
    String[] str3 = { "Ala", "ma", "kota" };

    System.out.println("Czy str1 == str2? " + equals(str1, str2));
    System.out.println("Czy str1 == str3? " + equals(str1, str3));
  }

  public static boolean equals(int a, int b) {
    return a == b;
  }

  public static boolean equals(double a, double b) {
    return a == b;
  }

  public static boolean equals(boolean a, boolean b) {
    return a == b;
  }

  public static boolean equals(char a, char b) {
    return a == b;
  }

  public static boolean equals(String a, String b) {
    return a.equals(b);
  }

  public static boolean equals(int[] a, int[] b) {
    if (a.length != b.length) {
      return false;
    }

    for (int i = 0; i < a.length; i++) {
      if (a[i] != b[i]) {
        return false;
      }
    }
```

```
    return true;
  }

  public static boolean equals(String[] a, String[] b) {
    if (a.length != b.length) {
      return false;
    }

    for (int i = 0; i < a.length; i++) {
      if (!a[i].equals(b[i])) {
        return false;
      }
    }

    return true;
  }
}
```

Wynik działania tego programu:

```
Czy 1 == 2? false
Czy 0 == 0? true
Czy 3.14 == 3.14? true
Czy 5.55 == 0? false
Czy true == false? false
Czy false == false? true
Czy 'a' == 'A'? false
Czy 'x' == 'x'? true
Czy kot == Kot? false
Czy kot == kot? true
Czy tab1 == tab2? false
Czy tab1 == tab3? true
Czy str1 == str2? false
Czy str1 == str3? True
```

# 8 Rozdział VIII – Testowanie kodu

## 8.1 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();
}
```

> Ta metoda nie powinna sama pobierać wartości od użytkownika, lecz powinna przyjmować argument, którego wartość do sześcianu ma zwrócić. Dzięki temu, w łatwy sposób będziemy mogli ją przetestować dla różnych wartości:

```
public static int szescian(int liczba) {
  return liczba * liczba * liczba;
}
```

   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);
}
```

> Ta metoda nie powinna wypisywać wyliczonej silni – zamiast tego, lepiej byłoby, gdyby metoda ta zwracała policzoną wartość. Dzięki temu, moglibyśmy w testach sprawdzić zwracane przez nią wyniki dla różnych argumentów:

```
public static int policzSilnie(int x) {
  int wynik = 1;

  for (int i = 1; i <= x; i++) {
    wynik = wynik * i;
  }

  return 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;
}
```

Ta metoda nie jest poprawna z dwóch powodów: po pierwsze, zmienna `liczbaZnakow` jest inicjalizowana nieprawidłową wartością – powinno to być `0`, a nie `1`. Po drugie, zmienna pętli `i` wyjdzie poza zakres znaków w argumencie `tekst` – warunek pętli używa nieprawidłowego operatora. Warunek ten powinien zostać zapisany przy pomocy operatora <.

Przypadki testowe, jakie powinniśmy rozważyć:

- argument `tekst` powinien być pustym stringiem `""`,
- szukanym znakiem powinna być litera np. `'a'`, a argument tekst powinien zawierać zarówno małe, jak i wielkie litery,
- argument `tekst` nie powinien zawierać ani jednego wystąpienia szukanego znaku,
- szukany znak powinien być na początku i na końcu argumentu `tekst`,
- argument `tekst` powinien składać się z jednego znaku – takiego, który został przesłany jako argument `znak`.

3. Jakie testy jednostkowe należałoby napisać do metody, która sortuje przesłaną do niej tablicę rosnąco?

Powinniśmy wziąć pod uwagę następujące przypadki testowe:

- sortowanie pustej tablicy,
- sortowanie tablicy, która jest posortowana,
- sortowanie tablicy, która jest posortowana w odwrotnej kolejności (czyli malejąco),
- sortowanie tablicy, której największy element jest pierwszym elementem,
- sortowanie tablicy, której najmniejszy element jest ostatnim elementem w tablicy,
- sortowanie tablicy z kilkoma dowolnymi wartościami,
- sortowanie bardzo dużej tablicy (jej zawartość można by wygenerować w pętli).

## 8.2 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.

### 8.2.1 Testy czyParzysta

*Napisz testy oraz metodę, która odpowiada na pytanie, czy podana liczba jest parzysta.*

Testowanie przeprowadzimy na następującym zbiorze danych testowych:

- liczba parzysta dodatnia,
- liczba parzysta ujemna,
- liczba nieparzysta dodatnia,
- liczba nieparzysta ujemna,
- liczba `0`.

Poniżej znajduje się przykładowe rozwiązanie tego zadania:

```
public class TestyCzyParzysta {
  public static void main(String[] args) {
    czyParzysta_liczbaParzystaDodatnia_zwrociParzysta();
    czyParzysta_liczbaParzystaUjemna_zwrociParzysta();
    czyParzysta_liczbaNieparzystaDodatnia_zwrociNieparzysta();
    czyParzysta_liczbaNieparzystaUjemna_zwrociNieparzysta();
    czyParzysta_liczbaZero_zwrociParzysta();
  }

  public static boolean czyParzysta(int liczba) {
    return liczba % 2 == 0;
  }

  public static void czyParzysta_liczbaParzystaDodatnia_zwrociParzysta() {
    assertEquals(true, czyParzysta(10));
  }

  public static void czyParzysta_liczbaParzystaUjemna_zwrociParzysta() {
    assertEquals(true, czyParzysta(-2));
  }

  public static void czyParzysta_liczbaNieparzystaDodatnia_zwrociNieparzysta() {
    assertEquals(false, czyParzysta(999));
  }

  public static void czyParzysta_liczbaNieparzystaUjemna_zwrociNieparzysta() {
    assertEquals(false, czyParzysta(-9));
  }
```

```
  public static void czyParzysta_liczbaZero_zwrociParzysta() {
    assertEquals(true, czyParzysta(0));
  }

  public static void assertEquals(boolean expected, boolean actual) {
    if (expected != actual) {
      System.out.println("Spodziewano sie " + actual +
          ", ale otrzymano: " + expected);
    }
  }
}
```

Program ten zawiera pięć testów, które wykorzystują dane spisane powyżej. Metody testowe nazywamy zgodnie z konwencją, którą poznaliśmy w rozdziale o testowaniu kodu – pierwszy człon nazwy to nazwa testowanej metody, drugi człon to dane wejściowe, a na końcu znajduje się spodziewany wynik testu.

W tym programie wykorzystujemy pomocniczą metodę `assertEquals`, która wypisuje na ekranie komunikat w przypadku, gdy jej argumenty nie są sobie równe. W każdej z metod testowych korzystamy z niej, by porównała ona spodziewaną wartość oraz wynik działania metody `czyParzysta`. Dzięki użyciu metody `assertEquals`, kod testów jest bardzo krótki i czytelny.

Po uruchomieniu programu, na ekranie nie zobaczymy żadnego komunikatu – nasza metoda `czyParzysta` działa poprawnie!

### 8.2.2 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.*

W tym programie warto sprawdzić zachowanie testowanej metody dla pierwszej liczby większej od zera, dla pierwszej liczby całkowitej mniejszej od zera, oraz dla zera. Możemy też sprawdzić wynik działania dla jeszcze jednej liczby dodatniej i jednej ujemnej. Będziemy więc mieli pięć testów.

Opisana powyżej funkcja, która zwraca `-1` dla liczb ujemnych, `0` dla zera, i `1` dla liczb dodatnich większych od `0`, nazywa się w matematyce *sign* – tak też będzie nazywała się testowana metoda w poniższym rozwiązaniu:

```
public class TestySprawdzaniaZnakuLiczby {
  public static void main(String[] args) {
    sign_pierwszaUjemna_zwrociMinusJeden();
    sign_zero_zwrociZero();
    sign_pierwszaDodatnia_zwrociJeden();
    sign_liczbaUjemna_zwrociMinusJeden();
    sign_liczbaDodatnia_zwrociJeden();
  }

  public static int sign(int liczba) {
    return liczba == 0 ? 0 : (liczba < 0 ? -1 : 1);
  }

  public static void sign_pierwszaUjemna_zwrociMinusJeden() {
    assertEquals(-1, sign(-1));
  }

  public static void sign_zero_zwrociZero() {
    assertEquals(0, sign(0));
  }

  public static void sign_pierwszaDodatnia_zwrociJeden() {
    assertEquals(1, sign(1));
  }

  public static void sign_liczbaUjemna_zwrociMinusJeden() {
    assertEquals(-1, sign(-20));
  }

  public static void sign_liczbaDodatnia_zwrociJeden() {
    assertEquals(1, sign(100));
  }

  public static void assertEquals(int expected, int actual) {
    if (expected != actual) {
      System.out.println("Spodziewano sie liczby " + actual +
          ", ale otrzymano: " + expected);
    }
  }
}
```

Podobnie jak w przypadku rozwiązania do poprzedniego zadania, używamy nazewnictwa metod

testowych, zgodnie z którym najpierw piszemy nazwę testowanej metody, następnie opis danych wejściowych, a na końcu – spodziewaną wartość.

Ponownie korzystamy z pomocniczej metody `assertEquals` – tym razem do porównywania wartości typu `int`.

Program nie wypisuje nic na ekran, co sugeruje nam, że testowana metoda `sign` działa poprawnie.

### 8.2.3 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 trzecim 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.*

Podczas testowania tej metody, warto sprawdzić wiele różnych przypadków:

- Jak zachowa się metoda, gdy przesłana do niej tablica będzie pusta?
- Jak zachowa się metoda, gdy szukana liczba będzie pierwszym elementem tablicy, a jak, gdy będzie ostatnim elementem? (dwa przypadki testowe)
- Jak zachowa się metoda, gdy przesłana tablica nie będzie pusta, ale nie będzie w niej szukanego elementu?
- Czy metoda zadziała poprawnie, gdy szukana liczba będzie znajdowała się gdzieś w środku tablicy?
- Czy metoda zwróci poprawnie pierwszy indeks szukanego elementu, jeżeli szukana liczba będzie występować więcej niż raz w tablicy?

Zgodnie z powyższym, będziemy mieli sześć metod testowych – oto przykładowe rozwiązanie tego zadania:

```
public class TestyZwracaniaIndeksuSzukanegoElementu {
  public static void main(String[] args) {
    indeksElementu_pustaTablica_zwrociMinusJeden();
    indeksElementu_elementNaPoczatku_zwrociIndeksZero();
    indeksElementu_elementNaKoncu_zwrociIndeksKoncowy();
    indeksElementu_elementuNieMa_zwrociMinusJeden();
    indeksElementu_elementWSrodku_zwrociIndeksElementu();
    indeksElementu_elementWieleRazy_zwrociIndeksPierwszego();
  }

  public static int indeksElementu(int[] tab, int szukanaLiczba) {
    for (int i = 0; i < tab.length; i++) {
      if (tab[i] == szukanaLiczba) {
        return i;
      }
    }
    // jezeli wartosci nie znaleziono, zwracamy -1
    return -1;
  }
```

```
public static void indeksElementu_pustaTablica_zwrociMinusJeden() {
  // given
  int[] tablica = {};
  int szukanaLiczba = 5;

  // when
  int indeksLiczby = indeksElementu(tablica, szukanaLiczba);

