Inżynieria oprogramowania na bazie języka Java

CodeCouple.pl

Version 1.1.15, 23-01-2019

Table of Contents

Przedmowa

Cześć, nazywam się **Krzysztof Chruściel**. Jestem **Java** developerem od kilku lat. Stworzyłem tą ksiażkę z myślą o ludziach, którzy chcą zacząć swoją przygodę z programowaniem w języku **Java**. Znajdziecie w niej bardzo szeroki zakres materiału. Na początku zaczniemy od kompilacji prostych plików, następnie przejdziemy przez cały proces wytwarzania oprogramowania, aby na końcu zbudować prostą aplikację.

Dlaczego kolejna książka? Na rynku istnieje bardzo wiele pozycji dotyczących programowaniem w języku **Java** jednakże niewiele z nich stawia na połączenie teorii z praktyką. Każdy rozdział składa się z krótkiego opisu teoretycznego oraz zestawu zadań związanych z omawianym tematem. Na końcu książki znajdują się odpowiedzi do zadań (rekomenduję pracę samodzielną ;))

Jeśli masz jakiekolwiek pytania lub sugestie odnośnie książki, proszę nie czuj się skrępowany tylko do mnie napisz!

- Blog: https://CodeCouple.pl
- Email: krzysztof.chrusciel@outlook.com

Dziel się wiedzą z innymi!

Wstęp

Zanim zaczniemy przygodę z programowaniem musimy przygotować sobie środowisko. W rodziale tym omówimy różnice pomiędzy **JRE** a **JDK** oraz przejdziemy przez cały proces tworzenia gotowej aplikacji.

JRE - Java Runtime Environment

Aby uruchomić aplikacje na platformie **Java**, musimy mieć zainstalowane środowisko uruchomieniowe **JRE**: JRE.

JDK - Java Development Kit

Aby tworzyć aplikacje na platformie **Java**, musimy mieć zainstalowane środowisko developerskie **JDK**: JDK.

Wersja Javy

Po instalacji **JRE/JDK** należy sprawdzić czy **Java** działa poprawnie. Aby sprawdzić aktualnie używaną wersję **Javy** możemy użyć wiersz poleceń:

Bash

```
java -version

java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) Client VM (build 25.121-b13, mixed mode)
```

HelloWorld

Wiemy już, że środowisko **Java** działa na naszej maszynie pora więc zacząć naszą przygodę z **programowaniem** w **Javie**. Użyjemy jednego z edytorów tekstowych takich jak Notepad++. Dodajmy nowy plik `Runner.java`. W tym pliku **napiszemy** (nie wklejajmy!) poniższe linie (treść tego pliku będzie wyjaśniona w następnych rozdziałach):

Runner.java

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

Kompilacja

Pliki z rozszerzeniem `.java` nie są jeszcze **skompilowane**. Aby **skompilować** źródła **Javowe** należy użyć wiersz poleceń (w folderze gdzie znajduje się plik z rozszerzeniem `.java`):

Bash

```
javac YourJavaFileName.java
```

W naszym przykładzie:

Bash

```
javac Runner.java
```

Po udanej kompilacji w folderze znajdują się dwa pliki:

- `Runner.java` - kod źródłowy
- `Runner.class` - skompilowana klasa z kodem bajtowym (więcej o kodziej bajtowym w bloku "Wprowadzenie do technologii JVM")

Uruchomienie

Aby uruchomić naszą mega aplikację, należy użyć wiersz poleceń (w folderze gdzie znajduje się plik z rozszerzeniem `.class`):

Bash

```
java YourJavaFileName
```

W naszym przykładzie:

Bash

```
java Runner
```

Wynik:

Bash

```
Hello World!
```

Archiwum

No dobra, mamy już działającą aplikację. Wyobraź sobie teraz sytuację w której udostępniasz ją innym osobą. Wysyłasz jeden plik i inni mogą korzystać z twojej aplikacji. A co jeśli twoja aplikacja

składa się z większej ilość plików? Tutaj pojawia się problem (możemy oczywiście wysyłać za każdym razem dużą ilość plików).

Rozwiązaniem tego problemu jest umieszczenie ich w archiwum. **Java** udostępnia mechanizm do tworzenia archiwów dedykowanych dla tego języka. Rodzaj archiwum zależy od typu aplikacji który tworzysz:

- **JAR - Java Archive** - dla prostych aplikacji
- **WAR - Web Application Archive** - dla aplikacji webowych (zawiera plik `web.xml`)
- **EAR - Enterprise Application Archive** - jest zbiorem plików **JAR** oraz **WAR**. Zawiera także informacje o powiązaniach pomiędzy nimi (zawiera plik `context.xml`)

Dla naszych aktualnych potrzeb archiwum typu **JAR** jest wystarczające.

JAR - Java Archive - Tworzenie

Aby stworzyć archiwum **JAR** należy użyć wiersz poleceń (w folderze gdzie znajdują się skompilowane źródła):

Bash

```
jar cf nazwa-pliku pliki-do-spakowania
```

- c - opcja, która oznacza tworzenie c - create
- f - opcja, która oznacza umieszczenie plików do archiwum

W naszym przykładzie

Bash

```
jar cf helloWorldApp.jar Runner.class
```

Po wydaniu polecenia powinien pokazać się nowy plik `helloWorldApp.jar` w naszym folderze.

JAR - Java Archive - Uruchamianie

Aby uruchomić plik **JAR** należy wydać polecenie (w folderze gdzie znajduje się plik z rozszerzeniem `.jar`):

Bash

```
java -jar jar-file-name
```

W naszym przykładzie

Bash

```
java -jar helloWorldApp.jar
```

Czyżby pojawił się błąd:

Bash

```
no main manifest attribute, in helloWorldApp.jar
```

Pamiętaj o pliku MANIFEST.MF, sprawdź zawartość archiwum

JAR - Java Archive - Manifest

Manifest służy do przechowywania meta-informacji o naszej aplikacji. Mogą to być informacje o autorze, zależnościach czy o klasie startowej. Aby dodać własny **Manifest** należy utworzyć plik `MANIFEST.MF` z treścią:

Pamiętaj o pustej linii na końcu!

MANIFEST.MF

```
Main-Class: Runner
```

Powyżej wskazaliśmy, która klasa ma zostać uruchomiona. Następnie możemy utworzyć **JAR** wraz z naszym **manifestem** wykonując polecenie:

Bash

```
jar cfm nazwa-pliku.jar plik-manifest pliki-do-spakowania
```

Gdzie:

- m - oznaczna wskazanie **manifestu**

W naszym przykładzie:

Bash

```
jar cfm helloWorldApp.jar MANIFEST.MF Runner.class
```

Spróbujmy teraz uruchomić naszą mega aplikację (`java -jar helloWorldApp.jar`)!

Więcej informacji na temat tworzenia plików **JAR** znajdziemy pod linkiem: Creating a JAR File.

Zadania

- Zainstaluj pakiet **JDK** dla **Javy 8**
- Sprawdź wersję **Javy**
- Utwórz klasę `Runner.java`, która wypisuje `Hello World!`
- Skompiluj klasę `Runner.java`
- Uruchom klasę `Runner`
- Stwórz archiwum **JAR** dla swojej aplikacji
- Uruchom stworzone archiwum
- Stwórz plik manifest ze wskazaniem klasy startowej `Runner`
- Stwórz archiwum **JAR** z własnym manifestem
- Uruchom stworzone archiwum

IDE - Integrated Development Environment

Jak widzieliśmy w poprzednich ćwiczeniach nie potrzeba żadnych specjalnych programów do tworzenia kodu. Jednakże mogą one bardzo pozytywnie wpłynąć na to jak wygląda nasza codzienna praca. Programy te określane są mianem **IDE - Integrated Development Environment**, ponieważ dostarczają wszystkie narzędzia oraz środowisko do **tworzenia/uruchamiania/debugowania** aplikacji. Na rynku istnieje wiele rozwiązań, które służą do tworzenia aplikacji w języku **Java**:

- **IntelliJ**
- **Eclipse**
- **NetBeans**
- Więcej...

IntelliJ

Najpopularniejszym narzędziem w środowisku **Javy** jest aplikacja **IntelliJ** tworzona przez firmę **JetBrains**. Jest to bardzo dobre narzędzie, które występuje w dwóch wersja:

- **Community** - wersja darmowa (wystarczająca dla naszych potrzeb)
- **Ultimate** - wersja płatna (więcej pomocnych funkcjonalności)

Meta-dane

Wszystkie informacje o ustawieniach naszego obszaru roboczego (workspace) przechowywane są w pliku `nazwa-projektu.iml` oraz w folderze `.idea`.

Skróty

Aby poprawić efektywność swojej pracy należy opanować skróty! Jest to podstawowy element rzemiosła każdego programisty. Na następne zajęcia należy wydrukować **keymap** dla programu **IntelliJ** link.

Live Templates

Narzędzia typu **IDE** dostarczają rozwiązania typu **Live Templates**. **Live Templates** pozwala na definiowe szablonów kodu do późniejszego reużywania:

- `psvm` - tworzy `public static void main(String[] args)`
- `inn` - tworzy `if(arg != null)`
- więcej...

Można tworzyć własne **live template**!

Pluginy

Pluginy to dodatki, które rozszerzają działanie **IDE**. Przydatnym pluginem jest **Key Promoter** (podpowiada skróty).

Zadania

- Pobrać i zainstalować **IntelliJ**
- Wydrukować kartkę ze skrótami
- Stworzyć nowy projekt **Java** korzystając z szablonu `Java Hello World`
- Sprawdzić czy dodał się plik `nazwa-projektu.iml` oraz folder `.idea`.
- Uruchomić aplikację z poziomu **IntelliJ**
- Dodać plugin "key promoter"

Zależności

W pierwszych zadaniach udało nam się stworzyć prostą aplikację. Niestety dystrubucja gotowego produktu (który nazywanym jest też artefaktem) jest mocno utrudniona, ponieważ polega ona na rozsyłaniu archiwum. Jeśli tworzymy rozwiązania, które chcemy współdzielić z innymi musimy skorzystać z narzędzi do zarządzania artefaktami. Na rynku istnieje kilka rozwiązań tego typu:

- **Maven**
- **Gradle**
- **Ant**
- Więcej...

Maven

Najpopularniejszym narzędziem do budowania w środowisku **JVM** jest aktualnie **Maven** (**Gradle** mocno goni ;)). Ponadto pozwala on w łatwy sposób zarządzać zależnościami.

Konwencja ponad konfiguracje

Maven zyskał tak dużą popularność poprzez podejście *convention over configuration*. Jest to podejście, w którym umawiamy się na dowolną konwencje przykładowo w jakich folderach będziemy trzymać źródła aplikacji. Konwencją dla narzędzia **Maven** są:

- `src` - folder zawierający kod źródłowy
- `src/main/java` - folder zawierający kod źródłowy (ale nie jest to kod testów)
- `src/main/resources` - folder zawierający dodatkowe pliki wymagane dla naszej aplikacji (mogą to być ustawienia)
- `src/test/java` - folder zawierający kod źródłowy testów naszej aplikacji

Po zbudowanie naszej aplikacji (jak budować korzystając z narzędzia **Maven** o tym później) pojawią się dodatkowe foldery:

- `target` - folder zawierający zbudowany kod źródłowy
- `target/classes` - folder zawierający skompilowane klasy

pom.xml

Wszystkie zależności związane z **Maven** umieszczamy w pliku `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.sda</groupId>
    <artifactId>posts-application</artifactId>
    <version>0.0.1-SNAPSHOT</version>


</project>
```

GAV

GAV jest akronimem od słów:

- **Group ID** - nazwa domeny organizacji pisana od tyłu (`sda.pl → pl.sda.produkt`)
- **Artifact ID** - nazwa artefaktu
- **Version** - wersja artefaktu

```
<groupId>pl.sda</groupId>
<artifactId>application-name</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

Semantic Versioning

Semantic Versioning jest uporządkowaną metodyką służącą do nadawania numeru wersji aplikacji. Każda wersja aplikacji składa się z trzech głównych elementów:

```
MAJOR.MINOR.PATCH
```

Gdzie:

- **MAJOR** - oznacza zmiany, które nie zapewniają kompatybilności
- **MINOR** - oznacza dodanie nowej funkcjonalności
- **PATCH** - oznacza drobne zmiany jak naprawa błędu lub refactoring (usprawnienie kodu)

Przykłady:

```
2.1.3 - druga wersja aplikacji, z jedną nową funkcjonalnością i trzema małymi
usprawnieniami
2.0.0 - druga wersja aplikacji
1.0.1 - pierwsza wersja aplikacji z jednym usprawnieniem
```

Dodatkowo w numerze wersji można umieszczać **meta-informacji** (czyli dodatkowe informację) na temat wydania. Tutaj istnieje dowolność jeśli chodzi o nazewnictwo, jednakże lepiej trzymać się standardów:

- wersja niestablina (testowa)
  - **SNAPSHOT** - wersja nie przetestowana produkcyjnie
  - **ALPHA** - autorzy doprowadzają do rzeczywistego działania programu, nawet w ograniczonym zakresie
  - **BETA** - kiedy program ma już pierwszych użytkowników, zwanych często beta testerami
  - **RC** - release candidate, kandydat do wydania

Więcej można poczytać tutaj.

Dependencies

Jak pisałem we wstępie **Maven** wykorzystywany jest do zarządzania zależnościami. Wyobraźmy sobie sytuację, w której chcemy skorzystać z biblioteki do sprawdzania daty. Moglibyśmy odnaleźć stronę projektu a następnie pobrać archiwum **JAR**. Pobrane archiwum umieścilibyśmy w projekcie aby w końcu móc z niego korzystać. Po pewnym czasie wychodzi nowa wersja tej biblioteki i powtarzamy cały proces. Dzięki **Maven'owi** możemy robić to w prosty sposób. Wszystkie zależności trzymane są na serwerze zdalnym zwanym **Maven Central** (o nim za chwilę). Jeśli wybraliśmy już interesującą nas zależność w **Maven Central**, w znanym już nam pliku `pom.xml` w sekcji `dependencies` dodajemy nową zależność:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.sda</groupId>
    <artifactId>posts-application</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>

        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.7</version>
        </dependency>

    </dependencies>

</project>
```

Maven Central

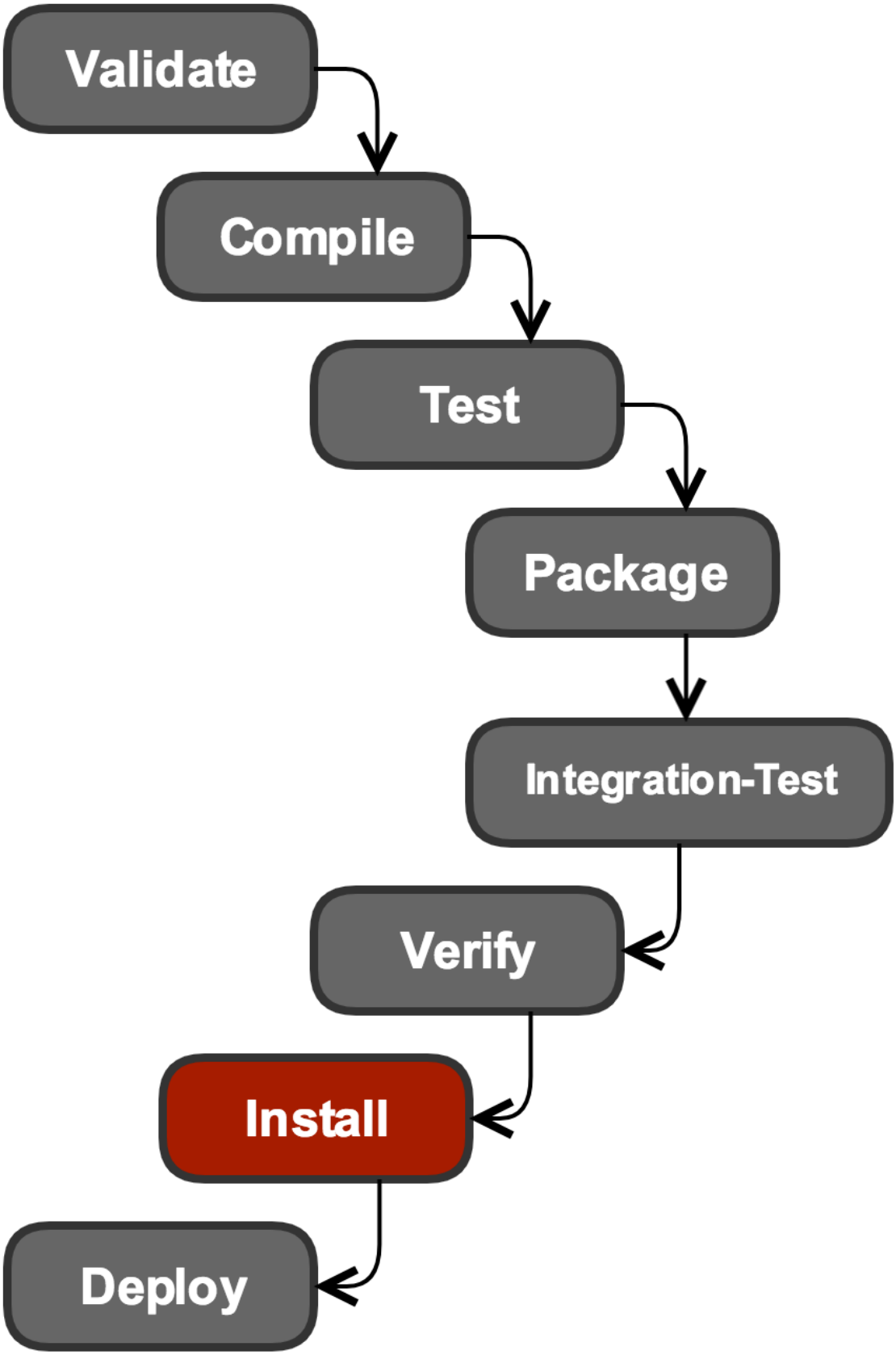
Jeśli poszukujemy artefaktów stworzonych przez innych musimy udać się do repozytorium zdalnego. Repozytorium zdalnym dla projektu **Maven** jest **Maven Central**, które jest dostępne pod adresem: https://mvnrepository.com/

Cykl życia

- **default** - odpowiedzialny za zbudowanie artefaktu
- **site** - odpowiedzialny za zbudowanie dokumentacji
- **clean** - odpowiedzialny za czyszczenie projektu

Fazy

Każdy cykl życia ma swoje **fazy**. Najważniejszym cyklem jest **default**, który ma następujące **fazy** (uruchamiane są sekwencyjnie):



- `validate` - sprawdza czy można skompilować projekt
- `compile` - kompiluje źródła (`javac`)

- `test` - odpala testy z `src/test/java`
- `package` - tworzy archiwum (domyślnie **JAR**)
- `integration-test` - uruchamia testy integracyjne (na tym etapie faza ta jest dla nas nieistotna)
- `verify` - sprawdza poprawność stworzonego archiwum
- `install` - instaluje artefakt w lokalnym repozytorium (`.m2`)
- `deploy` - instaluje artefakt w repozytorium zdalnym (na przykład **Maven Central**)

Cykl życia **clean** ma tylko jedną fazę:

- `clean` - usuwa ona zawartość folderu `target`

Na tym etapie pomijamy cykl **site**. Więcej informacji na temat cyklu życia.

.m2

Wszystkie pobrane do tej pory zależności oraz zbudowane przez nas **artefakty** znajdują się w folderze `.m2`. Folder `.m2` można znaleźć w folderze użytkownika. Dla systemu Windows: `C:\Users\krzysztof-chrusciel\.m2`. Jest to tak zwane lokalne repozytorium, w którym **instalowane** są artefakty.

Zadania

- Utworzyć nowy projekt **Maven**
- Nadać odpowiednie parametry **GAV**
- Dodać nową zależność do biblioteki `commons-lang3` (sprawdź **Maven Central**)
- Sprawdzić zawartość folderu `.m2`
- W folderze zgodnym z konwencją umieść klasę `Runner` (tą z poprzedniego ćwiczenia)
- Zainstaluj aplikację w lokalnym repozytorium
- Dokonaj modyfikacji w klasie `Runner` (zamień "HelloWorld" na "Maven")
- Zwiększy numer wersji w `pom.xml`
- Usuń zawartość folderu `target` (clean)
- Ponownie zainstaluj aplikację w lokalnym repozytorium
- Sprawdzić zawartość folderu `.m2`

Elementy języka

Pakiet (package)

Pakiety są niczym innym jak **folderami**. Służą one do ustrukturyzowania naszego kodu źródłowego:

```
pl.sda.cars - kod źródłowy związany z samochodami
pl.sda.cars.windows - kod źródłowy związany z szybami samochodów
pl.sda.cars.tires - kod źródłowy związany z oponami samochodów
pl.sda.cars.tires.winter - kod źródłowy związany z oponami zimowymi samochodów
pl.sda.cars.tires.summer - kod źródłowy związany z oponami letnimi samochodów
```

Po utworzeniu klasy w odpowiednim pakiecie, należy umieścić w niej informację o aktualnej lokalizacji. Informację o lokalizacji wskazujemy wykorzystując słowo kluczowe `package`:

Runner.java

```
package pl.sda.runner;

class Runner {

    // Ciało klasy

}
```

Klasa (class)

Klasa jestem podstawowym elementem programowania obiektowego. Jest ona reprezentacją rzeczywistości:

```
opcjonalny-modyfikator-dostępu class NazwaKlasy {

    // Ciało klasy

}
```

W naszym przykładzie:

Car.java

```
public class Car {

    // Ciało klasy

}
```

Tworzenie klasy (new)

Klasy są typami **referencyjnymi**. Aby stworzyć nową **instancję** klasy używamy słowa kluczowego **new**:

```
Car maluch = new Car();
```

Pole (field)

Pola służą do przechowywania stanu klasy. Pole może być **zadeklarowane** oraz **zainicjalizowane**.

Car.java

```
public class Car {

    public int numberOfWheels = 4; // inicjalizacja
    public String carName; // deklaracja

}
```

Używanie pól

```
Car maluch = new Car();
System.out.println("Liczba kół: " + maluch.numberOfWheels);
```

Metoda (method)

Metoda vs funkcja - metody są w klasie, funkcje po za klasą. W **Javie** mamy tylko metody!

Metody w klasie odpowiedzialne są za jej zachowania. Metoda może zwrócić wartość (poprzez użycie słowa kluczowego `return` - przerywa działanie metody). Jeśli nie chcemy zwracać wartości z metody używamy typu `void`.

```
opcjonalny-modyfikator-dostępu class NazwaKlasy {

    opcjonalny-modyfikator-dostępu typ-zwracany nazwaMetody(opcjonalne-argumenty) {
        //Ciało metody
    }

}
```

W naszym przykładzie:

Car.java

```
public class Car {

    //Pola
    public int numberOfWheels = 4;
    public String carName;

    //Metody
    public int getNumberOfWheels() {
        return numberOfWheels;
    }

    public int getTotalWheelPrice(int wheelPrice) {
        return numberOfWheels * wheelPrice; // zwraca wynik i przerywa działanie
metody
    }

    public void printNumberOfWheels() {
        System.out.println(getNumberOfWheels()); // metoda nic nie zwraca bo jest typu
void
    }

}
```

Wywołanie metody

```
Car maluch = new Car();
System.out.println("Cena za wszystkie koła: " + maluch.getTotalWheelPrice(4));
```

Zmienne (variables)

Pola są w klasie, zmienne w metodach!

Zmienne służą do przechowywania wartość w metodach (ich żywotność w pamięci trwa tyle ile działanie metody).

Używanie zmiennych

Car.java

```
public class Car {

    public int getTotalWheelPrice(int wheelPrice) {
        int variable = 20; // zmienna
        int secondVariable = 50; // druga zmienna
        return numberOfWheels * wheelPrice + variable + secondVariable;
    }

}
```

Konstruktor (constructor)

Jest specjalną metodą wywoływaną w momencie tworzenia (konstrułowania) obiektu.

```
opcjonalny-modyfikator-dostępu class NazwaKlasy {

    //Konstruktor
    opcjonalny-modyfikator-dostępu NazwaKlasy(opcjonalne-argumenty) {
        //Logika konstruktora
    }

}
```

W naszym przykładzie:

Car.java

```
public class Car {

    //Pola
    int numberOfWheels = 4;
    String carName;

    //Konstruktor domyślny - nie trzeba go tworzyć gdy nie ma innego
    Car() {}

    //Konstruktor
    Car(int numberOfWheels, String carName) {
        this.numberOfWheels = numberOfWheels;
        this.carName = carName;
    }

    //Metody
    public int getNumberOfWheels() {
        return numberOfWheels;
    }

    public int getTotalWheelPrice(int wheelPrice) {
        return numberOfWheels * wheelPrice;
    }

}
```

Wywołanie konstruktora

```
Car maluch = new Car(); //domyślny
System.out.println("Ilość kół: " + maluch.getNumberOfWheels());

Car scania = new Car(8, "Scania"); //stworzony przez nas
System.out.println("Ilość kół: " + scania.getNumberOfWheels());
```

Modyfikatory dostępu

- `public` - z elementów publiczny mogą korzystać wszyscy
- `protected` - z elementów chronionych mogą korzystać wszystkie dzieci klasy oraz klasy w obrębie tego samego pakietu
- `private` - elementy prywatne mogą być wykorzystywane tylko w obrębie klasy, w której się znajdują
- **package-scope** - elementy z dostępem pakietowym mogą być wykorzystywane w ramach tego samego pakiety, w którym się znajdują

Dziedziczenie (extends)

Dziedziczenie w **Javie** realizuje się poprzez słowo kluczowe `extends`. Dziedziczenie oznacza dziedziczenie cech od rodzica. Gdzie cechą mogą być pola i metody.

Maluch.java

```
class Maluch extends Car {

}
```

Importowanie (import)

Jeśli w obrębie naszej klasy, używamy klas, które znajdują się w innych pakietach musimy w naszej klasie wskazać gdzie się one znajdują. Wskazanie realizujemy korzystając ze słowa kluczowego `import`:

pl.sda.Car.java

```
package pl.sda;

public class Car {

}
```

pl.sda.cars.Maluch.java

```
package pl.sda.cars;

import pl.sda.Car;

class Maluch extends Car {

}
```

Zadania

- Stwórzyć nowy projekt **Maven** dla aplikacji bankowej
- Stworzyć pakiet:
  - `pl.sda.bank`
- W pakiecie `pl.sda.bank` stwórz klasę `Bank`, która ma dostęp publiczny
- W klasie `Bank` dodaj metodę o dostępie chroninym zwracającą imiona dłużników jako wartość tekstową
- Stworzyć pakiet:
  - `pl.sda.bank.pko`

- W pakiecie `pl.sda.bank.pko` stwórz klasę `BankPKO`, która ma dostęp publiczny i dziedziczy po klasie `Bank`
- W klasie `BankPKO` dodaj pole o dostępie prywatnym zawierającym oprocentowanie jako wartość liczbową
- W klasie `BankPKO` dodaj metodę o dostępie chroninym zwracającą wartość pola z oprocentowaniem
- Stworzyć pakiety:
  - `pl.sda.bank.pko.alior`
  - `pl.sda.bank.pko.ing`
- W pakiecie `pl.sda.bank.pko.alior` stwórz klasę `BankAlior`, która ma dostęp publiczny i dziedziczy po klasie `BankPKO`
- W klasie `BankAlior` dodaj pole o dostępie prywatnym przechowujące nazwę banku
- W klasie `BankAlior` dodaj metodę o dostępie prywatnym zwracającą prowizję jako wartość liczbową (prowizja to 10 + oprocentowanie)
- W klasie `BankAlior` dodaj metodę o dostępie publicznym zwracającą nazwe banku wraz z prowizją
- W pakiecie `pl.sda.bank.pko.ing` stwórz klasę `BankING`, która ma dostęp publiczny i dziedziczy po klasie `BankPKO`
- W klasie `BankING` dodaj pole o dostępie prywatnym przechowujące nazwę banku
- W klasie `BankING` dodaj metodę o dostępie prywatnym zwracającą prowizję jako wartość liczbową (prowizja to 15 + oprocentowanie)
- W klasie `BankING` dodaj metodę o dostępie publicznym zwracającą nazwe banku wraz z prowizją
- W pakiecie `pl.sda.bank` stwórz klasę `Runner`, która ma dostęp publiczny
- Dodaj metodę startową (`psvm` w **IntelliJ**)
- Stwórz nową instancję klasy `BankAlior` i wypisz nazwę banku wraz z prowizją i oprocentowaniem
- Stwórz nową instancję klasy `BankING` i wypisz nazwę banku wraz z prowizją i oprocentowaniem

Testowanie

Na początku naszej przygody poznaliśmy kroki jakie musimy wykonać aby stworzyć aplikację. Następnie poznaliśmy narzędzia takie jak Maven, które pozwalają nam współdzielić wyniki naszej pracy. Ostatni temat dotyczył elementów języka. Po poznaniu elementów języka chcielibyśmy poznać możliwości **Javy**. Każdy dobry programista stosuję metodykę pracy **TDD**, w której zaczynamy od napisania testu. **Testy** weryfikują poprawność aplikacji.

JUnit

Najpopularniejszą aktualnie biblioteką do testów jest biblioteka **JUnit** w wersji **5**. Aby zacząć korzystanie z rozwiązania **JUnit** wykorzystamy mechanizm **Maven**. Dodajemy nową zależność w naszym pliku `pom.xml`:

pom.xml

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```

Struktura testu

CarTest.java

```
public CarTest {

    @Test
    public void shouldReturnNumberOfWheels() {
        // Given
        Car maluch = new Car("Maluch");
        // When
        int numberOfWheels = maluch.getNumberOfWheels();
        // Then
        assertEquals(numberOfWheels, 4);
    }

}
```

GivenWhenThen

Musimy pamiętać, że w codziennej pracy programisty zachodzi zasada pareto **80/20**. **80%** czasu spędzamy na czytaniu kodu, natomiast tylko **20%** na pisaniu nowego. Jeśli więc więcej czasu spędzamy na czytaniu kodu, powinniśmy przykładać jak największą uwagę do czytelności testów! Jednym ze sposobów poprawny czytelności jest tak zwany trójpodział. Jest to podział testu na trzy

części logiczne części:

- **Given**
- **When**
- **Then**

Given

W sekcji `given` ustawiamy warunki początkowe testu:

CarTest.java

```
//Given
Car maluch = new Car("Maluch");
```

When (kiedy)

W sekcji `when` wywołujemy akcję (jej wynik przypisujemy do zmiennej), która jest testowana:

CarTest.java

```
//When
int numberOfWheels = maluch.getNumberOfWheels();
```

Then

W sekcji `then` sprawdzamy poprawność działania metody poprzez **asercje**:

CarTest.java

```
//Then
assertEquals(numberOfWheels, 4);
```

Asercje

Asercje służą do weryfikacji poprawności testu. Sprawdzają one czy spełniony został oczekiwany wynik. W naszym przykładzie, czy maluch ma cztery koła.

Asercje JUnit

Biblioteka **JUnit** dostarczą całą game assercji dzięki którym możemy sprawdzić poprawność kodu:

- assertEquals(excpected, actual) - `assertEquals(numberOfWheels, 4);`
- assertTrue(warunek-logiczny) - `assertTrue(10 > 3);`
- assertFalse(warunek-logiczny) - `assertFalse(3 > 10);`
- więcej...

Asercje AssertJ

Asercje wbudowane w bibliotekę **JUnit** są jak najbardziej poprawne, jednakże pamiętanie, który z elementów jest `excpeted` a który `actual` bywa uciążliwe. Rozwiązaniem tego problemu jest biblioteka **AssertJ**, która dostarcza tak zwane **fluent assertions** (czytelne asercje). Aby dodać **AssertJ** do naszego projektu, wykorzystamy dobrze już nam znany mechaznim do obsługi zależności, czyli **Maven**. Do pliku `pom.xml` dodajemy nową zależność:

pom.xml

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.8.0</version>
    <scope>test</scope>
</dependency>
```

Od teraz możemy korzystać z mechanizmu **fluent assertions** (poprawia to czytelność kodu testowego):

CarTest.java

```
public CarTest {

    @Test
    public void shouldReturnNumberOfWheels() {
        // Given
        Car maluch = new Car("Maluch", 4);
        // When
        int numberOfWheels = maluch.getNumberOfWheels();
        // Then
        assertThat(numberOfWheels).isEqualTo(4);
    }

}
```

Cykl życia

Każdy z testów ma swój cykl życia. Podczas testowania można wywoływać specjalne metody:

- `@BeforeAll` - metoda wywołana przed wszystkimi testami
- `@BeforeEach` - metoda wywołana przed każdym testem
- `@AfterEach` - metoda wywołana po każdym teście
- `@AfterAll` - metoda wywołana po wszystkich testach

StandardTests.java

```
class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

Zadania

- Stwórzyć nowy projekt **Maven** o nazwie `test-example`
- Dodać nową zależność do biblioteki **JUnit**
- Dodać klasę `Car`:

Car.java

```
public class Car {

    //Pola
    int numberOfWheels = 4;
    String carName;

    //Konstruktor domyślny - nie trzeba go tworzyć gdy niema innego
    Car() {}

    //Konstruktor
    Car(int numberOfWheels, String carName) {
        this.numberOfWheels = numberOfWheels;
        this.carName = carName;
    }

    //Metody
    public int getNumberOfWheels() {
        return numberOfWheels;
    }

    public int getTotalWheelPrice(int wheelPrice) {
        return numberOfWheels * wheelPrice;
    }

}
```

- Napisać test dla klasy `Car` sprawdzający czy nowo utworzony samochód ma nazwę "maluch" (korzystając z asercji **JUnit**)
- Dodaj nową zależność do biblioteki **AssertJ**
- Napisać test dla klasy `Car` sprawdzający czy nowo utworzony samochód z 6 kołami dobrze wylicza cene za koła (korzystając z asercji **AssertJ**)
- Napisać test dla klasy `Car` sprawdzający czy nowo utworzony samochód ma domyślnie cztery koła (korzystając z asercji **AssertJ**)

Typy danych

Każda klasa (czyli odzwierciedlenie rzeczywistego bytu w kodzie) reprezentowana jest przez elementy składowe jak pola i metody. Każde pole oraz metoda może zwracać określony typ danych. Typy danych dzielą się na **typy proste** (prymitywne) i **typy złożone** (obiektowe).

Java jest językiem **silnie typowanym**! **Każdy** obiekt musi posiadać typ.

Typy proste (prymitywne)

Są to typy, które z góry mają określony rozmiar w pamięci. Typy proste mogą przechowywać wartości liczbowe (całkowite i zmiennoprzecinkowe, logiczne oraz znaki.

byte

byte jest typem danych przeznaczonym dla liczb całkowitych w zakresie od **-128 do 127**. W pamięci zajmuję **1 bajt**.

byte - zadania

- Utwórz nowy projekt **Maven** o nazwie `types-example`
- Dodaj zależności do biblioteki **JUnit** oraz **AssertJ**
- Stwórz test `PrimitiveTypesTest` dla klasy `PrimitiveTypes`
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `byteDefault` typu `byte`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `byteExample` typu `byte`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

short

short jest typem danych przeznaczonym dla liczb całkowitych w zakresie od **-32_768 do 32_767**. W pamięci zajmuję **2 bajty**.

short - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `shortDefault` typu `short`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `shortExample` typu `short`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

int

int jest typem danych przeznaczonym dla liczb całkowitych w zakresie od **-2^31 do 2^31-1**. W

pamięci zajmuję **4 bajty**.

int - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `intDefault` typu `int`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `intExample` typu `int`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

long

long jest typem danych przeznaczonym dla liczb całkowitych w zakresie od **-2^63 do 2^63-1**. Przy deklaracji pola typu **long** o wartości przekraczającej zakres **int** należy użyć postfix **'l'/'L'**. W pamięci zajmuję **8 bajtów**.

long - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `longDefault` typu `long`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością `10000000000000L`) pole `longExample` typu `long`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

float

Pamiętaj o 'f'/'F' przy deklaracji

float jest typem danych przeznaczonym dla wartości zmiennoprzecinkowych. Może przechowywać do 6-7 cyfr po przecinku. W pamięci zajmuję **4 bajty**.

float - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `floatDefault` typu `float`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `floatExample` typu `float`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

double

double jest **domyślnym** typem danych przeznaczonym dla wartości zmiennoprzecinkowych. Może on przechowywać więcej liczb w pamięci w stosunku do **float** (około 15 cyfr po przecinku). W pamięci zajmuję **8 bajtów**.

double - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `doubleDefault` typu `double`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `doubleExample` typu `double`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

boolean

boolean jest typem logicznym, który może przechowywać dwie wartości `true` lub `false`. Ciężko jednoznacznie określić jego rozmiar w pamięci.

boolean - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `booleanDefault` typu `boolean`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `booleanExample` typu `boolean`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

char

Do przechowywania pojedynczego znaku wykorzystujemy typ **char**. Przechowuje on wszystkie znaki Unicode. Wartość deklarujemy w 'a'. W pamięci zajmuję **2 bajty**.

char - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- W klasie `PrimitiveTypes` zadeklaruj (bez wartości) pole `charDefault` typu `char`
- W klasie `PrimitiveTypes` zadeklaruj (z wartością) pole `charExample` typu `char`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

Typy złożone (obiektowe)

Typy obiektowe (czyli klasy stworzone z typów prymitywnych) w przeciwieństwie do typów prostych nie mają określonego rozmiaru z góry. Ich rozmiar wyliczany jest na podstawie typów zawartych w środku. Przykładowo:

```
class Window {

    int width; // 4 bajty
    int heigth; // 4 bajty

} // rozmiar = 4 + 4 = 8 bajtów

class Car {

    int numberOfWheels; // 4 bajty
    Window leftWindow; // 8 bajtów
    Window rigthWindow; // 8 bajtów

} // rozmiar = 4 + 8 + 8 = 20 bajtów
```

Domyślną wartością typów **obiektowych** jest wartość `null`.

Typy złożone - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- Utwórz klasę `ObjectTypes`
- W klasie `ObjectTypes` zadeklaruj pole `stringExample` typu `String` z wartością `text`
- W klasie `ObjectTypes` zadeklaruj pole `stringNull` typu `String` bez wartości
- W klasie `ObjectTypes` zadeklaruj pole `stringNewExample` typu `String` z wartością `new String("text")`
- W klasie `ObjectTypes` zadeklaruj pole `integerExample` typu `Integer` z wartością `new Integer(1)`
- Utwórz klasę `ObjectTypesTest`
- W testach sprawdź wartości tych pól
- Jaka jest wartość domyślna?

Autoboxing

Autoboxing jest procesem automatycznej konwersji z typu prostego na obiektowy.

Autoboxing

- Wykorzystaj projekt stworzony w zadaniu byte
- Utwórz klasę `Autoboxing`
- W klasie `Autoboxing` zadeklaruj pole `autoboxingExample` typu `Integer` z wartością `1`
- Utwórz klasę `AutoboxingTest`
- W teście sprawdź wartość tego pola

Autounboxing

Autounboxing jest procesem automatycznej konwersji z typu obiektowy na prosty.

Autounboxing - zadania

- Wykorzystaj projekt stworzony w zadaniu byte
- Utwórz klasę `Autounboxing`
- W klasie `Autounboxing` zadeklaruj pole `autounboxingExample` typu `int` z wartością `new Integer(12)`
- Utwórz klasę `AutounboxingTest`
- W teście sprawdź wartość tego pola

Operatory

Operatory arytmetyczne

Operatory arytmetyczne służą do wykonywania prostych **działań arytmetycznych**.

- Stwórz nowy projekt **Maven** o nazwie `operators-example`
- Dodaj zależności do biblioteki **JUnit** oraz **AssertJ**
- Stwórz klasę `Calculator`
- Stwórz klasę testową `CalculatorTest` dla klasy `Calculator`

Dodawanie (+)

Do wykonania operacji **dodawania** wykorzystuje się operator `+`.

- Stwórz test oraz implementację dla operacji **dodawania** (`int add(int first, int second)`)

Odejmowanie (-)

Do wykonania operacji **odejmowania** wykorzystuje się operator `-`.

- Stwórz test oraz implementację dla operacji **odejmowania** (`int sub(int first, int second)`)

Mnożenie (*)

Do wykonania operacji **mnożenia** wykorzystuje się operator `*`.

- Stwórz test oraz implementację dla operacji **mnożenia** (`int mul(int first, int second)`)

Dzielenie całkowite (/)

Do wykonania operacji **dzielenia całkowitego** wykorzystuje się operator `/`.

- Stwórz test oraz implementację dla operacji **dzielenia** (`int div(int first, int second)`) dla cyfr `17` i `4`
- Stwórz test oraz implementację dla operacji **dzielenia** (`double div(double first, int second)`) dla cyfr `17.0` i `4`
- Stwórz klasę `Runner` z metodą `main`
- W metodzie `main` wywołaj metodę `div` dla cyfr `17` i `0`

Reszta z dzielenia (modulo) (%)

Do wyliczenia **reszty z dzielenia** wykorzystuje się operator `%` (modulo).

- Stwórz test oraz implementację dla operacji **modulo** (`int mod(int first, int second)`) dla cyfr `17` i `4`

Skrócona wersja operatorów arytmetycznych

Operatory arytmetyczne mogą wystąpić także w **wersji skróconej**. Pozwala to na osiągnięcie bardziej **zwięzłej** składni:

```
int x = 10;
x += 1; // x = x + 1
x -= 1; // x = x - 1
x *= 2; // x = x * 2
x /= 2; // x = x / 2
x %= 2; // x = x % 2
```

Inkrementacja i Dekrementacja

W **Javie** oprócz podstawowych **operatorów arytmetyczne** występują dwa dodatkowe, bardzo pomocne operatory. Są to operatory **inkrementacji** i **dekrementacji**.

Operatory te bardzo często wykorzystywane są w pętlach.

Inkrementacja

Operator **inkrementacji** `++` służy do **zwiększania** wartość zmiennej o **jeden**:

```
int x = 10;
x++; // 10 + 1 = 11
```

Operator ten może występować w **dwóch** wersjach:

- **prefixowej** - wartość **zwiększana** jest **przed** przypisaniem

```
int a = 10;
int b = ++a; // b = 11 a = 11 PREfix
```

- **postfixowej** - wartość **zwiększana** jest **po** przypisaniu

```
int x = 10;
int y = x++; // y = 10 x = 11 POSTfix
```

- Stwórz test oraz implementację dla operacji **inkrementacji** (`int increment(int numer)`)

Dekrementacja

Operator **dekrementacji** `--` służy do **zmniejszania** wartość zmiennej o **jeden**:

```
int x = 10;
x--; // 10 - 1 = 9
```

- Stwórz test oraz implementację dla operacji odejmowania o jeden (`int decrement(int numer)`)

Podobnie jak operator **inkrementacji** występuje on w dwóch wersjach, **prefixowej** i **postfixowej**.

Operatory porównawcze

Operatory porównawcze służą do **porównywania wartości**. Zwracają one w wyniku wartość logiczną `true` lub `false`.

Równy (==)

Operator `==` sprawdza czy dwa obiekty są **równe**. O dwóch równych obiektach mówimy wtedy, gdy mają one **wskaźnik** na tę samą **referencję** (miejsce w pamięci):

```
// Typy obiektowe
Car firstCar = new Car();
Car secondCar = firstCar;
Car thirdCar = new Car();

firstCar == secondCar // true
firstCar == thirdCar // false

// Typy prymitywne
int firstValue = 10;
int secondValue = 10;
int thirdValue = 20;

firstValue == secondValue // true
firstValue == thirdValue // false
```

- Stwórz klasę `Comparator`
- Stwórz pustą klasę `Car`
- Stwórz klasę testową `ComparatorTest` dla klasy `Comparator`
- Napisz test i implementację dla metody `boolean compare(int value, int valueToCompare)`
- Napisz test i implementację dla metody `boolean compare(Car car, Car carToCompare)` dla tej samej instancji
- Napisz test i implementację dla metody `boolean compare(Car car, Car carToCompare)` dla dwóch różnych instancji

Różny (!=)

Operator `!=` sprawdza czy dwa obiekty są **różne**. O dwóch różnych obiektach mówimy wtedy, gdy

nie mają one **wskaźników** na tę samą **referencję** (miejsce w pamięci):

```
// Typy obiektowe
Car firstCar = new Car();
Car secondCar = firstCar;
Car thirdCar = new Car();

firstCar != secondCar // false
firstCar != thirdCar // true

// Typy prymitywne
int firstValue = 10;
int secondValue = 10;
int thirdValue = 20;

firstValue != secondValue // false
firstValue != thirdValue // true
```

- W klasie testowej `ComparatorTest` dodaj test dla metody `boolean areDifferent(int value, int valueToCompare)`
- W klasie testowej `ComparatorTest` dodaj test dla metody `boolean areDifferent(Car car, Car carToCompare)` dla tej samej instancji
- W klasie testowej `ComparatorTest` dodaj test dla metody `boolean areDifferent(Car car, Car carToCompare)` dla dwóch różnych instancji

Mniejszy/Większy (<, >)

Operatory `>` i `<` sprawdzają czy wartość jest **mniejsza** lub **większa**:

```
10 > 20 // false
20 < 100 // true
```

- W klasie testowej `ComparatorTest` dodaj test dla metody `boolean isLower(int number, int numberToCompare)`
- W klasie testowej `ComparatorTest` dodaj test dla metody `boolean isGreater(int number, int numberToCompare)`

Większy/Mniejszy bądź równy (>=, ⇐)

Operatory `>=` i `⇐` sprawdzają czy wartość jest **mniejsza bądź równa** lub **większa bądź równa**.