  // then
  assertEquals(-1, indeksLiczby);
}

public static void indeksElementu_elementNaPoczatku_zwrociIndeksZero() {
  // given
  int[] tablica = { 5, 10, 15, 100, 200 };
  int szukanaLiczba = 5;

  // when
  int indeksLiczby = indeksElementu(tablica, szukanaLiczba);

  // then
  assertEquals(0, indeksLiczby);
}

public static void indeksElementu_elementNaKoncu_zwrociIndeksKoncowy() {
  // given
  int[] tablica = { 5, 10, 15, 100, 200 };
  int szukanaLiczba = 200;

  // when
  int indeksLiczby = indeksElementu(tablica, szukanaLiczba);

  // then
  assertEquals(tablica.length - 1, indeksLiczby);
}

public static void indeksElementu_elementuNieMa_zwrociMinusJeden() {
  // given
  int[] tablica = { 5, 10, 15, 100, 200 };
  int szukanaLiczba = 500;

  // when
  int indeksLiczby = indeksElementu(tablica, szukanaLiczba);

  // then
  assertEquals(-1, indeksLiczby);
}

public static void indeksElementu_elementWSrodku_zwrociIndeksElementu() {
  // given
  int[] tablica = { 5, 10, 15, 100, 200 };
  int szukanaLiczba = 15;

  // when
  int indeksLiczby = indeksElementu(tablica, szukanaLiczba);

  // then
  assertEquals(2, indeksLiczby);
}
```

```
  public static void indeksElementu_elementWieleRazy_zwrociIndeksPierwszego() {
    // given
    int[] tablica = { 1, 2, 3, 2, 5 };
    int szukanaLiczba = 2;

    // when
    int indeksLiczby = indeksElementu(tablica, szukanaLiczba);

    // then
    assertEquals(1, indeksLiczby);
  }

  public static void assertEquals(int expected, int actual) {
    if (expected != actual) {
      System.out.println("Spodziewano sie liczby " + actual +
          ", ale otrzymano: " + expected);
    }
  }
}
```

W tym rozwiązaniu korzystamy z konwencji *given..when..then*. W każdym teście, najpierw przygotowujemy dane testowe (*given*), następnie wywołujemy testowaną metodą z danymi testowymi (*when*), a na końcu sprawdzamy wynik (*then*). Do porównania wartości wynikowej ze spodziewaną korzystamy z pomocniczej metody `assertEquals` (tak jak w dwóch poprzednich zadaniach).

Podobnie, jak w rozwiązaniach dwóch poprzednich zadań, także i tym razem nazywamy metody testowe stosując konwencję: `nazwaTestowanejMetody_daneWejsciowe_spodziewanyWynik`.

# 9 Rozdział IX – Klasy

## 9.1 Pytania do rozdziału "Czym są klasy i do czego służą?"

1. Czym różni się klasa od obiektu?

> Klasa to definicja nowego typu złożonego. W klasach zawarta jest informacja, jakie pola będą miały obiekty danej klasy, a także jakie metody będzie można wywoływać na rzecz tych obiektów. Obiekty to konkretne egzemplarze klasy – posiadają one własny, oddzielny stan (pola) oraz zestaw operacji, które można na nich wykonywać (metody).

2. Z czego składają się klasy?

> Klasy składają się z pól, które definiują stan wewnętrzny obiektów klas, oraz z metod, które można na rzecz obiektów tych klas wywoływać.

3. Jak utworzyć nowy obiekt klasy?

> Służy do tego słowo kluczowe `new`. Korzystając z niego, należy dopisać nazwę klasy, której obiekt chcemy utworzyć, po którym powinny następować nawiasy () oraz średnik, na przykład:

```
Samochod samochod = new Samochod();
```

4. Do czego służy i jakie wymagania powinna spełniać metoda `toString`?

> Jest to specjalna, pomocnicza metoda – zwraca ona tekstową reprezentację obiektu, na rzecz którego została wywołana. Jeżeli obiekt używany jest w kontekście, w którym spodziewany jest `String`, to metoda `toString` zostania automatycznie wywołana dla naszej wygody:

```
String opis = "Opis zmiennej samochod to: " + samochod;
```

> W powyższej linii, na obiekcie `samochod` zostanie automatycznie wywołana metoda `toString`, która zwróci tekstową reprezentację obiektu `samochod`.
>
> Metoda `toString` powinna spełniać następujące wymagania:

- Musi nazywać się `toString`.
- Musi zwracać `String`.
- Nie może przyjmować żadnych argumentów.
- Musi być publiczna (tzn. należy użyć modyfikatora `public`) i nie może być statyczna (nie może mieć modyfikatora `static`).

5. Czy poniższa linia kodu jest poprawna?

```
Samochod samochod = Samochod();
```

> To zależy – jeżeli chodziło o utworzenie nowego obiektu typu `Samochod`, to jest to błędny zapis – brakuje słowa kluczowego **`new`** przed nazwą klasy – linijka ta powinna wtedy wyglądać następująco:

```
Samochod samochod = new Samochod();
```

> Jednak linia ta mogłaby być w pewnym przypadku uznana za poprawną – jeżeli istniałaby *metoda* o nazwie `Samochod`, która zwracałaby obiekt typu `Samochod`:

```
public Samochod Samochod() {
  return new Samochod();
}
```

> Nie jest to jednak dobry pomysł – nazwy metod nie powinny nazywać się z wielkiej litery. Mogłyby wtedy zostać pomylone ze specjalnymi metodami nazywanymi *konstruktorami*.

6. Czy poniższa klasa jest poprawna?

*Nazwa pliku: PytaniaOKlasach.java*

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

> Nie, ponieważ nazwa tej klasy jest niezgodna z nazwą pliku, w którym jest zawarta. Albo nazwa klasy powinna zostać zmieniona na `PytaniaOKlasach`, albo nazwa pliku powinna zostać zmieniona na `Pytanie.java`.

7. Jaki będzie wynik uruchomienia poniższego programu?

```
public class PrzykladowaKlasa {
  private int x;
}
```

> Program zakończy się błędem, ponieważ Maszyna Wirtualna Java nie znajdzie w powyższej klasie metody `main`, której się spodziewa – jest to w końcu punkt wejścia do wykonywania naszego programu.

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);
  }
}
```

Na ekranie zobaczymy tekstowe reprezentacje dwóch obiektów typu `Punkt` – każdy z nich będzie miał inne wartości pól `x` oraz `y`, ponieważ każdy z tych obiektów zawiera własne egzemplarze pól `x` oraz `y`, które są niezależne od siebie:

```
X, Y: 10, 20
X, Y: 0, 5
```

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);
  }
}
```

Ten program w ogóle się nie skompiluje, ponieważ sygnatura metody `toString` jest nieprawidłowa – metoda `toString`, powinna zwracać wartość typu `String`, a zdefiniowana w powyższym programie metoda `toString` nic nie zwraca (`void`).

## 9.2 Zadania do rozdziału "Czym są klasy i do czego służą?"

### 9.2.1 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.*

Przykładowe rozwiązanie tego zadania, zgodne z powyższymi wymaganiami:

```
public class Osoba {
  private int wiek;
  private String imie;
  private String nazwisko;

  public void ustawWiek(int wartoscWieku) {
    wiek = wartoscWieku;
  }

  public void ustawImie(String imieOsoby) {
    imie = imieOsoby;
  }

  public void ustawNazwisko(String nazwiskoOsoby) {
    nazwisko = nazwiskoOsoby;
  }

  public String toString() {
    return "Osoba " + imie + " " + nazwisko +
        " ma " + wiek + " lat.";
  }

  public static void main(String[] args) {
    Osoba autor = new Osoba();
    autor.ustawImie("Stephen");
    autor.ustawNazwisko("King");
    autor.ustawWiek(71);

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

Program ten wypisuje na ekran:

```
Osoba Stephen King ma 71 lat.
```

## 9.3 Pytania do typów prymitywnych i referencyjnych

1. Jakie typy prymitywne są zdefiniowane w Javie?

> Java oferuje 8 typów prymitywnych: **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, oraz **`boolean`**.

2. Czym są typy referencyjne (złożone)? Jak się je definiuje?

> Typy referencyjne to typy, które definiowane są przez programistów. Każda klasa, którą napiszemy, to nowy typ złożony.

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;
  }
}
```

> W powyższej metodzie `main` tworzone są dwa obiekty – jeden typu `PytanieZagadka`, a drugi to tablica liczby całkowitych:

```
PytanieZagadka zagadka = new PytanieZagadka();
zagadka.liczby = new int[2];
```

> Liczba `5` to nie obiekt – jest to wartość typu prymitywnego. W ostatniej linii metody także nie tworzymy nowego obiektu, lecz definiujemy zmienną, która może wskazywać na obiekt typu `PytanieZagadka`. Zmienna `innaZagadka` pokazuje na ten sam obiekt w pamięci, co zmienna `zagadka`.

4. Który z poniższych zapisów jest niepoprawny i dlaczego?

```
// brakuje slowa kluczowego new
// linia moglaby byc poprawna, gdyby istniala
// metoda PytanieZagadka
PytanieZagadka zagadka = PytanieZagadka();

// blad - brakuje rozmiaru tablicy
int[] tablica1 = new int[];

// poprawna linia, chociaz nie powinnismy
// tworzyc obiektow typu String za pomoca
// slowa kluczowego new, lecz przypisywac
// wartosci bezposrednio: String tekst = "Witajcie!";
String tekst = new String("Witajcie!");
```

```
// blad - zla skladnia
int[] tablica2 = [];

// poprawna linia
String innyTekst = "Halo?!";

// poprawna linia
int[] tablica3 = new int[] {1, 2};

// poprawna linia
int[] tablica4 = {1, 2, 3};

// blad - jezeli podajemy wartosci,
// ktorymi tablica ma zostac zainicjalizowana,
// to nie powinnismy podawac rozmiaru tablicy
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]);
  }
}
```

> Na ekranie zobaczymy następujące wartości:

```
5
10
100
```

> Do zmiennej `innaZagadka` przypisujemy wskazanie na ten sam obiekt, na który wskazuje zmienna `zagadka`. Następnie przypisujemy do zmiennych `calkowita` i `tablica` wartości pól tego obiektu. Zmiana wartości zmiennej `calkowita` nie wpływa na wartość pola `liczba`, ponieważ są to zmienne typu prymitywnego, ale zmiana elementów tablicy `tablica` ma wpływ na tablicę `liczby`, ponieważ obie te zmienne wskazują na tą samą tablicę w pamięci.

## 9.4 Pytania do modyfikatorów dostępu

1. Do czego służą modyfikatory dostępu w Javie?

> Modyfikatory dostępu służą do ustawienia zakresu widoczności pól oraz metod klasy. Pozwalają one określić, jak będzie wyglądało użytkowanie klasy oraz kto, i w jakich okolicznościach, będzie mógł z pól i metod tej klasy korzystać.

2. Czy modyfikatory dostępu można używać tylko podczas definicji pól klasy?

> Nie, modyfikatory dostępu można używać także podczas definicji metod.

3. Czym różnią się modyfikatory `public` i `private`?

> Pola i metody publiczne są dostępne dla wszystkich innych klas, tzn. klasy używające danej klasy mogą się bezpośrednio odnosić do pól publicznych i wywoływać metody publiczne. Pola i metody prywatne są dostępne tylko z wnętrza klasy, w których są zawarte. Wszelkie inne klasy nie mają dostępu do pól i metod prywatnych.

4. Kto ma dostęp do prywatnych pól i metod klasy?

> Tylko klasa, w której te pola i metody zostały zdefiniowane.

5. Czy moglibyśmy wszystkie pola i metody zawsze definiować z dostępem `public`? Jeżeli tak, to czy jest to dobry pomysł?

> Moglibyśmy, ale nie jest to dobry pomysł – w ten sposób udostępnilibyśmy całą wewnętrzną implementację naszej klasy dla świata zewnętrznego – każdy, kto korzystałby z naszej klasy, mógłby dowolnie zmieniać pola obiektów tej klasy i wywoływać wszystkie jej metody. Wszelkie zmiany, jakie chcielibyśmy wprowadzić do naszej klasy najprawdopodobniej spowodowałyby, że inne, zależne od niej części naszych programów, przestałyby działać.

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);
  }
}
```

> Tak, powyższy kod jest poprawny, ponieważ odnosimy się do prywatnego pola `liczba` z wnętrza klasy, w której to pole jest zdefiniowane. Metoda `main` jest częścią klasy `KlasaZagadka`, więc odnoszenie się w niej do prywatnych pól i metod tej klasy jest dozwolone.