```
5 <= 5 // true
10 >= 5 // true
```

Operatory logiczne

Operatory logiczne służą do **łączenia** warunków logicznych.

Koniunkcja (AND) (&&)

Pierwszym **operatorem logicznym** służącym do **łączenia** warunków logiczny jest operator **koniunkcji** `&&`. W poniższym przykładzie zaprezentowane mamy **dwa warunki logiczne**. Pierwszy z nich sprawdza czy dana liczba jest **mniejsza** od stu, a drugi czy jest **większa** od dziesięciu. Spróbujmy połączyć te dwa warunki logiczne, korzystając z operatora **koniunkcji**:

```
int x = 26;
x < 100 && x > 10
```

W dziedzinie nauki zwanej **logiką** występują **tablice prawdy**. Tablice prawdy reprezentują tabelaryczną formę **zero-jedynkowych** kombinacji **wartości logiczny** dla danych **operatorów**. Tablica prawdy dla operatora **koniunkcji**:

p	q	p and q
0	0	0
0	1	0
1	0	0
1	1	1

Gdzie:

- 0 - fałsz
- 1 - prawda

Sprawdźmy teraz powyższy przykład korzystając z **tablicy prawdy**:

```
int x = 26;
x < 100 && x > 10

// w logice 1 and 1 = 1
// w Javie true and true = true
```

Jako **pierwszy** z **warunków logicznych** warto umieszczać ten mniej złożony, ponieważ jeśli pierwszy z nich **nie zostanie spełniony** (zwróci `false`) to drugi z nich **nie będzie nawet sprawdzany**.

Alternatywa (OR) (||)

Operator **alternatywy** (lub) `||` ponowanie jak operator **koniunkcji** służy do **łączenia warunków**. Różnicą pomiędzy nimi jest to, że w przypadku **alternatywy** wystarczy, iż tylko **jeden warunek**

zostanie spełniony (przy **koniunkcji wszystkie warunki muszą być spełnione!**). Przykładowo interesuje nas liczba **większa** od pięciu **lub mniejsza** od trzech:

```
int x = 2;
x > 5 || x < 3
```

Tablica prawdy dla operatora **alternatywy**:

p	q	p or q
0	0	0
0	1	1
1	0	1
1	1	1

Gdzie:

- 0 - fałsz
- 1 - prawda

Sprawdźmy teraz powyższy przykład korzystając z **tablicy prawdy**:

```
int x = 2;
x > 5 || x < 3

// w logice 0 or 1 = 1
// w Javie false or true = true
```

Negacja (NOT) (!)

Operator negacji ! służy do **negowania** wartości. Zwraca on wartość **przeciwną** do aktualnej:

```
boolean sdaIsBest = false;
boolean really = !sdaIsBest; // true
```

Tablica prawdy dla operatora **negacji**:

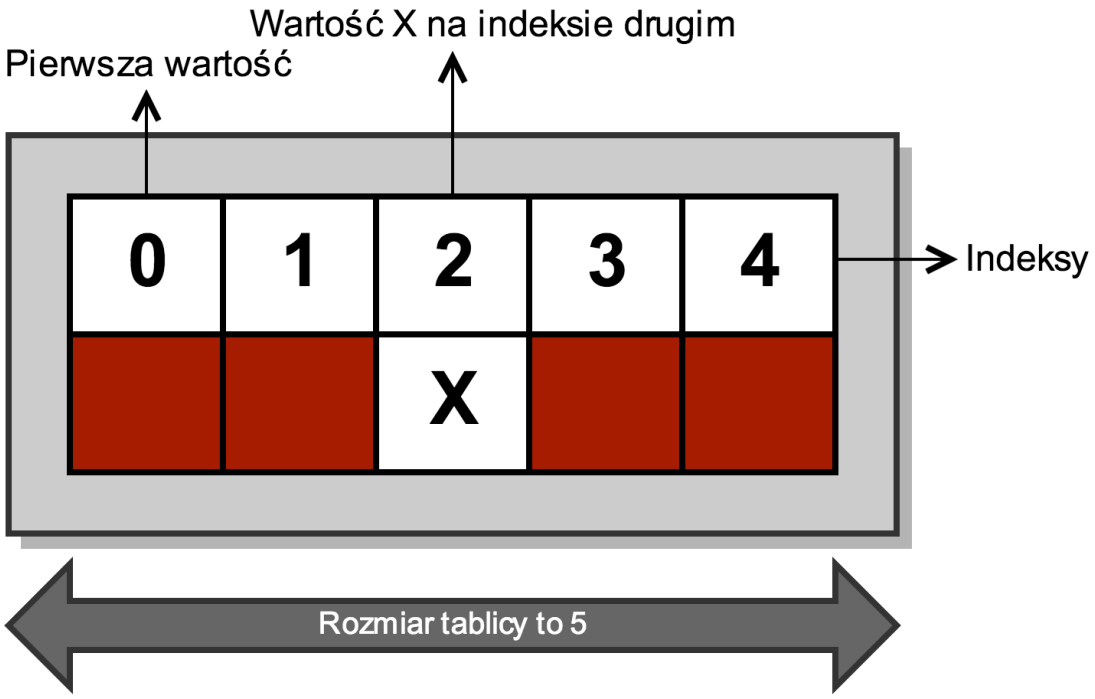
p	not(p)
0	1
1	0

Gdzie:

- 0 - fałsz
- 1 - prawda

Tablice

Tablica jest strukturą danych służącą do przechowywania kilku elementów w pamięci:



Tworzenie

Podczas tworzenia tablicy musimy z góry określić jej rozmiar. Związane jest to ze sposobem przechowywania tej struktury w pamięci:

```
int[] tabWithoutValues = new int[5]; //Rezerwacja obszaru pamięci, każdy element ma wartość 0

Car[] carsWithoutValues = new Car[5]; //Rezerwacja obszaru pamięci, każdy element ma wartość null

int[] tabWithValues = {1, 2, 3, 4, 5}; //Stworzenie tablicy z pięcioma elementami

int[] tabWithValuesSecondExample = new int[] {1, 2, 3}; //Stworzenie tablicy z trzema elementami
```

Odczyt

W **Javie** indeks zaczyna się od zera!

Odczytywanie odbywa się poprzez użycie indeksu:

```
int[] tabWithValues = {1, 2, 3, 4, 5}; //Stworzenie tablicy z pięcioma elementami

System.out.println(tabWithValues[0]);
System.out.println(tabWithValues[2]);
System.out.println(tabWithValues[4]);

// Wynik
1
3
5
```

Rozmiar

Aby sprawdzić rozmiar tablicy korzystamy z pola `length`:

```
int[] tab = new int[5];
tab.length; //Zwróci 5
```

Varargs

Czasem istnieje potrzeba stworzenia metody ze zmienną liczbą argumentów. Mechanizm realizujący to rozwiązanie nazywa się **varargs**:

Uwaga! parametr realizujący zmienną liczbę argumentów musi być na samym końcu deklaracji metody.

```
void someMethod(String ... strings) {
    // logika na wszystkich elementach
}
```

Wywołanie:

```
someInstance.someMethod("S");
someInstance.someMethod("S", "D");
someInstance.someMethod("S", "D", "A");
```

Zadania

- Utworzyć nowy projekt **Maven** o nazwie `arrays-example`
- Stworzyć klasę `ArrayExample`
- W klasie `ArrayExample` stworzyć tablicę liczb całkowitych z pięcioma elementami (na czwartej pozycji ustaw wartość 8)

- Stworzyć test dla klasy `ArrayExample` sprawdzający rozmiar tablicy
- Stworzyć test dla klasy `ArrayExample` sprawdzający czy element na pozycji czwartej to wartość 8
- W klasie `ArrayExample` zadeklarować tablicę (o nazwie `tabWithoutValues`) liczb całkowitych o rozmiarze pięć
- W klasie `ArrayExample` zadeklarować tablicę (o nazwie `stringsWithoutValues`) `String` o rozmiarze pięć
- Stworzyć test dla klasy `ArrayExample` sprawdzający element na indeksie 0 z tablicy `tabWithoutValues`
- Stworzyć test dla klasy `ArrayExample` sprawdzający element na indeksie 1 z tablicy `stringsWithoutValues`
- W klasie `ArrayExample` zadeklarować metodę `int manyArgs(String ⋯ strings)` zwracającą ilość przekazanych argumentów
- Stworzyć test dla metody `int manyArgs(String ⋯ strings)` sprawdzający ilość przekazanych argumentów dla `manyArgs()`
- Stworzyć test dla metody `int manyArgs(String ⋯ strings)` sprawdzający ilość przekazanych argumentów dla `manyArgs("S", "D", "A")`

Pętle (loops)

W poprzednim ćwiczeniu stworzyliśmy **tablicę z pięcioma elementami**. Wyobraźmy sobie tablicę w której mamy **1000 elementów**, które chcemy wypisać:

```
System.out.println(tab[0]);
System.out.println(tab[1]);
System.out.println(tab[2]);
...
System.out.println(tab[999]);
```

Musieliśmy powtórzyć powyższą linię **1000 razy**. Programowanie w taki sposób może być **męczące**. Rozwiązaniem tego problemu są **pętle**. Służą one do przeglądania (**iterowania**) zawartości tablicy lub kolekcji w łatwy sposób.

for

Pętla **for** służy do **iterowania** (bo zwyczajowo korzystamy z litery **i** dla **licznika**) po tablicy lub kolekcji (o kolekcjach w następnych rozdziałach):

```
for(wyrazenie-poczatkowe; warunek; modyfikator-licznika) {
    //Logika
}
```

W naszym przykładzie:

```
int[] tab = {1, 2, 3, 4, 5};

for(int i=0; i < tab.length; i++) {
    System.out.println("Wartość: " + tab[i]);
}

//Wynik:
Wartość: 1
Wartość: 2
Wartość: 3
Wartość: 4
Wartość: 5
```

while

Pętla **while** najczęściej wykorzystywana jest w miejscach gdzie zakładana ilość powtórzeń jest bliżej nieokreślona, ale znany jest warunek jaki musi być spełniony aby ją zakończyć:

```
while (warunek-logiczny) {
    //Logika
}
```

W naszym przykładzie

```
int[] tab = {1, 2, 3, 4, 5};
int counter = 0;

while (tab.length > counter) {
    System.out.println("Wartość: " + tab[counter]);
    counter++;
}

//Wynik:
Wartość: 1
Wartość: 2
Wartość: 3
Wartość: 4
Wartość: 5
```

do while

Pętla **do while** różni się tym od pętli **while**, że zostanie wywołana chociaż raz (warunek sprawdzany jest dopiero przy drugiej iteracji):

```
do {
    //Logika
}
while (warunek-logiczny);
```

W naszym przykładzie

```
int[] tab = {1, 2, 3, 4, 5};
int counter = 0;
do {
    System.out.println("Wartość: " + tab[counter]);
    counter++;
}
while (tab.length > counter);

//Wynik:
Wartość: 1
Wartość: 2
Wartość: 3
Wartość: 4
Wartość: 5
```

for each

Pętla **for each** jest bardzo użytecznym rodzajem pętli umożliwiającym **przeglądanie** tablicy lub kolekcji bez stosowania indeksów:

```
for (typ-wartości wartość : tablica) {
    //Logika
}
```

W naszym przykładzie:

```
int[] tab = {1, 2, 3, 4, 5};

for (int value : tab) {
    System.out.println("Wartość: " + value);
}

//Wynik:
Wartość: 1
Wartość: 2
Wartość: 3
Wartość: 4
Wartość: 5
```

Rekurencja

Ostatnim opisywanym sposobem przetwarzania danych jest **rekurencja**. Rekurencja jest techniką, w której metoda odwołuje się do samej "siebie":

```
int printer(int value) {
    System.out.println("Value: " + value);
    if (value == 10) {
        return 10;
    }
    return printer(++value);
}
```

W przypadku pętli oraz rekurencji należy pamiętać o warunku końcowym!

Zadania

- Utworzyć nowy projekt **Maven** o nazwie `loops-example`
- Stworzyć klasę `LoopExample`
- Stworzyć metodę `int[] fillFor(int value)` zwracającą tablicę wypełnioną wartościami do wartości zmiennej `value`
- Stworzyć klasę testową dla klasy `LoopExample` sprawdzający metodę `int[] fillFor(int value)`
- Stworzyć metodę `int[] fillWhile(int value)` zwracającą tablicę wypełnioną wartościami do wartości zmiennej `value`
- Stworzyć metodę testową dla klasy `LoopExample` sprawdzający metodę `int[] fillWhile(int value)`
- Stworzyć metodę `int[] fillDoWhile(int[] tab)` zwracającą tablicę ze zwiększonymi wartościami o jeden

```
int[] tab = {1, 2};
// Wynik
2, 3
```

- Stworzyć metodę testową dla klasy `LoopExample` sprawdzający metodę `int[] fillDoWhile(int[] tab)`
- Stworzyć metodę `void range(int start, int end)` wypisującą wartości z podanego przedziału wykorzystując rekurencję
- Stworzyć klasę `Runner` z `psvm`
- W metodzie `main` wywołać metodę `range` z liczbami `10` i `20`, wynik sprawdzić na konsoli

Literał (String)

Ten typ danych pojawił się już w poprzednich ćwieczeniach. Jednakże, jest to najpopularniejszy **obiektowy typ danych**, dlatego też jest mu poświęcony osobny dział. Służy on do przechowywania literałów (czyli po prostu tekstów). Wszystkie literały łańcuchowe przez to, iż są kosztowne w tworzeniu i utrzymywaniu, przechowywane są w specjalnym miejscu w pamięci zwanym **String Pool**. **String** jest typem, który jest **niemutowalny**. Oznacza to, iż każda modyfikacja powoduje powstanie nowego literału!

Tworzenie

String jest typem obiektowy dlatego też możemy tworzyć jego nową instancję poprzez użycie słowa kluczowego `new`:

```
String imie = "Krzysztof";
String drugieImie = "Krzysztof"; // to samo miejsce w pamięci co zmienna imie
String nazwisko = new String("Chruściel"); // nowy obszar pamięci
```

Konkatenacja

Operacja łączenia literałów łańcuchowych nazywana jest **konkatenacją**. **Konkatenacja** odbywa się poprzez znak `+`. **String** jest typem **niemutowalnym**, dlatego podczas operacji **konkatenacji** należy pamiętać, iż za każdym razem tworzony jest nowy obiekt:

```
"Krzysztof" + " " + "Chruściel" - > "Krzysztof Chruściel"
```

W momencie **konkatenacji** typu `String` z innym typem, wywoływana jest metoda `toString()`!

Konkatenacja - zadania

- Stworzyć klasę `StringExample`
- Stworzyć metodę `String concat(String first, String second)`, która w wyniku zwraca złączony `String`
- Stworzyć test w klasie `StringExampleTest` sprawdzający metodę `String concat(String first, String second)`

Metody

Zważając na to, iż typ **String** jest najpopularniejszym typem obiektowy posiada on szereg użytecznych metod. Poniżej znajduję się tylko część najczęściej używanych metod.

valueOf

Metoda `valueOf` zamienia podaną wartość na typ `String`:

```
String.valueOf(2.0f) -> "2.0"
String.valueOf(true) -> "true"
```

valueOf - zadania

- Napisz test w którym przetestujesz metodę `valueOf()`

trim

Metoda `trim` usuwa białe znaki z początku i końca literału:

```
"   zdanie z białym znakami " -> "zdanie z białymi znakami"
```

trim - zadania

- Napisz test w którym przetestujesz metodę `trim()`

toUpperCase

Metoda `toUpperCase` służy do zwiększania wszystkich znaków na podstawie literału:

```
"małe litery" -> "MAŁE LITERY"
```

toUpperCase - zadania

- Napisz test w którym przetestujesz metodę `toUpperCase()`

toLowerCase

Metoda `toLowerCase` służy do zmniejszania wszystkich znaków na podstawie literału:

```
"MAŁE LITERY" -> "małe litery"
```

- Napisz test w którym przetestujesz metodę `toLowerCase()`

toCharArray

Metoda `toCharArray` służy do tworzenia tablicy znaków (`char`) na podstawie literału:

```
"tablica" -> tab[] -> [0] = t [1] = a [2] = b [3] = l [4] = i [5] = c [6] = a
```

toCharArray - zadania

- Napisz test w którym przetestujesz metodę `toCharArray()` dla słowa "tablica"
- W teście sprawdź rozmiar zwróconej tablicy
- W teście sprawdź czy na indeksie 3 znajduję się znak `l`

substring

Metoda `substring` służy do wycinania innego literału na podstawie istniejącego literału:

```
String stringToSubstring = "first1second2third";
String after = stringToSubstring.substring(5);

System.out.println(after);

// Wynik
1second2third
```

substring - zadania

- Napisz test w którym przetestujesz metodę `substring()`
- Napisz test w którym przetestujesz metodę `substring(from, to)`

replace

Metoda `replace` zamienia znak na znak wybrany przez nas:

```
String stringToReplace = "first1second2third";
String after = stringToReplace.replace("first", "second");

System.out.println(after);

// Wynik
second1second2third
```

replace - zadania

- Napisz test w którym przetestujesz metodę `replace()`

length

Metoda `length` służy do obliczania długości literału:

```
"długość".length() -> 7 znaków
```

length - zadania

- Napisz test w którym przetestujesz metodę `length()`

indexOf

Metoda `indexOf` zwraca numer indeksu pierwszego wystąpienia znaku lub sekwencji znaków:

```
"first1second2third".indexOf('i') -> 1
```

indexOf - zadania

- Napisz test w którym przetestujesz metodę `indexOf()`

lastIndexOf

Metoda `lastIndexOf` zwraca numer indeksu ostatniego wystąpienia znaku lub sekwencji znaków:

```
"first1second2third".lastIndexOf('i') -> 15
```

lastIndexOf - zadania

- Napisz test w którym przetestujesz metodę `lastIndexOf()`

isEmpty

Metoda `isEmpty` służy do sprawdzenia czy dany literał jest pusty:

```
"" -> true | "niepusty" -> false
```

isEmpty - zadania

- Napisz test w którym przetestujesz metodę `isEmpty()` dla słowa ""
- Napisz test w którym przetestujesz metodę `isEmpty()` dla słowa "niepusty"

endsWith

Metoda `endsWith` służy do sprawdzenia czy dany literał kończy się zadaną frazą:

```
String janusz = "Janusz";
janusz.endsWith("sz"); // true
janusz.endsWith("jan"); // false
```

endsWith - zadania

- Napisz test w którym przetestujesz metodę `endsWith()`

contains

Metoda `contains` służy do sprawdzenia czy dany literał zawiera inny:

```
String janusz = "Janusz";
janusz.contains("Jan"); // true
janusz.contains("grazyna"); // false
```

contains - zadania

- Napisz test w którym przetestujesz metodę `contains()` dla słowa "SDA"
- W teście sprawdź czy dla `contains("A")` zwrócony wynik to `true`
- W teście sprawdź czy dla `contains("C")` zwrócony wynik to `false`

charAt

Metoda `charAt` służy do wycinania znaku (`char`) na podanym indeksie na podstawie istniejącego literału:

```
"charAt".charAt(3) -> 'r'
```

charAt - zadania

- Napisz test w którym przetestujesz metodę `charAt()` dla słowa "charAt"
- W teście sprawdź czy dla `charAt(3)` zwrócony znak to `r`

Object

Java realizuje paradygmat programowania obiektowego (dokładniejsze wyjaśnienie w dziale OOP). Każdy byt w języku **Java** jest obiektem, oznacza to, iż dziedziczy on **niejawnie** po klasie `Object`. Z klasy `Object` otrzymujemy kilka przydanych metod.

toString

Metoda `toString` służy do wyświetlania obiektu jako `String`. Domyślna implementacja zwraca:

```
nazwa.pakietu.NazwaKlasy@Hex(hashCode)

// Wynik
pl.sda.Runner@4554617c
```

hashCode

Metoda `hashCode` zwraca unikalny kod (nazywany hash codem), który w jednoznaczny sposób identyfikuje obiekt. Domyślna implementacja zależna jest od **JVM**:

> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

```
System.out.println(new Runner().hashCode());

// Wynik
1163157884
```

equals

Metoda `equals` służy do porównywania dwóch obiektów. Domyślna implementacja metody `equals` (sprawdza **referencje**, czyli czy obiekty zajmują te same miejsce w pamięci):

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

getClass

Metoda `getClass` zwraca nazwę klasy wraz z jej pakietem:

```
class pl.sda.Runner
```

wait, notify i notifyAll

Są to metody związane z wielowątkowością. Na tym poziomie nie będą one wykorzystywane.

clone

Metoda `clone` służy do klonowania (tworzenia jego kopii) obiektu. Domyślna implementacja jest pusta i rzuca wyjątek `CloneNotSupportedException`.

finalize

Metoda `finalize` wywoływana jest w momencie usuwania obiektu z pamięci (obiekt usuwany jest z pamięci poprzez mechanizm **GC**, więcej w bloku *Wprowadzenie do technologii JVM*). Domyślna implementacja jest pusta:

```
protected void finalize() throws Throwable {

}
```

Zadania

- Utworzyć nowy projekt **Maven** o nazwie `object-example`
- Stworzyć klasę `ObjectExample`
- Sprawdź jakie ma dostępne metody (2 x `Ctrl`+`F12`)
- Utworzyć klasę `Runner` z `psvm`
- W klasie `Runner` w metodzie `main` utworzyć instancję `ObjectExample`
- Z instancji `ObjectExample` wypisać domyślną implementację metody `getClass`
- Z instancji `ObjectExample` wypisać domyślną implementację metody `hashCode`
- Z instancji `ObjectExample` wypisać domyślną implementację metody `toString`
- Stworzyć klasę `ToStringObjectExample`
- W klasie `ToStringObjectExample` dodać dwa pola typu `int` o nazwach `first` i `second` z wartościami 5 i 10
- Nadpisz metodę `toString` korzystając z **IntelliJ**
- W klasie `Runner` w metodzie `main` utworzyć instancję `ToStringObjectExample`

- Z instancji `ToStringObjectExample` wypisać nadpisaną implementację metody `toString`

equals i hashCode

O metody `equals` i `hashCode` lubią pytać na rozmowie kwalifikacyjnej!

equals

Metoda `equals` służy do sprawdzenia czy dwa obiekty są takie same. Standardowy operator porównania w **Javie** `==` sprawdza czy obiektu znajdują się w tym samym miejscu w pamięci, a nie czy są takie same (choć wyglądają tak samo):

Runner.java

```
Car maluch = new Car(4, "Maluch");
Car maluchSecond = new Car(4, "Maluch");
if (maluch == maluchSecond) { // zwróci false
    // logika
}
```

Rozwiązanie tego problemu jest porównywanie obiektów korzystając z metody `equals`:

Runner.java

```
Car maluch = new Car(4, "Maluch");
Car maluchSecond = new Car(4, "Maluch");
if (maluch.equals(maluchSecond)) { // zwróci true, zwróci false w domyślnej
implementacji
    // logika
}
```

W kontekście nadpisywania metody `equals` musi ona spełniać **kilka warunków**:

- Powinna być **zwrotna**

Oznacza to, że dla porównania `x.equals(x)` zawsze powinna zwracać `true`

- Powinna być **symetryczna**

Oznacza to, że dla porównania `x.equals(y)` i `y.equals(x)` zawsze powinna zwracać `true`

- Powinna być **przechodnia**

Oznacza to, że jeśli mamy trzy obiekty `x`, `y` i `z` to dla porównania `x.equals(y)`, `y.equals(z)` również `x.equals(z)` jest prawdą.

- Powinna być **spójna**

Oznacza to, że dla wielokrotnego wywołania zawsze zwraca ten sam wynik (jeśli nie było żadnych zmian na obiektach)

- Powinna zwracać `false` przy porównaniu z `null`

Oznacza to, że dla porównania `x.equals(null)` zawsze powinna zwracać `false`

Metodę `equals` można nadpisać korzystając z **IDE** `alt+insert`

Nadpisana metoda `equals`:

Car.java

```
@Override
public boolean equals(Object o) {
    if (this == o) { // jeśli ten sam obiekt to true
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        // jeśli obiekt jest null lub z innej klasy to false
        return false;
    }
    Car car = (Car) o;
    return numberOfWheels == car.numberOfWheels && // sprawdzane są wszystkie pola
            Objects.equals(carName, car.carName);
}
```

hashCode

Metoda `hashCode` służy do generowania funkcji skrótu dla obiektu na podstawie pól, aby w jednoznaczny sposób określić unikalność obiektu. `hashCode` wykorzystywany jest najczęściej w kolekcjach (o kolekcjach w następnych rozdziałach), które do sortowania i umieszania obiektów używają funkcji skrótu:

Runner.java

```
Car maluch = new Car(4, "Maluch");
Car maluchSecond = new Car(4, "Maluch");
if (maluch.hashCode() == maluchSecond.hashCode()) { // zwróci false, ponieważ domyślna
implementacja hashCode zwraca hex(adres-w-pamięci)
    // logika
}
```

Metodę `hashCode` można nadpisać korzystając z **IDE** `alt+insert`

Nadpisana metoda `hashCode` "pod spodem" wykorzystyuje najczęściej mnożenie przez liczby pierwsze:

Car.java

```
@Override
public int hashCode() { // wersja z IDE
    return Objects.hash(numberOfWheels, carName);
}

@Override
public int hashCode() { // wersja "ręczna"
    return 31 * numberOfWheels + 7 * carName;
}
```

Kontrakt

Pomiedzy metodami `equals` i `hashCode` istnieje pewien kontrakt:

- Jeśli wartość metody `hashCode` jest taka sama `x.hashCode() == y.hashCode()` to nie jest wymagane aby `x.equals(y)` zwracało `true`
- Jeśli metoda `equals` zwraca `true` dla `x.equals(y)` to metoda `hashCode` również musi zwrócić `true` dla `x.hashCode() == y.hashCode()`
- Wielokrotne wywołanie metody `hashCode` na tym samym obiekcie (który nie był modyfikowany pomiędzy wywołaniami) musi zwrócić tę samą wartość

Zadania

- Utworzyć nowy projekt **Maven** o nazwie `equals-hashcode-example`
- Stworzyć klasę `PhoneEqualsExample` z dwoma polami `String name` i `int phoneNumber` ustawianymi w konstruktorze
- Stworzyć klasę `PhoneHashCodeExample` z dwoma polami `String name` i `int phoneNumber` ustawianymi w konstruktorze
- Stworzyć klasę `PhoneContractExample` z dwoma polami `String name` i `int phoneNumber` ustawianymi w konstruktorze
- W klasie `PhoneEqualsExample` wygenerować metodę `equals`
- Stworzyć klasę testową dla `PhoneEqualsExample` i przetestować metodę `equals`
- W klasie `PhoneHashCodeExample` wygenerować metodę `hashCode`
- Stworzyć klasę testową dla `PhoneHashCodeExample` i przetestować metodę `hashCode`
- W klasie `PhoneContractExample` wygenerować metodę `equals` i `hashCode`
- Stworzyć klasę testową dla `PhoneContractExample` i przetestować metodę `hashCode` i `equals`

Instrukcje warunkowe

Do tej pory kod, który uruchamialiśmy kończył się w jednym określony miejscu (poprzedzonym słowem kluczowym `return`). Jednakże bardzo często logika naszej aplikacji zależy od jakiś warunków. Przykładowo w metodzie chcemy sprawdzić czy podana liczba jest większa od dziesięć i w obu przypadkach wypisać inny napis lub zwrócić inny wynik. Możemy to zrealizować wykorzystując **instrukcje warunkowe**.

if

Instrukcja warunkowa `if` sprawdza dowolny warunek logiczny (warunki logiczne zwracają wartość `boolean`). Jeśli podany warunek jest **spełniony**, wykonuje się kod zawarty w `{ }`:

```
if (liczbaDoSprawdzenia == 10) {
    System.out.println("Podałeś liczbę 10!");
}
```

if, else

Instrukcja warunkowa `else`, a tak naprawdę kod w niej zawarty `{ }` wywoływany jest w momencie, gdy warunek logiczny `if` **nie został** spełniony:

```
if (liczbaDoSprawdzenia == 10) {
    System.out.println("Podałeś liczbę 10!");
} else {
    System.out.println("Nie podałeś liczby 10 :(");
}
```

if, else if, else

Konstrukcja `else if` służy do łączenia instrukcji warunkowych i działa jak połączenie `if` i `else`:

```
if (liczbaDoSprawdzenia == 1) {
    System.out.println("Podałeś liczbę 1!");
} else if (liczbaDoSprawdzenia == 10) {
    System.out.println("Podałeś liczbę 10!");
} else if (liczbaDoSprawdzenia == 100) {
    System.out.println("Podałeś liczbę 100!");
} else if (liczbaDoSprawdzenia === 1000) {
    System.out.println("Podałeś liczbę 1000!");
} else {
    System.out.println("Nie jest to 1, 10, 100, 1000 :(");
}
```

switch

Kolejnym typem instrukcji warunkowej jest konstrukcja `switch`. Wartość zmiennej wejściowej musi być już znana podczas kompilacji. Dlatego też, nie możemy wykorzystywać zmiennych czy wartości zwracanych z metod. Dostępne typy to:

- `byte` i `Byte`
- `short` i `Short`
- `char` i `Character`
- `int` i `Integer`
- `Enum`
- `String`

Instrukcja `break` kończy działanie `switch`

```
int liczbaDoSprawdzenia = 10;
switch(liczbaDoSprawdzenia) {
    case 1:
        System.out.println("Podałeś liczbę 1!");
        break;
    case 10:
        System.out.println("Podałeś liczbę 10!");
        break;
    case 100:
        System.out.println("Podałeś liczbę 100!");
        break;
    case 1000:
        System.out.println("Podałeś liczbę 1000!");
        break;
    default:
        System.out.println("Nie jest to 1, 10, 100, 1000 :(");
}
```

Sprawdź co się stanie, gdy usuniesz `break`

Zadania

- Utwórz nowy projekt **Maven** o nazwie `condition-example`
- Dodaj zależności do biblioteki **JUnit** oraz **AssertJ**
- Stwórz test `ConditionTest` dla klasy `Condition`
- Stwórz implementację i test dla metody `boolean isEven(int number)`, która sprawdzi czy podana liczba jest parzysta (w wyniku ma zwracać `true` lub `false`)
- Stwórz implementację i test dla metody `boolean isOdd(int number)`, która sprawdzi czy podana liczba jest nieparzysta (w wyniku ma zwracać `true` lub `false`)

- Stwórz implementację i test dla metody `int divisible(int number)`, która sprawdzi czy podana liczba jest podzielna przez 2, 5, 7 (w wyniku ma zwracać tę liczbę przez którą udało się podzielić bez reszty, w przeciwnym wypadku zwróć zero):

```
divisible(4); // wynik 2
divisible(15); // wynik 5
divisible(49); // wynik 7
divisible(1); // wynik 0
```

- Stwórz implementację i test dla metody `String getMonthNameBy(int number)` (korzystając z instrukcji `switch`), która zwraca nazwę miesiąca dla podanego numeru (pamiętaj, aby sprawdzić **corner case**)
- Stwórz implementację `void getMonthNamesBy(int number)` (korzystając z instrukcji `switch`), która wypisze nazwy miesięcy dla podanego numeru do końca roku (sprawdź to dodając klasę `Runner` z `psvm`, tam stwórz instancję klasy `Condition` i wywołaj metodę `getMonthNamesBy`)

```
getMonthNamesBy(10);

// październik
// listopad
// grudzień
```

Elementy statyczne i finalne

Elementy statyczne

Słowem kluczowym `static` oznaczamy elementy statyczne. Elementy statyczne **należą do klasy**, a nie do ich instancji. Oznacza to, iż możemy ich używać bez potrzeby tworzenia ich instancji. Oznaczenie elementu jako statyczny zależy od miejsca użycia.

Elementy statyczne - klasa

Klasy statyczne mogą być tworzone tylko jako klasy wewnętrzne:

ClassWithStaticClass.java

```
class ClassWithStaticClass {

    static class SomeStaticClass {
        int someField = 10;
    }

}
```

Runner.java

```
public class Runner {

    public static void main(String[] args) {
        int field = ClassWithStaticClass.SomeStaticClass().someField;
    }

}
```

Elementy statyczne - metoda

Metody statyczne służą do przechowywania wspólnej logiki, która nie jest zależna od stanu klasu. W metodach statycznych można odwoływać się tylko do innych statycznych elementów klasy:

ClassWithStaticMethod.java

```
class ClassWithStaticMethod {

    int value = 20;

    static int someMethod() {
        return 42;
    }

    static int someMethod() {
        return value + 42; // compliation error
    }

}
```

Runner.java

```
public class Runner {

    public static void main(String[] args) {
        int value = ClassWithStaticMethod.someMethod();
    }

}
```

Elementy statyczne - pole

Pole statyczne ma taką samą wartość we wszystkich instancjach. Jeśli pole statyczne zostanie zmienione, będzie ono dostępne jako zmienione dla wszystkich instancji. Pola statyczne najczęściej wykorzystywane są dla niezmienników wykorzystywanych w instancjach klasy. Może to być wartość liczby PI, która jest niezależna od instancji. Zmienne tego typu są bardziej optymalne w pamięci, ponieważ przez to, iż należą do klasy trzymane są tylko w jednym miejscu:

ClassWithStaticField.java

```
class ClassWithStaticField {

    static int staticValue = 10;

}
```

Runner.java

```
public class Runner {

    public static void main(String[] args) {
        int value = ClassWithStaticField.staticValue;
    }

}
```

Elementy statyczne - import static

W dziale elementy języka poznaliśmy słowo kluczowe `import`, które służy do wskazania lokalizacji klasy z której chcemy skorzystać. W **Javie** 1.5 pojawiło się nowe pojęcie jakim są **statyczne importy**. Umożliwiają one korzystanie ze **statycznych elementów** bez podawania nazwy klasy, z której pochodzą:

ClassWithStaticFields.java

```
class ClassWithStaticElements {

    static int staticValue = 10;

    static void assertThat(Object object) {
        // some logic
    }

}
```

Runner.java

```
import static ClassWithStaticElements.*;

public class Runner {

    public static void main(String[] args) {
        int value = staticValue;
        assertThat(value);
    }

}
```

Elementy finalne

Słowem kluczowym `final` oznaczamy elementy jako **finalne** ("ostateczne"). Użycia słowa `final` na klasie, metodzie, polu, zmiennej i parametrze ma inne znaczenie.

Elementy finalne - klasa

Słowo kluczowe `final` na **klasie** oznacza, iż nie można po tej klasie dziedziczyć:

FinalClass.java

```
final class FinalClass {

}
```

TryFinalClass.java

```
class TryFinalClass extends FinalClass { // compliation error

}
```

Elementy finalne - metoda

Słowo kluczowe `final` na **metodzie** oznacza, iż nie można tej metody nadpisać:

FinalMethod.java

```
final class FinalMethod {

    final void method() {
        // do nothing
    }

}
```

TryFinalMethod.java

```
class TryFinalMethod extends FinalMethod {

    @Override
    void method() { // compliation error
        // do nothing
    }

}
```

Elementy finalne - pole

Słowo kluczowe `final` na **polu** oznacza, iż w momencie tworzenia instancji należy określić wartość. Ponadto, po ustawieniu wartości nie można jej już zmieniać:

FinalField.java

```
final class FinalField {

    final int value = 10;
    final int secondValue;
    final int thirdValue; // compliation error

    FinalClass (int secondValue) {
        this.secondValue = secondValue;
    }

}
```

Elementy finalne - zmienna

Słowo kluczowe `final` na **zmiennej** oznacza, iż wartość można przypisać tylko raz:

FinalVariable.java

```
final class FinalVariable {

    void someMethod() {
        final int value = 20;
        value = 30; // compliation error
    }

}
```

Elementy finalne - parametr

Słowo kluczowe `final` na **parametrze** oznacza, iż nie można zmieniać wartości przekazanego parametru

FinalParameter.java

```
final class FinalParameter {

    void someMethod(final int value) {
        value = 20; // compliation error
    }

}
```

Elementy statyczne i finalne - stałe

Według konwencji stałe piszemy DRUKOWANYMI_LITERAMI

Elementy, które są `static` i `final` jednocześnie określane są mianem stałych:

Errors.java

```
public class Errors {

    public static final int INTERNAL_SERVER_ERROR = 500;
    public static final int SERVICE_UNAVAILABLE = 503;

}
```

Zadania

- Utwórzyć nowy projekt **Maven** o nazwie `static-final-example`
- Dodaj zależności do biblioteki **JUnit** oraz **AssertJ**
- Stwórz klasę finalną i spróbuj po niej podziedziczyć
- Stwórz metodę finalną i spróbuj ją nadpisać
- Stwórz pole finalne z wartością i spróbuj je zmienić w dowolnej metodzie
- Stwórz metodę z finalnym parametrem i spróbuj zmienić jego wartość w metodzie
- Stwórz metodę z finalną zmienną i spróbuj zmienić jej wartość
- Stwórz klasę `MonthConstants`
- W klasie `MonthConstants` dodaj nazwy miesięcy jako stałe
- W klasie `MonthConstants` stwórz statyczną metodę `static String getMonthNameBy(int number)`, która zwróci nazwę miesiąca (odpowiednią stała) dla podanego numeru
- Stworzyć test dla klasy `MonthConstants` sprawdzający czy zwrócona wartość to `"Maj"` dla cyfry `5`
- Stworzyć klasę `Author` z treścią:

```
class Author {
    private String firstName;
    private String lastName;
    private String city;
    private int age;

    private Author(final AuthorBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.city = builder.city;
        this.age = builder.age;
    }

    static class AuthorBuilder {
        private final String firstName;
        private String lastName;
        private String city;
        private int age;

        public AuthorBuilder (String firstName) {
            this.firstName = firstName;
        }

        public AuthorBuilder lastName(String lastName) {
            this.lastName = lastName;
            return this;
        }

        public AuthorBuilder city(String city) {
            this.city = city;
            return this;
        }

        public AuthorBuilder age(int age) {
            this.age = age;
            return this;
        }

        public Author build() {
            return new Author(this);
        }
    }
}
```

- Stwórz klasę `Runner` z `psvm`
- W klasie `Runner` wywołaj metodę klasy statycznej `AuthorBuilder` i **zbuduj** autora.

Interfejs

Interfejs służy do stworzenia **kontraktu**. W **kontrakcie** tym definiujemy jakie metody są dostępne. Ich **implementacja** znajduje się już w konkretnych klasach **implementujących** ten **interfejs**.

Tworzenie

Do tworzenia interfejsów używamy słowa kluczowego `interface`. Tak samo jak klasy, **interfejsy** tworzymy w nowych plikach:

MusicPlayer.java

```
interface MusicPlayer {

    void playSong(String songName);

}
```

Implementowanie

Nie można stworzyć nowej instancji interfejsu, jednakże można stworzyć nową instancję klasy **implementującej** ten interfejs. Gdy zdecydujemy się **zaimplementować** dany interefejs, musimy nadpisać wszystkie jego metody. Aby **zaimplementować** interfejs korzystamy ze słowa kluczowego `implements`:

MP3Player.java

```
class MP3Player implements MusicPlayer {

    @Override
    void playSong(String songName) {
        System.out.println("Play MP3: " + songName);
    }

}
```

WawPlayer.java

```
class WawPlayer implements MusicPlayer {

    @Override
    void playSong(String songName) {
        System.out.println("Play WAW: " + songName);
    }

}
```

Metody domyślne

W obrębie **interfejsu** można zadeklarować metody domyślne wykorzystując słowo kluczowe `default`. Metoda domyślna posiada implementację:

Pojawiły się dopiero w **Javie** 8!

MusicPlayer.java

```
interface MusicPlayer {

    void playSong(String songName);

    default String playerName() {
        return "Music";
    }

}
```

Dziedziczenie

Interfejsy mogą **dziedziczyć** zachowania innych interfejsów korzystając ze słowa kluczowego `extends`.

MusicPlayer.java

```
interface MusicPlayer {

    void playSong(String songName);

}
```

VideoPlayer.java

```
interface VideoPlayer {

    void playVideo(String videoName);

}
```

Player.java

```
interface Player extends MusicPlayer, VideoPlayer {

}
```

YoutubePlayer.java

```
class YoutubePlayer implements Player {

    @Override
    void playSong(String songName) {
        System.out.println("Play MP3: " + songName);
    }

    @Override
    void playVideo(String videoName) {
        System.out.println("Play video: " + videoName);
    }

}
```

Stałe w interfejsie

Z racji, iż nie można utworzyć instancji samego interfejsu, nie można w nim utworzyć zwykłych pól, tylko stałe. Wszystkie deklaracje są automatycznie oznaczane jako `public final static`, dlatego nie musimy dodawać tych właściwości:

Player.java

```
interface Player extends MusicPlayer, VideoPlayer {

    int field = 10; // same as public final static int field = 10;

}
```

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `interface-example`
- Utwórz interfejs `Vehicle.java` z metodą `void drive()`
- Utwórz interfejs `Payable.java` z metodą `void pay(int quantity)`
- Utwórz klasę `Bus.java` ze stałą o wartości `3.20`
- W klasie `Bus.java` zaimplementuj interfejsy `Vehicle.java` i `Payable.java`. W nadpisanej metodzie `void drive()` wypisz "Drive by bus", a w metodzie `void pay(int quantity)` wypisz wyliczoną cenę za bilety
- Utwórz klasę `Train.java` ze stałą o wartości `25.50`
- W klasie `Train.java` zaimplementuj interfejsy `Vehicle.java` i `Payable.java`. W nadpisanej metodzie `void drive()` wypisz "Drive by train", a w metodzie `void pay(int quantity)` wypisz wyliczoną cenę za bilety
- Utwórz klasę `Car.java`, która implementuje interfejs `Vehicle.java` i w nadpisanej metodzie wypisz "Drive by car"

- Utwórz klasę `Person.java`
- W klasie `Person.java` utwórz metodę `void driveBy(Vehicle vehicle)`, która wywoła odpowiednią metodę z interfejsu podanego w parametrze
- W klasie `Person.java` utwórz metodę `void buyTicketsFor(Payable payable, int quantity)`, która wywoła odpowiednią metodę z interfejsu podanego w parametrze
- Utwórz klasę `Runner.java`, w której stworzysz instancje obiektów:
  - `Person`
  - `Car`
  - `Bus`
  - `Train`
- Na utworzonej instancji klasy `Person` wywołaj kilka razy metody `driveBy` i `buyTicketsFor` z różnymi parametrami
- Uruchom klasę `Runner` z `psvm`
- Stwórz nowy pakiet `programmers`
- Utwórz interfejs `JavaProgrammer.java` z metodą `void typeJava()`
- Utwórz interfejs `TableSoccerPlayer.java` z metodą `void playTableSoccer()`
- Utwórz interfejs `AwesomeProgrammer.java` z metodą `void drinkCoffe()`, który dziedziczy po interfejsach `JavaProgrammer` i `TableSoccerPlayer`
- Utwórz klasę `Programmer.java`, która implementuje interfejs `AwesomeProgrammer`
- Utwórz klasę `Runner.java`, w której stworzysz instancję obiektu `Programmer`
- Na stworzonej instancji wywołaj metody:
  - `typeJava`
  - `playTableSoccer`
  - `drinkCoffe`

Klasa abstrakcyjna

Klasa **abstrakcyjna** jest specyficznym rodzajem klasy, którego nie można stworzyć wprost. Jest ona uogólnieniem, nie jest możliwe utworzenie instancji tej klasy, ponieważ sama w sobie jest zbyt ogólna (abstrakcyjna). Aby stworzyć klasę abstrakcyjną korzystamy ze słowa kluczowego `abstract`:

SomeClass.java

```
abstract class SomeClass {

}
```

SomeConcreteClass.java

```
class SomeConcreteClass extends SomeClass {

}
```

Metoda abstrakcyjna

Tylko w obrębie **klas abstrakcyjnych** można tworzyć **metody abstrakcyjne**. Podobnie jak w interfejsie są one tylko definicjami, które zostaną wypełnione w konkretnych implementacjach. Musimy nadpisać wszystkie metody abstrakcyjne:

SomeClass.java

```
abstract class SomeClass {

    abstract void someAbstractMethod();

    void someNormalMethod() {
        System.out.println("Method");
    }

}
```

SomeConcreteClass.java

```
class SomeConcreteClass extends SomeClass {

    @Override
    void someAbstractMethod() {
        // logic
    }

}
```

Klasa abstrakcyjna vs interfejs

Uwaga! o te **różnice** często pytają na rozmowach kwalifikacyjnych!

Musimy pamiętać, iż w **Javie** można dziedziczyć po jednej klasie, ale można implementować wiele interfejsów. Ponadto do **Javy** 7 włącznie nie można było tworzyć implementacji metod w interfejsach (to było główną różnicą). Od **Javy** 8 różnice te mocno się zacierają, ponieważ pojawiły się metody domyślne (`default`).

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `abstract-example`
- Utworzyć klasę abstrakcyjną `Drink` z metodami abstrakcyjnymi:
  - `abstract void showName()`
  - `abstract void addWater()`
  - `abstract void addAlcohol()`
  - `abstract void addJuice()`
  - `abstract void addIce()`
- W klasie `Drink` tworzymy metodę `void prepareDrink()`, w której wywołane będą wszystkie metody abstrakcyjne
- Utworzyć klasę `Mohito`, która dziedziczy po klasie `Drink`
- Utworzyć klasę `Malibu`, która dziedziczy po klasie `Drink`
- Utworzyć klasę `SexOnTheBeach`, która dziedziczy po klasie `Drink`
- Utwórz klasę `Runner.java` z `psvm`, w której stworzysz instancje:
  - `Mohito`
  - `Malibu`
  - `SexOnTheBeach`
- Na każdej stworzonej instancji wywołaj metodę `prepareDrink`

Enum

Enumerator to typ wyliczeniowy. Tworzymy go podobnie jak klasę w osobnym pliku, korzystając ze słowa kluczowego `enum`

Shape.java

```
enum Shape {

}
```

Służy on do definiowania elementów wyliczeniowych. Zazwyczaj są to wartości w skończonym zakresie jak dni tygodnia, miesiące czy rozmiary koszulek. Wartości enumeratora piszemy drukowanymi literami (jest to pewien rodzaj stałych) i odzielane są przecinkami:

Shape.java

```
enum Shape {

    RECTANGLE,
    CIRCLE,
    TRIANGLE

}
```

Wywołanie

Ponieważ wartości enumeratora są pewnego rodzaju stałymi, to nie tworzymy nowej instancji enumeratora:

```
void someMethod(Shape shapeToPrint) {

    if(Shape.RECTANGLE.equals(shapeToPrint)) {
        //logic
    }

}
```

Pola w enumeratorze

Każdy **enumerator** oprócz wartości wyliczeniowych może przechowywać dodatkowe atrybuty w polach:

Shape.java

```
enum Shape {

    RECTANGLE(4),
    CIRCLE(0),
    TRIANGLE(3);

    private int vertex;

    Shape(int vertex) {
        this.vertex = vertex;
    }

    int getVertex() {
        return vertex;
    }

}
```

```
void someMethod(int vertex) {

    if(Shape.RECTANGLE.getVertex() == vertex) {
        //logic
    }

}
```

Iteracja

Enumerator dostarcza metodę `values()`, która zwraca tablicę ze wszystkimi wartościami:

```
void someMethod() {

    for (Shape shape : Shape.values()) {
        // logic
    }

}
```

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `enum-example`
- Utwórz enumerator `WeekDay.java`, który zawiera wszystkie dni tygodnia
- Utwórz klasę `DayPrinter.java` z metodą `void printDayBy(WeekDay weekDay)`, która wypisze

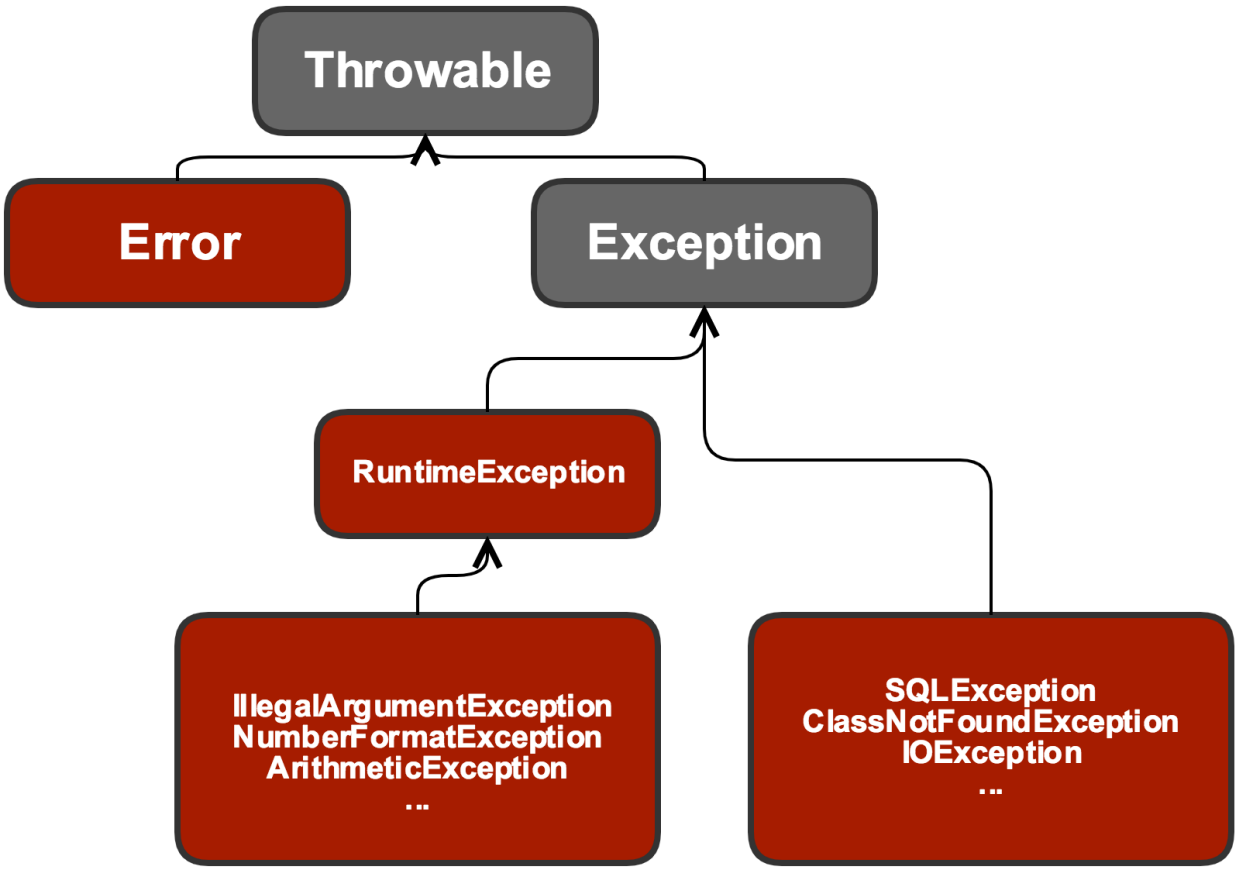
(skorzystaj z pętli `switch`):

  - dla `MONDAY` - "Loops"
  - dla `TUESDAY` - "Arrays"
  - dla `WEDNESDAY` - "Enums"
  - dla `THURSDAY` - "Classes"
  - dla `FRIDAY` - "Beer"
  - dla `SATURDAY` - "REST"
  - dla `SUNDAY` - "Java"

- Utwórz enumerator `Month.java`, który posiada dwa pola `private String monthName` i `private int monthNumber` ustawiane w konstruktorze
- W enumeratorze `Month.java` dodaj metodę `String getMonthBy(int monthNumber)`, która zwróci nazwę miesiąca (skorzystaj z pętli `for-each`) na podstawie podanego numeru
- Utwórz klasę `Runner.java` z `psvm`, w której stworzysz instancję `DayPrinter` i sprawdź metodę `printDayBy`
- W klasie `Runner` wywołaj metodę `getMonthBy` z enumeratora `Month`

Wyjątki

Wyjątki służą do obsługi sytuacji wyjątkowych. Wszystkie wyjątki dziedziczą po klasie `Throwable`:



Checked exceptions

Są to wyjątki, które trzeba obsłużyć (nie dziedziczą po `RuntimeException` lub po `Error`):

CheckedException.java

```
class CheckedException extends Exception {

}
```

Unchecked exceptions

Są to wyjątki, których nie trzeba obsługiwać (dziedziczą po `RuntimeException` lub po `Error`):

UncheckedException.java

```
class UncheckedException extends RuntimeException {

}
```

Rzucanie wyjątków (throw new Wyjątek())

Jeśli chcemy "rzucić" wyjątek używamy słowa kluczowego `throw` (możemy rzucić wyjątek **checked** i **unchecked**):

SomeClass.java

```
class SomeClass {

    void methodWhichThrowUncheckedException(int value) {
        if (value == 0) {
            throw new IllegalArgumentException();
        }
    }

    void methodWhichThrowCheckedException(File file) throws IOException {
        if (file == null) {
            throw new IOException();
        }
    }

}
```

Obsługa wyjątków (try/catch/finally)

Blocki catch i finally są opcjonalne, ale przynajmniej jeden z nich musi wystąpić.

Jeśli chcemy obsłużyć wyjątki musimy posłużyć się konstrukcją `try` oraz `catch`. W sekcji `catch` definiujemy jakie wyjątki chcemy obsłużyć. Możemy obsłużyć kilka wyjatków, jednakże należy pamiętać o hierarchii "od szczegółu do ogółu". Od **Javy** 7 bloki `catch` dla różnych wyjątków można łączyć za pomocą operatora `|`.

```
void someMethod() {
    try {
        someMethodWhichThrowException();
    } catch (Exception e) {
        // obsługa
    }
}
```

Dodatkowo możemy dodać blok `finally`, który wykona się niezależnie od tego, czy wyjątek wystąpi, czy nie:

```
void someMethod() {
    try {
        someMethodWhichThrowException();
    } catch (Exception e) {
        // obsługa
    } finally {
        System.out.println("Zawsze się wypiszę!");
    }
}
```

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `exceptions-example`
- Utwórz klasę `Runner.java`
- W klasie `Runner.java` dodaj punkt startowy
- Stwórz klasę `ExceptionExamples`
- Stwórz klasę `CheckedException`, która dziedziczy po `Exception`
- Stwórz klasę `UncheckedException`, która dziedziczy po `RuntimeException`
- W klasie `ExceptionExamples` dodaj metodę `void throwCheckedExample()`, w której rzucisz wyjątkiem `CheckedException`
- W klasie `ExceptionExamples` dodaj metodę `void throwUncheckedExample()`, w której rzucisz wyjątkiem `UncheckedException`
- W klasie `ExceptionExamples` dodaj metodę `catchExample`, w której wywołasz metodę `throwCheckedExample`
- Obsłuż wyjątek korzystając z bloku `try` oraz `catch`
- Dodaj blok `finally`, w którym wypiszesz "Finally"
- W klasie `Runner.java` uruchom metodę `catchExample`

JavaDoc

JavaDoc jest funkcjonalnością **Javy** służącą do opisu naszych klas. Oczywiście musimy starać się nazywać nasze klasy, pola i metoda w jak najlepszy sposób, jednakże czasem chcemy zawrzeć w opisie więcej informacji. Do tego wykorzystujemy **JavaDoc**.

Warto dokumentować publiczne metody!

Tworzenie

Informacje **JavaDoc** można umieścić na dowolnym elemencie:

Runner.java

```
/**
 *
 * This is a very awesome class which runs the whole world!
 *
 */
public class Runner {

    /**
     * This is a very awesome field.
     */
    public int publicField = 10;

    /**
     *
     * This is a very awesome method which runs the whole world!
     *
     */
    public void newPublicMethod(int value) {

    }

    /**
     *
     * This is a very awesome method which runs the whole world!
     *
     */
    public void oldPublicMethod() {

    }

    private void somePrivateMethod() {

    }

}
```

Dyrektywy

Dyrektywy w **JavaDoc** dostarczają dodatkowych informacji dla czytelników.

@author

Dyrektywa `@author` informuje o tym kto jest autorem danego elementu:

Runner.java

```
/**
 *
 * This is a very awesome class which runs the whole world!
 *
 * @author krzysztof.chrusciel
 */
public class Runner {

    // reszta klasy

}
```

@version

Dyrektywa `@version` informuje o numerze wersji danego elementu:

Runner.java

```
/**
 *
 * This is a very awesome class which runs the whole world!
 *
 * @author krzysztof.chrusciel
 * @version 1.0.1
 */
public class Runner {

    // reszta klasy

}
```

@since

Dyrektywa `@since` informuje od jakiej wersji element występuje:

Runner.java

```
/**
 *
 * This is a very awesome class which runs the whole world!
 *
 * @author krzysztof.chrusciel
 * @version 1.0.1
 * @since 1.0.0
 */
public class Runner {

    // reszta klasy

}
```

@depracted

Dyrektywa `@deprecated` informuje o tym, iż nie powinno używać się danego elementu, ponieważ istnieje nowsze rozwiązanie. Najczęściej w opisie tej adnotacji znajduje się informacja o najnowszej implementacji:

```
/**
 *
 * This is a very awesome method which runs the whole world!
 *
 */
public void newPublicMethod() {

}

/**
 *
 * This is a very awesome method which runs the whole world!
 * @deprecated please use newPublicMethod
 */
public void oldPublicMethod() throws IOException {
    throw new IOException();

}
```

@param

Dyrektywa `@param` informuje o parametrach w metodzie:

```
/**
 *
 * This is a very awesome method which runs the whole world!
 *
 * @param value the value for something
 */
public void newPublicMethod(int value) {

}
```

@throws

Dyrektywa `@throws` informuje o rzucanych wyjątkach w metodzie:

```
/**
 *
 * This is a very awesome method which runs the whole world!
 *
 * @throws IOException when read file
 */
public void oldPublicMethod() throws IOException {
    throw new IOException();

}
```

@link

Dyrektywa `@link` pozwala wskazać na element jako link:

```
/**
 *
 * This is a very awesome method which take {@link String} object.
 *
 * @param value the {@link String} value for something
 */
public void newPublicMethod(String value) {

}
```

@see

Dyrektywa `@see` informuje, gdzie możemy znaleźć więcej informacji:

```
/**
 *
 * This is a very awesome method which runs the whole world!
 *
 * @deprecated this method is deprecated
 * @see {@link Runner#newPublicMethod(String)}
 */
public void oldPublicMethod() throws IOException {
    throw new IOException();

}
```

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `java-doc-example`
- Poniższą klasę uzpełnić w **JavaDoc** (postaraj się wykorzystać wszystkie elementy):

Runner.java

```
public class Runner {

    public int publicField = 10;


    public void newPublicMethod(String value) {

    }

    public void oldPublicMethod() throws IOException {
        throw new IOException();

    }

    private void somePrivateMethod() {

    }

}
```

Adnotacje

Adnotacje pojawiły się w **Javie 1.5**. Służą one do przekazywania **dodatkowych informacji** (metadanych) na temat kodu. **Adnotacja** jest specjalnym rodzajem **interfejsu** (definicja znajduje się osobny pliku).

Tworzenie

Definicja własnej adnotacji:

NazwaAdnotacji.java

```
@Retention( RetentionPolicy.SOURCE )// Jak długo dane o adnotacji mają być
przechowywane
@Target( { ElementType.FIELD } ) //Ograniczenie gdzie możemy stosować adnotację.
public @interface NazwaAdnotacji {

    String parametr(); //Możemy definiować parametry
    String opcjonalnyParamter() default  "Value";

}
```

Używanie

Umieszczanie **adnotacji**:

AnnotationExample.java

```
class AnnotationExample {

    @NazwaAdnotacji(opcjonalnyParamter = "zmienna")
    private int value = 0;

}
```

Z **adnotacji** można korzystać wykorzystując **mechanizm refleksji** lub **programowania aspektowego**.

Przykładowe adnotacje wbudowane w język:

- `@Override` - wykorzystywana przez kompilator aby sprawdzić czy rzeczywiście istnieje taka metodą w nadklasie
- `@NotNull` - wykorzystywana przez **IDE** do oznaczania parametrów, które nie mogą przyjmować wartości `null`

Zasięg (target)

Adnotacje mają określony zasięg na którym mogą być używane. Zasięgi adnotacji:

- adnotacja - `ElementType.ANNOTATION_TYPE`
- konstruktor - `ElementType.CONSTRUCTOR`
- pole klasy - `ElementType.FIELD`
- zmienna lokalna - `ElementType.LOCAL_VARIABLE`
- metoda - `ElementType.METHOD`
- pakiet - `ElementType.PACKAGE`
- parametr metody - `ElementType.PARAMETER`
- klasa - `ElementType.TYPE`

Retencja

Retencja jest wartością określającą jak długo dane o adnotacji mają być przechowywane.

- `RetentionPolicy.CLASS` - umieszczane w skompilowanej klasie (wykorzystywna do modyfikacji byte kodu. Jest to domyślna retencja)
- `RetentionPolicy.RUNTIME` - dostępne w trakcie działania programu (refleksja)
- `RetentionPolicy.SOURCE` - usuwane przez kompilator w trakcie kompilacji

Zadania

Przesłanianie (@Override)

- Stwórz nowy projekt **Maven** o nazwie `annotation-example`
- Utwórz nową klase `Annotation`
- Nadpisz metodę `equals`
- Sprawdź czy pojawiła się adnotacja `@Override`
- Utwórz nową klase `Parent`
- W klasie `Parent` dodaj nową metodę:

```
void removeOverrideAnnotation() {

}
```

- Dodaj dziedziczenie klasie `Annotation` z `Parent`
- Nadpisz metodę `removeOverrideAnnotation` w klasie `Annotation`
- Usuń metodę `removeOverrideAnnotation` z klasy `Parent`

- Sprawdź czy pojawił się błąd

Tworzenie adnotacji

- Stwórz nową adnotację `Author`
- Adnotację `Author` można umieścić tylko na metodzie
- Adnotacja `Author` dostępna jest w runtime
- Adnotacja powinna posiadać dwa parametry `name` i `surname` typu `String`
- Adnotacja powinna posiadać domyślne wartości dla paramteru `name` i `surname`

Wyszukiwanie adnotacji

- Stwórzyć klasę `AnnotationExample`
- Dodać kilka metod (inwencja po twojej stronie ;))
- Na każdej (oprócz jednej) z metod umieścić adnotację z ćwiczenia tworzenie adnotacji
- Stwórz klasę `Runner` z metodą `main` (`psvm` w **IntelliJ**)
- W metodzie `main` wklej poniższą zawartość:

Runner.java

```
import java.lang.reflect.Method;

class Runner {

    public static void main(String[] args) {
        for(Method method : AnnotationExample.class.getDeclaredMethods()) {
            if (method.isAnnotationPresent(Author.class)) {
                System.out.println(method);
            }
        }

    }

}
```

- Uruchom aplikację i sprawdź wynik

Czas (LocalTime)

Obsługa czasu w okresie gdy nie było **Javy** w wersji 8 była bardzo trudna. Ponadto dotychczasowe API było mocno ograniczone. Od wersji 8 pojawiła się nowa klasa reprezentująca czas `LocalTime`.

Pamiętaj o ustawieniu **Javy** 8 w IDE!

Tworzenie

Do tworzenia czasu istnieje kilka wybranych metod. Pierwsza z nich pobiera aktualny czas korzystając z metody `now`:

```
LocalTime.now();
```

Metoda `parse` służy do parsowania (zmiany) czasu w formacie `String` na czas:

```
LocalTime time = LocalTime.parse("12:30:00");
```

Metoda `of` służy do tworzenia czasu z ich składowych:

```
LocalTime time = LocalTime.of(12, 30, 0); // 12:30:00
```

Zmiana czasu

Dodawanie czasu jest możliwe dzięki kilku metodą pomocniczym jak `plus` czy bardziej specyfikowanej metodzie `plusHours`:

```
time.plus(1, ChronoUnit.HOURS);
time.plusHours(1);
```

Analogicznie do dodawnia czasu, w taki sam sposób się go odejmuje. Wykorzystujemy do tego metody takie jak `minus` czy bardziej specyfikowaną metodę `minusHours`

```
time.minusHours(1);
time.minus(1, ChronoUnit.HOURS);
```

Elementy składowe

Jeśli chcemy pobrać **elementy składowe** czasu jak godziny, minuty czy sekundy możemy do tego wybrać gotowe metody:

```
time.getHour();
time.getMinute();
time.getSecond();
```

Sprawdzanie czasu

W nowym API, sprawdzanie czasu zostało bardzo ułatwione. Otrzymaliśmy dwie bardzo przydatne metody do sprawdzania czy dany czas jest przed lub po:

```
time.isAfter(timeToCompare);
time.isBefore(timeToCompare);
```

Zakres czasu

Zakres czasu korzysta ze standardu `ISO-8601`

Jeśli chcemy pobrać zakres czasu pomiędzy dwoma czasami wykorzystujemy klasę `Duration`:

```
Duration.between(firstTime, secondTime);
```

Zadania

- Stwórz nowy projekt **Maven** o nazwie `time-example`
- Stwórz klasę `Runner` z metodą `main` (`psvm` w **IntelliJ**)
- W metodzie `main`:
  - stwórz i wypisz czas korzystając z metody `now`
  - stwórz i wypisz czas korzystając z metody `of`
  - stwórz i wypisz czas korzystając z metody `parse`
  - pobierz aktualny czas i dodaj godzinę a następnie wypisz nowy czas
  - pobierz aktualny czas i odejmij 10 minut a następnie wypisz nowy czas
  - pobierz aktualny czas i wypisz godziny, minut i sekundy
  - pobierz aktualny czas i dodaj godzinę a następnie wypisz wynik metody `isAfter` porównując z aktualnym czasem
  - pobierz aktualny czas i dodaj godzinę a następnie wypisz wynik metody `isBefore` porównując z aktualnym czasem
  - stwórz dwa czasy a następnie wypisz zakres czasu w sekundach

Data (LocalDate)

Podobnie jak z czasem obsługa daty w okresie gdy nie było **Javy** w wersji 8 była bardzo trudna. Ponadto dotychczasowe API było mocno ograniczone. Od wersji 8 pojawiła się nowa klasa reprezentująca datę `LocalDate`.

Pamiętaj o ustawieniu **Javy** 8 w IDE!

Tworzenie

Do tworzenia daty istnieje kilka wybranych metod. Pierwsza z nich pobiera aktualną datę korzystając z metody `now`:

```
LocalDate.now();
```

Metoda `parse` służy do parsowania (zmiany) daty w formacie `String` na datę:

```
LocalDate date = LocalDate.parse("2018-12-12");
```

Metoda `of` służy do tworzenia daty z ich składowych:

```
LocalDate date = LocalDate.of(2018, 12, 12);
```

Zmiana daty

Dodawanie dat jest możliwe dzięki kilku metodą pomocniczym jak `plus` czy bardziej specyfikowanej metodzie `plusDays`:

```
date.plus(1, ChronoUnit.DAYS);
date.plusDays(1);
```

Analogicznie do dodawnia dat, w taki sam sposób się je odejmuje. Wykorzystujemy do tego metody takie jak `minus` czy bardziej specyfikowaną metodę `minusDays`

```
date.minusDays(1);
date.minus(1, ChronoUnit.DAYS);
```

Elementy składowe

Jeśli chcemy pobrać **elementy składowe** daty jak rok, miesiąc czy dzień możemy do tego wybrać gotowe metody:

```
date.getYear();
date.getMonthValue();
date.getDayOfMonth();
```

Sprawdzanie daty

W nowym API, sprawdzanie daty zostało bardzo ułatwione. Otrzymaliśmy dwie bardzo przydatne metody do sprawdzania czy dana data jest przed lub po:

```
date.isAfter(dateToCompare);
date.isBefore(dateToCompare);
```

Zakres dat

Zakres dat korzysta ze standardu `ISO-8601`

Jeśli chcemy pobrać zakres dat pomiędzy dwoma datami wykorzystujemy klasę `Period`:

```
Period.between(firstTime, secondTime);
```

Zadania

- Stwórz nowy projekt **Maven** o nazwie `date-example`
- Stwórz klasę `Runner` z metodą `main` (`psvm` w **IntelliJ**)
- W metodzie `main`:
  - stwórz i wypisz datę korzystając z metody `now`
  - stwórz i wypisz datę korzystając z metody `of`
  - stwórz i wypisz datę korzystając z metody `parse`
  - pobierz aktualną datę i dodaj jeden dzień a następnie wypisz nową datę
  - pobierz aktualną datę i odejmij jeden miesiąc a następnie wypisz nową datę
  - pobierz aktualną datę i wypisz dzień miesiąca, miesiąc i rok
  - pobierz aktualną datę i dodaj jeden dzień a następnie wypisz wynik metody `isAfter` porównując z aktualną datą
  - pobierz aktualną datę i dodaj jeden dzień a następnie wypisz wynik metody `isBefore` porównując z aktualną datą
  - stwórz dwie daty a następnie wypisz zakres dat w dniach

Data i czas

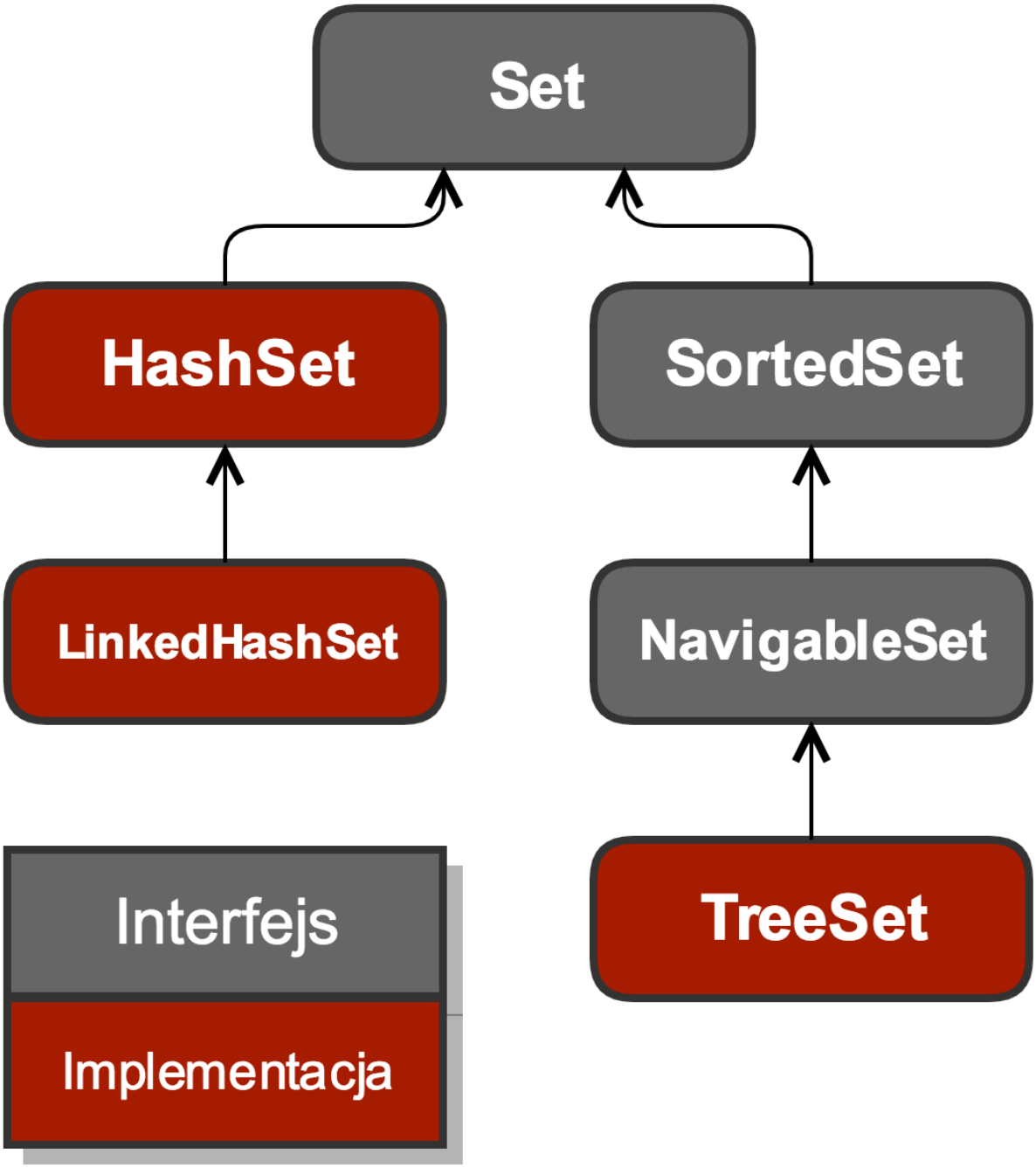
Jeśli chcemy pobrać jednocześnie datę i czas korzystamy z klasy `LocalDateTime`, która jest kombinacją `LocalDate` i `LocalTime`.

Kolekcje (Collections)

Do tej pory poznaliśmy jeden sposób do przechowywania **zbioru elementów** jakim były tablice. Niestety największą **wadą tablic jest potrzeba określenia jej rozmiaru z góry**. Wyobraźmy sobie system, w którym dynamicznie przybywają nam nowi użytkownicy. Jeśli przechowywalibyśmy ich w tablic to niestety co jakiś czas musielibyśmy **tworzyć nową**. Ponadto, należało by pamiętać ostatnio użyty indeks. To utrudnione wykorzystywanie tablic doprowadziło do postawnia frameworku **Java Collections**. Framework ten dostarcza nam **kolekcje**, które są implementacją podstawowych struktur danych. Struktury te są zarządzane w bardzo łatwy i przyjemny sposób.

Zbiory (Set)

Zbiór jest strukturą danych, która nie pozwala na przechowywania **duplikatów**. W standardowej implementacji nie zachowuje **kolejności** wstawiania. Najpopularniejsze implementacje to:



Wspólne metody

Interfejs `Set` udostępnia sporo przydatnych metod:

- `size` - zwraca ilość elementów w zbiorze
- `isEmpty` - sprawdza czy zbiór jest pusty
- `add` - dodaje nowy element do zbioru
- `addAll` - dodaje wszystkie elementy do zbioru
- `contains` - sprawdza czy zbiór zawiera element
- `remove` - usuwa element ze zbioru

HashSet

Jest zbiorem, który wykorzystuje **funkcje skrótu** do umieszczania nowych elementów (nie **gwarantuje** zachowania kolejności):

Obiekty przechowywane kolekcjach Hash* powinny mieć nadpisaną metodę `equals` i `hashCode` (więcej tutaj)

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("1");
hashSet.add("3");
hashSet.add("3");
hashSet.add("2");
hashSet.add("Janusz");
hashSet.add("2");
hashSet.add("5");
hashSet.add("5");
System.out.println(hashSet); // [1, 2, 3, Janusz, 5]
```

`HashSet` oferuje stałą szybkość działania O(1).

LinkedHashSet

Jest **zbiorem**, który wykorzystuje funkcje skrótu do umieszczania nowych elementów oraz zachowuje **kolejność** wstawiania:

```
LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("1");
linkedHashSet.add("3");
linkedHashSet.add("3");
linkedHashSet.add("2");
linkedHashSet.add("2");
linkedHashSet.add("5");
linkedHashSet.add("5");
System.out.println(linkedHashSet); // [1, 3, 2, 5]
```

TreeSet

Jest **zbiorem**, który w momencie wstawiania nowych elementów **sortuje** je według kolejności naturalnej lub według własnego komparatora:

`TreeSet` nie może przechowywać wartości `null`

```
TreeSet<String> hashTree = new TreeSet<>();
hashTree.add("1");
hashTree.add("3");
hashTree.add("2");
hashTree.add("5");
System.out.println(hashTree); // [1, 2, 3, 5]
```

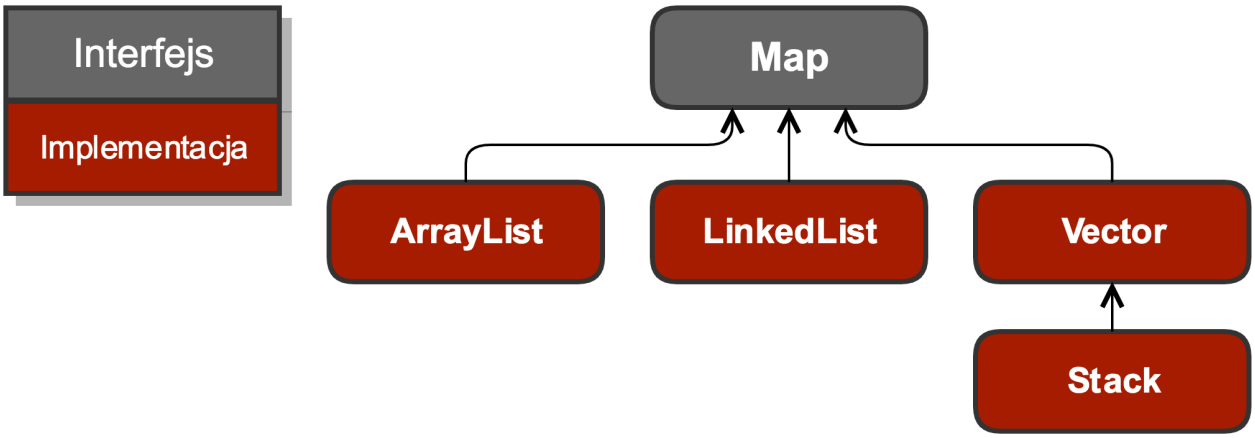
Szybkość działania kolekcji `TreeSet` określa się mianem O(log(n)).

Zadania

- Stwórz nowy projekt **Maven** o nazwie `collection-example`
- Stwórz klasę `Car` z jednym polem `final int value` ustawianym w konstruktorze
- Stwórz klasę testową `HashSetTest`, w której przetestujesz metodę:
  - `add` (spróbuj dodać kilka takich samych instancji klasy `Car`)
  - nadpisz `equals` i `hashCode` w klasie `Car`
  - przetesuj ponowanie metodę `add`

Listy (List)

Lista jest strukturą danych, w której można przechowywać duplikaty. Kolejność wstawianych elementów jest zachowania.