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?
  }
}
```

W metodzie `czyWTymSamymWieku` mamy dostęp do wszystkich pól i metod klasy `Osoba`. Możemy więc w tej metodzie odnieść się do pól `nazwisko` oraz `wiek`, a także wywołać metody `ustawWiek` i `wypiszNazwisko`.

Z drugiej strony, w klasie `UzycieTypuOsoba` mamy dostęp tylko do publicznych pól i metod obiektów klasy `Osoba`, więc w metodzie `main` klasy `UzycieTypuOsoba` możemy odnieść się jedynie do publicznego pola `nazwisko` obiektu `osoba`, oraz wywołać publiczne metody: `ustawWiek` oraz `czyWTymSamymWieku`.

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);
  }
}
```

> `KlasaZagadka` skompiluje się bez problemów, jednak próba kompilacji klasy `UzycieZagadki` zakończy się błędem, ponieważ próbujemy w tej klasy odnieść się do prywatnego pola `liczba` obiektu klasy `KlasaZagadka`.

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);
  }
}
```

> W każdym z zaznaczonych miejsc możemy umieścić modyfikator `public`. Najlepiej jednak byłoby ukryć te pola i metody, które stanowią wewnętrzną implementację klasy `InnaZagadka` – dlatego pole `liczba` oraz `tajnyAlgorytm` powinny być zdefiniowane z modyfikatorem `private`. **Metoda `toString` musi mieć modyfikator `public`** – taki jest wymóg odnośnie sygnatury metody `toString`. Metoda `ustawLiczbe` także powinna być publiczna, ponieważ korzystamy z niej w klasie `UzycieInnejZagadki`.

## 9.5 Pytania do pól klas

1. Jaka jest wartość domyślna dla typów referencyjnych?

> Wartość domyślna obiektów typów referencyjnych (o ile nie są one zmiennymi lokalnymi) to `null`. Wartość ta oznacza, że zmienna typu złożonego nie wskazuje na żaden obiekt.

2. Do czego są nam potrzebne gettery i settery?

> Gettery i settery to metody, które używane są do pobierania i ustawiania wartości pól klasy.

3. Jak powinny być pisane gettery?

> Gettery powinny:
>
> - zwracać wartość takiego samego typu, jakiego jest pole, z którego pobierają wartość,
> - zaczynać się od słowa *get* (chyba, że zwracają pole typu `boolean` – wtedy możemy zamiast *get* użyć słowa *is*, *has*, *should*, lub *can*) po którym powinna nastąpić nazwa pola, którego wartość zwracają, zapisana camelCasem,
> - nie mieć żadnych argumentów,
> - zwracać wartość danego pola.

4. Jak powinny być pisane settery?

> Settery powinny:
>
> - nie zwracać żadnej wartości,
> - zaczynać się od słowa *set*, po którym powinna nastąpić nazwa ustawianego pola, zapisana camelCasem,
> - przyjmować jeden argument takiego samego typu, jak pole, które ustawia,
> - mieć jeden argument o takie samej nazwie, jak nazwa pola, które ustawia,
> - przypisać do odpowiedniego pola przesłaną wartość.

5. Jakie powinny być nazwy getterów i setterów następujących pól?

   a) `String tytul;`

   b) `double mianownik;`

   c) `boolean uzytkownikZalogowany;`

> Settery i gettery tych pól powinny nazywać się: `getTytul` oraz `setTytul`, `getMianownik` oraz `setMianownik`, a także `isUzytkownikZalogowany` oraz `setUzytkownikZalogowany`. Getter pola `uzytkownikZalogowany` mógłby też nazywać się `getUzytkownikZalogowany`.

6. Czym jest i do czego służy `this`?

> `this` to referencja do obiektu, na rzecz którego wywoływane są metody klasy. Za pomocą `this` możemy odnieść się do pól obiektu, na rzecz którego wywołaliśmy np. setter, dzięki czemu możemy nazwa argument settera tak samo, jak nazwa pola, które ustawia.

7. Kiedy możemy zobaczyć błąd `Null Pointer Exception`?

> Błąd ten świadczy o próbie odniesienie się do pola bądź metody obiektu przez zmienną, która nie wskazywała na żaden obiekt, tzn. była nullem.

8. Jak uchronić się przed potencjalnym błędem `Null Pointer Exception`?

> Możemy skorzystać z instrukcji warunkowej `if`, w której warunku przyrównamy zmienną typu złożonego do wartości `null` – jeżeli warunek będzie prawdziwy, będzie to oznaczało, że zmienna "pokazuje na nic", więc nie powinniśmy za pomocą tej zmiennej próbować odnosić się do pól bądź metod.

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());
  }
}
```

> Na ekranie zobaczymy liczbę `0`, która jest domyślną wartością pól typu prymitywnego `int`. Dlaczego nie będzie to liczba `100`, którą ustawiamy za pomocą settera? W setterze, argument `liczba` zasłania pole o tej samej nazwie. Linia `liczba = liczba;` nie ma efektu – przypisujemy do argumentu `liczba` wartość argumentu `liczba`. Setter `setLiczba` powinien skorzystać z `this`, aby ustawić pole obiektu, na rzecz którego setter został wywołany:

```
this.liczba = liczba;
```

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;
}
```

> Powyższa klasa skompiluje się bez problemów.

b)

```
public class PytaniePola {
  private int x = y;
  private int y = 0;
}
```

> Kompilacja powyższej klasy zakończy się błędem – próbujemy skorzystać z wartości pola `y` zanim to pole zostanie zdefiniowane.

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());
  }
}
```

> Każdy z obiektów `o1` i `o2` ma własne pola `nazwa`. Na ekranie zobaczymy:

```
Pewna nazwa
Inna nazwa
```

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);
  }
}
```

Na ekranie zobaczymy domyślne wartości typów **`int`**, **`boolean`**, oraz `String`:

```
0 false null
```

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");
}
```

W wyniku działania tego fragmentu kodu, nic nie zostanie wypisane na ekran, ponieważ oba warunki instrukcji warunkowych nie będą spełnione – pierwszy, `tekst == null`, nie jest prawdziwy, ponieważ zmienna `tekst` ma przypisaną wartość.

Drugi warunek, `tekst != null`, nie będzie prawdziwy, ponieważ w linii wcześniej przypisujemy do zmiennej `tekst` wartość **`null`**.

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());
  }
}
```

W wyniku działania tego programu zobaczymy na ekranie domyślną wartość pola `liczba`, którą jest `0`, oraz błąd `Null Pointer Exception`, ponieważ próbujemy skorzystać z metody `toUpperCase` na obiekcie, który jest nullem – nie ustawiamy żadnej wartości w polu `nazwa`, a jako, że jest to pole typu złożonego, jego domyślną wartością jest **`null`**:

```
0
Exception in thread "main" java.lang.NullPointerException
    at UzycieWartosci.main(UzycieWartosci.java:17)
```

## 9.6 Zadania do pól klas

### 9.6.1 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`*.*

Przykładowe rozwiązanie tego zadania wraz metodą `main` korzystającą z obiektu klasy `Punkt`:

```
public class Punkt {
  private int x;
  private int 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 + ")";
  }

  public static void main(String[] args) {
    Punkt p = new Punkt();
    p.setX(10);
    p.setY(-5);

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

Wynik działania tego programu:

```
Punkt(x: 10, y: -5)
```

## 9.7 Pytania do konstruktorów

1. Do czego służą konstruktory?

> Konstruktory to specjalne metody służące do inicjalizacji tworzonych obiektów.

2. Czy każda klasa posiada konstruktor?

> Tak, każda klasa posiada konstruktor, nawet w przypadku, gdy programista sam nie zdefiniuje w klasie konstruktora. W takim przypadku, kompilator języka Java zdefiniuje w klasie *konstruktor domyślny*.

3. Jak zdefiniować konstruktor?

> Konstruktor to metoda, która powinna nazywać się tak samo, jak klasa, w której jest zawarty. Dodatkowo, konstruktory nie powinny mieć zwracanego typu – pomiędzy modyfikatorami a nazwą konstruktora nie definiujemy żadnego zwracanego typu – nie używamy nawet `void`.

4. Czym jest konstruktor domyślny i kiedy jest definiowany?

> Konstruktor domyślny to konstruktor, który nie ma argumentów oraz nie wykonuje żadnych instrukcji. Jest automatycznie definiowany przez kompilator języka Java w klasach, w których programiści nie zdefiniują żadnego konstruktora.

5. Czy klasa może mieć wiele konstruktorów?

> Tak, każda klasa może mieć wiele konstruktorów.

6. Jak z jednego konstruktora wywołać inny konstruktor?

> Należy skorzystać ze słowa kluczowego `this`, po którym powinny nastąpić nawiasy ( ) oraz średnik. W nawiasach należy umieścić argumenty konstruktora, który chcemy wywołać.

7. Jaki warunek musi spełniać wywołanie jednego konstruktora z drugiego konstruktora?

> Wywołanie innego konstruktora musi być pierwszą instrukcją w konstruktorze, z którego go wywołujemy. Dodatkowo, możemy wywołać tylko jeden inny konstruktor.

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);
  }
}
```

> Powyższy program się nie skompiluje, ponieważ w metodzie `main` próbujemy skorzystać z konstruktora bezargumentowego, a klasa `PytanieKonstruktor` takiego konstruktora nie posiada.

9. Ile konstruktorów posiada poniższa klasa?

```
public class PytanieKonstruktor {
  private int pewnePole;
}
```

> Ta klasa posiada jeden konstruktor – domyślny konstruktor wygenerowany automatycznie przez kompilator języka Java.

10. Czy poniższa klasa ma domyślny konstruktor?