Wspólne metody

Interfejs `List` udostępnia sporo przydatnych metod:

- `size` - zwraca ilość elementów w liście
- `isEmpty` - sprawdza czy lista jest pusta
- `add` - dodaje nowy element do listy
- `addAll` - dodaje wszystkie elementy do listy

- `contains` - sprawdza czy lista zawiera element
- `get` - pobiera element o określonym indeksie z listy

ArrayList

Jest **listą**, która "pod spodem" wykorzystuje tablice. Odczyt danych odbywa się w czasie O(1) natomiast wstawianie elementów w czasie O(n):

```
ArrayList<String> arrayList = new ArrayList<>();
```

LinkedList

Jest **listą**, która "pod spodem" implementuję listę dwukierunkową. Odczyt danych odbywa się w czasie O(n) natomiast wstawianie elementów w czasie O(1):

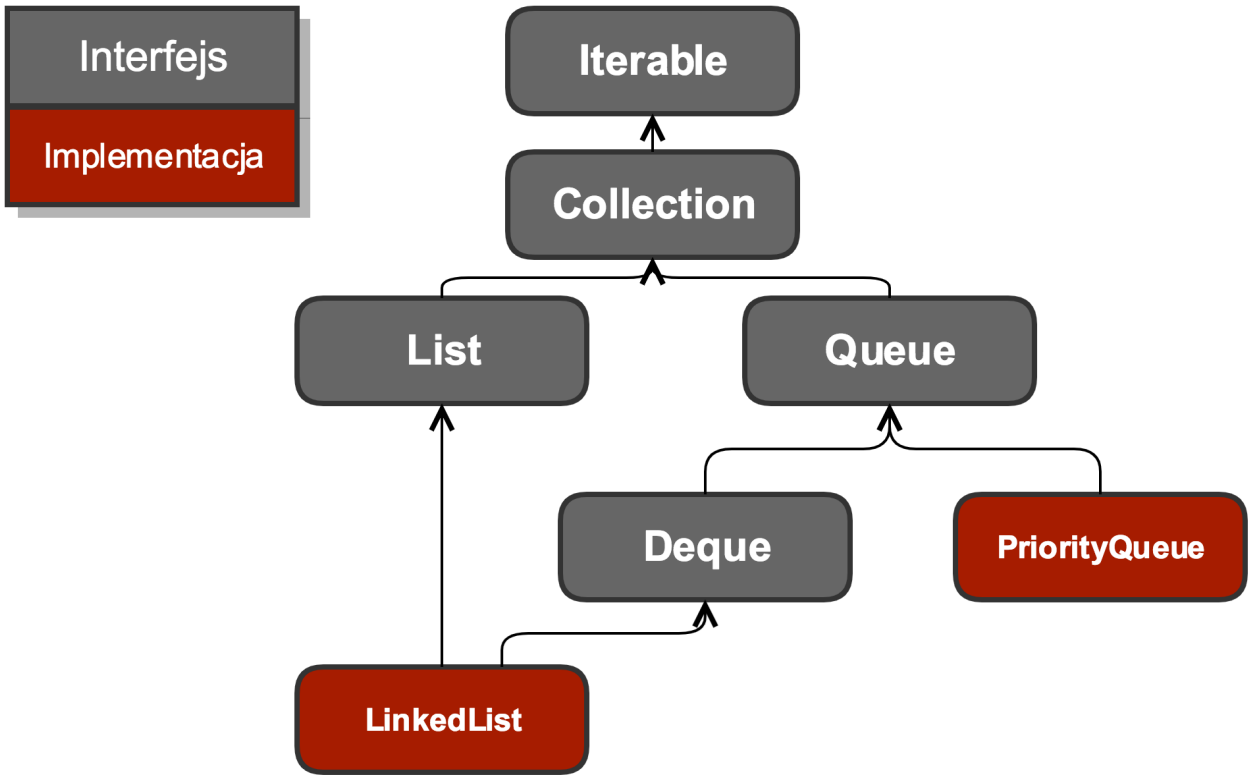
```
LinkedList<String> linkedList = new LinkedList<>();
```

Zadania

- Wykorzystaj projekt stworzony w zadanich ze zbiorami
- Stwórz klasę testową `ArrayListTest`, w której przetestujesz metody:
  - `size`
  - `isEmpty`
  - `add`
  - `addAll`
  - `contains`
  - `get`

Kolejki (Queue)

Kolejka jest kolejną **strukturą danych** zaimplementowaną we frameworku **Collections**:



Wspólne metody

Interfejs `Queue` udostępnia sporo przydatnych metod:

- `size` - zwraca ilość elementów w kolejce
- `isEmpty` - sprawdza czy kolejka jest pusta
- `add/offer` - dodaje nowy element do kolejki
- `clear` - czyści kolejkę
- `contains` - sprawdza czy kolejka zawiera element
- `peek` - pobiera pierwszy element z kolejki
- `poll` - pobiera i usuwa pierwszy element z kolejki

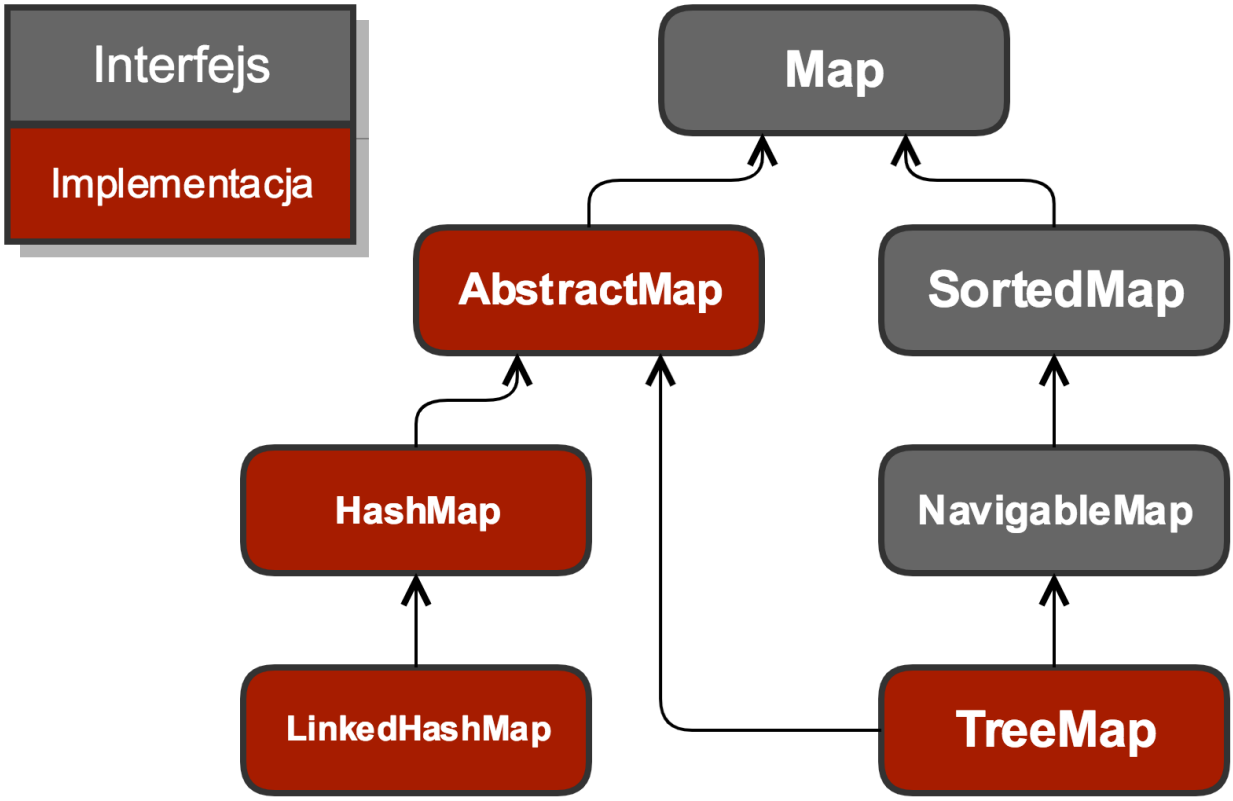
PriorityQueue

Jest **kolejką**, który w momencie wstawiania nowych elementów **sortuje** je według kolejności naturalnej lub według własnego komparatora:

```
PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
priorityQueue.add(20);
priorityQueue.add(21);
priorityQueue.add(18);
priorityQueue.add(19);
priorityQueue.add(23);
System.out.println(priorityQueue); // [18, 19, 20, 21, 23]
```

Mapy (Map)

Mapa, jest innym typem **kolekcji**. W przeciwieństwie do wcześniej opisanych typów nie dziedziczy ona po interfejsie `Collection`:



Przechowuje ona dane w formacie **klucz-wartość**.

Wspólne metody

Interfejs `Map` udostępnia sporo przydatnych metod:

- `size` - zwraca ilość elementów w mapie
- `isEmpty` - sprawdza czy mapa jest pusta
- `put` - dodaje nowy element do mapy
- `putAll` - dodaje wszystkie elementy do mapy
- `containsKey` - sprawdza czy mapa posiada zadany klucz
- `containsValue` - sprawdza czy mapa posiada zadaną wartość
- `remove` - usuwa element z mapy
- `get` - odczytuje wartość na podstawie podanego klucza

HashMap

Najpopularniejszą implementacją interfejsu `Map` jest `HashMap`. Wykorzystuje ona funkcję skrótu do umieszczania elementów:

```
HashMap<String, String> hashMap = new HashMap<>();
```

HashTable

`HashTable` działa podobnie jak `HashMap` z tą różnicą, iż jest kolekcja bezpieczna wątkowo. Wszystkie operacje wykonywane na niej są **synchronizowne**. Kolekcja ta nie zezwala na przechowywanie `null` jako wartości i klucza.

```
Hashtable<String, String> hashtable = new Hashtable<>();
```

TreeMap

`TreeMap` jest mapą, która sortuje klucze w kolejności naturalnej. Korzysta ona z tych samych mechanizmów co `TreeSet`:

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
treeMap.put(1, "1");
treeMap.put(3, "3");
treeMap.put(2, "2");
treeMap.put(5, "5");
System.out.println(treeMap); // {1=1, 2=2, 3=3, 5=5}
```

Szybkość działania kolekcji `TreeMap` określa się mianem O(log(n)).

Zadania

- Wykorzystaj projekt stworzony w zadanich ze zbiorami
- Stwórz klasę testową `HashMapTest`, w której przetestujesz metody:
  - `put`
  - `putAll`
  - `containsKey`
  - `containsValue`
  - `remove`
  - `get`
- Stwórz klasę testową `TreeMapTest`, w której sprawdzisz działanie kolekcji `TreeMap` dla liczb całkowitych.

Iterowanie

Iterowanie po kolekcjach typu `List` czy `Set` wykonuje się wykorzystując pętle `forEach`.

```
for (String value : arrayList) {

}
for (String value : set) {

}
```

Z racji, iż struktura `Map` przechowuje dwa elementy, można ją przeglądać na trzy sposoby. Przeglądając klucze, wartości lub klucze i wartości (`Entry`):

```
HashMap<String, String> hashMap = new HashMap<>();
// klucze
for (String key : hashMap.keySet()) {

}
// wartości
for (String value : hashMap.values()) {

}
// klucze i wartości
for (Map.Entry<String, String> value : hashMap.entrySet()) {
    value.getKey();
    value.getValue();
}
```

Iterator

Większość kolekcji, do wewnętrznego przeglądania zawartości wykorzystuje obiekt `Iterator`. Jest to obiekt, który pozwala odczytywać kolejne elementy z kolekcji:

```
Iterator<String> iterator = arrayList.iterator();
if (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

`Iterator` posiada kilka wbudowanych metod:

- `hasNext` - zwraca wartość `boolean` czy posiada kolejny element
- `next` - odczytuje element z kolekcji
- `remove` - usuwa element z kolekcji

Collections

Klasa `Collections` jest klasą pomocniczą związana z **kolekcjami**. Dostarcza ona zestaw pomocniczych metod:

```
List<Integer> ints = Arrays.asList(1,2,3,4,4,4,4);

Collections.emptyList(); // tworzy pustą listę
Collections.frequency(ints, 4); // 4
Collections.max(ints); // 4
Collections.min(ints); // 1
Collections.reverse(ints); // [4, 4, 4, 4, 3, 2, 1]
Collections.singletonList(1); // tworzy listę z jednym elementem
```

Zadania

- Wykorzystaj projekt stworzony w zadanich ze zbiorami
- Stwórz klasę testową `CollectionsTest`, w której przetestujesz metody z klasy `Collections`:
  - `frequency`
  - `max`
  - `min`
  - `reverse`

Dodatkowe zadania

- * Stwórz klasę testową, w której przetesujesz `PriorityQueue`
- * Stwórz klasę testową, w której przetesujesz `LinkedHashSet`
- * Stwórz klasę testową, w której przetesujesz `LinkedHashMap`
- * Stwórz klasę testową, w której przetesujesz `Iterator`
- * Wypisz wszystkie klucze i wartości z `HashMap`

Typy generyczne

Typy generyczne służą do tworzenia **wzorców/szablonów** dla klas. Umożliwiają one na tworzenie bardziej **złożony klas**, które mogą być w łatwy sposób **reużywane**. Ponadto dzięki typom **generycznym** możemy pobierać elementy bez potrzeby wcześniejszego **rzutowania**. Pozwala to na uniknięcie wielu **błędów na etapie kompilacji** a nie w runtime.

Tworzenie

Do tworzenia klas przy wykorzystaniu typów **generycznych** korzystamy z tak zwanego **operatora diamentowego** `<>`. Wewnątrz tego **operatora** umieszczamy nazwę paramateru. Zwyczajowo są to duże litery:

- `E` - element (to oznaczenie wykorzystywane jest najczęściej we frameworku Collections)
- `K` - klucz
- `N` - liczba
- `T` - typ
- `V` - wartość
- `S`, `U`, `V` - dla następnych typów

Mechanic.java

```
class Mechanic<T> {

    void repair(T carToRepair) {

    }

}
```

Typy generyczne działają tylko dla typów obiektowych!

Aby utworzyć klasę parametryzowaną typem generycznym musimy podać interesujący nas typ w **diamencie** `<Typ>`:

```
Mechanic<BMW> mechanicBMW = new Mechanic<>();
Mechanic<Skoda> mechanicSkoda = new Mechanic<>();
```

Ograniczenie typów

Słowo kluczowe `extends` wykorzystywane było do tej pory aby wskazać po jakiej klasie ma dziedziczyć nasza klasa. W typach generycznych jeśli chcemy ograniczyć typy wykorzystujemy słowo kluczowe `extends`:

Car.java

```
interface Car {

    void takeOffWhell();

}
```

Mechanic.java

```
class Mechanic<T extends Car> {

    void repair(T carToRepair) {
        carToRepair.takeOffWhell();
    }

}
```

Typy generyczne mogą być trudnym zagadnieniem! Na tym poziomie omówiliśmy podstawowe funkcjonalności.

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `generics-example`
- Stworzyć klasę `Food`
- Klasa `Food` powinna przyjmować i ustawiać dwa pola `protected final String name` i `protected final String weight` w konstruktorze
- Klasa `Food` powinna posiadać metodę abstrakcyjną `void prepare()` (w klasach nadpisujących tą metode należy wypisać wartość pola `name`)
- Stworzyć klasę `Nudle`, która dziedziczy po klasie `Food`
- Stworzyć klasę `Cabbage`, która dziedziczy po klasie `Food`
- Stworzyć klasę `Beef`, która dziedziczy po klasie `Food`
- Stworzyć klasę `Chef`, która jest parametryzowana przez klase rozszerzającą `Food`
- Klasa `Chef` powinna posiadać metodę `void prepareMeal(T foodToPrepare)` wywołującą metodę `prepare()` na obiekcie `foodToPrepare`
- Stworzyć klasę `Runner` z `psvm`
- W metodzie `main` stworzyć tylu kucharzy ile jest klas, które dziedziczą po `Food`
- Na każdym kucharzu wywołać metodę `prepareMeal`

Optional

Najpopularniejszym wyjątkiem w **Javie** jest wyjątek typu `NullPointerException`. Aby ułatwić pracę programistą powstała klasa "opakowująca" wartości. Nazywa się ona `Optional`.

Tworzenie

Klasa `Optional` oferuje trzy sposoby tworzenia "opakowania" na obiekt:

- `of` - opakowuje wartość (jeśli wrzucimy `null` dostaniemy wyjątek)
- `ofNullable` - opakowuje wartość, która może być `null`
- `empty` - tworzy pustego `Optional`

```
// Tworzenie Optional
Optional<String> stringAsOptional = Optional.of("wartość");

// Tworzenie Optional z wartości która może być null'em
Optional<String> stringAsOptionalFromNull = Optional.ofNullable(fieldWhichCanBeNull);

// Tworzenie pustego Optional
Optional.empty();
```

Odczyt

Odczytywanie wartości odbywa się przy użyciu metody `get`. Natomiast dobrą praktyką przed odczytem jest sprawdzanie czy `Optional` zawiera element. Aby to sprawdzić wykorzystujemy metodę `isPresent`.

```
// Odczytywanie wartości
if (optionalValue.isPresent()) { // sprawdzenie czy istnieje
    optionalValue.get() // odczyt
}
```

Wartość domyślna

`Optional` ułatwia nam zwrócenie wartości domyślnej poprzez metody `orElse`, `orElseGet` i `orElseThrow`:

```
String value = optional.orElse("default");
```

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `optional-example`
- Stworzyć klasę `NullableExample`
- W klasie `NullableExample` dodać pole `private final String string`
- W klasie `NullableExample` dodać metodę `Optional<String> getNull()` zwracającą wartość `Optional.ofNullable(null)`
- W klasie `NullableExample` dodać metodę `Optional<String> getString()` zwracającą wartość `Optional.of(string)`
- Stworzyć klasę `OptionalExample`
- Klasa `OptionalExample` w konstruktorze przyjmuje parametr typu `NullableExample`
- W klasie `OptionalExample` dodać metodę `String getOrDefault()` pobierającą wartość `getNull()` i zwracającą wartość domyślną `"Empty"` w przypadku gdy zwrócona wartość jest pusta
- W klasie `OptionalExample` dodać metodę `boolean get()` pobierającą wartość `getString()` i zwracającą z tej metody `true` jeśli wartość istnieje w przeciwnym przypadku `false`
- Napisać testy dla klasy `OptionalExample`

Interfejsy funkcyjne

Interfejsy funkcyjne są jednym z wielu nowych elementów w **Javie 8**. Wszystkie **interfejsy funkcyjne** znajdują się w pakiecie `java.util.function`.

Interfejsy funkcyjne wykorzystywane są przy współpracy z wyrażeniami **Lambda**. Możemy przygotować zachowania, które będziemy wykorzystywać podczas przetwarzania danych. **Interfejsy funkcyjne** posiadają tylko jedną metodę.

W pakiecie `java.util.function` znajdziemy wiele bardzo użytecznych **interfejsów funkcyjnych**. Do tych najbardziej podstawowych należą:

- `Function <T, R>` – przyjmuje dowolny obiekt i zwraca dowolny obiekt `(T, R)`
- `Consumer <T>` – przyjmuje dowolny obiekt, ale nic nie zwraca `(T, void)`
- `Supplier <T>` – nic nie przyjmuje, ale zwraca dowolny obiekt `(void, T)`
- `Predicate <T>` – przyjmuje dowolny obiekt, ale zwraca boolean `(T, boolean)`

Function <T, R>

Interfejs funkcyjny `Function <T, R>` przyjmuje dowolny obiekt i zwraca dowolny obiekt `(T, R)`. Domyślnie jest to metoda `apply`:

```
@FunctionalInterface
public interface Function<T, R> {

     R apply(T t);

}
```

Własne **function**:

MessageConsumer.java

```
class MessageFunction implements Function<String, String> {

    @Override
    public String apply(String stringToChange) {
        return stringToChange + "some message";
    }

}
```

Consumer <T>

Interfejs funkcyjny `Consumer <T>` przyjmuje dowolny obiekt, ale nic nie zwraca `(T, void)`. Domyślnie jest to metoda `accept`:

```
@FunctionalInterface
public interface Consumer<T> {

     void accept(T t);

}
```

Własny **consumer**:

MessageConsumer.java

```
class MessageConsumer implements Consumer<String> {

    @Override
    public void accept(String stringToConsume) {
        System.out.println(stringToConsume);
    }

}
```

Supplier <T>

Interfejs funkcyjny `Supplier <T>` nic nie przyjmuje, ale zwraca dowolny obiekt `(void, T)`. Domyślnie jest to metoda `get`:

```
@FunctionalInterface
public interface Supplier<T> {

    T get();

}
```

Własny **supplier**:

MessageSupplier.java

```
class MessageSupplier implements Supplier<String> {

    @Override
    public String get() {
        return "SDA!";
    }

}
```

Predicate <T>

Interfejs funkcyjny `Predicate <T>` przyjmuje dowolny obiekt, ale zwraca boolean `(T, boolean)`. Domyślnie jest to metoda `test`:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

}
```

Własny **predykat**:

MessageChecker.java

```
class MessageChecker implements Predicate<String> {

    @Override
    public boolean test(String stringToCheck) {
        return stringToCheck == null || stringToCheck.isEmpty();
    }

}
```

Własny interfejs funkcyjny

Aby stworzyć własny **interfejs funkcyjny** wystarczy stworzyć nowy interfejs tylko z **jedną metodą**. Ponadto, dobrą praktyką jest oznaczanie takiego interfejsu adnotacją `@FunctionalInterface`. Jest to adnotacja tylko informacyjna dla kompliatora aby sprawdzić czy mamy tylko jedną metodę. Dodatkowo, jeśli użyjemy tej adnotacji to **IDE** może nam przypominać o tym, że nie możemy dodawać nowych metod do tego interfejsu:

OwnFunctionalInterface.java

```
@FunctionalInterface
interface OwnFunctionalInterface {

    boolean someMethod();

}
```

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `functional-interface-example`
- Stworzyć klasę implementująca interfejs `Predicate` i sprawdzający czy podana liczba jest

parzysta

- Stworzyć klasę implementująca interfejs `Supplier` i zwracający losową liczbę całkowitą (sprawdź jak pobrać liczbę losową w **Javie**)
- Stworzyć klasę implementująca interfejs `Consumer` i wypisującego podaną liczbę
- Stworzyć klasę implementująca interfejs `Function`, która przyjmuję liczbę całkowitą i podnosi ją do potęgi o tą samą wartość co podana (4 -> 4^4) (sprawdź jak podnieść liczbę do potęgi w **Javie**)
- Stworzyć test dla funkcji i predykatu
- Stworzyć własny interfejs funkcyjny sprawdzający czy podana liczba jest nieparzysta
- Stworzyć test dla stworzonego interfejsu funkcyjnego

Lambda

Lambda to skrócona forma zapisu **interfejsu funkcyjnego**. Składa się ona z trzech elementów:

```
(opcjonalne-parametry) -> logika

(opcjonalne-parametry) -> { logika
                            na
                            kilka
                            linii }
```

Składnia

```
(x) -> System.out.println(x) // wypisze dowolnego x

// coś jest pobierane ale nic nie jest zwraca więc jest Consumer
Consumer<String> consumer = (x) -> System.out.println(x);
```

Jeśli przypominimy sobie jak wygląda **interfejs funkcyjny supplier** okaże się, że nie przyjmuje żadnego parametru (dlatego parametr w **lambdzie** jest opcjonalny):

```
() -> "Supplier"; // zwróci napis "Supplier"

Supplier<String> supplier = () -> "Supplier";
```

Typy

Składnia wyrażenia **lambda** może zaciemniać typ wprowadzonych parametrów, dlatego też można je podawać wprost (jest to opcjonalne):

```
(String x) -> System.out.println(x) // wypisze dowolnego x
(String x, Integer y) -> System.out.println(x + y) // wypisze konkatenacje x i y
```

Ciało lambdy

Najczęściej wyrażenia **lambda** jest jednolinijkowe aby poprawić czytelność kodu. Jednakże mechanizm ten umożliwia tworzenia bardziej złożony konstrukcji, które muszą znajdować się pomiedzy {}:

```
Supplier<String> supplier = () -> {
    if (wtorek) {
        return "wtorek";
    }
    return "inny dzień";
};

() -> {};
```

Zadania

- Stwórz nowy projekt **Maven** o nazwie `lambda-example`
- Utwórz nowy interfejs funkcyjny `StringSuppiler` z jedną metodą `String string();`
- Utwórz nową klase `Runner` z `psvm`
- W metodzie `main`:
  - Stwórz, przypisz i wywołaj lambdę, która przyjmuje jeden parametr i go wypisuje
  - Stwórz, przypisz i wywołaj lambdę, która nie przyjmuje parametrów, ale zwraca napis "SDA"
  - Stwórz, przypisz i wywołaj lambdę, która sprawdzi czy podany liczba jest parzysta
  - Stwórz, przypisz i wywołaj lambdę, która przyjmuje dwa parametry i wypisuje je połączone
  - Stwórz `Optional.ofNullable(null)` i wywołaj na nim metodę `ifPresent` i przekaż tam dowolnego consumer'a
  - Stwórz `Optional.ofNullable(null)` i wywołaj na nim metodę `orElseGet` i przekaż tam dowolnego supplier'a
  - Stwórz, przypisz i wywołaj lambdę dla `StringSuppiler`

Strumienie

Strumienie `Stream` są kolejnym dodatkiem wprowadzonym w **Javie** 8. Pozwalają one na relizowanie **paradygmatu funkcyjnego**, w którym mówimy co chcemy osiągnąć a nie jak.

Tworzenie

Strumienie można tworzyć na kilka sposób. Najpopularniejszym sposobem tworzenia strumieni jest metoda fabrykująca `of`:

```
Stream<String> stringStream = Stream.of("a", "b", "c");
```

Operacje pośrednie

Operacje pośrednie to metody, które operują na strumieniu, ale go nie zamykają. Dzięki temu, możliwy jest chaining:

```
Stream.of(1, 2, 3)
        .filter(value -> value > 2)
        .map(value -> String.valueOf(value));
```

Istnieje kilka najpopularniejszych **operacji pośrednich**:

- `map` - zmienia jeden typ na inny
- `filter` - przepuszcza tylko dane spełniające wymagania filtru

Operacje terminalne

Operacje terminalne są to operacje, które zamykają strumień. Kończą one pracę ze strumieniem.

Uwaga! Może być tylko jedna operacja kończąca, w przeciwnym wypadku wystąpi wyjątek.

```
List<String> numbersAsString = Stream.of(1, 2, 3)
        .filter(value -> value > 2)
        .map(value -> String.valueOf(value))
        .collect(Collectors.toList());
```

Istnieje kilka najpopularniejszych **operacji terminalnych**:

- `collect` - zbiera dane do kolekcji
- `count` - zlicza elementy
- `allMatch` - sprawdza czy wszystkie elementy spełniają dany warunek

- `anyMatch` - sprawdza czy dowolny jeden element spełnia dany warunek
- `findFirst` - zwraca pierwszy element spełniający dany warunek

Strumienie a interfejsy funkcyjne

Operacje pośrednie wykorzystują poznane już interfejsy funkcyjne. Przykładowo operacja `filter` przyjmuje **predykat**:

```
Stream<T> filter(Predicate<? super T> predicate);
```

Strumienie a kolekcje

Większość kolekcji z frameworku **Java Collections** dostarcza możliwość przetwarzania jej za pomocą strumieni:

```
arrayList.stream()
        .filter(string -> string.isEmpty())
        .collect(Collectors.toList());
```

Method reference

Kolejny dodatek z **Javy 8** to **referencja do metody**. Aby utworzyć referencję do metody korzystamy z symbolu `::`. Dzięki temu, możemy tworzyć metody, które są reużywalne. Referencje do metod można używać tylko w **interfejsach funkcyjnych**:

```
arrayList.stream()
        .filter(String::isEmpty)
        .map(String::toUpperCase)
        .collect(Collectors.toList());
```

Dzięki takiemu zapisowi nasz kod staje się jeszcze bardziej czytelny.

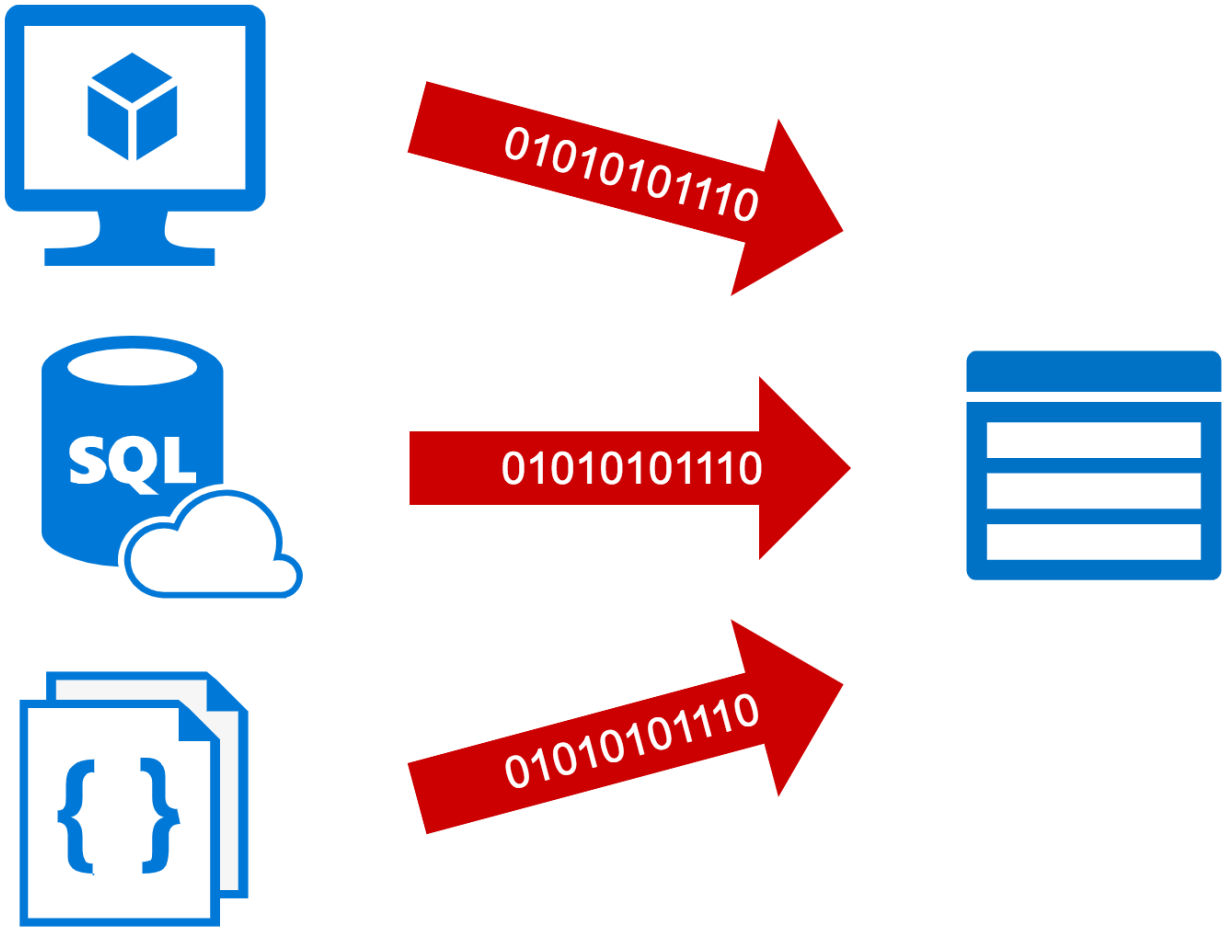
Zadania

- Stworzyć nowy projekt **Maven** z nazwą `streams-example`
- Stworzyć klasę testową `StreamTest`, w której napiszesz testy do:
  - stwórz strumień z pięcioma elementami typu `String`:
    - "first"
    - "second"
    - "third"
    - "fourth"

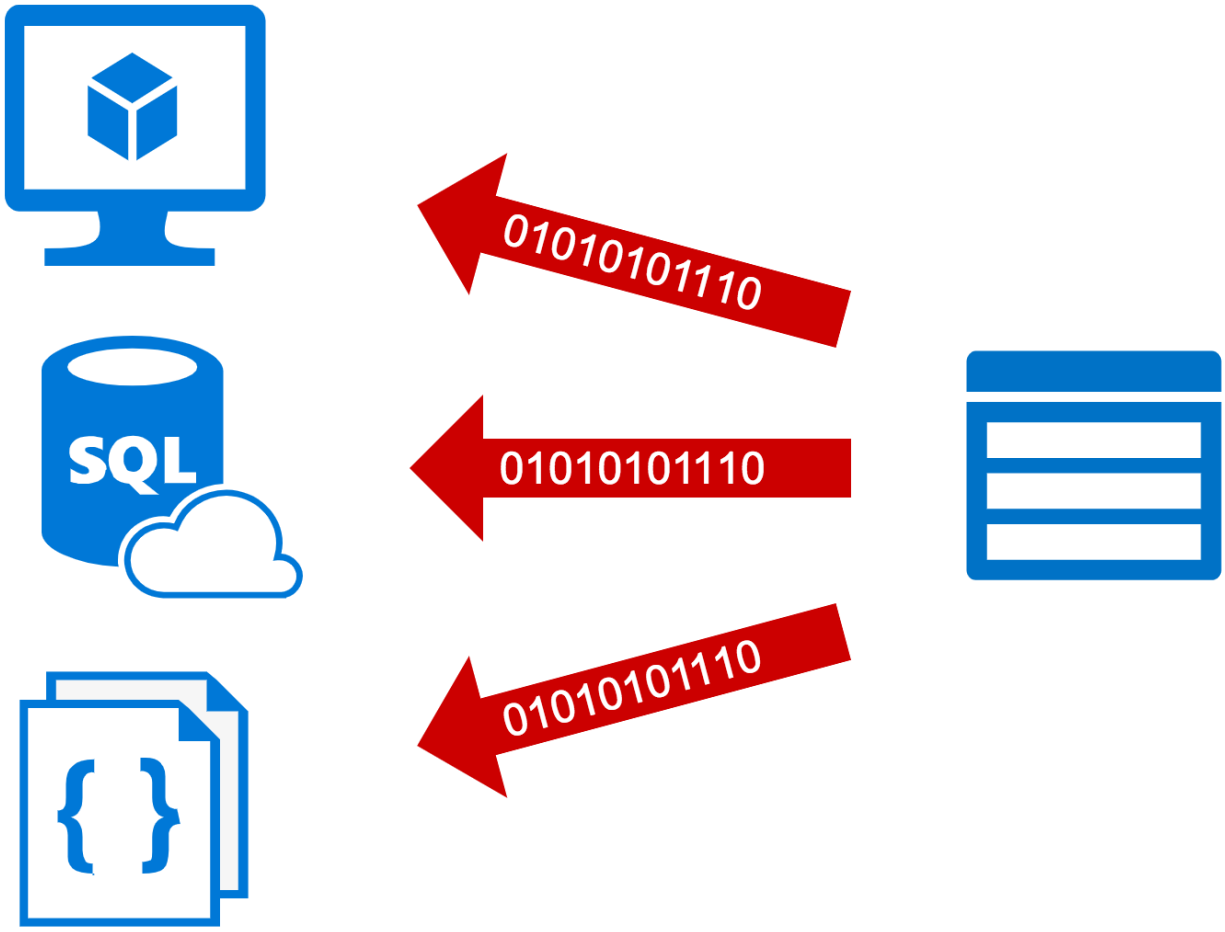
        - "fifth"
        - z utworzonego strumienia zwróć słowa dłuższe niż 5 znaków
        - oraz zwróć je w formacie UPPERCASE
    - stwórz strumień z pięcioma elementami typu `String`:
        - "first"
        - "second"
        - "third"
        - "fourth"
        - "fifth"
        - zwróć pierwszy element który ma więcej niż 7 znaków
    - stwórz strumień z pięcioma elementami typu `Integer`:
        - 1
        - 26
        - 30
        - 2
        - 45
        - zwróć liczby parzyste
    - stwórz strumień z pięcioma elementami typu `Integer`:
        - 1
        - 26
        - 30
        - 2
        - 45
        - zwróć maksymalną liczbę
    - stwórz strumień z pięcioma elementami typu `Integer`:
        - 1
        - 26
        - 30
        - 2
        - 45
        - zwróć listę liczb większych od 26 jako lista z `String`

InputOutput (IO)

Platforma **Java** do operacji **odczytu/zapisu** (I/O) wykorzystuje strumienie danych. Strumienie dzielimy na **wyjściowe** i **wejściowe**. Mogą one reprezentować różne rodzaje strumieni danych z takich źródeł jak pliki na dysku, urządzenia czy inne programy. Wszystkie strumienie są **sekwencjami danych**, które poruszają się w dwóch kierunkach. Strumień wejściowy:



Strumień wyjściowy:



Byte Streams

Najbardziej podstawowywm strumieniem danych jest **strumień bajtowy**. Przesyła on lub odbiera strumień **8-bitowych bajtów**. Wszystkie wariacje strumieni bajtów dziedziczą po `InputStream` dla danych wejściowych i `OutputStream` dla danych wyjściowych.

Pamiętaj, aby zawsze zamykać strumienie danych!

File Input/Output Stream

-1 oznacza koniec pliku (EOF end-of-file)!

Do odczytywania/wysyłania strumienia bajtów z/do pliku korzystamy z klas `FileInputStream` oraz `FileOutputStream`. Odczytują/zapisują one bajty w zakresie od 0 do 255. Wystąpienie -1 oznacza koniec pliku:

FileReader.java

```
FileInputStream file = null;
try {
    file = new FileInputStream(filePath);
    int byteValue;
    while((byteValue = file.read()) != -1) {
        System.out.println(byteValue);
    }
} finally {
    if (file != null) {
        file.close();
    }
}
```

FileWriter.java

```
FileOutputStream file = null;
try {
    file = new FileOutputStream(filePath);
    file.write(bytesToSave);
} finally {
    if (file != null) {
        file.close();
    }
}
```

Character Streams

Jeśli wiemy, iż nasz plik będzie przechowywał znaki to możemy wykorzystać strumienie znaków. Reprezentowane są one przez `FileReader` i `FileWriter`:

FileReader.java

```
FileReader fileReader = null;
try {
    fileReader = new FileReader("file.txt");
    int value;
    while ((value = fileReader.read()) != -1) {
        System.out.println((char)value);
    }
} finally {
    if (fileReader != null) {
        fileReader.close();
    }
}
```

FileWriter.java

```
FileWriter fileWriter = null;
try {
    fileWriter = new FileWriter("copy.txt");
    fileWriter.write("copy");
} finally {
    if (fileWriter != null) {
        fileWriter.close();
    }
}
```

Buffered Streams

Odpytywanie **systemu operacyjnego** o każdy znak z pliku jest bardzo kosztowną operacją. Aby zoptymalizować ten proces, wykorzystywany jest mechanizm buforowania. Umieszcza on odczytane/zapisane dane w buforze, aby na rządzanie tylko raz je zapisać lub odczytać. Każdy strumień można skonwertować na strumień buforowany:

BufferedExample

```
BufferedReader fileReaderBuffor = new BufferedReader(new FileReader("file.txt"));
String textLine = fileReaderBuffor.readLine();
do {
    System.out.println(textLine);

    textLine = fileReaderBuffor.readLine();
} while(textLine != null);
```

Do konwersji strumieni bajtowych wykorzystujemy `BufferedInputStream` i `BufferedOutputStream` natomiast dla znakowych `BufferedReader` i `BufferedWriter`. Dane trzymane są w buforze do czasu, aż plik zostanie zamkniemy, wywołamy metodę `flush` lub bufor się przepełni.

Formatowanie danych

Operowanie na odczytywanych bajtach może być trudnym zagadanieniem. Aby ułatwić ten proces powstały dwa mechanizmy, skanery przekształacjące strumień danych w tokeny i formatery formatujące tekst do żądanego formatu.

Scanner

Pamiętaj, aby zawsze zamykać obiekty typu `Scanner`!

Obiekty typu `Scanner` wykorzystywane są do przekształcania strumienia danych w tokeny. Tokenem może być wartość `int`, `long` czy `String`:

ScannerExample.java

```
// Strings
Scanner scanner = new Scanner(new BufferedReader(new FileReader("strings.txt")));
while(scanner.hasNext()) {
    System.out.println(scanner.next());
}

// Ints
Scanner scanner = new Scanner(new BufferedReader(new FileReader("ints.txt")));
while(scanner.hasNext()) {
    System.out.println(scanner.nextInt());
}
```

Standardowo scanner rozdziela tokeny korzystając z https://docs.oracle.com/javase/6/docs/api/java/lang/Character.html#isWhitespace%28char%29

Odczyt z terminala

Aby odczytać dane z linii poleceń korzystamy z `System.in`:

ScannerExample.java

```
Scanner in = new Scanner(System.in);
while(in.hasNext()) {
    System.out.println(in.next());
}
```

Formatter

Strumienie dostarczające funkcjonalność **formatowania** to `PrintWriter` dla strumienia znaków i `PrintStream` dla strumienia bajtów. Obie te klasy dostarczają zwykłe metody `write`, które służą do wysłania strumienia bajtów lub znaków. Ponadto dostarczają taki sam zbiór metod konwertujących dane do sformatowanego wyjścia:

- `print` i `println` - formatuje dane w sposób standardowy
- `format` - formatuje dane w sposób zdefiniowany przez użytkownika

Najczęściej stosowanym przez nas formaterem jest `PrintStream`, do którego można dostać się poprzez klasę `System`:

```
PrintStream printStream = System.out;
printStream.println("Hello World!");

System.out.println("Hello World!");
```

Metody `print` i `println` "pod spodem" wywołują na obiektach metodę `toString`, natomiast typy

proste rzutowane są na obiekty typu `String` korzystając najczęściej z metody `String.valueOf`:

PrintStream.java

```
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}

public void println(boolean x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}

public void print(boolean b) {
    write(b ? "true" : "false");
}
```

String.java

```
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

print

Metoda `print` służy do wyświetlnia w tej samej linii strumienia danych sformatownych w standardowy sposób:

Printer.java

```
System.out.print("Hello World!");
```

println

Metoda `println` służy do wyświetlnia w nowej linii strumienia danych sformatownych w standardowy sposób:

Printer.java

```
System.out.println("Hello World!");
```

format

Metoda `format` służy do formatowania tekstu według podanego formatu:

- `%` - poprzedza każdy znak dla formatowania
- `d` - formatuje wartość `int` jako liczbę całkowitą
- `s` - formatuje dowolną wartość jako `String`
- więcej

Formatter.java

```
System.out.format("To jest %s dowolny obiekt a tutaj %d to cyfra", "Obiekt", 20);
// To jest Obiekt dowolny obiekt a tutaj 20 cyfra
```

Data Streams

Pliki oprócz przechowywania zwykłego tekstu mogą przechowywać też dane. Jeśli chcemy zapisać/odczytać dane (boolean, char, byte, short, int, long, float, double i String) w/z pliku możemy wykorzystać interfejsy `DataInput` i `DataOutput`. Najpopularniejsze implementacje tego interfejsu to `DataInputStream` i `DataOutputStream`.

DataOutputStreamExample

```
DataOutputStream out = new DataOutputStream(
                           new BufferedOutputStream(
                               new FileOutputStream("data.txt")));

for (int i = 0; i < 10; i ++) {
    out.writeDouble(10.0 + i);
    out.writeInt(i);
    out.writeUTF("Value:" + i);
}
```

Koniec pliku sygnalizowany jest poprzez wyjątek `EOFException`!

Object Streams

W poprzednim rozdziale poznaliśmy strumienie danych dla typów prymitywnych, teraz czas na typy obiektowe. Zapisywanie obiektów odbywa się dzięki implementacjom interfejsów `ObjectInput` i `ObjectOutput`. Najpopularniejsze implementacje to `ObjectInputStream` oraz `ObjectOutputStream`.

ObjectOutputStreamExample

```
ObjectOutputStream out = new ObjectOutputStream(
        new BufferedOutputStream(
                new FileOutputStream("data.txt")));

out.writeObject(someObject);

ObjectInputStream in = new ObjectInputStream(
        new BufferedInputStream(
                new FileInputStream("data.txt")));

in.readObject();
```

Zadania

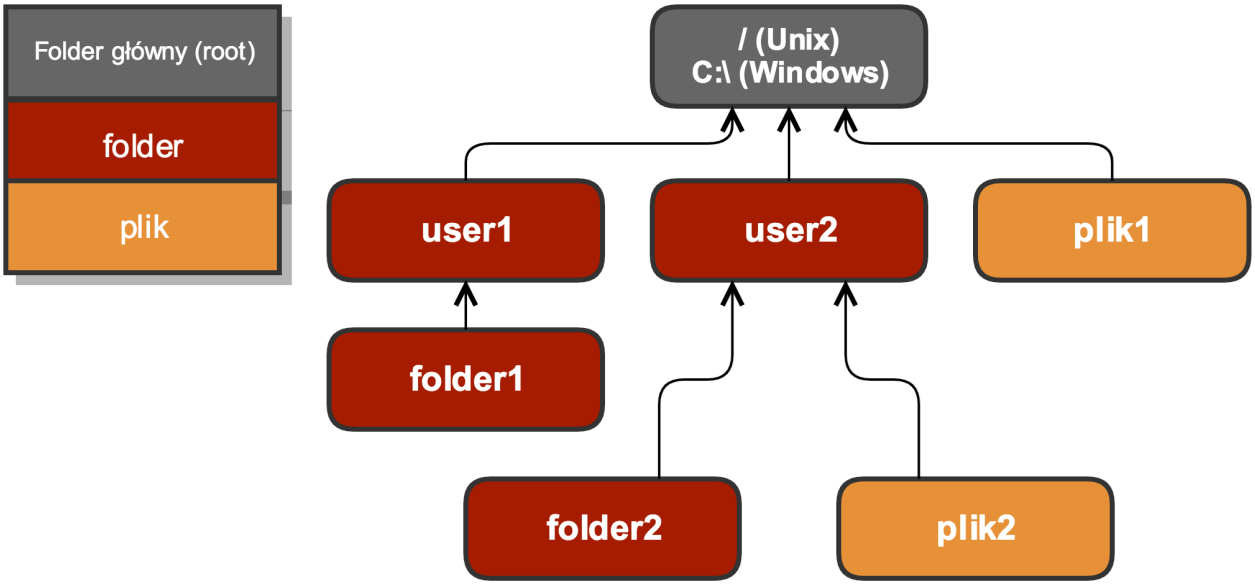
- Stworzyć nowy projekt **Maven** z nazwą `io-example`
- Utworzyć klasę `Runner` z `psvm`
- Stworzyć nowy plik `file.txt` w folderze projektu i dodać do niego kilka linii tekstu
- W metodzie `main` wypisać bajty z pliku na konsoli korzystając z `FileInputStream`
- W metodzie `main` skopiować zawartość pliku `file.txt` do innego pliku z nazwą `kopia.txt` korzystając z `FileOutputStream`
- W metodzie `main` wypisać znaki z pliku `file.txt` na konsoli korzystając z `FileReader`
- W metodzie `main` wypisać wszystkie linie z pliku `file.txt` na konsoli korzystając z `BufferedReader`
- W metodzie `main` pobrać i wypisać dane z terminala
- W metodzie `main` zapisać dane (`int = 20`, `double = 15.5`, `String = "file"`) w pliku `data.txt` korzystając z `DataOutputStream`
- W metodzie `main` odczytać i wypisać dane z pliku `data.txt` korzystając z `DataInputStream`
- W metodzie `main` zapisać dane (`new BigDecimal(-2)`) w pliku `dataObjects.txt` korzystając z `ObjectOutputStream`
- W metodzie `main` odczytać i wypisać dane z pliku `dataObjects.txt` korzystając z `ObjectInputStream` (rzutuj odczytany obiekt na `BigDecimal` i użyj metody `negate`)

New InputOutput (NIO)

Praca ze strumieniami nie należała do zbyt intuicyjnych. Aby ułatwić tą pracę pojawił się nowy pakiet zwany **New I/O** `java.nio.file`. Większość operacji wykonywana jest przy użyciu klas `Path` i `Files`.

Ścieżka (Path)

Systemy operacyjne przechowują pliki w strukturach drzewiastych. Na samej górze struktury znajduje się główny folder zwany `root`. Pod głównym węzłem umieszczone są pliki i foldery.



Tworzenie ścieżki

Aby stworzyć nowy obiekt typu `Path` możemy wykorzystać metody fabrykujące z klasy `Paths`:

PathExample.java

```
Path path = Paths.get("some/path");
Path pathShorcutFor = FileSystems.getDefault().getPath("some/path");
```

Pliki (Files)

Kolejną ważna klasą po `Path` w pakiecie `java.nio.file` jest klasa `Files`. Zawiera ona zbiór przydatnych metod statycznych.

Informacje o folderze/pliku

Klasa `Files` dostarcza kilka metod ułatawiających sprawdzenie informacji na temat folderu/pliku:

FilesExample.java

```
Files.isHidden(file); // czy plik jest ukryty
Files.isReadable(file); // czy można odczytać plik
Files.isExecutable(file); // czy jest plikiem wykonywalnym
Files.exists(file); // czy plik istnieje
```

Usuwanie folderu/pliku

Aby usunąć folder/plik wykorzystujemy metodę `delete` z klasy `Files`:

DeleteExample.java

```
Files.delete(file); // usuwa plik, może rzucić wyjątek, że plik nie istnieje
Files.deleteIfExists(file); // usuwa plik jeśli istnieje
```

Kopiowanie folderu/pliku

Aby skopiować folder/plik wykorzystujemy metodę `copy` z klasy `Files`. Operacja kopiowania może zawierać dodatkowe ustawienia:

- `REPLACE_EXISTING` - podmienia istniejący folder/plik
- `COPY_ATTRIBUTES` - kopiuje atrybuty folderu/pliku

CopyExample.java

```
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

Przenoszenie folderu/pliku

Aby przenieść folder/plik wykorzystujemy metodę `move` z klasy `Files`. Podobnie jak metoda `copy` przyjmuje ona różne opcje do operacji przenoszenia:

- `REPLACE_EXISTING` - podmienia istniejący folder/plik
- `ATOMIC_MOVE`- wykonuje operację jako atomową (nikt nie może wykonywać żadnych operacji na tym pliku/folderze podczas kopiowania)

MoveExample.java

```
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
```

Metoda ta przenosi tylko aktualny folder, nie wykonuje tego rekursywnie (nie wchodzi w głąb struktury)

Tworzenie plików/folderów

Aby utworzyć nowy plik lub folder ponownie wykorzystujemy klasę `Files`:

CreateExample.java

```
Files.createFile(path);
Files.createDirectory(path);
```

Czytanie z pliku

Odczytywanie plików w **New I/O** zostało bardzo mocno uproszczone:

ReadExample.java

```
byte[] bytes = Files.readAllBytes(path);
List<String> allLines = Files.readAllLines(path);
BufferedReader bufferedReader = Files.newBufferedReader(path);
InputStream inputStream = Files.newInputStream(path);
```

Zapisywanie do pliku

Podobnie jak przy odczytywaniu plików, zapis został bardzo mocno uproszczony:

WriteExample.java

```
Path writeBytes = Files.write(path, bytes);
Path write = Files.write(path, allLines);
BufferedWriter bufferedWriter = Files.newBufferedWriter(path);
OutputStream outputStream = Files.newOutputStream(path);
```

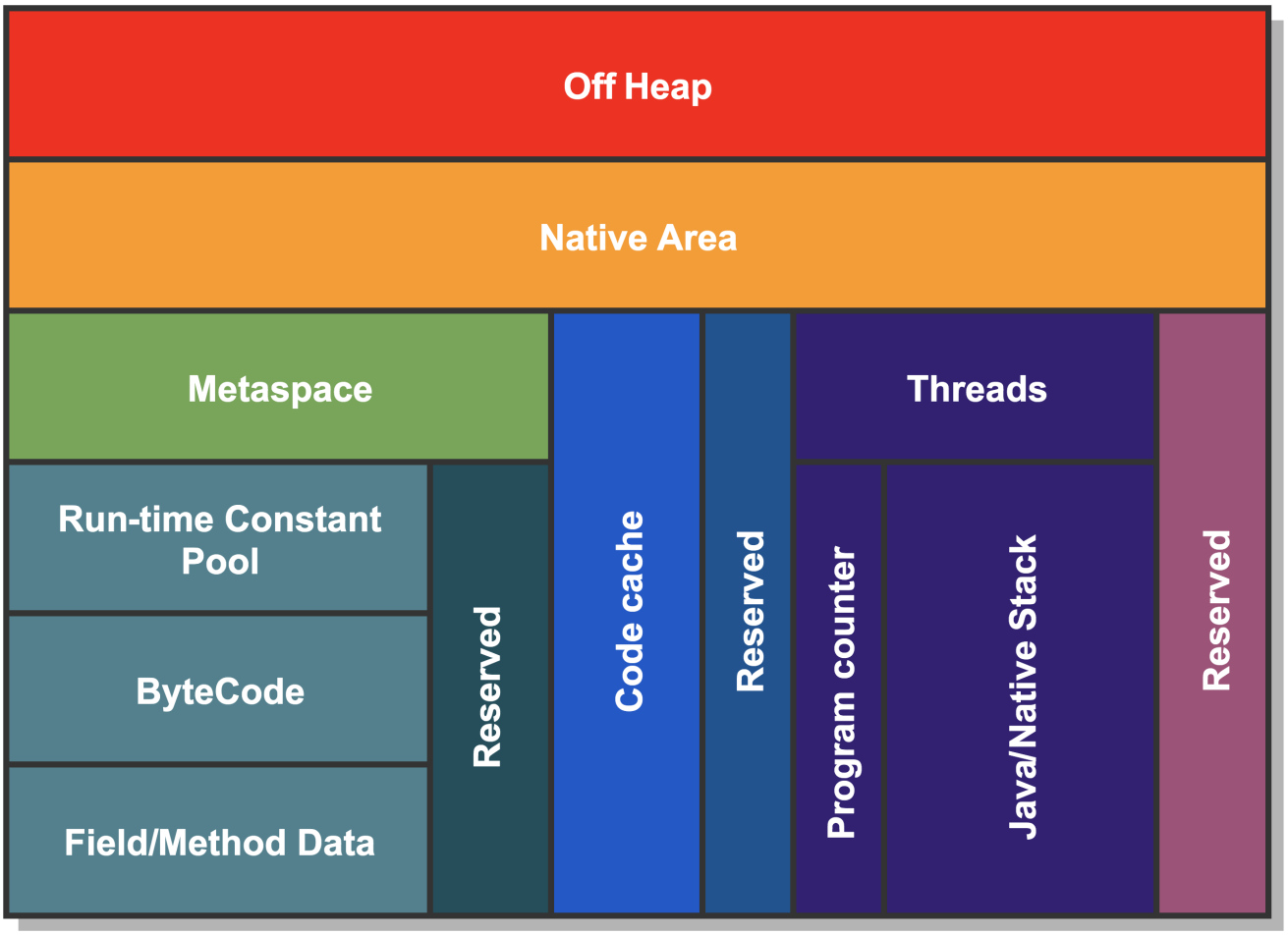
Zadania

- Stworzyć nowy projekt **Maven** z nazwą `new-io-example`
- Utworzyć klasę `Runner` z `psvm`
- Stworzyć nowy plik `file.txt` w folderze projektu i dodać do niego kilka linii tekstu
- Stworzyć nowy plik `fileSecond.txt` w folderze projektu i dodać do niego kilka linii tekstu
- Stworzyć nowy folder `files` w folderze projektu
- W metodzie `main` stworzyć `Path` do pliku `file.txt`
- W metodzie `main` sprawdzić i wypisać czy plik `file.txt`:
  - jest ukryty
  - można go odczytać
  - istnieje

- W metodzie `main` usunąć nieistniejący plik `fileThird.txt`
- W metodzie `main` skopiować plik `file.txt` do tego samego folderu z nazwą `fileCopy.txt`
- W metodzie `main` przenieść plik `fileCopy.txt` do folderu z nazwą `files`
- W metodzie `main` stworzyć nowy folder `io`
- W metodzie `main` stworzyć nowy plik `io.txt` w folderze `io`
- W metodzie `main` odczytać i wypisać zawartość pliku `file.txt`
- W metodzie `main` zapisać do pliku `io.txt` kilka linii tekstu (`Files.write`)
- W metodzie `main` odczytać i wypisać zawartość pliku `io.txt`

Wielowątkowość

Na pewnym etapie tworzenia aplikacji dochodzimy do momentu, w którym chcemy **przyśpieszyć** prace naszej aplikacji. Jednym ze sposobów **przyśpieszenia** pracy jest wykonywanie zadań **równolegle**. **Zrównoleglenie** pracy możemy osiągnąć poprzez stworzenie nowego **wątku**. Uruchomiony program nazywany jest **procesem**, natomiast w środku procesu uruchomione są **wątki** (minimum jeden). Należy jednak pamiętać o kosztach związanych z tworzeniem wątku. Przyjmuje się, iż jeden wątek zajmuje w pamięci około 1 MB. Alokowany jest on w specjalnym **obszarze pamięci**. Obszar ten znajduje się na tak zwanym **off heapie** w bloku zwanym **metaspace**:



Main

W **Javie** główny wątek programu nazywany jest wątkiem `main`. Jeśli chcemy pobrać aktualny **wątek** to korzystamy ze statycznej metody `Thread.currentThread()`:

Runner.java

```
class Runner {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread()); // Thread[main,5,main]
        System.out.println(Thread.currentThread().getName()); // main
    }
}
```

Thread

`Thread` jest klasą, która reprezentuje **wątek**. Możemy myśleć o nowym **wątku** jako o **pracowniku**. Aby stworzyć nowego **pracownika** to tworzymy nową klasę, która dziedziczy po klasie `Thread` oraz nadpisujemy metodę `run`. Samo dziedziczenie pozwala nam jedynie na stworzenie nowego **pracownika**, natomiast w nadpisanej metodzie `run` określamy **prace** jaka ma zostać przez niego wykonana:

Fred.java

```
class Fred extends Thread {

    @Override
    public void run() {
        //logic
    }
}
```

run vs start

O **różnicę** pomiędzy tymi metodami pytają na **rozmowach kwalifikacyjnych**!

Udało nam się stworzyć **pracownika**. Teraz pora zlecić mu wykonanie pracy. W kontekście klasy `Thread` pojawiają się dwie metody `run` i `start`. Choć z pozoru wyglądają bardzo podobnie mają one inne **przeznaczenie**. W metodzie `run` umieszczamy logikę (**zadanie**), która będzie wykonana przez **pracownika**. Natomiast metoda `start` uruchamia nowy wątek (nasz **pracownik** zaczyna wykonywać pracę w nowym **wątku**):

Fred.java

```
class Fred extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
```

Runner.java

```
class Runner {
    public static void main(String[] args) {
        Fred fred = new Fred();
        fred.run(); // to odpali tylko logikę
        fred.start(); // to odpali logikę w nowym wątku
        // wynik
        // main
        // Thread-0
    }
}
```

Runnable

`Runnable` jest **interfejsem funkcyjnym**, który posiada tylko jedną metodę typu `void run()`. Z tego interfejsu korzystamy wtedy, kiedy chcemy utworzyć nowe **zadanie** (a nie **pracownika**):

Runnable.java

```
@FunctionalInterface
public interface Runnable {

    public abstract void run();

}
```

Job.java

```
class Job implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

}
```

Samo `Runnable` służy do stworzenia **zadania**, aby to zadanie uruchomić musimy wrzucić je do nowego **wątku** (pracownika):

Runner.java

```
class Runner {
    public static void main(String[] args) {
        Job job = new Job();
        Thread fred = new Thread(job);
        fred.run(); // to odpali tylko logikę
        fred.start(); // to odpali logikę w nowym wątku
    }
}
```

Pula wątków

Podczas tworzenia zadań **asynchronicznych** możemy wskazać własną **pulę wątków**. Takie rozwiązanie jest lepsze niż operowanie na **domyślnej puli wątków**, ponieważ pozwala nam kontrolować jej parametry. **Java** dostarcza przyjazny mechanizm `Executors`, który umożliwia tworzenie **puli wątków**.

Pojedynczy wątek

Metodą ułatwiającą stworzenie nowego, pojedynczego wątku jest `newSingleThreadExecutor`:

```
Executors.newSingleThreadExecutor();
```

Określona ilość

Jeśli potrzebujemy **pulę wątków** o określonej wielkości możemy wykorzystać metodę `newFixedThreadPool`, która jako parametr przyjmuje ilość wątków:

```
Executors.newFixedThreadPool(int numberOfThreads);
```

Używanie

Po stworzeniu interesującej nas **puli wątków** pora ją wykorzystać:

```
ExecutorService threadPool = Executors.newFixedThreadPool(5);
threadPool.submit(job);
threadPool.submit(job);
threadPool.submit(job);
threadPool.submit(job);
threadPool.submit(job);
```

Zamykanie

Należy pamiętać, aby po zakończonej pracy zamknąć **pulę wątków**. Jeśli tego nie zrobimy nasz

program nie zakończy działania. Zamknięcie **puli wątków** odbywa się między innymi przy użyciu metody `shutdown`:

Pula wątków

```
ExecutorService threadPool = Executors.newFixedThreadPool(5);

threadPool.submit(job);
threadPool.submit(job);
threadPool.submit(job);
threadPool.submit(job);
threadPool.submit(job);

threadPool.shutdown();
```

Dział wielowątkowości jest dużo większy i bardziej złożony!

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `threads-example`
- Stworzyć klasę `Runner` z `psvm`
- W metodzie `main` wypisać nazwę aktualnego wątku
- Stworzyć nową klasę dziedziczącą po `Thread` i dodać w niej logikę wypisania nazwy wątku
- W metodzie `main` uruchomić nowo stworzony wątek korzystając z metody `start`
- W metodzie `main` uruchomić nowo stworzony wątek korzystając z metody `run`
- Stworzyć nową klasę implmentującą interfejs `Runnable` i dodać w niej logikę wypisania nazwy wątku
- W metodzie `main` uruchomić nowo stworzony wątek korzystając z metody `start`
- W metodzie `main` uruchomić nowo stworzony wątek korzystając z metody `run`
- W metodzie `main` stworzyć nową pulę `10 wątków`
- Na nowo utworzonej puli uruchomić wcześniej utworzone zadanie `Runnable`
- Pamiętaj o zamknięciu puli wątków!

Problemy wielowątkowości

Zastosowanie wielowątkowości może przyśpieszyć działanie naszego programu. Jednakże wielowątkowy kod który jest źle napisany może prowadzić do nieprzywidywalnych problemów jak i blokad naszej aplikacji. Napopularniejsze błędy związane z wielowątkowością to:

- **race condition** - wyścig, w którym inny niepożądany wątek jako pierwszy dostał się do sekcji krytycznej
- **deadlock** - zakleszczenie, zablokowanie wątku w oczekiwaniu na zakończenie pracy
- **starvation** - zagłodzenie, w którym wątki o mniejszym priorytecie nigdy nie zaczną swojej

pracy

Race condition - wyścig

Uruchomym poniższy kod kilka razy:

```
class Counter {
    private int value;

    public void increment() {
        value += 1;
    }

    public int getValue() {
        return value;
    }
}

public class RaceCondition {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) {
                c.increment();
            }
        };

        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        Thread t3 = new Thread(r);

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        System.out.println(c.getValue());
    }
}
```

Jaki wynik otrzymałeś? Prawdopodobnie wynik za każdym razem był inny. Jest to związane z tak zwany wyścigiem ang. *race condition*, w którym wątki wywłaszczają czas procesora jeden po drugim, posiadając lokalną kopię w swojej przestrzeni pamięci. Rozwiązaniem problemu wyścigu jest synchronizacja.

DeadLock - zakleszczenie

Deadlock jest sytuacją gdy wątki czekając na siebie nawzajem, blokują swoją pracę. Bardzo często prowadzi to do zablokowania aplikacji, lub ograniczenia jej działania:

```
public class Deadlock {

  public static void main(String[] args) {

    final String resource1 = "resource1";
    final String resource2 = "resource2";

    Thread t1 = new Thread() {
      public void run() {
          synchronized (resource1) {
           System.out.println("Thread 1: locked resource 1");

           try { Thread.sleep(100);} catch (Exception e) {}

           synchronized (resource2) {
            System.out.println("Thread 1: locked resource 2");
           }
         }
      }
    };

    Thread t2 = new Thread() {
      public void run() {
        synchronized (resource2) {
          System.out.println("Thread 2: locked resource 2");

          try { Thread.sleep(100);} catch (Exception e) {}

          synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };


    t1.start();
    t2.start();
  }
}
```

Deadlock jest łatwy do wykrycia w działającej aplikacji

Starvation - zagłodzenie

Zagłodzenie opisuje sytuacje w której wątek nie jest w stanie wykonywać swojej pracy, ponieważ współdzielone zasoby są zablokowane. Bardzo często blokowane są przez inny wątek, który blokuje te zasoby na dłuższy okres czasu. Możemy sobie wyobrazić sytuację w której jeden wątek wykonuje operację aktualizacji danych co minutę. Proces ten trwa około trzydziestu sekund, następnie inne wątki chcą pobrać najnowsze dane, mają więc tylko trzydzieści sekund aby to zrobić. W skrajne sytuacji mogą tego nie zrobić nigdy co doprowadziło do **zagłodzenia** innych wątków.

Synchronizacja

Jednym z rozwiązań problemów związanych z **wielowątkowością** jest **synchronizacja**. **Synchronizacja** jest procesem, w którym wykorzystując wbudowane mechanizmy sprawiamy, iż nasza praca wykonuje się w pożądanych przez nas sposób. **Java** dostarcza wiele mechanizów, pozwalających zrealizować synchronizację. My skupimy się na dwóch:

- słowo kluczowe `synchronized`
- blokady - locki

Synchronized

Pierwszym sposobem realizacji **synchronizacja** jest użycie słowa kluczowego `synchronized`. Słowo to można umieścić w dwóch miejscach:

- jako słowo kluczowe w nagłówku metody
- jako oznaczenie bloku synchronizowanego

Metoda

Synchronized

```
public synchronized void synchronizedMethod() {
    // logika
}
```

Blok

Blok synchronized

```
public void synchronizedBlock() {
    synchronized(this) {
        // logika
    }
}
```

Zadania

- W projekcie `threads-example` stworzyć klasę `SynchronizedRunner` z `psvm`
- W metodzie `main` stworzyć nową pulę `10 wątków`
- Stworzyć klasę `Synchronized`
- W klasie `Synchronized` dodać dwie metody `synchronizedMethod` oraz `synchronizedBlock`
- W nagłowku metody `synchronizedMethod` wykorzystać słowo kluczowe `synchronized`
- W metodzie `synchronizedMethod` na wejściu wypisać nazwę wątków, zatrzymać wątek na trzy sekundy
- W metodzie `synchronized` na wejściu wypisać nazwę wątków, w bloku synchronizowanym wypisać nazwę aktualnie przetwarzanego wątku, zatrzymać wątek na trzy sekundy na końcu wypisać nazwę wątku wychodzącego
- Wywołać klasę `Synchronized` w metodzie `main`

Locks

Drugim sposobem synchronizacji jest wykorzystanie interefejsu `Lock`. Interfejs ten dostarcza kilka metod oraz implementacji pozwalających zrealizować synchronizację. Pierwszym z nich jest `ReentrantLock`:

ReentrantLock

```
public class SomeClass {

    ReentrantLock lock = new ReentrantLock();

    public void synchronizedMethod() {
        lock.lock();
        try {
            // sekcja krytyczna
        } finally {
            lock.unlock(); // pamiętamy o odblokowaniu
        }
    }
}
```

Interfejs `Lock` dostarczył metody `lock` oraz `unlock`. Pierwsza z nich zakłada blokadę na sekcj krtycznej natomiast druga tą blokadę ściąga.

Pamiętam aby umieścić metodę `unlock` w sekcji `finally`

W przypadku podziału operacji na **zapis/odczyt** możemy użyć bardziej wyszukanego mechanizmu jakim jest `ReentrantReadWriteLock`. Oferuje on dwie blokady, jedną na zapis a drugą na odczyt. Co ciekawe, w momencie założenia blokady na sekcji zapisu nie mogą być wykonywane żadne operacje zapisu oraz na odwrót:

ReentrantReadWriteLock

```
public class ReadWriteLock {

    Map<String,String> syncHashMap = new HashMap<>();
    ReadWriteLock lock = new ReentrantReadWriteLock();

    Lock readLock = lock.readLock();
    Lock writeLock = lock.writeLock();

    public void put(String key, String value) {
        try {
            writeLock.lock();
            syncHashMap.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }

    public String get(String key){
        try {
            readLock.lock();
            return syncHashMap.get(key);
        } finally {
            readLock.unlock();
        }
    }

}
```

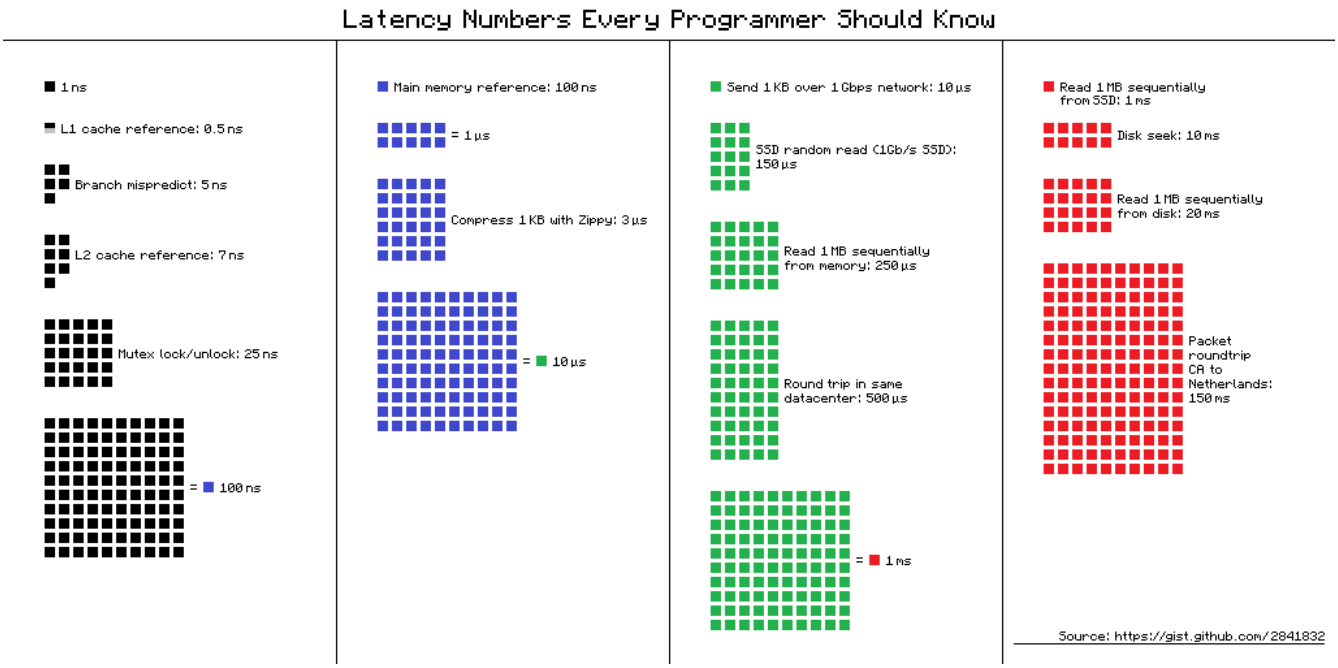
Zadania

- W projekcie `threads-example` stworzyć klasę `LocksRunner` z `psvm`
- W metodzie `main` stworzyć nową pulę `10 wątków`
- Stworzyć klasę `Locks`
- Stworzyć klasę `RWLocks`
- W klasie `Locks` dodać `ArrayList` przechowującą obiekty typu `String`
- W klasie `RWLocks` dodać `ArrayList` przechowującą obiekty typu `String`
- W klasie `Locks` dodać metody `add` i `get` dla dodanej listy (operacja dodawnia i odczytu z listy, zatrzymać wątek na trzy sekundy)
- W klasie `RWLocks` dodać metody `add` i `get` dla dodanej listy (operacja dodawnia i odczytu z listy, zatrzymać wątek na trzy sekundy)
- W klasie `Locks` zabepieczyć metody `add` i `get` korzystając z `ReentrantLock`
- W klasie `Locks` w metodach `add` i `get` wypisać wejście, przetwarzanie i wyjście wątku
- W klasie `RWLocks` zabepieczyć metody `add` i `get` korzystając z `ReentrantReadWriteLock`

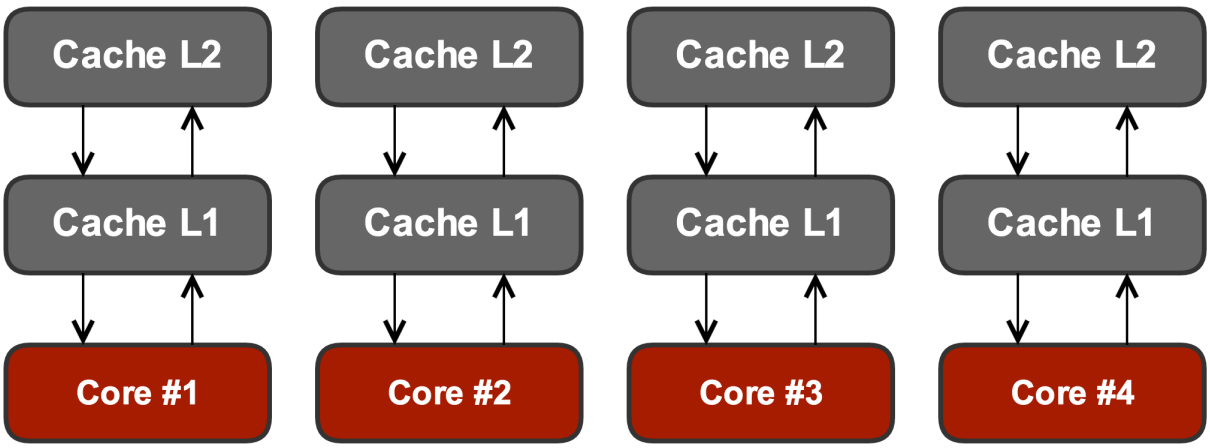
- W klasie `RWLocks` w metodach `add` i `get` wypisać wejście, przetwarzanie i wyjście wątku
- Wywołać klasę `Locks` i `RWLocks` w metodzie `main`

volatile

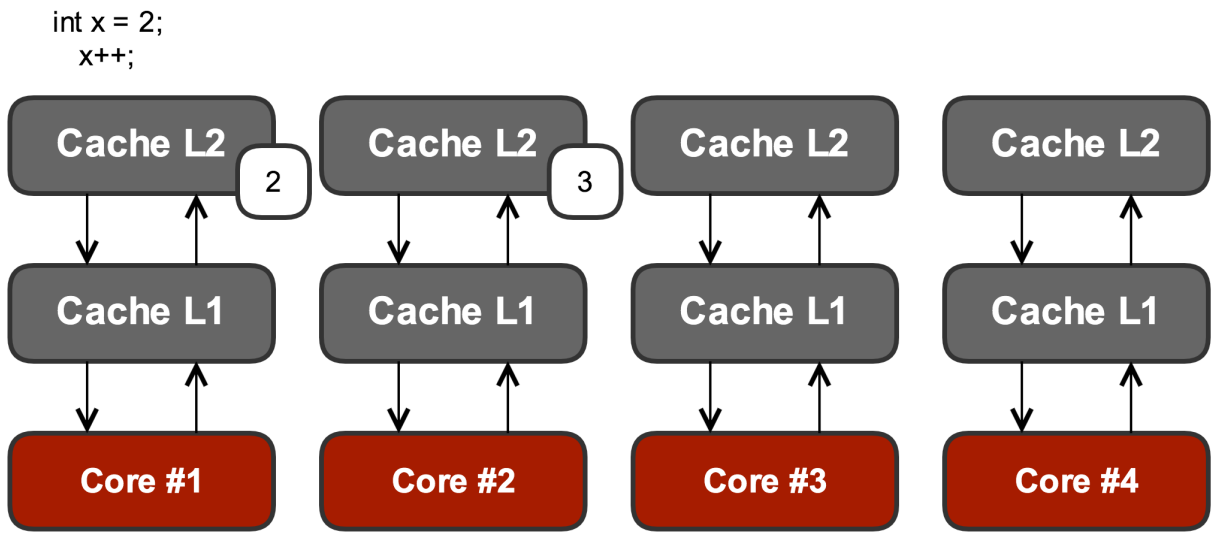
Odczyt z pamięci jest bardzo kosztownym procesem:



Dlatego też, aby uniknąć **częstego** odpytywania "odległych" obszarów pamięci wątki **alokują** pamięć "jak nabliżej" siebie:



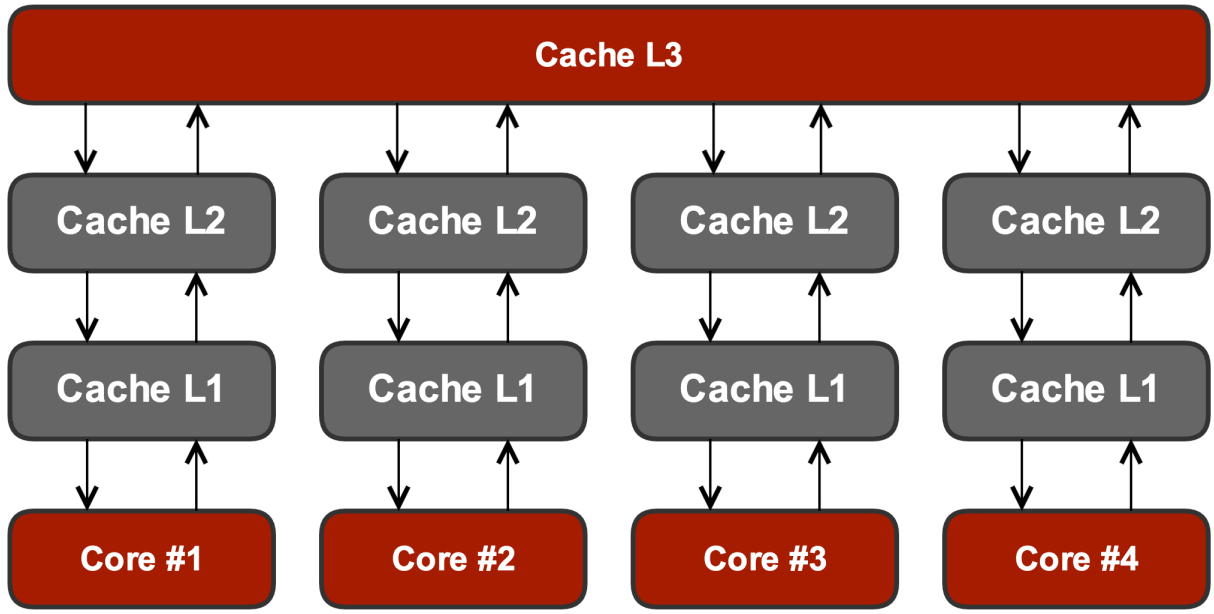
Może powodować to **problemy** z niespójnymi danymi:



Rozwiązaniem tego problemu jest użycie słowa kluczowego `volatile`:

```
volatile SomeObject someName;
```

Powoduje to **umieszczenie** wartości bezpośrednio w **cache L3**:



Zadania

- Naprawić poniższy kod:

```
public class HowToUnlock {
   private static boolean enabled = true;

   public static void main(String... args) throws InterruptedException {
      new Thread(HowToUnlock::loop).start();
      Thread.sleep(1000);
      enabled = false;
      System.out.println("Shutdown");
   }

   private static void loop() {
      System.out.println("Start");
      while (isEnabled()) {}
      System.out.println("End");
   }