```
public class PytanieKonstruktor {
  private int pewnePole;

  public PytanieKonstruktor() {

  }
}
```

> Nie, ta klasa nie posiada konstruktora domyślnego, ponieważ w tej klasie zdefiniowany został już inny konstruktor – tak się składa, że jest on zgodny z konstruktorem domyślnym, który zostałby wygenerowany przez kompilator, gdyby w tej klasie nie było zdefiniowanego konstruktora.

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");
  }
}
```

Ta klasa się nie skompiluje. Istnieje sposób (za pomocą użycia pierwszego konstruktora) na utworzenie obiektu tej klasy, w którym jedno z pól `final` nie zostanie zainicjalizowane. Kompilator jest w stanie wychwycić taki przypadek i nie pozwoli na kompilację klasy.

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);
  }
}
```

Ta klasa się nie skompiluje, ponieważ wywołanie z konstruktora innego konstruktora powinno być pierwszą instrukcją. W bezargumentowym konstruktorze w powyższej klasie korzystamy z instrukcji `System.out.println` przed wywołaniem innego konstruktora, co powoduje błąd kompilacji.

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");
  }
}
```

Ta klasa się nie skompiluje, ponieważ w pierwszym konstruktorze, po wywołaniu drugiego konstruktora, próbujemy przypisać do pola `liczba` wartość argumentu. Pole `liczba` jest `final` i jest ustawiane jednorazowo w drugim konstruktorze. Próba ponownego ustawienia tego pola powoduje błąd kompilacji.

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);
  }
}
```

Na ekranie zobaczymy komunikat `x = 0`. W powyższej klasie kompilator języka Java automatycznie wygeneruje konstruktor domyślny, ponieważ klasa ta nie posiada konstruktora. Metoda `PytanieKonstruktor`, która jest w tej klasie zdefiniowana, to nie konstruktor, ponieważ zawiera zwracany typ – `void`, co sprawia, że jest to zwykła metoda, a nie konstruktor.

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);
  }
}
```

Ta klasa się nie skompiluje. Metoda `Pytaniekonstruktor` miała najprawdopodobniej być konstruktorem klasy `PytanieKonstruktor`, jednak nie spełnia wymagania odnośnie konstruktorów: nie nazywa się tak samo, jak nazwa klasy, w którym jest zdefiniowana. W nazwie popełniony został błąd – mała litera k powinna być zastąpiona wielką literą K.

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);
  }
}
```

Na ekranie zobaczymy:

```
x = 10
x = 0
```

Drugi konstruktor, przyjmujący argument typu `int`, nie ustawia pola klasy o nazwie `x`, lecz przypisuje do argumentu jego własną wartość – zabrakło użycia słowa kluczowego `this`:

```
this.x = x;
```

## 9.8 Zadania do konstruktorów

### 9.8.1 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`.*

Przykładowe rozwiązanie tego zadania mogłoby być następujące:

```
public class Adres {
  private String miejscowosc;
  private String kodPocztowy;
  private String nazwaUlicy;
  private int nrDomu;

  public Adres(String miejscowosc, String kodPocztowy,
               String nazwaUlicy, int nrDomu) {
    this.miejscowosc = miejscowosc;
    this.kodPocztowy = kodPocztowy;
    this.nazwaUlicy = nazwaUlicy;
    this.nrDomu = nrDomu;
  }

  public String toString() {
    return nazwaUlicy + " " + nrDomu + ", "  +
        kodPocztowy + " " + miejscowosc;
  }
}
```

Konstruktor klasy `Adres` przyjmuje cztery argumenty, którymi inicjalizuje pola tworzonego obiektu.

### 9.8.2 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.*

Przykładowe rozwiązanie tego zadania:

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private final int rokUrodzenia;
  private Adres adres;

  public Osoba(String imie, String nazwisko,
               int rokUrodzenia, Adres adres) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.rokUrodzenia = rokUrodzenia;
    this.adres = adres;
  }

  public Osoba(String imie, String nazwisko, int rokUrodzenia,
               String miejscowosc, String kodPocztowy,
               String nazwaUlicy, int nrDomu) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.rokUrodzenia = rokUrodzenia;

    this.adres = new Adres(
        miejscowosc, kodPocztowy, nazwaUlicy, nrDomu
    );
  }

  public String toString() {
    return "Osoba " + imie + " " + nazwisko +
        " urudzona w roku " + rokUrodzenia +
        " mieszka pod adresem " + adres;
  }
```

```
  public static void main(String[] args) {
    Adres adres = new Adres("Warszawa", "01-123", "Krucza", 8);
    Osoba osoba1 = new Osoba("Jan", "Kowalski", 1982, adres);

    Osoba osoba2 = new Osoba(
        "Jan", "Nowak", 1980,
        "Warszawa", "01-231", "Grzybowska", 20
        );

    System.out.println(osoba1);
    System.out.println(osoba2);
  }
}
```

Pierwszy konstruktor przyjmuje cztery argumenty, których wartościami zostaną zainicjalizowane pola tworzonego obiektu.

Drugi konstruktor przyjmuje siedem argumentów – cztery ostatnie są używane do utworzenia obiektu typu `Adres`.

W metodzie `main` tworzymy dwa obiekty klasy `Osoba`, korzystając z dwóch różnych konstruktorów – pierwszy oczekuje obiektu typu `Adres` jako argumentu, więc tworzymy najpierw obiekt typu `Adres` i przekazujemy go jako argument do konstruktora.

Wynik działania tego programu jest następujący:

```
Osoba Jan Kowalski urudzona w roku 1982 mieszka pod adresem Krucza 8, 01-
123 Warszawa
Osoba Jan Nowak urudzona w roku 1980 mieszka pod adresem Grzybowska 20, 01-
231 Warszawa
```

*Obie klasy, `Adres` oraz `Osoba`, powinny znajdować się w tym samym katalogu – inaczej próba skompilowania powyższej klasy `Osoba` zakończy się błędem.*

## 9.9 Pytania do porównywania obiektów

1. Czym różni się porównywanie obiektów za pomocą operatora porównania == i metody `equals`?

> Operator == użyty do porównywania zmiennych typu złożonego odpowiada na pytanie: *Czy obie zmienne pokazują na ten sam obiekt w pamięci?* Metoda `equals` może natomiast zostać napisana w taki sposób, aby sprawdzała, czy pola porównywanych obiektów są takie same, dzięki czemu może ona sprawdzić równość dwóch obiektów nie na podstawie tego, czy są one tym samym obiektem w pamięci, lecz czy mają taki sam stan (takie same wartości wszystkich bądź części pól).

2. Czy do przyrównywania zmiennych do wartości **`null`** możemy korzystać z operatorów == i !=, czy powinniśmy korzystać z metody `equals`?

> Jeżeli chcemy przyrównać zmienną typu złożonego do wartości **`null`**, to możemy (i powinniśmy, ponieważ taki zapis jest krótszy) korzystać z operatorów == oraz !=.

3. Jak powinna wyglądać sygnatura metody `equals`?

> Metoda ta powinna być publiczna, zwracać wartość typu **`boolean`**, oraz przyjmować jeden argument typu `Object`.

4. Czy dla dwóch zmiennych `x` i `y`, wskazujących na ten sam obiekt w pamięci, metoda `x.equals(y)` może zwrócić **`false`**?

> Tak – wszystko zależy od tego, jak zapisana zostanie metoda `equals`. Moglibyśmy napisać metodę `equals`, która zawsze zwracałaby wartość **`false`** – nie miałaby ona jednak raczej sensu, a na pewno nie przestrzegałaby kontraktu `equals`, który specyfikuje, że ten sam obiekt zawsze powinien być równy sobie.

5. Na podstawie jakich warunków metoda `equals` powinna zwrócić **`true`** lub **`false`**?

> Zależy to od programisty – metoda `equals` może brać pod uwagę część lub wszystkie pola porównywanych obiektów i na podstawie ich wartości odpowiedzieć na pytanie, czy dwa obiekty są sobie równe, czy nie.

6. Co to jest kontrakt `equals`?

> Jest to zbiór reguł określony przez twórców języka Java, które powinna spełniać każda metoda `equals`. Kontrakt `equals` definiuje m. in., że każdy obiekt powinien być równy samemu sobie. Spis reguł można znaleźć w oficjalnej dokumentacji języka Java.

7. Jak rzutuje się wartość danego typu na wartość innego typu?

Aby zrzutować wartość jednego typu na wartość innego typu, zapisujemy docelowy typ w nawiasach przed wartością, którą chcemy zrzutować, np. `(int) 3.14` powoduje zrzutowanie liczby rzeczywistej na liczbę całkowitą. Wynikiem tej operacji będzie liczba całkowita `3` bez części ułamkowej.

8. Do czego służy metoda `getClass`?

Metoda ta zwraca informację, jakiego typu jest obiekt, na rzecz którego wywołaliśmy metodę `getClass`. Możemy skorzystać z tej metody np. w metodzie `equals` aby sprawdzić, czy przesłany jako argument obiekt do porównania jest tego samego typu, co obiekt, na rzecz którego metoda `equals` została wywołana.

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?

Po sprawdzeniu, czy przesłany jako argument obiekt nie jest nullem, oraz czy jego typ to `PewnaKlasa`, powinniśmy sprawdzić wartości pól `liczba`, `tekst`, oraz `znaki`. Pole `liczba` porównamy za pomocą operatora ==, ponieważ jest to pole typu prymitywnego. Następnie, sprawdzimy, czy pole `tekst` nie jest nullem – jeżeli nie, to porównamy je za pomocą metody `equals` typu `String`. Na końcu sprawdzimy, czy tablica `znaki` w obu obiektach jest taka sama. Tablica ta jest typu prymitywnego `char`, więc elementy obu tablic porównamy za pomocą operatora ==.

10. Do czego służy metoda `Arrays.equals`, przyjmująca jako argumenty dwie tablice?

Metoda ta zwraca `true`, jeżeli obie tablice przesłane jako argumenty są takie same, a `false` w przeciwnym razie. Dwie tablice są uznawane za takie same, jeżeli obie mają ten sam rozmiar i takie same elementy na odpowiadających sobie indeksach, lub jeżeli obie tablice są nullem.

11. Czy metoda `equals`, gdy jej argumentem jest `null`, na przykład `x.equals(null)`, może zwrócić `true`?

Może – implementacja metody `equals` to zadanie programisty. Można by ją napisać w taki sposób, by zwracała `true` dla argumentu, który jest nullem. Nie jest to jednak dobry pomysł – takie działanie byłoby niezgodne z kontraktem `equals`.

```java
// 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?

```java
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));
}
```

> Kod się skompiluje – co prawda nie napisaliśmy metody `equals`, ale dziedziczymy ją z klasy-rodzica, która jest klasą nadrzędną wszystkich klas w języku Java – klasy `Object`. Dziedziczona metoda `equals` sprawdza, czy dwie zmienne wskazują na ten sam obiekt w pamięci, więc na ekranie zobaczymy:

```
a1 rowne a2? true
a1 rowne a3? false
```

13. Czy poniższa metoda `equals` byłaby poprawna dla klasy `A`?

```java
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }

  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }

  return this.liczba == o.liczba;
}
```

> Nie – taka metoda `equals` spowodowałaby błąd kompilacji klasy. Próbujemy odnieść się do pola `liczba` obiektu `o`, jednak zmienna ta jest typu `Object`. Powinniśmy najpierw zrzutować argument `o` na obiekt typu `A`:

```java
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }

  if (o == null || this.getClass() != o.getClass()) {
    return false;
  }
```

```
  A other = (A) o;

  return this.liczba == other.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));
}
```

Na ekranie zobaczymy komunikaty `a1 rowne a2? true` oraz `a1 rowne a3? true`, a także błąd `Null Pointer Exception`. W ostatniej linii metody `main` próbujemy porównać obiekt `a1` do wartości `null` – zauważmy, że w powyższej metodzie `equals` najpierw sprawdzamy, czy klasa obiektu wskazywanego przez `this` i obiektu `o` przesłanego jako argument się zgadza, a dopiero potem sprawdzamy, czy przypadkiem argument `o` nie jest nullem – powinniśmy sprawdzać te warunki w odwrotnej kolejności:

```
if (o == null || this.getClass() != o.getClass()) {
  return false;
}
```

## *9.10 Zadania do porównywania obiektów*

### 9.10.1 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.*

Przykładowe rozwiązanie tego zadania:

```
public class Punkt {
  private int x;
  private int y;

  public Punkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }

    if (o == null || this.getClass() != o.getClass()) {
      return false;
    }

    Punkt other = (Punkt) o;

    return this.x == other.x && this.y == other.y;
  }

  public static void main(String[] args) {
    Punkt p1 = new Punkt(10, 20);
    Punkt p2 = new Punkt(10, 20);
    Punkt p3 = new Punkt(10, -5);
    Punkt p4 = new Punkt(0, 20);

    System.out.println(p1.equals(p2));
    System.out.println(p1.equals(p3));
    System.out.println(p1.equals(p4));
    System.out.println(p1.equals(null));
    System.out.println(p1.equals(p1));
  }
}
```

Metoda `equals` sprawdza, czy przesłany obiekt nie jest tym samym obiektem, co obiekt, na rzecz którego wywołaliśmy metodę `equals` (czyli `this`). Następnie, sprawdzamy, czy obiekt ten nie jest nullem i czy jest aby na pewno typu `Punkt`. Na końcu rzutujemy argument `o` na typ `Punkt` i porównujemy pola `x` i `y`. W metodzie `main` używamy kilka razy metody `equals`, by sprawdzić, czy działa poprawnie – na ekranie zobaczymy:

```
true
false
false
false
true
```

### 9.10.2 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.*

Klasę `Figura` umieszczamy w tym samym katalogu, co klasę `Punkt` z poprzedniego zadania, abyśmy mogli z niej skorzystać. Przykładowe rozwiązanie mogłoby wyglądać następująco:

```
import java.util.Arrays;

public class Figura {
  private Punkt[] wierzcholki;

  public Figura(Punkt[] wierzcholki) {
    this.wierzcholki = wierzcholki;
  }

  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }

    if (o == null || this.getClass() != o.getClass()) {
      return false;
    }

    Figura other = (Figura) o;

    return Arrays.equals(this.wierzcholki, other.wierzcholki);
  }

  public static void main(String[] args) {
    Figura kwadrat = new Figura(new Punkt[] {
        new Punkt(0, 0),
        new Punkt(10, 0),
        new Punkt(10, 10),
        new Punkt(0, 10)
    });

    Figura podobnyKwadrat = new Figura(new Punkt[] {
        new Punkt(0, 0),
        new Punkt(10, 0),
        new Punkt(10, 10),
        new Punkt(0, 10)
    });

    Figura innyKwadrat = new Figura(new Punkt[] {
        new Punkt(2, 2),
        new Punkt(4, 2),
        new Punkt(4, 4),
        new Punkt(2, 4)
    });

    Figura trojkat = new Figura(new Punkt[] {
        new Punkt(10, 10),
        new Punkt(20, 20),
        new Punkt(10, 30)
    });
```

```
        System.out.println(kwadrat.equals(podobnyKwadrat));
        System.out.println(kwadrat.equals(innyKwadrat));
        System.out.println(kwadrat.equals(trojkat));
        System.out.println(kwadrat.equals("Witaj!"));
        System.out.println(kwadrat.equals(null));
        System.out.println(kwadrat.equals(kwadrat));
    }
}
```

W metodzie `equals` wykonujemy podstawowe sprawdzenia, a na końcu korzystamy z metody `equals` klasy `Arrays`, którą zaimportowaliśmy do naszego programu na jego początku, aby porównać tablice `wierzcholki` obu obiektów.

W wyniku działania tego programu, na ekranie zobaczymy:

```
true
false
false
false
false
true
```

## 9.11 Pytania do referencji do obiektów

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());
  }
}
```

> Na ekranie zobaczymy liczbę `100`, ponieważ zmienne `z1` i `z2` wskazują na ten sam obiekt w pamięci.

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;
  }
}
```

> Na ekranie zobaczymy liczbę `5.0` – zmiany wartości zmiennych prymitywnych nie są wykonywane na oryginalnych zmiennych.

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);
  }
}
```

Na ekranie zobaczymy tekst `Witaj!`. Przypisanie do argumentu `obiekt` w metodzie `zmienObiekt` nie zmieni, na co pokazuje zmienna `o` w metodzie `main` – zmieni się jedynie argument metody `zmienObiekt` o nazwie `obiekt` – będzie on pokazywał na nowo utworzony obiekt.

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());
  }
}
```

Na ekranie zobaczymy dwa razy liczbę `10`. Oba obiekty odwołują się do tej samej tablicy w pamięci – tej, którą tworzymy na początku metody `main`. Jeżeli zmienimy element tej tablicy, to pośrednio wpłyniemy także na stan obiektów `o1` i `o2`.

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]);
  }
}
```

Powyższa klasa się nie skompiluje – nie możemy przypisać do zmiennej `liczby` nowej tablicy, ponieważ ta zmienna jest **`final`**.

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]);
  }
}
```

> Na ekranie zobaczymy liczbę `5` – co prawda zmienna `liczby` jest **`final`**, ale tablica, na którą pokazuje, nie jest – możemy modyfikować jej elementy.

7. Co charakteryzuje obiekty niemutowalne?

> Zmiana stanu (wartości pól) obiektów niemutowalnych jest niemożliwa – obiekty niemutowalne, raz utworzone i zainicjalizowane pewnymi wartościami, posiadają te wartości do końca ich "życia".

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;
  }
}
```

> Obiekty tej klasy są niemutowalne – co prawda pole `x` jest polem publicznym, ale wartości pola typu prymitywnego z modyfikatorem **`final`** nie da się zmodyfikować.

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;
  }
}
```

> Obiekty tej klasy nie są niemutowalne, ponieważ istnieje możliwość zmiany wartości pola `komunikat` – za pomocą settera `setKomunikat`.

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;
  }
}
```

Obiekty tej klasy nie są niemutowalne, ponieważ do prywatnego pola `slowa` przypisujemy tablicę przesłaną jako argument – nic nie stoi na przeszkodzie, by klasa, która będzie tworzyć obiekty typu `ZagadkaMutowalne3`, zmieniła elementy tablicy przesłanej jako argument do konstruktora, już po utworzeniu obiektu klasy `ZagadkaMutowalne3`.

11. Co to jest sterta i stos?

Sterta to obszar pamięci naszego programu, w którym przechowywane są tworzone przez nas obiekty typów złożonych. Stos to obszar pamięci naszego programu, w którym m. in. przechowywane są zmienne typów prymitywnych.

12. Czym różnią się typy prymitywne od typów referencyjnych (złożonych)?

Poniższa tabela podsumowuje różnice pomiędzy typami złożonymi a typami prymitywnymi:

| | **Typy Prymitywne** | **Typy Referencyjne** |
|---|---|---|
| **Wartości** | konkretne, np. `5`, `true`, `'a'` | adresy obiektów |
| **Tworzenie** | zdefiniowanie zmiennej | operator `new` |
| **Domyślne wartości** | zmienne lokalne – brak,<br>pola klas – zależne od typu | zmienne lokalne – brak,<br>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 |

## *9.12 Zadania do referencji do obiektów*

### 9.12.1 Niemutowalna Ksiazka i Biblioteka

*Napisz niemutowalną klasę `Ksiazka` z polami `tytul`, `autor`, oraz `cena`, oraz metodą `toString`. Następnie, napisz niemutowalną klasę `Biblioteka`, która będzie zawierała tablicę obiektów typu `Ksiazka`. W klasie `Biblioteka` zawrzyj metodę `getKsiazki`, która będzie zwracała tablicę z obiektami typu `Ksiazka`, które przechowuje obiekt typu `Biblioteka`.*

Zacznijmy od klasy `Ksiazka` – pola w tej klasie będą finalne. Nie udostępnimy żadnego sposobu na modyfikację pól obiektów tej klasy, dzięki czemu obiekty tej klasy będą niemutowalne:

```
public class Ksiazka {
  private final String tytul;
  private final String autor;
  private final double cena;

  public Ksiazka(String tytul, String autor, double cena) {
    this.tytul = tytul;
    this.autor = autor;
    this.cena = cena;
  }

  public String toString() {
    return "Ksiazka " + tytul + ", autorstwa " + autor +
        " kosztuje " + cena;
  }
}
```

Klasa `Biblioteka` ma zawierać tablicę obiektów typu `Ksiazka`. Jako, że klasa `Biblioteka` ma być także niemutowalna, musimy ustrzec się przed potencjalnymi modyfikacjami stanu obiektów tej klasy – dlatego nie przypiszemy tablicy-argumentu przesłanego do konstruktora bezpośrednio do pola `ksiazki`, lecz stworzymy kopię tablicy. Tak samo zachowamy się w metodzie `getKsiazki` – nie zwrócimy bezpośrednio pola `ksiazki`, lecz zwrócimy jego kopię – dzięki temu, ustrzeżemy się przed próbami modyfikacji tablicy zawartej w obiekcie klasy `Biblioteka`:

```
public class Biblioteka {
  private final Ksiazka[] ksiazki;

  public Biblioteka(Ksiazka[] ksiazki) {
    this.ksiazki = new Ksiazka[ksiazki.length];

    for (int i = 0; i < ksiazki.length; i++) {
      this.ksiazki[i] = ksiazki[i];
    }
  }

  public Ksiazka[] getKsiazki() {
    Ksiazka[] rezultat = new Ksiazka[ksiazki.length];

    for (int i = 0; i < ksiazki.length; i++) {
      rezultat[i] = ksiazki[i];
    }

    return rezultat;
  }
```

```
  public static void main(String[] args) {
    Ksiazka[] mrocznaWieza = {
        new Ksiazka("Roland", "Stephen King", 39.99),
        new Ksiazka("Powolanie trojki", "Stephen King", 39.99),
        new Ksiazka("Ziemie jalowe", "Stephen King", 39.99),
        new Ksiazka("Czarnoksieznik i krysztal", "Stephen King", 45.99),
        new Ksiazka("Wilki z Calla", "Stephen King", 39.99),
        new Ksiazka("Piesn Susannah", "Stephen King", 29.99),
        new Ksiazka("Mroczna Wieza", "Stephen King", 49.99),
    };

    Biblioteka biblioteka = new Biblioteka(mrocznaWieza);

    System.out.println("Ksiazki w bibliotece:");

    for (Ksiazka ksiazka : biblioteka.getKsiazki()) {
      System.out.println(ksiazka);
    }
  }
}
```