   private static boolean isEnabled() {
      return enabled;
   }
}
```

Debugowanie

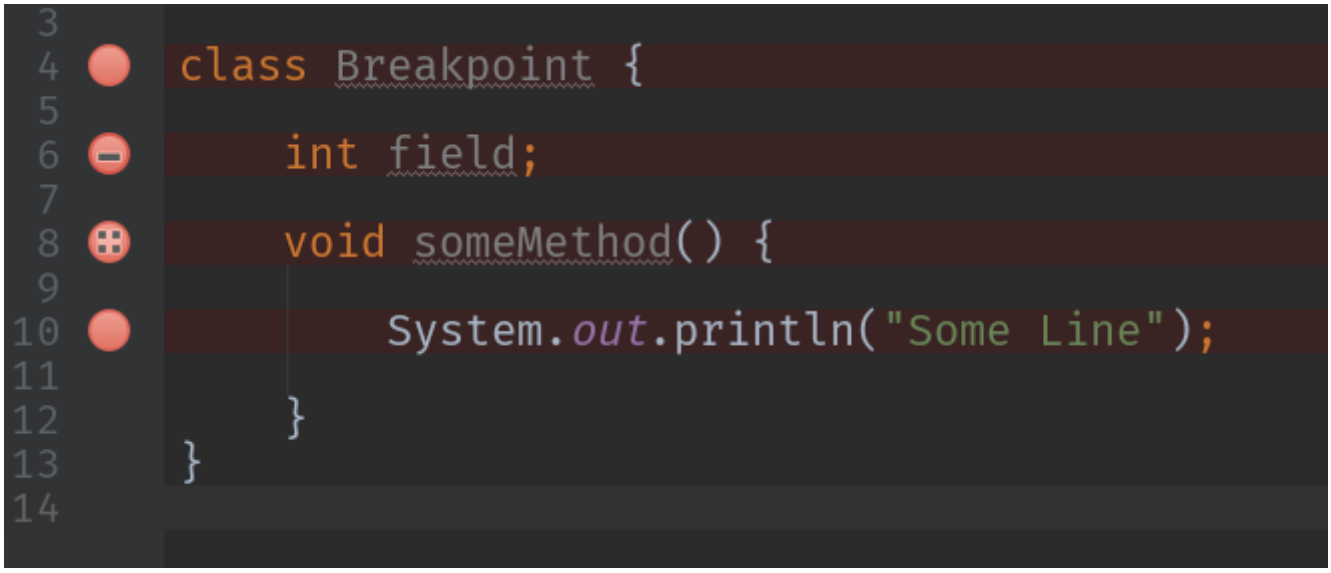
Debugowanie jest bardzo przydatną (jak nie, obowiązkową) umiejętnością każdego programisty. Jest to proces śledzenia kodu w celu wykrycia **błędu** lub **zrozumienia algorytmu**. Podczas tego procesu możemy podstawiać inne dane niż te w aktualnych wykonaniu aby sprawdzić inne gałęzie programu.

Uruchomienie aplikacji w trybie **debug** może bardzo zwolnić aplikację!

Breakpoint

Miejsce, w którym zatrzymać ma się aplikacja nazywamy **breakpoint**. W **IntelliJ** jest to taka duża czerwona kropka:

Breakpoint można wstawiać na różnych miejsach w kodzie:

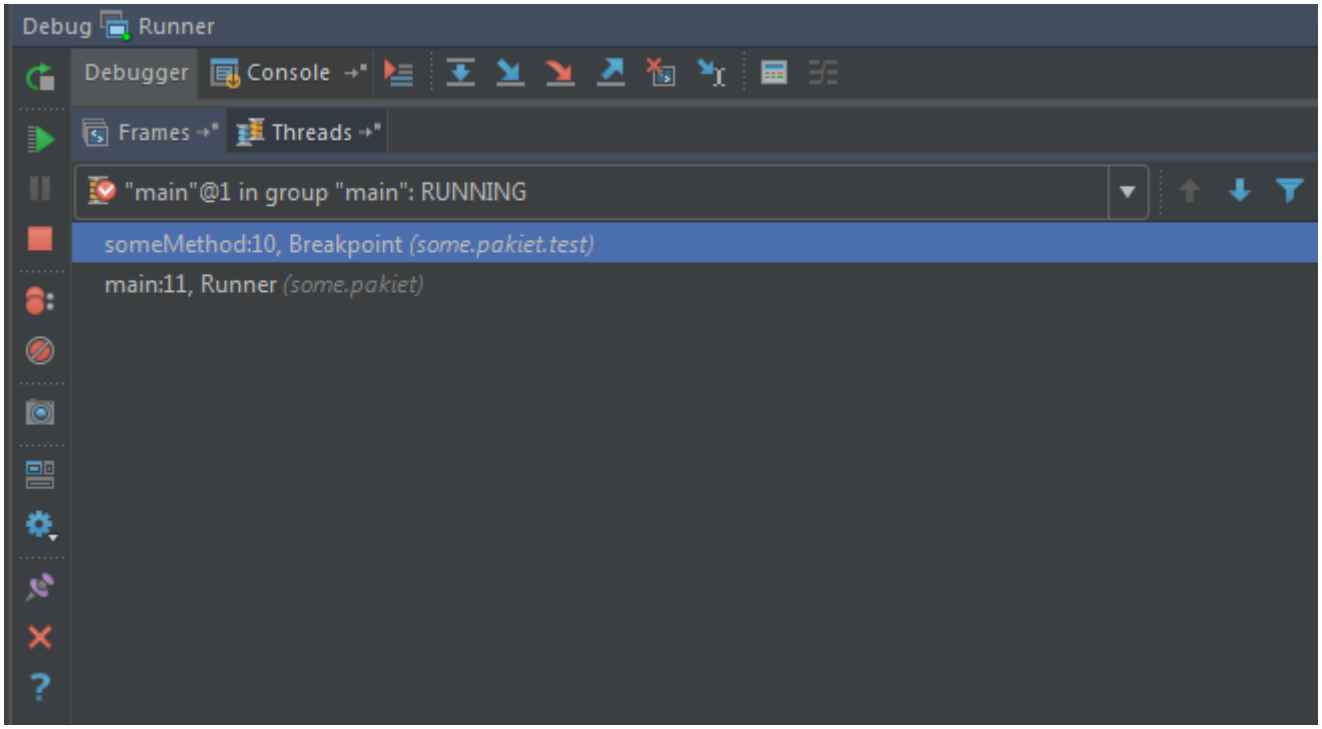


Odpowiednie ustawienie działa inaczej w zależności od wybranego elementu:

- **na klasie** - zostanie złapane każde wywołanie na tej klasie
- **na metodzie** - zostanie złapane każde wywołanie na tej metodzie
- **na polu** - zostanie złapane każde wywołanie na tym polu
- **na linii** - zostanie złapane w linii umieszczenia

Ramki

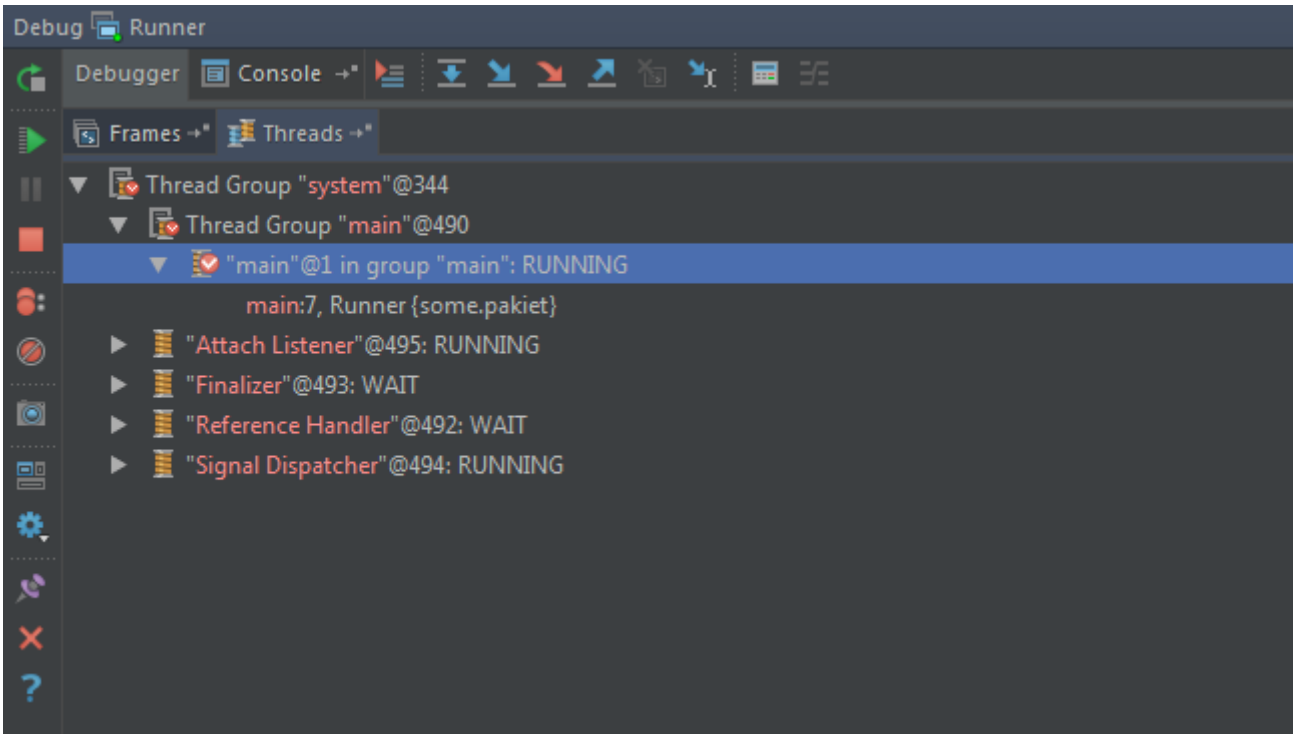
Przy każdym wywołaniu kodu "odkładana" jest ramka na stos. W widoku poniżej możemy sprawdzić cały stos wywołań:



Często istnieje także możliwość cofnięcia wywołania tak zwane **zrzucenie ramki**.

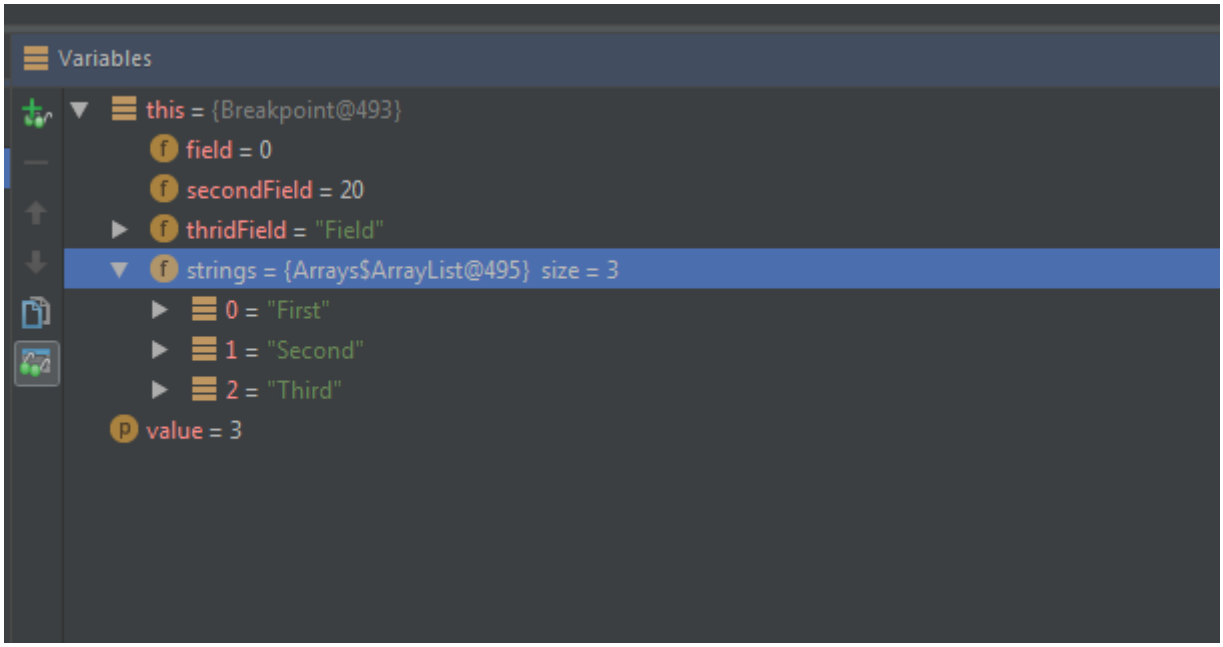
Wątki

IntelliJ daje nam możliwość podejrzenia, który wątek jest aktualnie wykonywany:



Zmienne

Podczas **debugowania** możemy podejrzeć wszystkie pola w aktualnej instancji klasy:

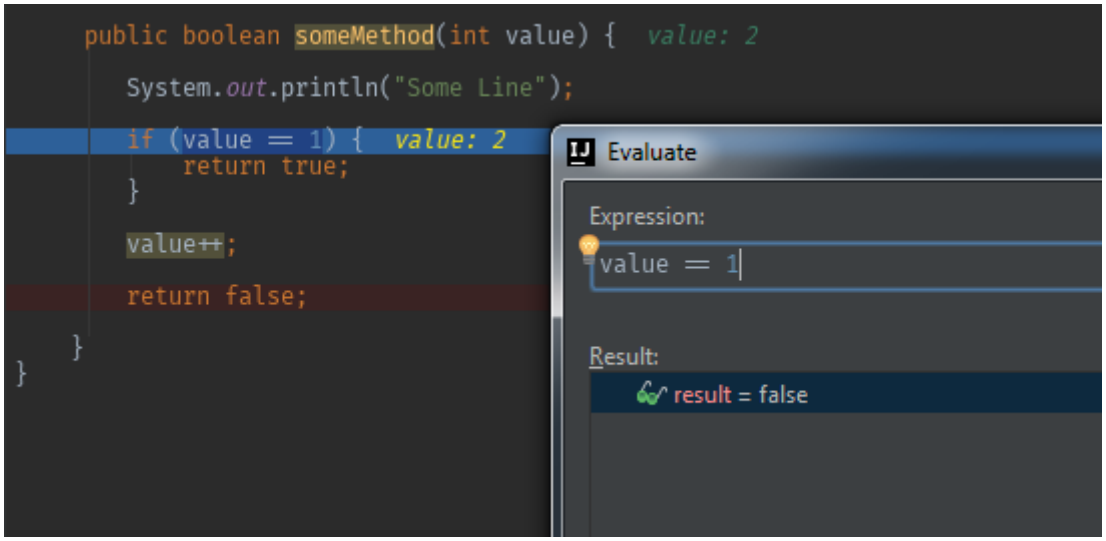


Ponadto, **IntelliJ** w kodzie pokazuje nam aktualne wartości:

```
public class Breakpoint {

    int field;  field: 0
    int secondField = 20;  secondField: 20
    String thridField = "Field";  thridField: "Field"
    List<String> strings = Arrays.asList("First", "Second", "Third");  strings:  size = 3

     public boolean someMethod(int value) {  value: 3

        System.out.println("Some Line");

        if (value == 1) {
            return true;
        }

        value++;  value: 3

        return false;

    }
}
```

Wywołanie

Jeśli chcemy wywołać pewień fragmentu kodu wystarczy, że go zaznaczymy i już możemy wykonać **ewaluację wyrażenia**:



Skróty

Bardzo ważną umiejętnością podczas **debugowania** jest umiejętność korzystania ze skrótów:

- `CTRL+F9` - uruchomienie kodu w trybie **debug**
- `ALT+F8` - ewaluacja wyrażenia (można też wykonać to wciskając `ALT`)
- `F7` - wejście do środka
- `F8` - przejście linia niżej
- `F9` - skok do następnego breakpoint'u
- `CTRL+SHIFT+F8` - więcej informacji na temat breakpoint'ów

Zadania

- Stworzyć nowy projekt **Maven** z nazwą `debug-example`
- Utworzyć klasę `Job`:

Job.java

```
class Job implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

}
```

- Utworzyć klasę `Debug`:

Debug.java

```
class Debug {

    int value = 20;

    int divisible(int number) {
        value++;
        if (number % 2 == 0) {
            return 2;
        } else if (number % 5 == 0) {
            return 5;
        } else if (number % 7 == 0) {
            return 7;
        } else {
            value++;
            return 0;
        }
    }

}
```

- Utworzyć klasę `Runner` z `psvm`:

Runner.java

```
class Runner {

    public static void main(String[] args) throws InterruptedException {
        new Debug().divisible(7);
        new Thread(new Job()).start();
        while(true) {
            Thread.sleep(5000);
            System.out.println("Janusz");
        }
    }

}
```

- Postaw breakpoint na linii `value++;`
- Postaw breakpoint na linii `System.out.println(Thread.currentThread().getName());`
- Uruchom aplikacje w trybie **debug**
- Po zatrzymaniu się na linii `value++;` użyj `F8`
- Obserwuj wartość pola `value`
- Wykonaj linie `number % 2 == 0` użyj `ALT`
- Debuguj aż dojedziesz do linii `number % 7 == 0`, zrzuć ramkę

- Przejdź do breakpoint'a na `System.out.println(Thread.currentThread().getName());`
- Podejrzyj aktualne wątki
- Postaw breakpoint na linii `System.out.println("Janusz");`
- Przejdź do kolejnego breakpoint'u

OOP (Object Oriented Programming)

OOP jest **paradygmatem programowania obiektowego**, w którym programy definiuje się za pomocą **obiektów**. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Obiekt

Obiekt jest elementem łączącym **stan** i **zachowania**. W **Javie** obiektem jest **klasa**. Każdy **obiekt** składa się z czterech elementów:

- **tożsamość**
- **struktura**
- **stan**
- **zachowanie**

Tożsamość

Jest cechą umożliwiającą **identyfikację** i **odróżnienie** od innych obiektów. W środowisku obiektowego języka programowania **tożsamość** obiektu realizowana jest przez **unikatowy odnośnik (referencję) do obiektu**, dzięki któremu można się jednoznacznie do niego odwoływać. Odnośnik do obiektu może być implementowany na różne sposoby np. jako wskaźnik (adres).

Struktura

Każdy obiekt posiada **strukturę** określaną przez dostępne pola (atrybuty). Przykładowo obiekt posiada dwa pola, liczbę użytkowników oraz nazwę systemu. Oznacza to, iż struktura obiektu zbudowana jest z dwóch elementów, liczby i nazwy.

Stan

Wartości pól opisanych w strukturze określają **stan** obiektu. Przykładowo obiekt posiada pole przechowujące liczbę użytkowników w systemie. Jeśli ktoś odpyta o wartość tego pola to tak naprawdę zadaje pytanie "w jakim stanie znajduje się aktualnie ten obiekt?".

Zachowanie

Zachowania związane z obiektem realizowane są poprzez **metody**, które zazwyczaj związane są ze **stanem obiektu**. Przykładowo obiekt przechowujący liczbę użytkowników w systemie posiada zachowanie, które potrafi zwrócić tą liczbę.

Założenia

Abstrakcja

Abstrakcją w programowaniu nazywamy pewnego rodzaju uproszczenie rozpatrywanego

problemu, polegające na ograniczeniu zakresu cech manipulowanych obiektów wyłącznie do cech kluczowych dla algorytmu, a jednocześnie niezależnych od implementacji. Cel stosowania abstrakcji jest dwojaki: ułatwienie rozwiązania problemu i zwiększenie jego ogólności.

Hermetyzacja

Hermetyzacja określana jest również jako **enkapsulacja** lub **kapsułkowanie**. Jest to mechanizm pozwalający na **ukrywanie** stanu oraz zachowań wewnątrz obiektu. Element obiektu ukrywane są poprzez zadeklarowanie ich jako **prywatne** lub **chronione**.

Polimorfizm

Polimorfizm jest to słowo pochodzące z greki oznaczające **wielopostaciowość**. Obiekty jednego typu mogą mieć wiele postaci. Dzięki temu możliwe jest **wyabstrahowanie wyrażeń** od konkretnych typów. Referencje i kolekcje obiektów mogą dotyczyć **obiektów różnego typu**, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla **pełnego typu obiektu wywoływanego**.

Dziedziczenie

Porządkuje i wspomaga polimorfizm i enkapsulację dzięki umożliwieniu definiowania i tworzenia specjalizowanych obiektów na podstawie bardziej ogólnych. Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy.

SOLID

SOLID jest zbiorem **zasad**, których stosowanie sprawia, że nasz kod staje się lepszy. Są to zasady zebrane przez **Roberta C. Martina** (znanego jako wujek Bob). Akronim **SOLID** został zaprezentowany przez **Michaela Feathersa**. W skład tego akronimu wchodzą:

- S - Single Responsibility Principle
- O - Open Close Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

SRP - Single Responsibility Principle

Pierwsza z reguł to **SRP**. Jest to reguła, która głosi:

Klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy).

Podczas programowania powinniśmy dążyć do tego, aby nasze klasy miały jak najmniejsze odpowiedzialności (aby robił jak najmniej). Dzięki temu podczas wystąpienia nieuniknionych zmian, będziemy mieli pewność, że nie popsujemy czegoś dookoła:

Naruszenie SRP

```
public class FileManager {

    String fileName = "file.txt";

    public void printFile() {
        // logika bazująca na fileName
    }

    public void writeToFile() {
        // logika bazująca na fileName
    }

}
```

Powyższy kod prezentuje naruszenie zasady **SRP**. Wyobraźmy sobie sytuację, w której chcemy zmienić zachowanie metody `writeToFile`, przykładowo dodać możliwość zapisu do pliku `csv`. Istnieje bardzo duże prawdopodobieństwo, że podczas zmieniania odpowiedzialności zapisu klasy `FileManager` możemy popsuć działanie metody `printFile`. W tym przypadku istnieje **więcej niż jeden powód** do modyfikacji tej klasy. Rozwiązaniem tego problemu jest zastosowanie **SRP**, czyli rozbicie odpowiedzialności na klasy:

SRP

```
public class FilePrinter {

    String fileName = "file.txt";

    public void printFile() {
        // logika bazująca na fileName
    }

}

public class FileWriter {

    String fileName = "file.csv";

    public void writeToFile() {
        // logika bazująca na fileName
    }

}
```

OCP - Open Close Principle

Kolejna reguła to **OCP**. Jest to reguła, która głosi:

Klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje.

Chociaż opis tej reguły nie jest do końca jasny to spróbujmy przełożyć to na życiowy przykład. W momencie podjęcia decyzji o wybudowania domu zaczynamy jego budowę od fundamentów. Od teraz zamkneliśmy rozmiar naszego domu na modyfikacje (rozmiar jest już niezmienny) natomiast układ ścian jest dowolny. Dodatkowo możemy układać ściany w dowolny sposób (otwarty na rozszerzenia) **bez potrzeby zmiany fundamentów**. W przyszłości możemy wyburzyć ściany ale fundamenty dalej zostają.

Naruszenie OCP

```
public class FilePrinter {

    private String type;

    public FilePrinter(String type) {
        this.type = type;
    }

    void print(String fileName) {
        if ("CSV".equals(type)) {
            System.out.println("CSV");
        } else if ("TXT".equals(type)) {
            System.out.println("TXT");
        } else {
            System.out.println("Zły typ");
        }
    }

}
```

Powyżej przedstawiony jest kod, który narusza zasadę **OCP**. W momencie dodania nowego typu pliku, musimy zmodyfikować klasę `FilePrinter` (nasze fundamenty). Spróbujmy naprawić ten kod stosując **OCP** co w **Javie** realizowane jest za pomocą abstrakcji (klasa abtrakcyjna lub interfejs):

OCP

```
public class FilePrinter {

    void print(FileProvider provider) {
        System.out.println(provider.getText());
    }

}

interface FileProvider {

    String getText();

}

class CsvProvider implements FileProvider {

    String getText() {
        return "CSV";
    }

}

class TxtProvider implements FileProvider {

    String getText() {
        return "CSV";
    }

}
```

Teraz w przypadku nowego typu `FileProvider` nie zmieniamy naszej głównej klasy. Jesteśmy zamknięci na zmiany, ale otwarci na modyfikacje.

LSP - Liskov Substitution Principle

Trzecia reguła to **LSP**. Jest to reguła, która głosi:

Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.

W przypadku stosowania tej reguły nie powinniśmy zauważyć różnicy między używaniem podtypu i typu bazowego:

Naruszenie LSP

```
public interface FileOperator {

    void readFile();
    void writeToFile();

}
```

Przy wykorzystaniu powyższego **interfejsu** chcielibyśmy stworzyć klasę z **jedną** odpowiedzialnością związana z odczytywaniem plików:

Efekt naruszenia LSP

```
public class FileReader implements FileOperator {

    void readFile() {
        // logika
    }

    void writeToFile() {
        throw new UnsupportedOperationException();
    }

}

public class FileReaderWriter implements FileOperator {

    void readFile() {
        // logika
    }

    void writeToFile() {
        // logika
    }

}

public class Client {

    void someMethod() {
         List<FileOperator> files = new ArrayList<>();
         files.add(new FileReader());
         files.add(new FileReaderWriter());

         files.forEach(file -> file.read());
         files.forEach(file -> file.writeToFile()); //wyjątek naruszenie LSP
    }

}
```

Aby spełnić zasadę **LSP** zastosujemy zasadę **ISP**:

LSP

```
public interface Reader {

    void readFile();

}

public interface Writer {

    void writeToFile();

}

public class FileReader implements Reader {

    void readFile() {
        // logika
    }

}

public class FileWriter implements Writer {

    void writeToFile() {
        // logika
    }

}

public class Client {

    void someMethod() {
         List<Reader> fileReaders = new ArrayList<>();
         files.add(new FileReader());
         List<Writer> fileWriters = new ArrayList<>();
         files.add(new FileWriter());

         fileReaders.forEach(file -> file.read());
         fileWriters.forEach(file -> file.writeToFile());
    }

}
```

ISP - Interface Segregation Principle

Jest to reguła, która głosi:

Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.

Jest to reguła, która jest bardzo podobna do reguły **SRP**. Dotyczy ona natomiast **interfejsów**. Stosując **interfejsy** w naszych projektach powinniśmy dbać o to aby miały one dokładnie jedną **odpowiedzialność**. Dla przykładu:

Naruszenie ISP

```
public interface FileOperator {

    void readFile();
    void writeToFile();

}
```

Przy wykorzystaniu powyższego **interfejsu** chcielibyśmy stworzyć klasę z **jedną** odpowiedzialnością związana z odczytywaniem plików. Bez zastosowania **ISP** musimy nadmiarowo nadpisać metodę `writeToFile`:

Efekt naruszenia ISP

```
public class FileReader implements FileOperator {

    void readFile() {
        // logika
    }

    void writeToFile() {
        // co tutaj?
    }

}
```

Rozwiązaniem tego **naruszenia** jest zastosowanie zasady **Interface Segregation Principle**. W **Javie** możemy to zrealizować za pomocą **dziedziczenia** interfejsów:

ISP

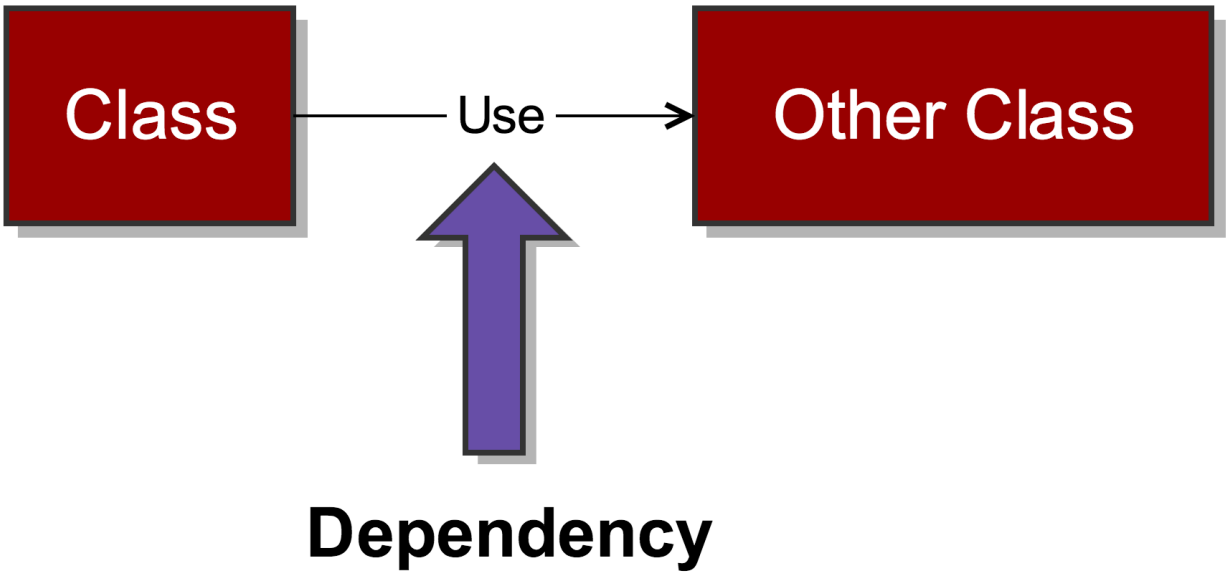
```
public interface Reader {

    void readFile();

}

public interface Writer {

    void writeToFile();

}

public interface FileOperator extends Reader, Writer {

}

public class FileReader implements Reader {

    void readFile() {
        // logika
    }

}
```

DIP - Dependency Inversion Principle

Jest to reguła, która głosi:

Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych - zależności między nimi powinny wynikać z abstrakcji.

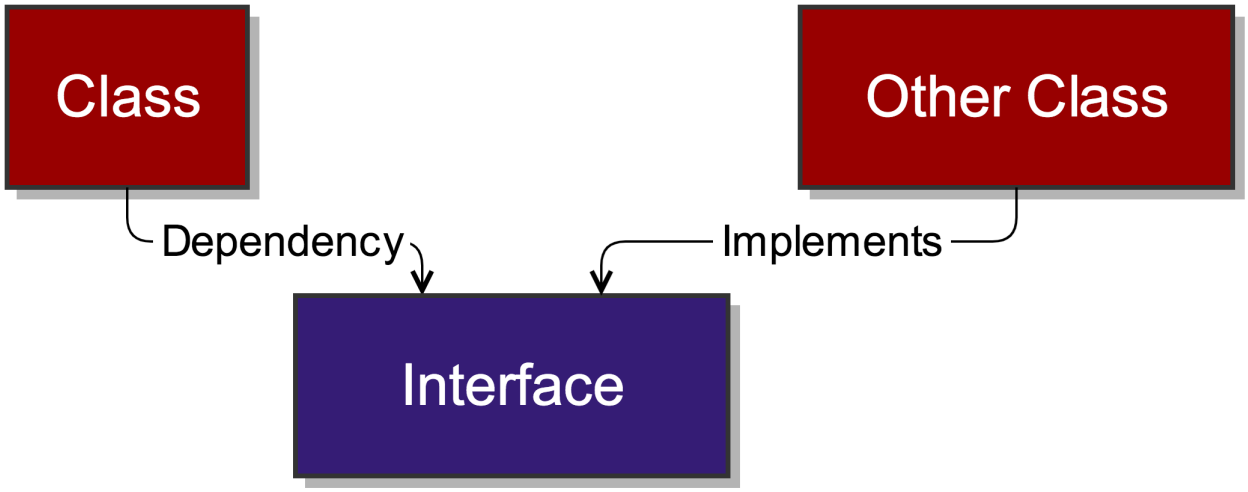
Podczas programowania z zachowaniem **DIP** obiekt nie tworzy obiektów, które wykorzystywane są wewnątrz. Dzięki temu, **nie wiążemy się** z konkretną implementacją (najlepiej operować na **interfejsach**), a także nie musimy znać parametrów **konstruowanego** obiektu. Operując na **interfejsach** stajemy się niezależni od konkretnej implementacji. W podejściu bez stosowania **Dependency Inversion Principle** nasz kod wygląda następująco:



Car.java

```
class Car {

    //Concrete class
    private WinterTire tire = new WinterTire(195);

    void drive() {
        tire.turn();
    }

}
```

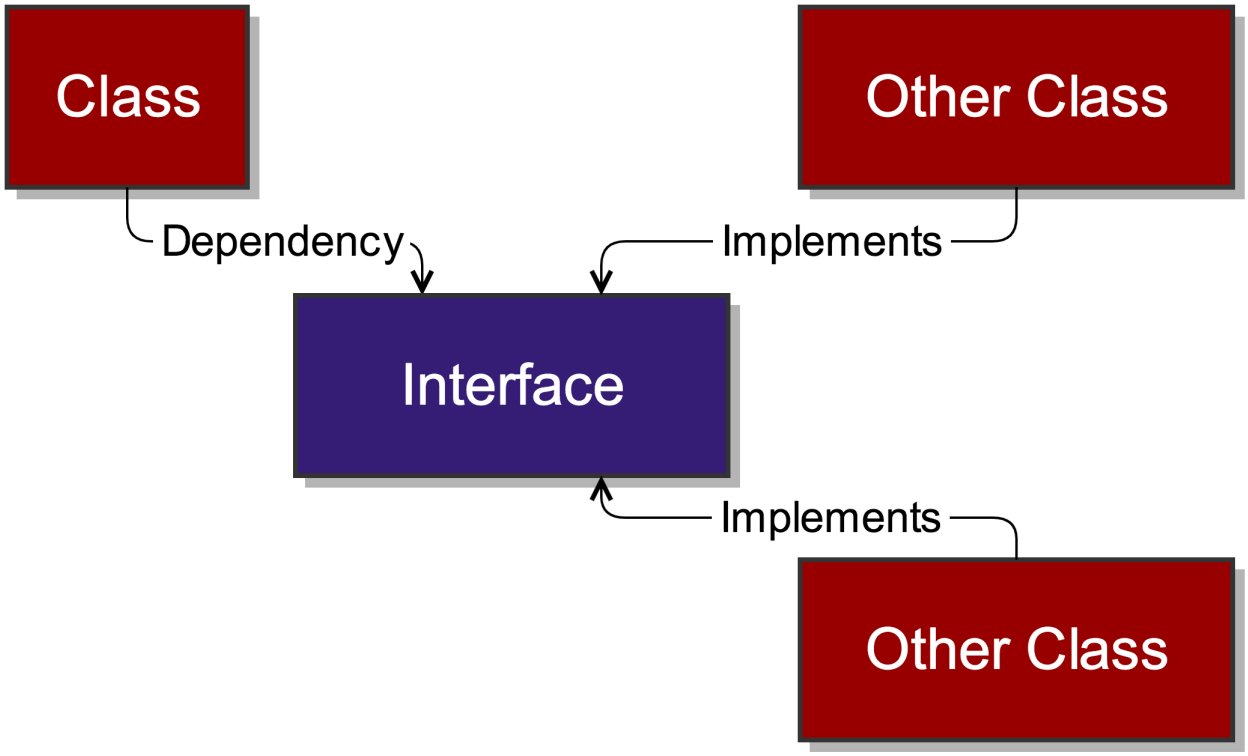
Obiekt główny, czyli `Car`, odpowiedzialny jest za stworzenie obiektu, który wykorzystywany jest wewnątrz (klasa `WinterTire`). Ponadto musi znać jego parametry (wartość `195`) w momencie tworzenia. Wyobraźmy sobie teraz sytuację, w której chcemy stworzyć samochód z innymi oponami, musimy stworzyć nową klasę `Car` (na przykład `CarWithSummerTire`) i w niej tworzymy konkretną implementację. Natomiast po zastosowaniu **DIP**:



Car.java

```
class Car {

    private Tire tire;

    Car(Tire tire) {
        this.tire = tire;
    }

    void drive() {
        tire.turn();
    }

}
```

W drugim fragmencie kodu zastosowaliśmy **wstrzykiwanie przez konstruktor**. Teraz wyobraźmy sobie, że testujemy nasz kod. W bardzo prosty sposób w klasie testowej możemy wstrzyknąć implementację interfejsu `Tire`, która będzie działać jak zaślepka rzeczywistej klasy. Ponadto, jeśli zdecydujemy się dodać nową implementację interfejsu `Tire` nasza główna klasa nie ulegnie zmianie, co oznacza, że nie jesteśmy związani z konkretną implementacją, a o to nam chodziło.



Inne

Oprócz zasady **SOLID** istnieje wiele bardzo popularnych akronimów, które kryją w sobie lekcje. Najpopularniejsze **akronimy** to:

- **KISS**
- **DRY**
- **YAGNI**

KISS

KISS jest akronimem od ang. *Keep It Simple, Stupid*. Jest to reguła, który mówi o tym, że nasz kod powinien być jak najbardziej prosty. Dzięki stosowaniu tej reguły, nikt nie będzie miał problemów ze zrozumieniem naszego kodu. Musimy pamiętać o zasadzie **Pareta 80/20**. W kontekście programowania **80%** czasu spędzamy na czytaniu i analizie kodu natomiast na samo pisanie zostaje nam tylko **20%** czasu.

KISS doczekał się swojej polskiej wersji **BUZI** (Bez Udziwnień Zapisu, Idioto)

DRY

DRY jest akronimem od ang. *Don't repeat yourself*. Jest to reguła, który mówi o tym aby unikać powtórzeń w kodzie. Unikanie **powtórzeń** powinno odbywać się na każdym poziomie projektu, zaczynając od prostych metod a kończąc na procesie ciągłej integracji.

YAGNI

YAGNI jest akronimem od ang. *You aren't gonna need it*. Jest to reguła, który mówi o tym, że bardzo często w naszym kodzie piszemy kod "na przyszłość". Dodajemy duże ilości dodatkowej funkcjonalności, która może okazać się w **przyszłości** przydatna. Niestety jak pokazuje praktyka, bardzo rzadko wracamy do tego kodu a dalej należy go utrzymywać!

Wzorce projektowe

Wzorce projektowe są gotowymi rozwiązaniami powtarzających się problemów. Zaprojektowane są one w bardzo uniwersalny sposób sprawiający, że znajdują zastosowanie w dowolnym języku realizującym paradygmat programowania obiektowego. Bardzo dobrą praktyką dla programistów jest ich znajomość jak i stosowanie. Istnieje bardzo dużo **wzorców projektowych**, które klasyfikuje się w trzech kategoriach:

- **kreacyjne** - rozwiązujące problemy związane z tworzeniem obiektów
- **strukturalne** - rozwiązujące problemy związane ze strukturą obiektów
- **behawioralne** - rozwiązujące problemy związane z zachowaniem obiektów

Singleton

Pierwszy z opisywanych **wzorców** należy do kategorii **kreacyjnych**. Słowo *single* z angielskiego oznacza jeden, w przypadku tego **wzorca** chcielibyśmy mieć stworzoną (dlatego **kreacyjny**) dokładnie **jedną** instancję danego obiektu. Bardzo często jako przykład zastosowania tego wzorca podaje się połączenie do **bazy danych**. Nawiązywanie połączenia jest bardzo **kosztownym** procesem dlatego też lepiej jest trzymać i używać ponownie tego jednego połączenia. Istnieją dwa sposoby realizacji tego wzorca w **Javie**, my zajmiemy się tym **najpopularniejszym**:

Singleton

```
public final class Singleton {

    // należy zwrócić uwagę na użycie słowa kluczowego volatile
    private static volatile Singleton instance = null;

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    // żeby uniknąć automatycznego tworzenia domyślnego, publicznego, bezargumentowego
konstruktora
    private Singleton() {
    }

}
```

Singleton bardzo często przedstawiany jest również jako **antywzorzec**. Przypomnijmy sobie o

założeniach **programowania obiektowego**, które mówi między innymi o tym, że obiekty powinny być **rozszerzalne**. W tym przypadku straciliśmy tą możliwość.

Fasada

Kolejnym opisywanym wzorcem jest **Fasada**. Jest to **wzorzec**, który ma na celu **ułatwić** klientom korzystanie z naszego API. Przykładowo jeśli w systemie znajduje się duży obiekt składający się z wielu mniejszych obiektów (ponieważ zastosowaliśmy wzorzec **Pyłek**) to stworzenie takiego obiektu może być bardzo skomplikowane. Aby nie utrudniać tego zadania klientom naszego API przygotowujemy **jeden** punkt wejścia do tworzenia tych obiektów, czyli **fasadę**:

Brak fasady

```
public class Client {

    BigObject someMethod() {
        FirstObject first = new FirstObject();
        SecondObject second = new SecondObject();
        ThirdObject third = new ThirdObject();
        FourthObject fourth = new FourthObject();
        return new BigObject(first, second, third, fourth);
    }

}
```

Fasada

```
public class Client {

    private BigObjectFacade facade = new BigObjectFacade();

    BigObject someMethod() {
        return facade.someMethod();
    }

}

public class BigObjectFacade {

    BigObject someMethod() {
        FirstObject first = new FirstObject();
        SecondObject second = new SecondObject();
        ThirdObject third = new ThirdObject();
        FourthObject fourth = new FourthObject();
        return new BigObject(first, second, third, fourth);
    }

}
```

Adapter

O wzorcu **adapter** powinniśmy myśleć jako o **przejściówce** dla dwóch niekompatybilnych interfejsów. Dzięki takiej przejściówce możliwa jest dalsza komunikacja:

.

```
public class USBReader {

    String readUSB(USB usb) {
        return usb.read(0, 0, 0);
    }

    String readUSB20(USB20 usb) {
        return usb.read(0, 0);
    }

}

public interface USB {

    String read(int x, int y, int z);

}

public class USB20 {

    String read(int x, int y) {
        return "USB20";
    }

}

public class USB30 implements USB {

    String read(int x, int y, int z) {
        return "USB30";
    }

}
```

Adapter

```
public class USB20Adapter implements USB {

    private final USB20 adaptee;

    public USB20Adapter() {
        this.adaptee = adaptee;
    }

    String read(int x, int y, int z) {
        return adaptee.read(x, y)
    }

}

public class USBReader {

    String readUSB(USB usb) {
        return usb.read(0, 0, 0);
    }

}
```

Strategia

Ostatnim opisywane wzorcem jest wzorzec **strategii**. Podobnie jak **singleton** jest to jeden z najpopularniejszych wzorców wykorzystywanych w kodzie obiektowym. Pozwala on na wymienne stosowanie algorytmów (strategii), które zamknięte są w postaci klas, niezależnie od korzystających z nich użytkowników:

Strategia

```
public interface Strategy {

    void someMethod();