Zwróćmy uwagę, że w konstruktorze klasy `Biblioteka`, oraz w getterze `getKsiazki`, tworzymy kopię tablicy, odpowiednio, przesłanej jako argument, oraz tej przechowywanej w polu `ksiazki`. Na ekranie zobaczymy:

```
Ksiazki w bibliotece:
Ksiazka Roland, autorstwa Stephen King kosztuje 39.99
Ksiazka Powolanie trojki, autorstwa Stephen King kosztuje 39.99
Ksiazka Ziemie jalowe, autorstwa Stephen King kosztuje 39.99
Ksiazka Czarnoksieznik i krysztal, autorstwa Stephen King kosztuje 45.99
Ksiazka Wilki z Calla, autorstwa Stephen King kosztuje 39.99
Ksiazka Piesn Susannah, autorstwa Stephen King kosztuje 29.99
Ksiazka Mroczna Wieza, autorstwa Stephen King kosztuje 49.99
```

## *9.13 Zadania do pól i metod statycznych*

### 9.13.1 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`.*

Kod liczący silnię pisaliśmy przy okazji zadania 5.2.2. Policz silnię, a metodę `sumaLiczba` – w zadaniu 7.7.3. Metoda sumująca liczby w tablicy. Klasa `Obliczenia`, bazująca na tamtych rozwiązaniach, mogłaby wyglądać następująco:

```
public class Obliczenia {
  public static int silnia(int liczba) {
    int silnia = 1;

    for (int i = 1; i <= liczba; i++) {
      silnia = silnia * i;
    }

    return silnia;
  }

  public static int sumaLiczb(int[] tab) {
    int suma = 0;

    for (int i : tab) {
      suma += i;
    }

    return suma;
  }
}
```

Klasa `WykonywanieObliczen`, korzystająca z klasy `Obliczenia`, mogłaby wygladać następująco:

```
public class WykonywanieObliczen {
  public static void main(String[] args) {
    System.out.println("Silnia 6 to " + Obliczenia.silnia(6));

    int[] liczby = { 10, -50, 343, 42, 5 };
    System.out.println(
        "Suma liczb wynosi " + Obliczenia.sumaLiczb(liczby)
    );
  }
}
```

W wyniku uruchomienia tego programu, na ekranie zobaczymy:

```
Silnia 6 to 720
Suma liczb wynosi 350
```

## 9.14 Pytania do pakietów i importowania klas

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.");
  }
}
```

> Pełna nazwa tej klasy to nazwa klasy wraz z pakietem, w którym jest zwarta, więc pełna nazwa tej klasy to `com.kursjava.TestowaKlasa`.

2. Jaką komendą należy skompilować, a jaką uruchomić, klasę z powyższego zadania?

> Do kompilacji i uruchomienia tej klasy należy, odpowiednio, użyć komend:

```
javac com/kursjava/TestowaKlasa.java
java com.kursjava.TestowaKlasa
```

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
```

> Nie, Maszyna Wirtualna Java zgłosi błąd – nie znajdzie ona klasy, którą chcemy uruchomić. Powinniśmy wywołać powyższą komendę z katalogu `C:\programy`:

```
C:\programy> java com.kursjava.TestowaKlasa
```

4. Czy poniższa próba kompilacji klasy `TestowaKlasa` powiedzie się?

```
javac com.kursjava.TestowaKlasa.java
```

> Nie, ponieważ podając plik z klasą do kompilacji powinniśmy podać jego lokalizację za pomocą znaków slash / zamiast kropek:

```
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();
  }
}
```

> Nie, ponieważ kompilator nie znajdzie klasy `TestowaKlasa` – klasa ta znajduje się w pakiecie `com.kursjava`, a powyżej próbujemy zaimportować wszystkie klasy z pakietu `com` – **użycie gwiazdki nie powoduje importu z podpakietów**!

6. Czy poniższa klasa skompiluje się i wykona się bez błędów?

```
public class WykorzystanieTestowejKlasy {
  public static void main(String[] args) {
    com.kursjava.TestowaKlasa.wypiszKomunikat();
  }
}
```

> Tak, klasa skompiluje się i wykona bez błędów – odnosimy się do klasy `TestowaKlasa` za pomocą jej pełnej nazwy. Kompilator będzie wiedział, gdzie szukać tej klasy – w katalogu `kursjava`, który z kolei będzie znajdował się w katalogu `com`.

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();
  }
}
```

> Kompilacja tej klasy zakończy się błędem, ponieważ słowo kluczowe **`package`** powinno być pierwszą instrukcją w pliku z kodem źródłowym Java.

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);
  }
}
```

> Tak, ponieważ klasa `Math` znajduje się w pakiecie `java.lang`, który jest dla naszej wygody automatycznie importowany do naszych klas, dzięki czemu nie musimy pisać importu klas takich jak `Math` oraz `String`.

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);
  }
}
```

> Tak, powyższy kod jest poprawny i wypisze na ekran liczbę `3`, chociaż program ten nie powinien importować statycznie stałej z pakietu `Math`, skoro klasa `TestowaKlasa` posiada własne pole o tej samej nazwie, które zasłania pole z klasy `Math`.

10. Czym jest classpath? Jak ustawić wartość classpath?

> *classpath* to lista lokalizacji, w których kompilator javac i Maszyna Wirtualnej Java szukają klas do skompilowania i uruchomienia. Aby ustawić classpath, możemy przekazać wartość dla argumentu `-classpath` podczas uruchamiania kompilatora javac i Maszyny Wirtualnej Java.

11. Dlaczego w naszych programach nie musimy importować typu `String` z Biblioteki Standardowej Java?

> Klasa `String` znajduje się w pakiecie `java.lang` w Bibliotece Standardowej Java. Pakiet ten jest automatycznie importowany do naszych programów dla naszej wygody.

12. Czym różni się *dostęp domyślny* od dostępów definiowanych za pomoc modyfikatorów **`private`** oraz **`public`**?

> *Dostęp domyślny* nie ma własnego słowa kluczowego. Jest on prawie tak restrykcyjny jak dostęp definiowany przez modyfikator **`private`**, ale zezwala na korzystanie z pól i metod tym klasom, które zdefiniowane są w tym samym pakiecie.

13. Jak zdefiniować, że pole bądź metoda klasy ma mieć *dostęp domyślny*?

> Jako, że *dostęp domyślny* nie ma własnego słowa kluczowego, aby pole bądź metoda miało *dostęp domyślny*, należy pominąć przy jego definicji modyfikatory **`public`**, **`private`**, oraz **`protected`**.

14. Czy klasy mogą być niepubliczne? Jeżeli tak, to czym takie klasy różnią się od klas publicznych?

> Tak, klasy mogą być niepubliczne, jeżeli podczas ich definicji nie użyjemy słowa kluczowego **`public`** – klasa będzie wtedy miała *dostęp domyślny*, czyli będzie dostępna do użytku tylko dla klas zdefiniowanych w tym samym pakiecie.

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");
  }
}
```

Klasa `C`, która zdefiniowana jest w tym samym pakiecie, co klasy `A` oraz `B`, ma dostęp do wszystkich publicznych pól i metod oraz tych zdefiniowanych z *dostępem domyślnym*, czyli:

- pola `y` klasy `A`,
- pola `z` klasy `A`,
- metody `pewnaMetoda` klasy `A`,
- pola `n` klasy `B`,
- pola `m` klasy `B`,
- metody `innaMetoda` klasy `B`.

Z kolei klasa `D`, która jest w innym pakiecie, niż klasy `A` oraz `B`, ma dostęp jedynie do publicznego pola `y` klasy `A`. **Do klasy `B` klasa `D` w ogóle nie ma dostępu, ponieważ klasa `B` jest zdefiniowana z *dostępem domyślnym*, tzn. klasa `B` dostępna jest tylko dla klas z tego samego pakietu – klasa `B` jest klasą *niepubliczną*.**

# 10 Rozdział XI – Wyjątki

## *10.1 Pytania*

1. Do czego służą wyjątki?

> Wyjątki służą do sygnalizowania, że coś w programie poszło nie tak. Zawierają także informację o ścieżce wykonania programu, która doprowadziła do zaistnienia problemu.

2. Co to jest *stack trace*?

> Stack trace to lista wywołań kolejnych metod, które doprowadziły do sytuacji, w której doszło do rzucenia wyjątku w programie. Stack trace zawiera nie tylko nazwy metod, posortowane (patrząc od dołu) od pierwszej wywołanej do ostatniej, ale także numer linii kodu, które określają miejsce w metodzie, w którym nastąpiło przejście do następnej metody. Dzięki temu możemy prześledzić ścieżkę wykonania programu, która zakończyła się rzuceniem wyjątku.

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)
```

> Wyjątek rzucany jest w ostatniej wykonanej metodzie, która w stack trace znajduje się na szczycie listy metod. W tym przypadku jest to metoda o nazwie `innaMetoda` i to w niej rzucony został wyjątek.

4. Jak w języku Java obsługuje się wyjątki?

> Wyjątki obsługuje się umieszczając instrukcje, których wykonanie może zakończyć się rzuceniem wyjątku, w bloku **`try`**. Obsługa konkretnego typu wyjątku powinna zostać umieszczona w sekcji **`catch`**.

5. Czy musimy stosować obsługę wyjątków jeżeli metoda, z której chcemy skorzystać, może rzucić wyjątek?

> Nie musimy – jeżeli rzucany jest wyjątek rodzaju Unchecked, to w ogóle nie musimy takiego wyjątku obsługiwać w naszym kodzie, jeżeli nie mamy takiej potrzeby.
>
> Jeżeli jednak rzucany jest wyjątek rodzaju Checked, a nie chcemy go obsłużyć, to musimy dodać do sygnatury metody, która korzysta z metody rzucającej wyjątek, klauzulę **`throws`** i wyszczególnić ten typ rzucanego wyjątku. W ten sposób oddelegowujemy złapanie wyjątku do metody, która będzie korzystała z naszej metody – prędzej czy później, ktoś potencjał rzucenia tego wyjątku będzie musiał obsłużyć.

6. Do czego służy sekcja **`finally`** i czy jest wymagana?

W tej opcjonalnej sekcji, która może być częścią obsługi wyjątków, zawieramy instrukcje, które mają zostać wykonane niezależnie od tego, czy wyjątek w sekcji **`try`** został rzucony, czy nie.

7. Jak rzuca się wyjątki?

Wyjątki rzuca się stosując słowo kluczowe **`throw`**, po którym podany powinien zostać wyjątek. Możemy albo rzucić już złapany wyjątek, albo utworzyć nowy za pomocą słowa kluczowego **`new`** dokładnie w ten sam sposób, jak do tej pory tworzyliśmy obiekty różnych klas.

8. Do czego służy słowo kluczowe **`throws`**?

To słowo kluczowe nie powinno być mylone z podobnym do niego słowem kluczowym **`throw`**. Słowo kluczowe **`throws`** stosujemy w sygnaturach metod, aby wskazać, że metoda może rzucić pewny typ wyjątku bądź wyjątków.

9. Jaką regułę muszą spełniać klasy, aby były klasami wyjątków?

Aby klasa była uznawana za klasę wyjątku, musi ona rozszerzać (dziedziczyć po) klasę `Throwable` lub którąś z jej pochodnych. Zazwyczaj jako klasy bazowe dla wyjątków stosuje się klasy `Exception` lub `RuntimeException`, które są pochodnymi klasy `Throwable`.

10. Co trzeba zrobić, aby pobrać komunikat skojarzony z wyjątkiem?

Należy na obiekcie wyjątku wywołać metodę `getMessage`, która ten komunikat zwraca. Metoda ta dziedziczona jest z klasy bazowej.

11. Czym różnią się wyjątki Checked oraz Unchecked?

Wyjątki Checked to wyjątki, które nie mają w swojej hierarchii dziedziczenia klasy `RuntimeException`. Wyjątki pochodzące od `RuntimeException` to wyjątki Unchecked. Jeżeli w wyniku wykonania metody może zostać rzucony wyjątek typu Checked, to muszą one być obsłużone, a metoda ta musi zadeklarować potencjał rzucenia tych wyjątków w swojej sygnaturze za pomocą słowa kluczowego **`throws`**. Wyjątków Unchecked ani nie trzeba obsługiwać, ani deklarować ich za pomocą **`throws`**, chociaż można to zrobić.

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`

Wyjątki `EOFException` i `SQLException` to wyjątki rodzaju Checked, ponieważ nie mają w swojej hierarchii dziedziczenia klasy `RuntimeException`. Pozostałe dwa wyjątki są rodzaju Unchecked.

13. Czy wyjątek `NullPointerException` to wyjątek rodzaju Checked czy Unchecked?

`NullPointerException` to wyjątek rodzaju Unchecked, ponieważ dziedziczy po klasie `RuntimeException`.

14. Co to jest *silent catch* (*połykanie wyjątków*)?

Silent catch to łapanie wyjątków za pomocą **`try`**..**`catch`** bez jakiejkolwiek obsługi tych wyjątków – blok z instrukcjami dla **`catch`** w takich przypadkach jest pusty.

15. Do czego służy *try-with-resources* i jak się tego mechanizmu używa?

Ten dodany w wersji Java 1.7 mechanizm służy do skrótowego zapisu kodu obsługi wyjątków, gdy korzystamy z pewnych zasobów, np. obiektu służącego do czytania bądź zapisywania do pliku. Taki zasób będzie po zakończeniu bloku **`try`**..**`catch`** automatycznie zamknięty (wykonana zostania na nim jego metoda `close`). Oznacza to, że my, jako programiści, nie musimy pisać sami fragmentu kodu służącego do zamknięcia tego zasobu. Obiekt, który ma zostać dla nas obsłużony, podajemy w nawiasach po słowie kluczowym **`try`**, na przykład:

```
try (FileReader fr = new FileReader(f)) {
```

"Zasoby", z jakich możemy korzystać w *try-with-resources*, to obiekty klas, które implementują interfejs `AutoCloseable` lub `Closeable`.

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();
  }
}
```

Nie, ponieważ próbujemy korzystać z klasy `MojWyjatekException` jakby była to klasa wyjątku, ale klasa ta nie dziedziczy po żadnej innej klasie wyjątku – kod się nie skompiluje.

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();
}
```

Nie, ponieważ próbujemy skorzystać ze zmiennej `x` poza blokiem `try`. Zmienna ta zdefiniowana jest w bloku `try` i tylko w nim istnieje. Kompilator zgłosi błąd nieznanego identyfikatora "x" w linii `if (x >= 0) {`

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 {

}
```

Nie, ponieważ kolejność obsługi wyjątków w metodzie `pewnaMetoda` jest nieprawidłowa. Najbardziej ogólne typy wyjątków powinny być zawsze po mniej ogólnych. W tym przypadku wszystkie wyjątki zostaną dopasowane do pierwszego bloku `catch`, ponieważ wszystkie wyjątki mogą być traktowane jako `Exception` (polimorfizm i dziedziczenie), więc kompilator wykryje, że drugi blok `catch` nigdy nie ma szansy się wykonać.

19. Czy poniższa metoda skompiluje się bez błędów?

```
public static void pewnaMetoda() throw IllegalArgumentException {
  // pewne instrukcje moga rzucic wyjatek
}
```

Nie, ponieważ skorzystaliśmy z nieprawidłowego słowa kluczowego. Deklarowanie rzucanych przez metodę wyjątków odbywa się za pomocą słowa **throws**, a nie **throw**.

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();
}
```

W metodzie `main` kompilator zgłosi błąd kompilacji, ponieważ wykryje, że zmienna `wynik` może zostać użyta bez nadania jej wpierw wartości. Jeżeli metoda `getInt` rzuci wyjątek, to instrukcja `wynik = x * x;` nigdy się nie wykona, a tym samym zmienna `wynik` pozostanie niezainicjalizowana żadną wartością.

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();
  }
}
```

Tak, ale po uruchomieniu zobaczymy na ekranie informację, że rzucony został wyjątek `NullPointerException`. Korzystając z `pewnaMetoda` nie musimy używać z **try..catch** do złapania potencjalnego wyjątku `NullPointerException`, ponieważ ten wyjątek jest rodzaju Unchecked. Wyjątków Unchecked nie trzeba koniecznie łapać w **try..catch**.

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.");
  }
}
```

Ten kod się nie skompiluje, ponieważ próbujemy rzucić wyjątek korzystając z konstruktora, który przyjmuje komunikat błędu. Klasa `MojWyjatekException` nie udostępnia takiego konstruktora – kompilator zaprotestuje.

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 {

}
```

Metoda `pewnaMetoda` skompiluje się bez błędów – `IllegalArgumentException` jest rodzaju Unchecked, więc nie musimy deklarować potencjału jego rzucenia.

W metodzie `innaMetoda` kompilator zgłosi błąd – rzucamy wyjątek rodzaju Checked, a nie informujemy o tym w sygnaturze metody za pomocą **`throws`**.

Metoda `kolejnaMetoda` jest poprawna – co prawda jest pusta, ale to nie problem. Metoda może sygnalizować, że może rzucić wyjątek za pomocą **`throws`** nawet, jeżeli tego nie robi.

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
  }
}
```

Nie, ten kod się nie skompiluje, ponieważ łapiąc wyjątki za pomocą | (ang. pipe), musimy wylistować wykluczające się wyjątki. W powyższym kodzie wyjątek `Exception` jest klasą bazową dla wyjątku `MojWyjatekException`, więc wyjątek `MojWyjatekException` i tak załapałby się do tej sekcji `catch` – kompilator zgłosi błąd.

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();
}
```

Druga metoda skompiluje się bez błędów – rzucamy wyjątek typu `Exception`, deklarujemy w klauzuli `throws`, że taki wyjątek może być rzucony – wszystko w porządku.

Pierwsza metoda się nie skompiluje, ponieważ rzuca ona wyjątek typu `Exception` w sekcji `catch` (ponownie rzuca już złapany wyjątek), a nie zawiera klauzuli `throws` w swojej sygnaturze.

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;
  }
}
```

Ten kod kompiluje się bez błędów. Metoda `pewnaMetoda` nie musi zgłaszać za pomocą **`throws`**, że rzuca `IllegalArgumentException`, ponieważ ten typ wyjątku jest rodzaju Unchecked. W metodzie `main` wywołujemy metodę `pewnaMetoda` z argumentem, który spowoduje rzucenie wyjątku. Zostanie on dopasowany do sekcji **`catch`**, ponieważ `Exception` jest jedną z klas, ktore `IllegalArgumentException` rozszerza.

Na ekranie zobaczymy napis "null", ponieważ rzucany wyjątek nie zawiera żadnego komunikatu.

## *10.2 Zadania*

### 10.2.1 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`, gdy jej argument będzie ujemny. Skorzystaj z tej metody w `main`, obsługując potencjalny wyjątek.*

Wykonując to zadanie, musimy pamiętać o:

- utworzeniu klasy wyjątku `BlednaWartoscDlaSilniException`, która będzie dziedziczyć po klasie `Exception`,
- zawarciu w klasie wyjątku konstruktora, który będzie przyjmował jako argument treść komunikatu, ponieważ rzucając wyjątek zawrzemy w nim informację o błędzie,
- dodaniu do sygnatury metody silni klauzuli **`throws`** z informacją o rzucanym wyjątku,
- sprawdzaniu wartości przekazanej jako argument i rzuceniu wyjątku, jeżeli okaże się nieprawidłowa,
- opakowaniu wywołania metody liczącej silnię w blok **`try`**..**`catch`** w metodzie `main`.

Przykładowa implementacja rozwiązania mogłaby wyglądać następująco:

```
class BlednaWartoscDlaSilniException extends Exception {
  public BlednaWartoscDlaSilniException(String message) {
    super(message);
  }
}

public class SilniaZObslugaLiczbUjemnych {
  public static void main(String[] args) {
    try {
      System.out.println("Silnia 5 = " + silnia(5));
      System.out.println("Silnia -2 = " + silnia(-2));
    } catch (BlednaWartoscDlaSilniException e) {
      System.out.println("Wystpil blad: " + e.getMessage());
    }
  }

  public static int silnia(int n)
      throws BlednaWartoscDlaSilniException {

    if (n < 0) {
      throw new BlednaWartoscDlaSilniException(
          "Silnia moze byc liczona tylko dla n >= 0"
      );
    }

    int result = 1;

    for (int i = 2; i <= n; i++) {
      result *= i;
    }

    return result;
  }
}
```

Wynik działania powyższego programu jest następujący:

```
Silnia 5 = 120
Wystpil blad: Silnia moze byc liczona tylko dla n >= 0
```

### 10.2.2 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`****.*

W tym programie musimy utworzyć nowy typ wyjątku udostępniający konstruktor, do którego będziemy mogli przekazać komunikat błędu.

Ponadto, zgodnie z założeniami zadania, w konstruktorze klasy `Osoba` nie powinniśmy od razu rzucać wyjątku, gdy sprawdzimy, że pewny argument przesłany do konstruktora jest nieprawidłowy – zamiast tego powinniśmy do zmiennej typu `String` dodawać kolejne informacje o błędnych danych.

Po sprawdzeniu wszystkich argumentów, jeżeli jakikolwiek komunikat o błędzie został dodany do używanej zmiennej, rzucimy wyjątek `NieprawidlowyAdresException`, którego komunikatem błędu będzie właśnie ten komunikat, zawierający informację o wszystkich błędnych wartościach.

Jeżeli jednak wszystkie pola są poprawne, to po prostu przypiszemy je do pól klasy.

Przykładowa implementacja – w metodzie `main` tworzymy kilka obiektów typu `Adres`, aby zobaczyć, jak program się zachowa:

```
class NieprawidlowyAdresException extends Exception {
  public NieprawidlowyAdresException(String message) {
    super(message);
  }
}

public class Adres {
  private String ulica;
  private int numerDomu;
  private String kodPocztowy;
  private String miasto;

  public Adres(String ulica,
               int numerDomu,
               String kodPocztowy,
               String miasto)
      throws NieprawidlowyAdresException {