}

public class StrategyA implements Strategy {

    void someMethod() {
        System.out.println("A");
    }

}

public class StrategyB implements Strategy {

    void someMethod() {
        System.out.println("B");
    }

}

public class Client {

    void someClientMethod(Strategy strategy) {
        strategy.someMethod();
    }

}
```

Test

1. Jak nazywa się kompilator Javy:
    a. javacompiler
    b. javac
    c. javak
    d. groovyc
2. Jakie rozszerzenie posiada skompilowany plik:
    a. class
    b. java
    c. compiled
3. Który z poniższych typów to package-scope:
    a. private
    b. public
    c. protected
    d. nic nie piszemy
4. Do uruchomienia programów w języku Java wystarczy:
    a. JDK
    b. JRE
5. Jakim poleceniem sprawdzamy wersje Javy:
    a. javaversion
    b. java-version
    c. java -v
    d. java -version
6. Który typ archiwum jest najpopularniejszy w Javie:
    a. CAR
    b. WAR
    c. JAR
    d. EAR
7. Czym jest artefakt:
    a. produktem końcowym procesu budowania
    b. rodzajem klasy z jednym konstruktorem
    c. metodą startową
8. Która z poniższych opcji jest prawdziwa:
    a. protected static void main(String args[])

b. public void static main(String args[])
c. public static void main(Integer args[])
d. Żadne z powyższych

9. Aby poprawnie skompilować plik Runner.java moja klasa powinna:
    a. nazywać się tak samo jak plik
    b. nazywać się tak samo jak plik ale nazwa powinna być pisana małą literą
10. Jaka jest wartość domyślna dla typu boolean:
    a. true
    b. false
11. Jak nazywa się ten mechanizm: int a = new Integer(2):
    a. Wrapping
    b. Boxing
    c. Autoboxing
    d. Autounboxing
12. Jaka jest domyślna wartość dla typu char:
    a. ' '
    b. ''
    c. 'a'
    d. '\u0000'
13. Poprzez jakie słowo realizujemy dziedziczenie:
    a. class
    b. extends
    c. public
    d. int
14. Który z poniższych typów przechowuje liczby całkowite:
    a. double
    b. int
    c. float
    d. long
    e. boolean
    f. char
    g. short
    h. byte
15. Czym jest Maven:
    a. kompilatorem

b. biblioteka do testowania
c. narzędziem do budowania
d. narzędziem do wskazywania zależności

16. Do czego służy plik MANIFEST.MF:
    a. Do przechowywania dodatkowych informacji o archiwum
    b. Do tworzenia klas
    c. Zawiera informacje o Maven
    d. Wymagany przy kompilacji
17. Jakim skrótem wstawiamy nową klase:
    a. alt+n/option+n
    b. alt+insert/command+n
    c. ctrl+F12/command+F12
18. W jakim folderze w IntelliJ przechowywane są informacje o projekcie:
    a. target
    b. src/main
    c. .idea
    d. src/main/test
19. W jakim folderze umieszczamy dodatkowe pliki zgodnie z konwencją Maven:
    a. src/main
    b. src
    c. src/test
    d. src/main/resources
20. Jak nazywa się plugin do podpowiadania skrótów:
    a. ShortCutter
    b. Key Promoter X
    c. Keys
    d. KeyMapper
21. W jakim pliku umieszczamy zależności do projektu:
    a. ron.xml
    b. maven.xml
    c. Runner.java
    d. pom.xml
22. Jak nazywa się lokalne repozytorium:
    a. .m3
    b. .mvn

c. .m2

d. .repo

23. Cykl Maven odpowiedzialny za budowanie dokumentacji to:
    a. default
    b. clean
    c. site
    d. doc
24. Która z poniższych faz tworzy archiwum:
    a. package
    b. clean
    c. jar
    d. archive
25. Pola znajdują się w:
    a. metodzie
    b. klasie
    c. w psvm
    d. Maven
26. Do czego używamy importów:
    a. Aby wskazać gdzie znajduje się dana klasa
    b. Aby mieć nowe zależności
    c. Jest to wymaganie Maven
27. Jak tworzymy nową instancje klasy:
    a. Car car;
    b. Car car = "4, Maluch";
    c. Car car = new Car();
    d. Car car = instance Car();
28. Jak nazywa się biblioteka do testowania:
    a. JUnit
    b. JavaUnit
    c. JIntegration
    d. JMH
29. Jak nazywa się biblioteka do asercji z fluent-assertions:
    a. FestAssert
    b. FluentAssertions
    c. AssertJ

d. Asserts

30. Jaka adnotacją oznaczymy test:
    a. @Test
    b. @Subject
    c. @TestRun
    d. @Runner
31. Akronim GWT w kontekście testowania oznacza:
    a. Good Written Tests
    b. Given When Then
    c. Good When Run
32. Ile bajtów w pamięci zajmuje typ int:
    a. 1
    b. 8
    c. 4
    d. 2
33. Jaka jest domyślna wartość typu obiektowego:
    a. 1
    b. 0
    c. nie ma
    d. null
34. Jakim operatorem obliczymy resztę z dzielenia:
    a. /
    b. *
    c. %
    d. +
35. Jaki będzie wynik dzielenia 17/4 w Javie gdy jedna z wartości to double:
    a. 1
    b. 4
    c. 4.25
    d. Pojawi się wyjątek
36. Który z operatorów sprawdza czy dwa obiekty zajmują inne miejsce w pamięci:
    a. ==
    b. >>
    c. !=
    d. =!

37. Który z operatorów logiczny jest alternatywą:
    a. ||
    b. &&
    c. !
38. Jaki będzie wynik dla x = 7: (x > 6 && x < 7) || (x >= 7 || x < 8):
    a. true
    b. false
39. Jaki wynik zostanie zwrócony z metody: int getValue() { int x = 10; int i = x; i; return --i;}:
    a. 9
    b. 10
    c. 11
    d. 12
40. Który zapis odczytuje pierwszą wartość z tablicy:
    a. int x = tab[1];
    b. int x = tab(1);
    c. int x = tab.getValue[0];
    d. int x = tab[0];
41. Czy można stworzyć tablicę bez określonego z góry rozmiaru:
    a. tak
    b. nie
42. Odczytanie wartości na indeksie 6 z tablicy String[] tab = new String[5] zwróci:
    a. null
    b. ""
    c. wyjątek
    d. nic się nie stanie
43. Czy ten zapis jest poprawny: void method(String … strings, int value):
    a. tak
    b. nie
44. Jak można odczytać wielkość tablicy:
    a. tab.size
    b. tab.length()
    c. tab.length
    d. tab.getSize()
45. Odczytanie wartości na indeksie 2 z tablicy int[] tab = new int[5] zwróci:
    a. 0

b. 1
c. wyjatek
d. nic

46. Jakie mamy rodzaje pętli:
    a. for each
    b. loop
    c. for
    d. while do
    e. while
    f. enums
    g. counter
    h. do while
47. Który rodzaj pętli nie zwraca informacji o indeksie:
    a. for
    b. while do
    c. counter
    d. for each
48. Jaki będzie wynik pętli: int x = 0; while(true) { if (x = 10) { done; } ++x }:
    a. x = 10
    b. nieskonczona petla
    c. kod się nie skompiluje
    d. x = 11
49. String jest:
    a. Niemutowalny
    b. Typem prymitywnym
    c. Typem obiektowym
    d. Przechowuje liczby całkowite
50. Która z poniższych inicjalizacji przechowywana jest w String Pool:
    a. String abc = new String("abc");
    b. String abc = "abc";
51. Operacja łączenia Stringów nazywana jest:
    a. kotygnacją
    b. konkatenacją
    c. koniunkcją
52. Która z poniższych operacji służy do usuwania białych znaków:

a. isEmpty()
b. removeWhitespaces()
c. removeSpaces()
d. trim()

53. Jaki będzie wynik operacji: "text".substring(1,3):
    a. ex
    b. ext
    c. tex
54. Która z metod wywoływana jest w momencie usuwania obiektu z pamięci:
    a. finally()
    b. afterRemove()
    c. finalize()
    d. drop()
55. Jaka metoda wywoływana jest "pod spodem" na obiekcie w System.out.print(new Integer(1)):
    a. hashCode();
    b. equals();
    c. append();
    d. convertToString();
    e. toString();
56. Standardowa implementacja metody equals sprawdza:
    a. czy dwa obiekty są takie same
    b. czy dwa obiekty zajmują takie same miejsce w pamięci
    c. czy dwa obiekty mają taka samą ilość pól
57. Jes li wartos c metody hashCode jest taka sama x.hashCode() == y.hashCode() to czy x.equals(y) musi zwrócić true:
    a. tak
    b. nie
58. Metoda equals powinna być:
    a. symetryczna
    b. konkretna
    c. kontraktowa
    d. spójna
    e. przechodznia
    f. zawrotna
    g. zwrotna

59. Wynik dla null.equals(null) to:
    a. true
    b. false
    c. NullPointerException
60. Wynik dla x.equals(null) to:
    a. false
    b. true
    c. NullPointerException
61. Czy zapis: switch(true) jest poprawny:
    a. tak
    b. nie
62. Jak sprawdzić czy liczba jest nieparzysta:
    a. x % 2 != 0
    b. x / 2 != 0
    c. x * 2 != 0
63. Czy można nadpisać niefinalną metodę:
    a. tak
    b. nie
64. Elementy statyczne należą do:
    a. instancji
    b. klasy
    c. kompilatora
    d. metody
65. Który z zapisów jest poprawny i zgodny z konwencją dla stałych:
    a. public final static String fieldWithName = "field";
    b. public static final String FIELDWITHNAME = "field";
    c. public static String fieldWithName = "field";
    d. public static final String FIELD_WITH_NAME = "field";
66. Czy można utworzyć nową instancję klasy abstrakcyjnej:
    a. Tak
    b. Nie
67. Czy interfejs może dziedziczyć po innych klasach:
    a. Tak
    b. Nie
68. Zakładając że pole int x jest statyczne to czy można je odczytać: SomeClass test = new

SomeClass(); int x = test.x:

a. Tak
b. Nie

69. Do utworzenia klasy abstrakcyjnej i interfejsu korzystamy ze słów kluczowych:
    a. class i interface
    b. abstract class i interface
    c. interface i abstract interface
    d. static class i interface
70. Adnotacja @Override służy do:
    a. przeciazania
    b. przysłaniania
71. Czy można dziedziczyć po finalnej klasie:
    a. Tak
    b. Nie
72. Typ wyliczeniowy to:
    a. class
    b. abstract class
    c. enum
    d. counter
73. Czy interfejs może implementować inne interfejsy:
    a. Tak
    b. Nie
74. Czy enum może zawierać pola i metody:
    a. Tak
    b. Nie
75. Czy można utworzyć instancje Enum'a:
    a. Tak
    b. Nie
76. Czy w interfejsie mogą znajdować się metody z implementacją:
    a. Tak
    b. Nie
77. Czy metoda oznaczona jako abstract może mieć ciało:
    a. Tak
    b. Nie
78. Czy można utworzyć nową instancję interfejsu:

a. Tak
b. Nie

79. Czy można utworzyć stałe w interfejsie:
    a. Tak
    b. Nie
80. Czy w nieabstrakcyjnej metodzie możemy wywoływać metody abstrakcyjne:
    a. Tak
    b. Nie
81. Która metoda zwraca wszystkie wartości enum'a:
    a. Enum.values()
    b. Enum.enums()
    c. Enum.tabs()
    d. Enum.constants()
82. Czy pola w enumeratorze mogą być przed stałymi:
    a. Tak
    b. Nie
83. Aby rzucić wyjątkiem musi on dziedziczyć po klasie:
    a. Exceptions
    b. ThrowableExceptions
    c. Throwable
    d. UncheckedExceptions
84. Blok finally:
    a. Wywoła się jak wystąpi wyjątek
    b. Wywoła się zawsze
    c. Wywoła się gdy wyjątek nie wystąpi
    d. Wywoła się tylko gdy operujemy na plikach
85. Czy wyjątki które nie dziedziczą po RuntimeException trzeba obsłużyć:
    a. Tak
    b. Nie
86. Na jakie sposoby można obsłużyć wyjątki:
    a. try/catch
    b. try/finally
    c. try/catch/finally
    d. throws w deklaracji metody
87. Który z operatorów służy do łączenia wyjątków w sekcji catch:

a. ||
b. &&
c. &
d. |

88. Która z adnotacji decyduje gdzie można umieścić adnotację:
    a. @Retention
    b. @Place
    c. @Target
    d. @Destination
89. Który z poniższych typów oznacza iż adnotacje będzie można umieścić tylko na klasie:
    a. METHOD
    b. CLASS
    c. FINAL
    d. TYPE
    e. FIELD
    f. DEFINITION
90. Adnotacja @Override wykorzystywana jest:
    a. podczas kompilacji
    b. w czasie wykonywania programu - runtime
    c. w bytekodzie
91. Definicję adnotacji tworzymy poprzez słowo kluczowe:
    a. @enum
    b. @class
    c. @annotation
    d. @interface
92. Która z kolekcji zachowuje kolejność:
    a. LinkedList
    b. HashSet
    c. ArrayList
    d. LinkedHashSet
93. Która z kolekcji nie pozwala na duplikaty:
    a. HashSet
    b. LinkedList
    c. TreeSet
    d. ArrayList

94. Która ze złożoności algorytmicznej jest najlepsza:
    a. O(1)
    b. O(log n)
    c. O(n^2)
    d. O(n)
95. Czy TreeSet może przechowywać wartość null:
    a. Tak
    b. Nie
96. Czy equals i hashCode jest wymagany do poprawnego działania kolekcji Hash*:
    a. Tak
    b. Nie
97. Odczyt wartości z LinkedListy ma złożoność:
    a. O(1)
    b. O(n^2)
    c. O(n)
    d. O(log n)
98. Czy można odczytać wartość z HashMap'y po indeksie:
    a. Tak
    b. Nie
99. Która z poniższych metod pobiera i usuwa pierwszą wartość z kolejki:
    a. get(0)
    b. poll()
    c. peek()
    d. put()
100. Czy można zrobić tak: List<Integer> list = Arrays.asList(1); list.add(2):
    a. Tak
    b. Nie
101. Czy ten zapis: Optional.of(null) jest poprawny:
    a. Tak
    b. Nie
102. Która z metod Iteratora pobiera kolejną wartość:
    a. get()
    b. next()
    c. take()
    d. iter()

103. Która z metod sprawdza czy Optional nie jest pusty:
    a. isPresent()
    b. ifPresent()
    c. isEmpty()
    d. exists()
104. Zwyczajowo w typach generycznych dla elementów dajemy literę:
    a. K
    b. E
    c. T
    d. S
    e. V
105. Typy generyczne można ograniczać poprzez słowo kluczowe:
    a. extends
    b. implements
    c. more
    d. generic
106. Interfejs funkcyjny:
    a. musi posiadać adnotację @FunctionalInterface
    b. musi mieć dokładnie jedną deklarację metody
    c. musi być enumeratorem
    d. musi mieć pola
107. W jakim pakiecie znajdują się domyślne interfejsy funkcyjne:
    a. java.util.functions
    b. java.util.interfaces
    c. java.lang
    d. java.util.function
108. Który z interfejsów funkcyjnych nic nie przyjmuje ale coś zwraca:
    a. Function
    b. Supplier
    c. Predicate
    d. Consumer
109. W którym z domyślnych interfejsów mamy metodę apply():
    a. Function
    b. Supplier
    c. Predicate

d. Consumer

110. Który z interfejsów funkcyjnych przyjmuje dowolny typ ale nic nie zwraca:
    a. Function
    b. Supplier
    c. Predicate
    d. Consumer
111. W którym z domyślnych interfejsów mamy metodę test():
    a. Function
    b. Supplier
    c. Predicate
    d. Consumer
112. W javadoc taki znak @ to:
    a. adnotacja
    b. dyrektywa
    c. opis
    d. wskazanie klasy
113. Która z metod to potęgowanie:
    a. Math.sqrt()
    b. Math.abs()
    c. Maths.power()
    d. Math.pow()
114. Jaki typ interfejsu funkcyjnego reprezentuje ten kod: () → "SDA":
    a. Supplier
    b. Predicate
    c. Function
    d. Consumer
115. Który z interfejsów funkcyjnych konsumuje dwa elementy:
    a. BiFunction
    b. BiSupplier
    c. BiConsumer
    d. BiPredicate
116. Czy taki zapis lambda: () → {}; jest poprawny:
    a. Tak
    b. Nie
117. Które z operacji zamykają strumień:

a. map
b. filter
c. collect
d. findAny

118. Czy może być więcej niż jedną operację terminalną w strumieniu:
    a. Tak
    b. Nie
119. Który z operatorów zamienia jeden typ na drugi:
    a. map
    b. filter
    c. peek
    d. collect
120. Który z zapisów to method reference:
    a. System.out.println()
    b. System.out::println()
    c. System.out::println
    d. System.out:println()
121. W File Input Stream koniec pliku oznaczamy przez:
    a. -1
    b. 0
    c. 1
    d. Exception
122. W jakiej sekcji należy zamykać pliki:
    a. catch()
    b. finally()
    c. try()
    d. close()
123. Który mechanizm przekształca strumień danych w tokeny:
    a. Scanner
    b. Formatter
    c. BufferedStream
    d. FileWriter

Odpowiedzi

Wstęp

Sprawdź wersję **Javy**

Bash

```
java -version
```

Utwórz klasę `Runner.java`, która wypisuje `Hello World!`

Runner.java

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

Skompiluj klasę `Runner.java`

Bash

```
javac Runner.java
```

Uruchom klasę `Runner`

Bash

```
java Runner
```

Stwórz archiwum **JAR** dla swojej aplikacji

Bash

```
jar -cf helloWorldApp.jar Runner.class
```

Uruchom stworzone archiwum:

Bash

```
java -jar helloWorldApp.jar
```

Stwórz plik manifest ze wskazaniem klasy startowej `Runner`:

MANIFEST.MF

```
Main-Class: Runner
```

Stwórz archiwum **JAR** z własnym manifestem

Bash

```
jar -cfm helloWorldApp.jar MANIFEST.MF Runner.class
```

Uruchom stworzone archiwum

Bash

```
java -jar helloWorldApp.jar
```

Zależności

Wszystkie zależności związane z **Maven** umieszczamy w pliku `pom.xml`:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.sda</groupId>
    <artifactId>posts-application</artifactId>
    <version>1.1.0</version>

    <dependencies>

        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.7</version>
        </dependency>

    </dependencies>

</project>
```

Elementy języka

Klasa `Bank`:

Bank.java

```
package pl.sda.bank;


public class Bank {

    protected String imionDluznikow() {
        return "Jan" + "Tomasz";
    }

}
```

Klasa `BankPKO`:

BankPKO.java

```
package pl.sda.bank.pko;

import pl.sda.bank.Bank;


public class BankPKO extends Bank {

    private int oprocentowanie = 20;

    protected int zwrocOprocentowanie() {
        return oprocentowanie;
    }
}
```

Klasa `BankAlior`:

BankAlior.java

```
package pl.sda.bank.pko.alior;

import pl.sda.bank.pko.BankPKO;


public class BankAlior extends BankPKO {

    private String nazwa = "Alior";

    private int zwrocProwizje() {
        return 10 + zwrocOprocentowanie();
    }

    public String zwrocInformacje() {
        return nazwa + " " + zwrocProwizje();
    }

}
```

Klasa `BankING`:

BankING.java

```
package pl.sda.bank.pko.ing;

import pl.sda.bank.pko.BankPKO;

public class BankING extends BankPKO {

    private String nazwa = "ING";

    private int zwrocProwizje() {
        return 15 + zwrocOprocentowanie();
    }

    public String zwrocInformacje() {
        return nazwa + " " + zwrocProwizje();
    }

}
```

Klasa `Runner`:

Runner.java

```
package pl.sda;

import pl.sda.bank.pko.alior.BankAlior;
import pl.sda.bank.pko.ing.BankING;

public class Runner {

    public static void main(String[] args) {
        BankAlior alior = new BankAlior();
        System.out.println(alior.zwrocInformacje());
        BankING ing = new BankING();
        System.out.println(ing.zwrocInformacje());
    }

}
```

Testowanie

Plik z zależnościami `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.sda</groupId>
    <artifactId>test-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.2.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.assertj</groupId>
            <artifactId>assertj-core</artifactId>
            <version>3.8.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

Testy dla klasy `Car`:

CarTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

public class CarTest {

    @Test
    public void shouldReturnCorrectCarName() {
        // Given
        Car maluch = new Car(4, "Maluch");
        // When
        String carName = maluch.carName;
        // Then
        assertThat(carName).isEqualTo("Maluch");
    }

    @Test
    public void shouldReturnCorrectWheelPrice() {
        // Given
        Car maluch = new Car(6, "Maluch");
        // When
        int totalWheelPrice = maluch.getTotalWheelPrice(20);
        // Then
        assertThat(totalWheelPrice).isEqualTo(120);
    }

    @Test
    public void shouldReturnDefaultWheelNumber() {
        // Given
        Car maluch = new Car();
        // When
        int numberOfWheels = maluch.getNumberOfWheels();
        // Then
        assertThat(numberOfWheels).isEqualTo(4);
    }

}
```

Typy danych

Typy proste (prymitywne)

Testy dla klasy `PrimitiveTypes`:

PrimitiveTypesTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class PrimitiveTypesTest {

    @Test
    void shouldReturnDefaultByteValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        byte byteDefault = primitiveTypes.byteDefault;
        // Then
        assertThat(byteDefault).isEqualTo((byte)0);
    }

    @Test
    void shouldReturnByteValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        byte byteValue = primitiveTypes.byteExample;
        // Then
        assertThat(byteValue).isEqualTo((byte)100);
    }

    @Test
    void shouldReturnDefaultShortValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        short shortDefault = primitiveTypes.shortDefault;
        // Then
        assertThat(shortDefault).isEqualTo((short)0);
    }

    @Test
    void shouldReturnShortValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
```

```
        // When
        short shortValue = primitiveTypes.shortExample;
        // Then
        assertThat(shortValue).isEqualTo((short)100);
    }

    @Test
    void shouldReturnDefaultIntValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        int intDefault = primitiveTypes.intDefault;
        // Then
        assertThat(intDefault).isEqualTo(0);
    }

    @Test
    void shouldReturnIntValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        int intValue = primitiveTypes.intExample;
        // Then
        assertThat(intValue).isEqualTo(100);
    }

    @Test
    void shouldReturnDefaultLongValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        long longDefault = primitiveTypes.longDefault;
        // Then
        assertThat(longDefault).isEqualTo(0);
    }

    @Test
    void shouldReturnLongValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        long longValue = primitiveTypes.longExample;
        // Then
        assertThat(longValue).isEqualTo(10000000000000L);
    }

    @Test
    void shouldReturnDefaultFloatValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
```

```
        float floatDefault = primitiveTypes.floatDefault;
        // Then
        assertThat(floatDefault).isEqualTo(0.0f);
    }

    @Test
    void shouldReturnFloatValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        float floatValue = primitiveTypes.floatExample;
        // Then
        assertThat(floatValue).isEqualTo(1.0f);
    }

    @Test
    void shouldReturnDefaultDoubleValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        double doubleDefult = primitiveTypes.doubleDefult;
        // Then
        assertThat(doubleDefult).isEqualTo(0.0f);
    }

    @Test
    void shouldReturnDoubleValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        double doubleExample = primitiveTypes.doubleExample;
        // Then
        assertThat(doubleExample).isEqualTo(1.0f);
    }

    @Test
    void shouldReturnDefaultBooleanValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        boolean booleanDefault = primitiveTypes.booleanDefault;
        // Then
        assertThat(booleanDefault).isEqualTo(false);
    }

    @Test
    void shouldReturnBooleanValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        boolean booleanValue = primitiveTypes.booleanExample;
```

```
        // Then
        assertThat(booleanValue).isEqualTo(true);
    }

    @Test
    void shouldReturnDefaultCharValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        char charDefault = primitiveTypes.defaultChar;
        // Then
        assertThat(charDefault).isEqualTo('\u0000');
    }

    @Test
    void shouldReturnCharValue() {
        // Given
        PrimitiveTypes primitiveTypes = new PrimitiveTypes();
        // When
        char charValue = primitiveTypes.charExample;
        // Then
        assertThat(charValue).isEqualTo('A');
    }

}
```

Implementacja klasy `PrimitiveTypes`:

PrimitiveTypes.java

```
package pl.codecouple;

class PrimitiveTypes {

    byte byteDefault;
    byte byteExample = 100;

    short shortDefault;
    short shortExample = 100;

    int intDefault;
    int intExample = 100;

    long longDefault;
    long longExample = 10000000000000L;

    float floatDefault;
    float floatExample = 1.0f;

    double doubleDefult;
    double doubleExample = 1.0;

    boolean booleanDefault;
    boolean booleanExample = true;

    char defaultChar;
    char charExample = 'A';

}
```

Typy złożone (obiektowe)

Testy dla klasy `ObjectTypes`:

ObjectTypesTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class ObjectTypesTest {

    @Test
    void shouldReturnStringValue() {
        // Given
        ObjectTypes objectTypes = new ObjectTypes();
        // When
        String stringExample = objectTypes.stringExample;
        // Then
        assertThat(stringExample).isEqualToIgnoringCase("text");
    }

    @Test
    void shouldReturnStringNullValue() {
        // Given
        ObjectTypes objectTypes = new ObjectTypes();
        // When
        String stringExample = objectTypes.stringNull;
        // Then
        assertThat(stringExample).isNull();
    }

    @Test
    void shouldReturnStringNewValue() {
        // Given
        ObjectTypes objectTypes = new ObjectTypes();
        // When
        String stringExample = objectTypes.stringNewExample;
        // Then
        assertThat(stringExample).isEqualToIgnoringCase("text");
    }

    @Test
    void shouldReturnIntegerValue() {
        // Given
        ObjectTypes objectTypes = new ObjectTypes();
        // When
        Integer integerExample = objectTypes.integerExample;
        // Then
        assertThat(integerExample).isEqualTo(1);
    }

}
```

Implementacja klasy `ObjectTypes`:

ObjectTypes.java

```
package pl.codecouple;

class ObjectTypes {

    String stringExample = "text";
    String stringNull;
    String stringNewExample = new String("text");

    Integer integerExample = new Integer(1);

}
```

Autoboxing

Testy dla klasy `Autoboxing`:

AutoboxingTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class AutoboxingTest {

    @Test
    void shouldReturnAutoboxedValue() {
        // Given
        Autoboxing autoboxing = new Autoboxing();
        // When
        Integer result = autoboxing.autoboxingExample;
        // Then
        assertThat(result).isEqualTo(1);
    }

}
```

Implementacja klasy `Autoboxing`:

Autoboxing.java

```
package pl.codecouple;

class Autoboxing {

    Integer autoboxingExample = 1;

}
```

Autounboxing

Testy dla klasy `Autounboxing`:

AutounboxingTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class AutounboxingTest {

    @Test
    void shouldReturnAutounboxedValue() {
        // Given
        Autounboxing autounboxing = new Autounboxing();
        // When
        int result = autounboxing.autounboxingExample;
        // Then
        assertThat(result).isEqualTo(1);
    }

}
```

Implementacja klasy `Autounboxing`:

Autounboxing.java

```
package pl.codecouple;

class Autounboxing {

    int autounboxingExample = new Integer(1);

}
```

Operatory

Operatory arytmetyczne

Testy dla klasy `Calculator`:

CalculatorTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class CalculatorTest {

    @Test
    void shouldAddTwoNumbers() {
        // Given
        Calculator calculator = new Calculator();
        // When
        int resultToCheck = calculator.add(10, 20);
        // Then
        assertThat(resultToCheck).isEqualTo(30);
    }

    @Test
    void shouldSubTwoNumbers() {
        // Given
        Calculator calculator = new Calculator();
        // When
        int resultToCheck = calculator.sub(10, 20);
        // Then
        assertThat(resultToCheck).isEqualTo(-10);
    }

    @Test
    void shouldMulTwoNumbers() {
        // Given
        Calculator calculator = new Calculator();
        // When
        int resultToCheck = calculator.mul(10, 20);
        // Then
        assertThat(resultToCheck).isEqualTo(200);
    }

    @Test
```

```
    void shouldDivTwoNumbers() {
        // Given
        Calculator calculator = new Calculator();
        // When
        int resultToCheck = calculator.div(17, 4);
        // Then
        assertThat(resultToCheck).isEqualTo(4);
    }

    @Test
    void shouldDivTwoNumbersWithOneDouble() {
        // Given
        Calculator calculator = new Calculator();
        // When
        double resultToCheck = calculator.div(17.0, 4);
        // Then
        assertThat(resultToCheck).isEqualTo(4.25);
    }


    @Test
    void shouldModTwoNumbers() {
        // Given
        Calculator calculator = new Calculator();
        // When
        int resultToCheck = calculator.mod(17, 4);
        // Then
        assertThat(resultToCheck).isEqualTo(1);
    }


}
```

Implementacja klasy `Calculator`:

Calculator.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class Calculator {

    int add(int first, int second) {
        return first + second;
    }

    int sub(int first, int second) {
        return first - second;
    }

    int mul(int first, int second) {
        return first * second;
    }

    int div(int first, int second) {
        return first / second;
    }

    double div(double first, int second) {
        return first / second;
    }

    int mod(int first, int second) {
        return first % second;
    }

}
```

Implementacja klasy `Runner`:

Calculator.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class Runner {

    public static void main(String[] args) {
        new Calculator().div(17, 0);
    }

}
```

Inkrementacja

Testy dla klasy `Incrementation`:

IncrementationTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class IncrementationTest {

    @Test
    void shouldAddOne() {
        // Given
        Incrementation incrementation = new Incrementation();
        // When
        int result = incrementation.addOne(10);
        // Then
        assertThat(result).isEqualTo(11);
    }

}
```

Implementacja klasy `Incrementation`:

Incrementation.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class Incrementation {

    int addOne(int number) {
        return ++number;
    }

}
```

Dekrementacja

Testy dla klasy `Decrementation`:

DecrementationTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class DecrementationTest {

    @Test
    void shouldAddOne() {
        // Given
        Decrementation decrementation = new Decrementation();
        // When
        int result = decrementation.subOne(10);
        // Then
        assertThat(result).isEqualTo(9);
    }

}
```

Implementacja klasy `Decrementation`:

Decrementation.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class Decrementation {

    int subOne(int number) {
        return --number;
    }

}
```

Operatory porównawcze

Testy dla klasy `Comparator`:

ComparatorTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class ComparatorTest {

    @Test
    void shouldReturnTrueWhenTwoIntValuesAreSameInMemory() {
        // Given
        int value = 10;
        int valueToCompare = 10;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.compare(value, valueToCompare);
        // Then
        assertThat(result).isTrue();
    }

    @Test
    void shouldReturnTrueWhenTwoCarValuesAreSameInMemory() {
        // Given
        Car car = new Car();
```

```
        Car carToCompare = car;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.compare(car, carToCompare);
        // Then
        assertThat(result).isTrue();
    }

    @Test
    void shouldReturnFalseWhenTwoCarValuesAreNotSameInMemory() {
        // Given
        Car car = new Car();
        Car carToCompare = new Car();

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.compare(car, carToCompare);
        // Then
        assertThat(result).isFalse();
    }

    @Test
    void shouldReturnTrueWhenTwoIntValuesAreDifferent() {
        // Given
        int value = 10;
        int valueToCompare = 20;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.areDifferent(value, valueToCompare);
        // Then
        assertThat(result).isTrue();
    }

    @Test
    void shouldReturnTrueWhenTwoCarValuesAreDifferent() {
        // Given
        Car car = new Car();
        Car carToCompare = new Car();

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.areDifferent(car, carToCompare);
        // Then
        assertThat(result).isTrue();
    }
```

```
    @Test
    void shouldReturnFalseWhenTwoCarValuesAreNotDifferent() {
        // Given
        Car car = new Car();
        Car carToCompare = car;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.areDifferent(car, carToCompare);
        // Then
        assertThat(result).isFalse();
    }


    @Test
    void shouldReturnTrueWhenNumberIsLower() {
        // Given
        int number = 1;
        int numberToCompare = 2;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.isLower(number, numberToCompare);
        // Then
        assertThat(result).isTrue();
    }

    @Test
    void shouldReturnFalseWhenNumberIsNotLower() {
        // Given
        int number = 2;
        int numberToCompare = 1;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.isLower(number, numberToCompare);
        // Then
        assertThat(result).isFalse();
    }

    @Test
    void shouldReturnFalseWhenNumberIsNotGreater() {
        // Given
        int number = 1;
        int numberToCompare = 10;

        Comparator comparator = new Comparator();
```

```
        // When
        boolean result = comparator.isGreater(number, numberToCompare);
        // Then
        assertThat(result).isFalse();
    }

    @Test
    void shouldReturnTrueWhenNumberIsGreater() {
        // Given
        int number = 10;
        int numberToCompare = 1;

        Comparator comparator = new Comparator();

        // When
        boolean result = comparator.isGreater(number, numberToCompare);
        // Then
        assertThat(result).isTrue();
    }

}
```

Implementacja klasy `Comparator`:

Comparator.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class Comparator {

    boolean compare(int value, int valueToCompare) {
        return value == valueToCompare;
    }

    boolean compare(Car car, Car carToCompare) {
        return car == carToCompare;
    }

    boolean areDifferent(int value, int valueToCompare) {
        return value != valueToCompare;
    }

    boolean areDifferent(Car car, Car carToCompare) {
        return car != carToCompare;
    }

    boolean isLower(int number, int numberToCompare) {
        return number < numberToCompare;
    }

    boolean isGreater(int number, int numberToCompare) {
        return  number > numberToCompare;
    }

}
```

Tablice

Testy dla klasy `ArrayExample`:

ArrayExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class ArrayExampleTest {

    @Test
    void shouldReturnTableLength() {
        // Given
        ArrayExample arrayExample = new ArrayExample();
        // When
        int tabLength = arrayExample.tabWithValues.length;
        // Then
        assertThat(tabLength).isEqualTo(5);
    }


    @Test
    void shouldReturnCorrectValue() {
        // Given
        ArrayExample arrayExample = new ArrayExample();
        // When
        int tabLength = arrayExample.tabWithValues[3];
        // Then
        assertThat(tabLength).isEqualTo(8);
    }


    @Test
    void shouldReturnDefaultValue() {
        // Given
        ArrayExample arrayExample = new ArrayExample();
        // When
        int value = arrayExample.tabWithoutValues[0];
        // Then
        assertThat(value).isEqualTo(0);
    }

    @Test
    void shouldReturnNullValue() {
        // Given
        ArrayExample arrayExample = new ArrayExample();
        // When
```

```
        String withoutValue = arrayExample.stringsWithoutValues[1];
        // Then
        assertThat(withoutValue).isNull();
    }

    @Test
    void shouldReturnVarargsSize() {
        // Given
        ArrayExample arrayExample = new ArrayExample();
        // When
        int size = arrayExample.varargs();
        // Then
        assertThat(size).isEqualTo(0);
    }

    @Test
    void shouldReturnVarargsSizeWithElements() {
        // Given
        ArrayExample arrayExample = new ArrayExample();
        // When
        int size = arrayExample.varargs("S", "D", "A");
        // Then
        assertThat(size).isEqualTo(3);
    }

}
```

Implementacja klasy `ArrayExample`:

ArrayExample.java

```
package pl.codecouple;

class ArrayExample {

    int[] tabWithValues = {1, 2, 3, 8, 5};
    int[] tabWithoutValues = new int[5];

    String[] stringsWithoutValues = new String[5];

    int varargs(String ... strings) {
        return strings.length;
    }

}
```

Pętle (loops)

Testy dla klasy `LoopExample`:

LoopExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class LoopExampleTest {

    LoopExample loopExample;

    @BeforeEach
    void setUp() {
        loopExample = new LoopExample();
    }

    @Test
    void shouldFillArrayViaForLoop() {
        // When
        int[] result = loopExample.fillFor(10);
        // Then
        assertThat(result).hasSize(10);
        assertThat(result[9]).isEqualTo(9);
    }

    @Test
    void shouldFillArrayViaWhileLoop() {
        // When
        int[] result = loopExample.fillWhile(10);
        // Then
        assertThat(result).hasSize(10);
        assertThat(result[9]).isEqualTo(9);
    }

    @Test
    void shouldIncreaseArrayValueViaDoWhileLoop() {
        // Given
        int[] array = {1, 2, 3};
        // When
        int[] result = loopExample.fillDoWhile(array);
        // Then
        assertThat(result).hasSize(3);
        assertThat(result[2]).isEqualTo(4);
    }
}
```

Implementacja klasy `LoopExample`:

LoopExample.java

```
package pl.codecouple;

class LoopExample {

    int[] fillFor(int value) {
        int[] values = new int[value];
        for(int i=0; i < value; i++) {
            values[i] = i;
        }
        return values;
    }

    int[] fillWhile(int value) {
        int[] values = new int[value];
        int counter = 0;
        while(counter < value) {
            values[counter] = counter;
            counter++;
        }
        return values;
    }

    int[] fillDoWhile(int[] tab) {
        int counter = 0;
        do {
            tab[counter] += 1;
            counter++;
        } while (counter < tab.length);
        return tab;
    }

    void range(int start, int end) {
        System.out.println("Value: " + start);
        if (start == end) {
            return;
        }
        range(++start, end);
    }

}
```

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

public class Runner {

    public static void main(String[] args) {
        final LoopExample loopExample = new LoopExample();
        loopExample.range(10, 20);
    }

}
```

Literał (String)

Testy dla klasy `StringExample`:

StringExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class StringExampleTest {

    @Test
    void shouldAddTwoStrings() {
        // Given
        StringExample stringExample = new StringExample();
        // When
        String concat = stringExample.concat("first", "second");
        // Then
        assertThat(concat).isEqualToIgnoringCase("firstsecond");
    }

    @Test
    void shouldReturnValueOf() {
        // When
        String result = String.valueOf(10);
        // Then
        assertThat(result).isEqualToIgnoringCase("10");
    }

    @Test
    void shouldReturnTrim() {
        // Given
        String testString = "   trim    ";
        // When
        String result = testString.trim();
        // Then
        assertThat(result).isEqualToIgnoringCase("trim");
    }

    @Test
    void shouldReturnToUpper() {
        // Given
        String testString = "abc";
        // When
        String result = testString.toUpperCase();
        // Then
        assertThat(result).isEqualTo("ABC");
    }
```

```
    @Test
    void shouldReturnToLower() {
        // Given
        String testString = "ABC";
        // When
        String result = testString.toLowerCase();
        // Then
        assertThat(result).isEqualTo("abc");
    }

    @Test
    void shouldReturnToCharArray() {
        // Given
        String testString = "tablica";
        // When
        char result[] = testString.toCharArray();
        // Then
        assertThat(result.length).isEqualTo(7);
        assertThat(result[3]).isEqualTo('l');
    }

    @Test
    void shouldReturnToSubstring() {
        // Given
        String testString = "tablica";
        // When
        String substring = testString.substring(3);
        // Then
        assertThat(substring).isEqualTo("lica");
    }

    @Test
    void shouldReturnToSubstringWithConditions() {
        // Given
        String testString = "tablica";
        // When
        String substring = testString.substring(3, 5);
        // Then
        assertThat(substring).isEqualTo("li");
    }

    @Test
    void shouldReturnReplace() {
        // Given
        String testString = "tablica";
        // When
        String substring = testString.replace('a', 'c');
        // Then
        assertThat(substring).isEqualTo("tcblicc");
    }
```

```
    @Test
    void shouldReturnLength() {
        // Given
        String testString = "tablica";
        // When
        int length = testString.length();
        // Then
        assertThat(length).isEqualTo(7);
    }

    @Test
    void shouldReturnIndexOf() {
        // Given
        String testString = "tablica";
        // When
        int indexOf = testString.indexOf("a");
        // Then
        assertThat(indexOf).isEqualTo(1);
    }


    @Test
    void shouldReturnLastIndexOf() {
        // Given
        String testString = "tablica";
        // When
        int lastIndexOf = testString.lastIndexOf("a");
        // Then
        assertThat(lastIndexOf).isEqualTo(6);
    }

    @Test
    void shouldReturnTrueForIsEmpty() {
        // Given
        String testString = "";
        // When
        boolean empty = testString.isEmpty();
        // Then
        assertThat(empty).isTrue();
    }

    @Test
    void shouldReturnFalseForIsEmpty() {
        // Given
        String testString = "   ";
        // When
        boolean empty = testString.isEmpty();
        // Then
        assertThat(empty).isFalse();
    }
```

```
    @Test
    void shouldReturnEndsWith() {
        // Given
        String testString = "tablica";
        // When
        boolean endsWith = testString.endsWith("lica");
        // Then
        assertThat(endsWith).isTrue();
    }

    @Test
    void shouldReturnTrueForContains() {
        // Given
        String testString = "SDA";
        // When
        boolean contains = testString.contains("A");
        // Then
        assertThat(contains).isTrue();
    }

    @Test
    void shouldReturnFalseForContains() {
        // Given
        String testString = "SDA";
        // When
        boolean contains = testString.contains("C");
        // Then
        assertThat(contains).isFalse();
    }
    @Test
    void shouldReturnCharAt() {
        // Given
        String testString = "charAt";
        // When
        char charAt = testString.charAt(3);
        // Then
        assertThat(charAt).isEqualTo('r');
    }


}
```

Implementacja klasy `StringExample`:

StringExample.java

```
package pl.codecouple;

class StringExample {

    String concat(String first, String second) {
        return first + second;
    }

}
```

Object

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

public class Runner {

    public static void main(String[] args) {
        ObjectExample objectExample = new ObjectExample();
        System.out.println(objectExample.hashCode());
        System.out.println(objectExample.getClass());
        System.out.println(objectExample.toString());
        System.out.println(objectExample);

        ToStringObjectExample toStringObjectExample = new ToStringObjectExample();
        System.out.println(toStringObjectExample.toString());
        System.out.println(toStringObjectExample);
    }

}
```

Implementacja klasy `ObjectExample`:

ObjectExample.java

```
package pl.codecouple;

class ObjectExample {

}
```

Implementacja klasy `ToStringObjectExample`:

ToStringObjectExample.java

```
package pl.codecouple;

class ToStringObjectExample {

    int first = 5;
    int second = 10;

    @Override
    public String toString() {
        return "ToStringObjectExample{" +
                "first=" + first +
                ", second=" + second +
                '}';
    }
}
```

equals i hashCode

Testy dla klasy `PhoneEqualsExample`:

PhoneEqualsExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class PhoneEqualsExampleTest {

    @Test
    void shouldReturnTrueWhenSameObjectsAreGiven() {
        // Given
        PhoneEqualsExample first = new PhoneEqualsExample("name", 777);
        PhoneEqualsExample second = new PhoneEqualsExample("name", 777);
        // When
        boolean equals = first.equals(second);
        // Then
        assertThat(equals).isTrue();
    }

}
```

Testy dla klasy `PhoneHashCodeExample`:

PhoneHashCodeExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class PhoneHashCodeExampleTest {

    @Test
    void shouldReturnTrueWhenSameObjectsAreGiven() {
        // Given
        PhoneHashCodeExample first = new PhoneHashCodeExample("name", 777);
        PhoneHashCodeExample second = new PhoneHashCodeExample("name", 777);
        // When
        int hashCode = first.hashCode();
        // Then
        assertThat(hashCode).isEqualTo(second.hashCode());
    }

}
```

Testy dla klasy `PhoneContractExample`:

PhoneContractExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class PhoneContractExampleTest {

    @Test
    void shouldReturnTrueWhenSameObjectsAreGiven() {
        // Given
        PhoneContractExample first = new PhoneContractExample("name", 777);
        PhoneContractExample second = new PhoneContractExample("name", 777);
        // When
        boolean equals = first.equals(second);
        // Then
        assertThat(equals).isTrue();
        assertThat(first.hashCode()).isEqualTo(second.hashCode());
    }

}
```

Implementacja klasy `PhoneEqualsExample`:

PhoneEqualsExample.java

```
package pl.codecouple;

import java.util.Objects;

class PhoneEqualsExample {

    String name;
    int phoneNumber;

    PhoneEqualsExample(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PhoneEqualsExample that = (PhoneEqualsExample) o;
        return phoneNumber == that.phoneNumber &&
                Objects.equals(name, that.name);
    }

}
```

Implementacja klasy `PhoneHashCodeExample`:

PhoneHashCodeExample.java

```
package pl.codecouple;

import java.util.Objects;

class PhoneHashCodeExample {

    String name;
    int phoneNumber;

    PhoneHashCodeExample(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, phoneNumber);
    }
}
```

Implementacja klasy `PhoneContractExample`:

PhoneContractExample.java

```
package pl.codecouple;

import java.util.Objects;

class PhoneContractExample {

    String name;
    int phoneNumber;

    PhoneContractExample(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PhoneContractExample that = (PhoneContractExample) o;
        return phoneNumber == that.phoneNumber &&
                Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {

        return Objects.hash(name, phoneNumber);
    }
}
```

Instrukcje warunkowe

Testy dla klasy `Condition`:

ConditionTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class ConditionTest {

    private Condition condition;

    @BeforeEach
    void setUp() {
        condition = new Condition();
    }

    @Test
    void shouldReturnTrueWhenGivenNumberIsOdd() {
        // When
        boolean result = condition.isOdd(11);
        // Then
        assertThat(result).isTrue();
    }

    @Test
    void shouldReturnTrueWhenGivenNumberIsEven() {
        // When
        boolean result = condition.isEven(12);
        // Then
        assertThat(result).isTrue();
    }

    @Test
    void shouldReturnCorrectNumberWhenGivenNumberIsDivisibleBySeven() {
        // When
        int divisible = condition.divisible(49);
        // Then
        assertThat(divisible).isEqualTo(7);
    }

    @Test
    void shouldReturnCorrectMonthName() {
        // When
        String monthName = condition.getMonthNameBy(10);
        // Then
```

```
        assertThat(monthName).isEqualToIgnoringCase("Październik");
    }

    @Test
    void shouldReturnDefaultMonthName() {
        // When
        String monthName = condition.getMonthNameBy(30);
        // Then
        assertThat(monthName).isEqualToIgnoringCase("Taki miesiąc nie istnieje!");
    }

}
```

Implementacja klasy `Condition`:

Condition.java

```
package pl.codecouple;

class Condition {

    boolean isEven(int number) {
        if (number % 2 == 0) {
            return true;
        }
        return false;
    }

    boolean isOdd(int number) {
        if (number % 2 == 0) {
            return false;
        } else {
            return true;
        }
    }

    int divisible(int number) {
        if (number % 2 == 0) {
            return 2;
        } else if (number % 5 == 0) {
            return 5;
        } else if (number % 7 == 0) {
            return 7;
        } else {
            return 0;
        }
    }

    String getMonthNameBy(int number) {
        switch (number) {
            case 1:
```

```
            return "Styczeń";
        case 2:
            return "Luty";
        case 3:
            return "Marzec";
        case 4:
            return "Kwiecień";
        case 5:
            return "Maj";
        case 6:
            return "Czerwiec";
        case 7:
            return "Lipiec";
        case 8:
            return "Sierpień";
        case 9:
            return "Wrzesień";
        case 10:
            return "Październik";
        case 11:
            return "Listopad";
        case 12:
            return "Grudzień";
        default:
            return "Taki miesiąc nie istnieje!";
    }
}

void getMonthNamesBy(int number) {
    switch (number) {
        case 1:
            System.out.println("Styczeń");
        case 2:
            System.out.println("Luty");
        case 3:
            System.out.println("Marzec");
        case 4:
            System.out.println("Kwiecień");
        case 5:
            System.out.println("Maj");
        case 6:
            System.out.println("Czerwiec");
        case 7:
            System.out.println("Lipiec");
        case 8:
            System.out.println("Sierpień");
        case 9:
            System.out.println("Wrzesień");
        case 10:
            System.out.println("Październik");
        case 11:
```

```
                System.out.println("Listopad");
            case 12:
                System.out.println("Grudzień");
                break;
            default:
                System.out.println("Taki miesiąc nie istnieje!");
        }
    }