    String znalezionBledy = "";
```

```
    if (ulica == null) {
      znalezionBledy += "Ulica nie moze byc nullem. ";
    }

    if (numerDomu <= 0) {
      znalezionBledy += "Numer domu musi byc liczba dodatnia. ";
    }

    if (kodPocztowy == null) {
      znalezionBledy += "Kod pocztowy nie moze byc nullem. ";
    }

    if (miasto == null) {
      znalezionBledy += "Miasto nie moze byc nullem.";
    }

    if (!znalezionBledy.equals("")) {
      throw new NieprawidlowyAdresException(znalezionBledy);
    }

    this.ulica = ulica;
    this.numerDomu = numerDomu;
    this.kodPocztowy = kodPocztowy;
    this.miasto = miasto;
  }

  public static void main(String[] args) {
    try {
      Adres adres = new Adres("Jasna", 1, "05-025", "Warszawa");
      System.out.println("Obiekt typu Adres utworzony.");
    } catch (NieprawidlowyAdresException e) {
      System.out.println("Blad tworzenia adresu: " + e.getMessage());
    }

    try {
      Adres adres = new Adres("Koszykowa", -5, null, "Warszawa");
    } catch (NieprawidlowyAdresException e) {
      System.out.println("Blad tworzenia adresu: " + e.getMessage());
    }

    try {
      Adres adres = new Adres(null, 0, null, null);
    } catch (NieprawidlowyAdresException e) {
      System.out.println("Blad tworzenia adresu: " + e.getMessage());
    }
  }
}
```

Wynik działania tego programu jest następujący:

```
Obiekt typu Adres utworzony.
Blad tworzenia adresu: Numer domu musi byc liczba dodatnia. Kod pocztowy
nie moze byc nullem.
Blad tworzenia adresu: Ulica nie moze byc nullem. Numer domu musi byc
liczba dodatnia. Kod pocztowy nie moze byc nullem. Miasto nie moze byc
nullem.
```

### 10.2.3 Liczba znaków w pliku

*Skorzystaj z try-with-resources w programie, które pobierze od użytkownika lokalizację pliku z rozszerzeniem* `.txt` *na dysku, a następnie 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!*

W tym programie możemy skorzystać z kodu przykładu z rozdziału o mechanizmie *try-with-resources*. W programie nie musimy wypisywać odczytanych z pliku znaków, a jedynie je policzyć – wystarczy więc inkrementować zmienną zliczającą znaki. Będziemy to robić tak długo, aż metoda `read` obiektu klasy `FileReader`, z której skorzystamy, nie zwróci liczby `-1` świadczącej o odczytaniu już całego pliku.

Program będzie zawierał znaną już metodę `getString`, pobierającą od użytkownika tekst, a także metodę `pobierzLokalizacjePliku`, która w pętli będzie próbować pobrać od użytkownika taki tekst, który kończy się na `.txt` (bo takiego pliku oczekujemy).

Odczytywanie pliku ujmiemy w *try-with-resources*, a o potencjalnych błędach poinformujemy użytkownika w odpowiednich sekcjach **`catch`**:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class LiczbaZnakowWPliku {
  public static void main(String[] args) {
    File f = new File(pobierzLokalizacjePliku());
    int liczbaZnakow = 0;

    try (FileReader fileReader = new FileReader(f)) {
      while (fileReader.read() != -1) {
        liczbaZnakow++;
      }

      System.out.println("Liczba znakow: " + liczbaZnakow);
    } catch(FileNotFoundException e) {
      System.out.println("Podany plik nie istnieje!");
    } catch (IOException e) {
      System.out.println(
          "Wystapil blad podczas odczytywania pliku: " + e.getMessage()
      );
    }
  }

  public static String pobierzLokalizacjePliku() {
    String sciezkaDoPliku;

    while (true) {
      System.out.print("Podaj lokalizacje pliku txt na dysku: ");
      sciezkaDoPliku = getString();

      if (!sciezkaDoPliku.endsWith(".txt")) {
```

```
            System.out.println("Zle rozszerzenie pliku!");
        } else {
            return sciezkaDoPliku;
        }
    }
  }

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

Dwa przykładowe uruchomienia tego programu:

```
Podaj lokalizacje pliku txt na dysku: C:/pewienPlik
Zle rozszerzenie pliku!
Podaj lokalizacje pliku txt na dysku: C:/pewienPlik.txt
Liczba znakow: 36
```

```
Podaj lokalizacje pliku txt na dysku: C:/nieistniejacyPlik.txt
Podany plik nie istnieje!
```

### 10.2.4 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.*

Klasa `Stack` to ciekawe zadanie, ponieważ pomimo prostoty tej struktury danych, trzeba wziąć pod uwagę kilka rzeczy. W pierwszej kolejności musimy zastanowić się jak przechowywać dane – w tym celu możemy skorzystać z tablicy liczb typu `int`, którą nazwiemy `values`.

W konstruktorze będziemy tworzyć tablicę `values` o rozmiarze przekazanym jako argument konstruktora. Najpierw jednak sprawdzimy, czy liczba ta nie jest ujemna – w takim przypadku rzucimy wyjątek `IllegalArgumentException`.

Musimy także w jakiś sposób zapisywać informację o liczbie już przechowywanych na stosie elementów – będzie ona nam potrzebna w każdej z pozostałych metod klasy `Stack`.

Aktualny rozmiar stosu będziemy zapisywać w polu o nazwie `currentSize`, które będzie miało wartość (domyślną) zero dla każdego nowego obiektu typu `Stack`.

Następnie, będziemy odpowiednio operować tym polem w każdej z metod:

- `size` – po prostu zwracamy wartość pola `currentSize`.
- `clear` – ustawiamy `currentSize` na `0` (nie musimy "zerować" tablicy z danymi, bo przechowujemy liczby typu prymitywnego – jeżeli nasz stos przechowywałby obiekty pewnej klasy, to powinniśmy całą tablicę "wynullować", aby Garbage Collector wiedział, że przechowywane tam wcześniej obiekty nie są już potrzebne i powinny zwolnić pamięć).
- `push` – dodajemy do tablicy `values` element przekazany jako argument – indeks, pod którym wstawiamy element, to aktualna wartość pola `currentSize`. Wstawiając element, dodatkowo zwiększymy wartość przechowywaną w polu `currentSize`. Dla przykładu, jeżeli stos jest pusty i wywołamy metodę `push` z argumentem `5`, to wstawimy tą wartość do elementu tablicy `values[0]`, ponieważ `currentSize` wynosi `0`. Dodatkowo zwiększymy `currentSize` z `0` do `1`. Na początku metody `push` musimy jeszcze sprawdzić, czy liczba aktualnie przechowywanych elementów nie jest już równa rozmiarowi tablicy – będzie to oznaczało, że stos jest pełny i nie możemy już dodać do niego nowych elementów – zamiast tego rzucimy wyjątek `StackFullException`.
- `pop` – zwracamy aktualny element i zmniejszamy `currentSize` o `1`. Poprzednio dodany element znajduje się zawsze w elemencie tablicy o indeksie mniejszym o `1` od wartości `currentSize`, dlatego korzystamy z operatora predekrementacji w metodzie `pop`, co zobaczysz w kodzie poniżej. Jeżeli `currentSize` jest równy zero, czyli stos jest pusty, to zamist tego rzucimy wyjątek `StackEmptyException`.
- `top` – zwracamy ostatnio dodaną do stosu wartość (tą na "szczycie" stosu) – ponieważ tablice indeksujemy od `0`, od wartości `currentSize` musimy odjąć `1`, ponieważ gdy w stosie będzie przechowywany jeden element, to będzie on w elemencie `values[0]`, a `currentSize` będzie wtedy miało wartość `1`. Jeżeli jednak `currentSize` wynosi `0`, czyli stos jest pusty, to rzucimy wyjątek `StackEmptyException`.

Implementacja klasy `Stack` mogłaby wyglądać następująco – w ramach treningu przeanalizuj poniższy kod:

```
class StackEmptyException extends RuntimeException {}
class StackFullException extends RuntimeException {}

public class Stack {
  private int[] values;
  private int currentSize;

  public Stack(int size) {
    if (size < 0) {
      throw new IllegalArgumentException("size musi byc >= 0");
    }
    this.values = new int[size];
```

```
  }

  public void push(int value) {
    if (currentSize == values.length) {
      throw new StackFullException();
    }
    values[currentSize++] = value;
  }

  public int pop() {
    if (currentSize == 0) {
      throw new StackEmptyException();
    }
    return values[--currentSize];
  }

  public int top() {
    if (currentSize == 0) {
      throw new StackEmptyException();
    }
    return values[currentSize - 1];
  }

  public int size() {
    return currentSize;
  }

  public void clear() {
    currentSize = 0;
  }
}
```

Klasę `Stack` używamy w osobnej klasie `StackUzycie`:

```
public class StackUzycie {
  public static void main(String[] args) {
    try {
      Stack s = new Stack(-1);
    } catch (Exception e) {
      e.printStackTrace();
    }

    try {
      Stack s = new Stack(0);
      s.push(1);
    } catch (Exception e) {
      e.printStackTrace();
    }

    try {
      Stack s = new Stack(1);
      s.push(1);
      System.out.println("Rozmiar: " + s.size());
      System.out.println("Top: " + s.top());
      System.out.println("Wartosc: " + s.pop());
      System.out.println("Rozmiar: " + s.size());
      System.out.println("Top: " + s.top());
    } catch (Exception e) {
      e.printStackTrace();
    }
```

```
    try {
      Stack s = new Stack(3);
      s.push(5);
      s.push(10);
      s.push(15);
      s.clear();
      s.push(20);
      System.out.println("Wartosc: " + s.pop());
      System.out.println("Top: " + s.top());
    } catch (Exception e) {
      e.printStackTrace();
    }

    try {
      Stack s = new Stack(5);
      s.push(10);
      s.push(200);
      s.push(3000);
      s.push(40000);
      s.push(500000);
      System.out.println("Rozmiar: " + s.size());
      System.out.println("Top: " + s.top());
      System.out.println("Wartosc: " + s.pop());
      System.out.println("Top: " + s.top());
      s.push(5);
      System.out.println("Wartosc: " + s.pop());
      System.out.println("Wartosc: " + s.pop());
      System.out.println("Wartosc: " + s.pop());
      System.out.println("Top: " + s.top());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Wynik działania klasy `StackUzycie` jest następujący:

```
java.lang.IllegalArgumentException: size musi byc >= 0
     at wyjatki.Stack.<init>(Stack.java:12)
     at wyjatki.StackUzycie.main(StackUzycie.java:4)

wyjatki.StackFullException
     at wyjatki.Stack.push(Stack.java:19)
     at wyjatki.StackUzycie.main(StackUzycie.java:11)

Rozmiar: 1
Top: 1
Wartosc: 1
Rozmiar: 0
wyjatki.StackEmptyException
     at wyjatki.Stack.top(Stack.java:33)
     at wyjatki.StackUzycie.main(StackUzycie.java:23)

Wartosc: 20
wyjatki.StackEmptyException
     at wyjatki.Stack.top(Stack.java:33)
     at wyjatki.StackUzycie.main(StackUzycie.java:36)

Rozmiar: 5
```

```
Top: 500000
Wartosc: 500000
Top: 40000
Wartosc: 5
Wartosc: 40000
Wartosc: 3000
Top: 200
```