}
```

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

class Condition {

    boolean isEven(int number) {
        if (number % 2 == 0) {
            return true;
        }
        return false;
    }

    boolean isOdd(int number) {
        if (number % 2 == 0) {
            return false;
        } else {
            return true;
        }
    }

    int divisible(int number) {
        if (number % 2 == 0) {
            return 2;
        } else if (number % 5 == 0) {
            return 5;
        } else if (number % 7 == 0) {
            return 7;
        } else {
            return 0;
        }
    }

    String getMonthNameBy(int number) {
        switch (number) {
            case 1:
                return "Styczeń";
            case 2:
```

```
            return "Luty";
        case 3:
            return "Marzec";
        case 4:
            return "Kwiecień";
        case 5:
            return "Maj";
        case 6:
            return "Czerwiec";
        case 7:
            return "Lipiec";
        case 8:
            return "Sierpień";
        case 9:
            return "Wrzesień";
        case 10:
            return "Październik";
        case 11:
            return "Listopad";
        case 12:
            return "Grudzień";
        default:
            return "Taki miesiąc nie istnieje!";
    }
}

void getMonthNamesBy(int number) {
    switch (number) {
        case 1:
            System.out.println("Styczeń");
        case 2:
            System.out.println("Luty");
        case 3:
            System.out.println("Marzec");
        case 4:
            System.out.println("Kwiecień");
        case 5:
            System.out.println("Maj");
        case 6:
            System.out.println("Czerwiec");
        case 7:
            System.out.println("Lipiec");
        case 8:
            System.out.println("Sierpień");
        case 9:
            System.out.println("Wrzesień");
        case 10:
            System.out.println("Październik");
        case 11:
            System.out.println("Listopad");
        case 12:
```

```
                System.out.println("Grudzień");
                break;
            default:
                System.out.println("Taki miesiąc nie istnieje!");
        }
    }

}
```

Elementy statyczne

Test dla klasy `MonthConstants`:

MonthConstantsTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class MonthConstantsTest {

    @Test
    void shouldReturnCorrectMonth() {
        // When
        String monthName = MonthConstants.getMonthNameBy(5);
        // Then
        assertThat(monthName).isEqualTo("Maj");
    }

}
```

Implementacja klasy `MonthConstants`:

MonthConstants.java

```
package pl.codecouple;

public class MonthConstants {

    public static final String STYCZEN = "Styczen";
    public static final String LUTY = "Luty";
    public static final String MARZEC = "Marzec";
    public static final String KWIECIEN = "Kwiecien";
    public static final String MAJ = "Maj";
    public static final String CZERWIEC = "Czerwiec";
    public static final String LIPIEC = "Lipiec";
    public static final String SIERPIEN = "Sierpien";
    public static final String WRZESIEN = "Wrzesien";
    public static final String PAZDZIERNIK = "Pazdziernik";
    public static final String LISTOPAD = "Listopad";
    public static final String GRUDZIEN = "Grudzien";

    public static String getMonthNameBy(int number) {
        switch (number) {
            case 1:
                return STYCZEN;
            case 2:
                return LUTY;
            case 3:
                return MARZEC;
            case 4:
                return KWIECIEN;
            case 5:
                return MAJ;
            case 6:
                return CZERWIEC;
            case 7:
                return LIPIEC;
            case 8:
                return SIERPIEN;
            case 9:
                return WRZESIEN;
            case 10:
                return PAZDZIERNIK;
            case 11:
                return LISTOPAD;
            case 12:
                return GRUDZIEN;
            default:
                return "Błąd";
        }
    }

}
```

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

public class Runner {

    public static void main(String[] args) {
        Author author = new Author.AuthorBuilder("Jan")
                .age(50)
                .city("Katowice")
                .lastName("Nowak")
                .build();
    }

}
```

Interfejs

Implementacja interfejsu `Vehicle.java`:

Vehicle.java

```
package pl.codecouple;

interface Vehicle {

    void drive();

}
```

Implementacja interfejsu `Payable.java`:

Payable.java

```
package pl.codecouple;

interface Payable {

    void pay(int quantity);

}
```

Implementacja klasy `Bus.java`:

Bus.java

```
package pl.codecouple;

class Bus implements Vehicle, Payable {

    private static final double price = 3.20;

    @Override
    public void pay(int quantity) {
        System.out.println(quantity * price);
    }

    @Override
    public void drive() {
        System.out.println("Drive by bus");
    }

}
```

Implementacja klasy `Train.java`:

Train.java

```
package pl.codecouple;

class Train implements Vehicle, Payable {

    private static final double price = 25.50;

    @Override
    public void pay(int quantity) {
        System.out.println(quantity * price);
    }

    @Override
    public void drive() {
        System.out.println("Drive by train");
    }

}
```

Implementacja klasy `Car.java`:

Car.java

```
package pl.codecouple;

class Car implements Vehicle {

    @Override
    public void drive() {
        System.out.println("Drive by car");
    }

}
```

Implementacja klasy `Person.java`:

Person.java

```
package pl.codecouple;

class Person {

    void driveBy(Vehicle vehicle) {
        vehicle.drive();
    }

    void buyTicketsFor(Payable payable, int quantity) {
        payable.pay(quantity);
    }

}
```

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

public class Runner {

    public static void main(String[] args) {
        Person person = new Person();
        Bus bus = new Bus();
        Train train = new Train();
        Car car = new Car();

        person.driveBy(bus);
        person.driveBy(train);
        person.driveBy(car);

        person.buyTicketsFor(train, 10);
        person.buyTicketsFor(bus, 10);
    }

}
```

Implementacja interfejsu `JavaProgrammer.java`:

JavaProgrammer.java

```
include::../../../sda-sources/i-introduction-to-java-language/interface/src/main/java
/pl/codecouple/programmers/JavaProgrammer
.java[]
```

Implementacja interfejsu `TableSoccerPlayer.java`:

TableSoccerPlayer.java

```
include::../../../sda-sources/i-introduction-to-java-language/interface/src/main/java
/pl/codecouple/programmers
/TableSoccerPlayer.java[]
```

Implementacja interfejsu `AwesomeProgrammer.java`:

AwesomeProgrammer.java

```
include::../../../sda-sources/i-introduction-to-java-language/interface/src/main/java
/pl/codecouple/programmers
/AwesomeProgrammer.java[]
```

Implementacja klasy `Programmer.java`:

Programmer.java

```
include::../../../sda-sources/i-introduction-to-java-language/interface/src/main/java
/pl/codecouple/programmers
/Programmer.java[]
```

Implementacja klasy `Runner.java`:

Runner.java

```
include::../../../sda-sources/i-introduction-to-java-language/interface/src/main/java
/pl/codecouple/programmers
/Runner.java[]
```

Klasa abstrakcyjna

Implementacja klasy `Drink.java`:

Drink.java

```
package pl.codecouple;

abstract class Drink {

    abstract void showName();
    abstract void addWater();
    abstract void addAlcohol();
    abstract void addJuice();
    abstract void addIce();

    void prepareDrink() {
        showName();
        addWater();
        addAlcohol();
        addJuice();
        addIce();
    }

}
```

Implementacja klasy `Mohito.java`:

Mohito.java

```
package pl.codecouple;

class Mohito extends Drink {

    @Override
    void showName() {
        System.out.println("Mohito");
    }

    @Override
    void addWater() {
        System.out.println("150ml");
    }

    @Override
    void addAlcohol() {
        System.out.println("100ml");
    }

    @Override
    void addJuice() {
        System.out.println("50ml");
    }

    @Override
    void addIce() {
        System.out.println("Yes");
    }

}
```

Implementacja klasy `Malibu.java`:

Malibu.java

```
package pl.codecouple;

class Malibu extends Drink {

    @Override
    void showName() {
        System.out.println("Malibu");
    }

    @Override
    void addWater() {
        System.out.println("150ml");
    }

    @Override
    void addAlcohol() {
        System.out.println("50ml");
    }

    @Override
    void addJuice() {
        System.out.println("250ml");
    }

    @Override
    void addIce() {
        System.out.println("No");
    }
}
```

Implementacja klasy `SexOnTheBeach.java`:

SexOnTheBeach.java

```
include::../../../sda-sources/i-introduction-to-java-language/abstract-class/src/main
/java/pl/codecouple/SexOnTheBeach
.java[]
```

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

class Runner {

    public static void main(String[] args) {
        Mohito mohito = new Mohito();
        Malibu malibu = new Malibu();
        SexOnTheBeach sexOnTheBeach = new SexOnTheBeach();

        mohito.prepareDrink();
        malibu.prepareDrink();
        sexOnTheBeach.prepareDrink();
    }

}
```

Enum

Implementacja enumeratora `WeekDay.java`:

WeekDay.java

```
package pl.codecouple;

enum WeekDay {

    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY

}
```

Implementacja klasy `DayPrinter.java`:

DayPrinter.java

```
package pl.codecouple;

class DayPrinter {

    void printDayBy(WeekDay weekDay) {
        switch (weekDay) {
            case MONDAY:
                System.out.println("Loops");
            case TUESDAY:
                System.out.println("Arrays");
        }
    }

}
```

Implementacja enumeratora `Month.java`:

Month.java

```
package pl.codecouple;

enum Month {

    JANUARY("january", 1),
    FEBRUARY("february", 2),
    MARCH("march", 3),
    APRIL("april", 4),
    MAY("may", 5),
    JUNE("june", 6),
    JULY("july", 7),
    AUGUST("august", 8),
    SEPTEMBER("september", 9),
    OCTOBER("october", 10),
    NOVEMBER("november", 11),
    DECEMBER("december", 12);

    private String monthName;
    private int monthNumber;

    Month(String monthName, int monthNumber) {
        this.monthName = monthName;
        this.monthNumber = monthNumber;
    }

     static String getMonthBy(int monthNumber) {
         for (Month month : Month.values()) {
            if (month.monthNumber == monthNumber) {
                return month.monthName;
            }
        }
        return "Zły numer :(";
    }
}
```

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

class Runner {

    public static void main(String[] args) {
        DayPrinter printer = new DayPrinter();
        printer.printDayBy(WeekDay.MONDAY);
        System.out.println(Month.getMonthBy(12));
        System.out.println(Month.getMonthBy(13));
    }

}
```

Wyjątki

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

public class Runner {

    public static void main(String[] args) {
        ExceptionExamples exceptionExamples = new ExceptionExamples();
        exceptionExamples.catchExample();
    }

}
```

Implementacja klasy `CheckedException`:

CheckedException.java

```
package pl.codecouple;

class CheckedException extends Exception {
}
```

Implementacja klasy `UncheckedException`:

UncheckedException.java

```
package pl.codecouple;

class UncheckedException extends RuntimeException {
}
```

Implementacja klasy `ExceptionExamples`:

ExceptionExamples.java

```
package pl.codecouple;

class ExceptionExamples {

    void throwCheckedExample() throws CheckedException {
        throw new CheckedException();
    }

    void throwUncheckedExample() {
        throw new UncheckedException();
    }

    void catchExample() {
        try {
            throwCheckedExample();
        } catch (CheckedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Finally");
        }
    }

}
```

JavaDoc

Adnotacje

Implementacja klasy `Annotation.java`:

Annotation.java

```
package pl.codecouple;

public class Annotation extends Parent {

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    void removeOverrideAnnotation() {

    }
}
```

Implementacja klasy `Parent.java`:

Parent.java

```
package pl.codecouple;

class Parent {

    void removeOverrideAnnotation() {

    }

}
```

Implementacja adnotacji `Author.java`:

Author.java

```
package pl.codecouple;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Author {

    String name() default "Jan";
    String surname() default "Nowak";

}
```

Implementacja klasy `AnnotationExample.java`:

AnnotationExample.java

```
include::../../../sda-sources/i-introduction-to-java-language/annotations/src/main
/java/pl/codecouple
/AnnotationExample.java[]
```

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

import java.lang.reflect.Method;

class Runner {
    public static void main(String[] args) {
        for(Method method : AnnotationExample.class.getDeclaredMethods()) {
            if (method.isAnnotationPresent(Author.class)) {
                System.out.println(method);
                System.out.println(method.getDeclaredAnnotation(Author.class).name());
                System.out.println(method.getDeclaredAnnotation(Author.class).surname
());
            }
        }
    }
}
```

Czas (LocalTime)

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

import java.time.Duration;
import java.time.LocalTime;

class Runner {

    public static void main(String[] args) {
        System.out.println(LocalTime.now());
        System.out.println(LocalTime.of(10, 10, 10));
        System.out.println(LocalTime.parse("10:10:10"));
        System.out.println(LocalTime.now().plusHours(1));
        System.out.println(LocalTime.now().minusMinutes(10));
        System.out.println(LocalTime.now().getHour());
        System.out.println(LocalTime.now().getMinute());
        System.out.println(LocalTime.now().getSecond());
        System.out.println(
                Duration.between(LocalTime.now().plusHours(1), LocalTime.now())
.getSeconds());

    }

}
```

Data (LocalDate)

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

import java.time.LocalDate;
import java.time.Period;

class Runner {

    public static void main(String[] args) {
        System.out.println(LocalDate.now());
        System.out.println(LocalDate.of(2018, 10, 10));
        System.out.println(LocalDate.parse("2017-02-02"));
        System.out.println(LocalDate.now().plusDays(1));
        System.out.println(LocalDate.now().minusDays(10));
        System.out.println(LocalDate.now().getDayOfMonth());
        System.out.println(LocalDate.now().getMonthValue());
        System.out.println(LocalDate.now().getYear());
        System.out.println(
                Period.between(LocalDate.now(), LocalDate.now().plusDays(1)).getDays(
));

    }

}
```

Kolekcje (Collections)

Testy dla klasy `HashSet`:

HashSetTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import java.util.HashSet;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class HashSetTest {

    @Test
    void shouldReturnOnlyOneElement() {
        // Given
        HashSet<String> hashSet = new HashSet<>();
        // When
        hashSet.add("aaa");
        hashSet.add("aaa");
        hashSet.add("aaa");
        hashSet.add("aaa");
        // Then
        assertThat(hashSet).hasSize(1);
        assertThat(hashSet).contains("aaa");
    }

}
```

Testy dla klasy `ArrayList`:

ArrayListTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
```

```
 * Created by CodeCouple.pl
 */
class ArrayListTest {

    @Test
    void shouldReturnSize() {
        // Given
        ArrayList<String> arrayList = new ArrayList<>();
        arrayList.add("janusz");
        // When
        int size = arrayList.size();
        // Then
        assertThat(size).isEqualTo(1);
    }

    @Test
    void shouldTrueWhenListIsEmpty() {
        // Given
        ArrayList<String> arrayList = new ArrayList<>();
        // When
        boolean empty = arrayList.isEmpty();
        // Then
        assertThat(empty).isTrue();
    }

    @Test
    void shouldTrueWhenAddCorrectly() {
        // Given
        ArrayList<String> arrayList = new ArrayList<>();
        // When
        boolean add = arrayList.add("Value");
        // Then
        assertThat(add).isTrue();
    }

    @Test
    void shouldReturnNewSizeAfterAddAll() {
        // Given
        ArrayList<String> arrayList = new ArrayList<>();
        ArrayList<String> arrayListToAdd = new ArrayList<>();
        arrayListToAdd.add("Value");
        // When
        boolean addAll = arrayList.addAll(arrayListToAdd);
        // Then
        assertThat(addAll).isTrue();
    }

    @Test
    void shouldReturnTrueWhenListContainsElement() {
        // Given
        ArrayList<String> arrayList = new ArrayList<>();
```

```
        arrayList.add("Value");
        // When
        boolean contains = arrayList.contains("Value");
        // Then
        assertThat(contains).isTrue();
    }

    @Test
    void shouldReturnCorrectElement() {
        // Given
        List<String> arrayList = Arrays.asList("Value");
        // When
        String value = arrayList.get(0);
        // Then
        assertThat(value).isEqualTo("Value");
    }

}
```

Testy dla klasy `HashMap`:

HashMapTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import java.util.HashMap;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class HashMapTest {

    @Test
    void shouldPutNewElement() {
        // Given
        HashMap<String, String> map = new HashMap<>();
        // When
        map.put("key", "value");
        // Then
        assertThat(map.get("key")).isEqualTo("value");
    }

    @Test
    void shouldPutAllNewElements() {
        // Given
        HashMap<String, String> map = new HashMap<>();
        HashMap<String, String> newMap = new HashMap<>();
```

```
        map.put("key", "value");
        // When
        map.putAll(newMap);
        // Then
        assertThat(map.get("key")).isEqualTo("value");
    }

    @Test
    void shouldReturnTrueWhenMapContainsKey() {
        // Given
        HashMap<String, String> map = new HashMap<>();
        map.put("key", "value");
        // When
        boolean containsKey = map.containsKey("key");
        // Then
        assertThat(containsKey).isTrue();
    }

    @Test
    void shouldReturnTrueWhenMapContainsValue() {
        // Given
        HashMap<String, String> map = new HashMap<>();
        map.put("key", "value");
        // When
        boolean containsValue = map.containsValue("value");
        // Then
        assertThat(containsValue).isTrue();
    }

    @Test
    void shouldReturnRemovedElement() {
        // Given
        HashMap<String, String> map = new HashMap<>();
        map.put("key", "value");
        // When
        String value = map.remove("key");
        // Then
        assertThat(value).isEqualTo("value");
    }

    @Test
    void shouldReturnGetElement() {
        // Given
        HashMap<String, String> map = new HashMap<>();
        map.put("key", "value");
        // When
        String value = map.get("key");
        // Then
        assertThat(value).isEqualTo("value");
    }
```

```
}
```

Testy dla klasy `TreeMap`:

TreeMapTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import java.util.Map;
import java.util.TreeMap;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class TreeMapTest {

    @Test
    void shouldSortNewKeysOnTreeMap() {
        // Given
        TreeMap<Integer, String> treeMap = new TreeMap<>();
        // When
        treeMap.put(2, "value");
        treeMap.put(2, "value");
        treeMap.put(3, "value");
        treeMap.put(8, "value");
        treeMap.put(1, "value");
        // Then
        assertThat(treeMap.firstKey()).isEqualTo(1);
    }

    @Test
    void shouldSortNewKeysOnMap() {
        // Given
        TreeMap<Integer, String> treeMap = new TreeMap<>();
        // When
        treeMap.put(2, "value");
        treeMap.put(2, "value");
        treeMap.put(3, "value");
        treeMap.put(8, "value");
        treeMap.put(1, "value");
        // Then
        assertThat(getFirstElementFrom(treeMap)).isEqualTo(1);
    }

    private int getFirstElementFrom(TreeMap<Integer, String> treeMap) {
        for (Map.Entry<Integer, String> entry : treeMap.entrySet()) {
            return entry.getKey();
```

```
        }
        return 0;
    }

}
```

Testy dla klasy `Collections`:

CollectionsTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.Collections;

import static org.assertj.core.api.Java6Assertions.assertThat;

/**
 * Created by CodeCouple.pl
 */
class CollectionsTest {

    @Test
    void shouldReturnCorrectFrequency() {
        // Given
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(1);
        list.add(1);
        list.add(1);
        list.add(2);
        // When
        int frequency = Collections.frequency(list, 1);
        // Then
        assertThat(frequency).isEqualTo(4);
    }

    @Test
    void shouldReturnCorrectMaxValue() {
        // Given
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(10);
        list.add(17);
        list.add(1);
        // When
        int max = Collections.max(list);
        // Then
        assertThat(max).isEqualTo(17);
```

```
    }

    @Test
    void shouldReturnCorrectMinValue() {
        // Given
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(10);
        list.add(17);
        list.add(1);
        // When
        int min = Collections.min(list);
        // Then
        assertThat(min).isEqualTo(1);
    }

    @Test
    void shouldReturnReversedList() {
        // Given
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(10);
        list.add(17);
        // When
        Collections.reverse(list);
        // Then
        assertThat(list.get(0)).isEqualTo(17);
    }

}
```

Typy generyczne

Implementacja klasy `Food`:

Food.java

```
package pl.codecouple;

abstract class Food {

    protected final String name;
    protected final String weight;

    protected Food(String name, String weight) {
        this.name = name;
        this.weight = weight;
    }

    abstract void prepare();

}
```

Implementacja klasy `Nudle`:

Nudle.java

```
package pl.codecouple;

class Nudle extends Food {

    protected Nudle(String name, String weight) {
        super(name, weight);
    }

    void prepare() {
        System.out.println(name);
    }
}
```

Implementacja klasy `Cabbage`:

Cabbage.java

```
package pl.codecouple;

class Cabbage extends Food {

    protected Cabbage(String name, String weight) {
        super(name, weight);
    }

    void prepare() {
        System.out.println(name);
    }
}
```

Implementacja klasy `Beef`:

Beef.java

```
package pl.codecouple;

class Beef extends Food {

    protected Beef(String name, String weight) {
        super(name, weight);
    }

    void prepare() {
        System.out.println(name);
    }

}
```

Implementacja klasy `Chef`:

Chef.java

```
package pl.codecouple;

class Chef<T extends Food> {

    void prepareMeal(T foodToPrepare) {
        foodToPrepare.prepare();
    }

}
```

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

class Runner {

    public static void main(String[] args) {
        Chef<Beef> beefChef = new Chef<>();
        Chef<Nudle> nudleChef = new Chef<>();
        Chef<Cabbage> cabbageChef = new Chef<>();

        Beef beef = new Beef("Beef", "100");
        beefChef.prepareMeal(beef);

        Nudle nudle = new Nudle("Nudle", "100");
        nudleChef.prepareMeal(nudle);

        Cabbage cabbage = new Cabbage("Cabbage", "100");
        cabbageChef.prepareMeal(cabbage);
    }

}
```

Optional

Testy dla klasy `OptionalExample`:

OptionalExampleTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class OptionalExampleTest {

    @Test
    void shouldReturnDefaultString() {
        // Given
        NullableExample nullableExampe = new NullableExample(null);
        OptionalExample optionalExample = new OptionalExample(nullableExampe);
        // When
        String defaultValue = optionalExample.getOrDefault();
        // Then
        assertThat(defaultValue).isEqualTo("Empty");
    }

    @Test
    void shouldReturnTrueWhenValueIsGiven() {
        // Given
        NullableExample nullableExampe = new NullableExample("String");
        OptionalExample optionalExample = new OptionalExample(nullableExampe);
        // When
        boolean value = optionalExample.get();
        // Then
        assertThat(value).isTrue();
    }

}
```

Implementacja klasy `NullableExample`:

NullableExample.java

```
package pl.codecouple;

import java.util.Optional;

class NullableExample {

    private final String string;

    NullableExample(String string) {
        this.string = string;
    }

    Optional<String> getNull() {
        return Optional.ofNullable(null);
    }

    Optional<String> getString() {
        return Optional.of(string);
    }


}
```

Implementacja klasy `OptionalExample`:

OptionalExample.java

```
package pl.codecouple;

class OptionalExample {

    private final NullableExample nullableExample;

    OptionalExample(NullableExample nullableExample) {
        this.nullableExample = nullableExample;
    }

    String getOrDefault() {
        return nullableExample.getNull().orElse("Empty");
    }

    boolean get() {
        return nullableExample.getString().isPresent();
    }
}
```

Interfejsy funkcyjne

Testy do klasy `IsEvenPredicate`:

IsEvenPredicateTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Java6Assertions.assertThat;

class IsEvenPredicateTest {

    @Test
    void shouldReturnTrueWhenNumberIsEven() {
        // Given
        IsEvenPredicate isEvenPredicate = new IsEvenPredicate();
        // When
        boolean isEven = isEvenPredicate.test(2);
        // Then
        assertThat(isEven).isTrue();
    }

    @Test
    void shouldReturnFalseWhenNumberIsNotEven() {
        // Given
        IsEvenPredicate isEvenPredicate = new IsEvenPredicate();
        // When
        boolean isEven = isEvenPredicate.test(3);
        // Then
        assertThat(isEven).isFalse();
    }

}
```

Implementacja klasy `IsEvenPredicate`:

IsEvenPredicate.java

```
package pl.codecouple;

import java.util.function.Predicate;

class IsEvenPredicate implements Predicate<Integer> {

    public boolean test(Integer integer) {
        return integer % 2 == 0;
    }

}
```

Implementacja klasy `RandomSupplier`:

RandomSupplier.java

```
package pl.codecouple;

import java.util.Random;
import java.util.function.Supplier;

class RandomSupplier implements Supplier<Integer> {

    @Override
    public Integer get() {
        return new Random().nextInt();
    }

}
```

Implementacja klasy `NumberConsumer`:

NumberConsumer.java

```
package pl.codecouple;

import java.util.function.Consumer;

class NumberConsumer implements Consumer<Integer> {

    @Override
    public void accept(Integer integer) {
        System.out.println(integer);
    }

}
```

Implementacja klasy `PowerFunction`:

PowerFunction.java

```
package pl.codecouple;

import java.util.function.Function;

class PowerFunction implements Function<Integer, Double> {

    @Override
    public Double apply(Integer integer) {
        return Math.pow(integer, integer);
    }

}
```

Testy do klasy `IsOdd`:

IsOddTest.java

```
include::../../../sda-sources/i-introduction-to-java-language/functional-interface/
src/test/java/pl/codecouple/IsOddTest
.java[]
```

Implementacja interfejsu `IsOddPredicate`:

IsOddPredicate.java

```
package pl.codecouple;

@FunctionalInterface
interface IsOddPredicate {

    boolean isOdd(int numberToCheck);

}
```

Implementacja klasy `IsOdd`:

IsOdd.java

```
package pl.codecouple;

class IsOdd implements IsOddPredicate {

    @Override
    public boolean isOdd(int numberToCheck) {
        return numberToCheck % 2 != 0;
    }

}
```

Lambda

Implementacja interfejsu `StringSupplier.java`:

StringSupplier.java

```
package pl.codecouple;

interface StringSupplier {

    String string();

}
```

Implementacja klasy `Runner.java`:

Runner.java

```
package pl.codecouple;

import java.util.Optional;
import java.util.function.BiConsumer;
import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.Supplier;

class Runner {

    public static void main(String[] args) {
        Consumer<String> consumer = (x) -> System.out.println(x);
        Consumer<String> consumerMethodReference = System.out::println;

        consumer.accept("consumer");
        consumerMethodReference.accept("consumerMethodReference");

        Supplier<String> supplier = () -> "SDA";
        System.out.println(supplier.get());

        Predicate<Integer> isEven = (x) -> x % 2 == 0;
        System.out.println(isEven.test(2));

        BiConsumer<String, String> biConsumer = (x, y) -> System.out.println(x + " " +
y);
        biConsumer.accept("SDA", "Roxx!");

        Optional.ofNullable(null).ifPresent(System.out::println);
        Optional.ofNullable(null).orElseGet(() -> "SDA");

        StringSupplier stringSupplier = () -> "SDA";
        System.out.println(stringSupplier.string());
    }

}
```

Strumienie

Testy dla klasy `StreamTest`:

StreamTest.java

```
package pl.codecouple;

import org.junit.jupiter.api.Test;

import java.util.List;
import java.util.Optional;
import java.util.OptionalInt;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

import static org.assertj.core.api.Java6Assertions.assertThat;

class StreamTest {

    @Test
    void shouldReturnWordsHigherThanFiveCharsInUpperCase() {
        // Given
        Stream<String> words = Stream.of("first", "second", "third", "fourth", "fifth
");
        // When
        List<String> result = words.filter(x -> x.length() > 5)
                .map(String::toUpperCase)
                .collect(Collectors.toList());
        // Then
        assertThat(result).hasSize(2);
        assertThat(result).contains("SECOND", "FOURTH");
    }

    @Test
    void shouldReturnWordsHigherThanSeven() {
        // Given
        Stream<String> words = Stream.of("first", "second", "third", "fourth", "fifth
");
        // When
        Optional<String> result = words.filter(x -> x.length() > 7)
                .findFirst();
        // Then
        assertThat(result).isNotNull();
        assertThat(result.isPresent()).isFalse();
    }

    @Test
    void shouldReturnEvenNumbers() {
        // Given
```

```
        Stream<Integer> numbers = Stream.of(1, 26, 30, 2, 45);
        // When
        List<Integer> result = numbers.filter(x -> x % 2 == 0)
                .collect(Collectors.toList());
        // Then
        assertThat(result).hasSize(3);
        assertThat(result).contains(26, 30, 2);
    }

    @Test
    void shouldReturnMaxNumber() {
        // Given
        IntStream numbers = IntStream.of(1, 26, 30, 2, 45);
        // When
        OptionalInt max = numbers.max();
        // Then
        assertThat(max).isNotNull();
        assertThat(max.getAsInt()).isEqualTo(45);
    }

    @Test
    void shouldReturnNumbersHigherThanTwentySixAsString() {
        // Given
        Stream<Integer> numbers = Stream.of(1, 26, 30, 2, 45);
        // When
        List<String> result = numbers.filter(x -> x > 26)
                .map(String::valueOf)
                .collect(Collectors.toList());
        // Then
        assertThat(result).hasSize(2);
        assertThat(result).contains("30", "45");
    }


}
```

InputOutput (IO)

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

import java.io.*;
import java.math.BigDecimal;
import java.util.Scanner;

class Runner {

    public static void main(String[] args) throws IOException, ClassNotFoundException
{
        exerciseOne();
        exerciseTwo();
        exerciseThree();
        exerciseFour();
        exerciseFive();
        exerciseSix();
        exerciseSeven();
        exerciseEight();
        exerciseNine();
    }

    private static void exerciseOne() throws IOException {
        FileInputStream file = null;
        try {
            file = new FileInputStream("file.txt");
            int byteValue;
            while((byteValue = file.read()) != -1) {
                System.out.println(byteValue);
            }
        } finally {
            if (file != null) {
                file.close();
            }
        }
    }

    private static void exerciseTwo() throws IOException {
        FileInputStream file = null;
        FileOutputStream fileToSave = null;
        try {
            file = new FileInputStream("file.txt");
            fileToSave = new FileOutputStream("copy.txt");
            int byteValue;
            while((byteValue = file.read()) != -1) {
                fileToSave.write(byteValue);
```

```
            }
        } finally {
            if (file != null) {
                file.close();
            }
            if (fileToSave != null) {
                fileToSave.close();
            }
        }
    }

    private static void exerciseThree() throws IOException {
        FileReader fileReader = null;
        try {
            fileReader = new FileReader("file.txt");
            int value;
            while ((value = fileReader.read()) != -1) {
                System.out.println((char)value);
            }
        } finally {
            if (fileReader != null) {
                fileReader.close();
            }
        }
    }

    private static void exerciseFour() throws IOException {
        BufferedReader fileReader = null;
        try {
            fileReader = new BufferedReader(new FileReader("file.txt"));
            String line;
            while ((line = fileReader.readLine()) != null) {
                System.out.println(line);
            }
        } finally {
            if (fileReader != null) {
                fileReader.close();
            }
        }
    }

    private static void exerciseFive() {
        Scanner in = new Scanner(System.in);
        while(in.hasNext()) {
            System.out.println(in.next());
            break;
        }
    }

    private static void exerciseSix() throws IOException {
        DataOutputStream out = null;
```

```
        try {
            out = new DataOutputStream(
                    new BufferedOutputStream(
                            new FileOutputStream("data.txt")));


            out.writeInt(20);
            out.writeDouble(15.5);
            out.writeUTF("file");
        } finally {
            if (out != null) {
                out.close();
            }
        }

    }

    private static void exerciseSeven() throws IOException {
        DataInputStream in = null;
        try {
            in = new DataInputStream(
                    new BufferedInputStream(
                            new FileInputStream("data.txt")));


            System.out.println(in.readInt());
            System.out.println(in.readDouble());
            System.out.println(in.readUTF());
        } finally {
            if (in != null) {
                in.close();
            }
        }

    }

    private static void exerciseEight() throws IOException {
        ObjectOutputStream out = null;
        try {
            out = new ObjectOutputStream(
                    new BufferedOutputStream(
                            new FileOutputStream("dataObjects.txt")));


            out.writeObject(new BigDecimal(2));
        } finally {
            if (out != null) {
                out.close();
            }
        }
```

```
    }

    private static void exerciseNine() throws IOException, ClassNotFoundException {
        ObjectInputStream in = null;
        try {
            in = new ObjectInputStream(
                    new BufferedInputStream(
                            new FileInputStream("dataObjects.txt")));


            System.out.println(((BigDecimal)in.readObject()).negate());
        } finally {
            if (in != null) {
                in.close();
            }
        }

    }

}
```

New InputOutput (NIO)

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.Arrays;
import java.util.List;

class Runner {
    public static void main(String[] args) throws IOException {
        Files.createDirectories(Paths.get("files"));

        Path path = Paths.get("file.txt");

        Files.isHidden(path);
        Files.isReadable(path);
        Files.exists(path);

        Files.deleteIfExists(Paths.get("fileThird.txt"));

        Files.copy(Paths.get("file.txt"), Paths.get("fileCopy.txt"),
                StandardCopyOption.REPLACE_EXISTING);
        Files.copy(Paths.get("file.txt"), Paths.get("files/fileCopy.txt"),
                StandardCopyOption.REPLACE_EXISTING);

        Files.createDirectories(Paths.get("io"));

        Files.createFile(Paths.get("io/io.txt"));

        System.out.println(Files.readAllLines(Paths.get("file.txt")));

        List<String> strings = Arrays.asList("a", "b", "c");
        Files.write(Paths.get("io/io.txt"), strings);

        System.out.println(Files.readAllLines(Paths.get("io/io.txt")));
    }
}
```

Wielowątkowość

Implementacja klasy `Runner`:

Runner.java

```
package pl.codecouple;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * Created by CodeCouple.pl
 */
class Runner {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName());
        new NewThread().start();
        new NewThread().run();
        new Thread(new NewRunnable()).start();
        new Thread(new NewRunnable()).run();

        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        for (int i = 0; i < 10; i++) {
            threadPool.submit(new NewRunnable());
        }

        threadPool.shutdown();
    }

}
```

Implementacja klasy `NewThread`:

NewThread.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class NewThread extends Thread {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

}
```

Implementacja klasy `NewRunnable`:

NewRunnable.java

```
package pl.codecouple;

/**
 * Created by CodeCouple.pl
 */
class NewRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

}
```

Implementacja klasy `SynchronizedRunner`:

SynchronizedRunner.java

```
package pl.codecouple;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SynchronizedRunner {

    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        final Synchronized sync = new Synchronized();

        for (int i = 0; i < 10; i++) {
            threadPool.submit(sync::synchronizedBlock);
        }

        for (int i = 0; i < 10; i++) {
            threadPool.submit(sync::synchronizedMethod);
        }

        threadPool.shutdown();
    }

}
```

Implementacja klasy `Synchronized`:

Synchronized.java

```
package pl.codecouple;

public class Synchronized {

    public synchronized void synchronizedMethod() {
        System.out.println("Thread sleeping: " + Thread.currentThread().getName());
        try {
            Thread.sleep(3_000);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void synchronizedBlock() {
        System.out.println("Thread waiting: " + Thread.currentThread().getName());

        synchronized(this) {
            try {
                System.out.println("Thread sleeping: " + Thread.currentThread()
.getName());
                Thread.sleep(3_000);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Thread out: " + Thread.currentThread().getName());
    }

}
```

Implementacja klasy `LocksRunner`:

LocksRunner.java

```
package pl.codecouple;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class LocksRunner {

    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        RWLocks locks = new RWLocks();

        for (int i = 0; i < 5; i++) {
            threadPool.submit(() -> locks.add("Text"));
        }

        for (int i = 0; i < 5; i++) {
            threadPool.submit(() -> locks.get(1));
        }

        threadPool.shutdown();
    }

}
```

Implementacja klasy `Locks`:

Locks.java

```
package pl.codecouple;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Locks {

    Lock lock = new ReentrantLock();

    List<String> stringList = new ArrayList<>();

    String get(int index) {
        System.out.println("Thread waiting on get: " + Thread.currentThread().getName
());
        try {
            lock.lock();
            try {
```

```
                System.out.println("Thread sleeping on get: " + Thread.currentThread(
).getName());
                Thread.sleep(3_000);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
            return stringList.get(index);
        } finally {
            lock.unlock();
            System.out.println("Thread out on get: " + Thread.currentThread().getName
());
        }
    }

    void add(String stringToAdd) {
        System.out.println("Thread waiting on add: " + Thread.currentThread().getName
());
        try {
            lock.lock();
            try {
                System.out.println("Thread sleeping on add: " + Thread.currentThread(
).getName());
                Thread.sleep(3_000);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
            stringList.add(stringToAdd);
        } finally {
            lock.unlock();
            System.out.println("Thread out on add: " + Thread.currentThread().getName
());
        }
    }

}
```

Implementacja klasy `RWLocks`:

RWLocks.java

```
package pl.codecouple;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class RWLocks {

    ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
    Lock readLock = lock.readLock();
    Lock writeLock = lock.writeLock();

    List<String> stringList = new ArrayList<>();

    String get(int index) {
        System.out.println("Thread waiting on get: " + Thread.currentThread().getName
());
        try {
            readLock.lock();
            try {
                System.out.println("Thread sleeping on get: " + Thread.currentThread(
).getName());
                Thread.sleep(3_000);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
            return stringList.get(index);
        } finally {
            readLock.unlock();
            System.out.println("Thread out on get: " + Thread.currentThread().getName
());
        }
    }

    void add(String stringToAdd) {
        System.out.println("Thread waiting on add: " + Thread.currentThread().getName
());
        try {
            writeLock.lock();
            try {
                System.out.println("Thread sleeping on add: " + Thread.currentThread(
).getName());
                Thread.sleep(3_000);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
            stringList.add(stringToAdd);
        } finally {
            writeLock.unlock();
            System.out.println("Thread out on add: " + Thread.currentThread().getName
());
        }
    }
}
```

Odpowiedzi do testu

Jak nazywa się kompilator Javy:

- javacompiler
- **javac**
- javak
- groovyc

Jakie rozszerzenie posiada skompilowany plik:

- **class**
- java
- compiled

Który z poniższych typów to package-scope:

- private - tylko w obrębie tej samej klasy
- public - dostęp dla wszystkich
- protected - dostęp w obrębie pakietu i klas dziedziczących
- **Nic nie piszemy**

Do uruchomienia programów w języku Java wystarczy:

- JDK - środowisko wymagane do wytwarzania oprogramowania w języku **Java**
- **JRE**

Jakim poleceniem sprawdzamy wersje Javy:

- javaversion
- java-version
- java -v
- **java -version**

Który typ archiwum jest najpopularniejszy w Javie:

- CAR
- WAR
- **JAR**
- EAR

Czym jest artefakt:

- **produktem końcowym procesu budowania**

- rodzajem klasy z jednym konstruktorem
- metodą startową - metodą startową jest metoda main

Która z poniższych opcji jest prawdziwa:

- protected static void main(String args[]) - metoda ta musi być publiczna
- public void static main(String args[]) - zła kolejność static i void
- public static void main(Integer args[]) - metoda ta przyjmuje argumenty typu Integer
- **Żadne z powyższych** - poprawna metoda public static void main(String args[])

Aby poprawnie skompilować plik Runner.java moja klasa powinna:

- **nazywać się tak samo jak plik**
- nazywać się tak samo jak plik ale nazwa powinna być pisana małą literą

Jaka jest wartość domyślna dla typu boolean:

- true
- **false**

Jak nazywa się ten mechanizm: int a = new Integer(2):

- Wrapping
- Boxing
- Autoboxing
- **Autounboxing**

Jaka jest domyślna wartość dla typu char:

- ' '
- ''
- 'a'
- **'\u0000'** - pierwszy znak w tablicy unicode

Poprzez jakie słowo realizujemy dziedziczenie:

- class
- **extends**
- public - modyfikator dostępu
- int - typ danych

Który z poniższych typów przechowuje liczby całkowite:

- double - liczby zmiennoprzecinkowe o większej precyzji
- **int**

- float - liczby zmiennoprzecinkowe
- **long**
- boolean - przechowuje wartości logiczne
- char - przechowuje pojedyncze znaki
- **short**
- **byte**

Czym jest Maven:

- kompilatorem - jedynie korzysta z kompilatora
- biblioteka do testowania - jedynie uruchamia testy
- **narzędziem do budowania**
- **narzędziem do wskazywania zależności**

Do czego służy plik MANIFEST.MF:

- **Do przechowywania dodatkowych informacji o archiwum**
- Do tworzenia klas - nie jest wymagany podczas tworzenia klas
- **Zawiera informacje o Maven** - informacje o projekcie Maven znajdują się w pliku pom.xml
- Wymagany przy kompilacji - nie jest wymagany podczas kompilacji

Jakim skrótem wstawiamy nową klase:

- alt+n/option+n
- **alt+insert/command+n**
- ctrl+F12/command+F12

W jakim folderze w IntelliJ przechowywane są informacje o projekcie:

- target - skompilowane pliki
- src/main - pliki źródłowe zwykłych klas
- **.idea**
- src/test - pliki źródłowe testów

W jakim folderze umieszczamy dodatkowe pliki zgodnie z konwencją **Maven**:

- src/main - pliki źródłowe zwykłych klas
- src - pliki źródłowe dla testów i zwykłych klas
- src/test - pliki źródłowe testów
- **src/main/resources**

Jak nazywa się **plugin** do podpowiadania skrótów:

- ShortCutter
- **Key Promoter X**
- Keys
- KeyMapper

W jakim **pliku** umieszczamy zależności do projektu gdy korzystamy z **Maven**:

- ron.xml - taki plik nie istnieje w kontekście **Maven**
- maven.xml - taki plik nie istnieje w kontekście **Maven**
- Runner.java - to jest zwykła klasa
- **pom.xml**

Jak nazywa się **lokalne repozytorium**:

- .m3 - taki folder nie istnieje w kontekście **Maven**
- .mvn - taki folder nie istnieje w kontekście **Maven**
- **.m2**
- .repo - taki folder nie istnieje w kontekście **Maven**

Cykl Maven odpowiedzialny za budowanie dokumentacji to:

- default - domyślny cykl odpowiedzialny za budowanie artefaktu
- clean - służy do usuwania folderu target
- **site**
- doc - taki cykl nie istnieje w kontekście **Maven**

Która z poniższych faz tworzy archiwum:

- **package**
- clean - to jest cykl a nie faza
- jar - taka faza nie istnieje w kontekście **Maven**
- archive - taka faza nie istnieje w kontekście **Maven**

Pola znajdują się w:

- metodzie - w metodach znajdują się zmienne
- **klasie**
- w psvm - to jest metoda więc w środku są zmienne
- Maven - zły kontekst

Do czego używamy importów:

- **Aby wskazać gdzie znajduje się dana klasa**

- Aby mieć nowe zależności - nowe zależności dodajemy w pliku pom.xml
- Jest to wymaganie Maven - nie jest

Jak tworzymy nową instancje klasy:

- Car car; - tutaj jedynie rezerwujemy obszar pamięci
- Car car = "4, Maluch"; - brak użycia konstruktora
- **Car car = new Car();**
- Car car = instance Car(); - brak użycia słowa new

Jak nazywa się biblioteka do testowania:

- **JUnit**
- JavaUnit
- JIntegration
- JMH

Jak nazywa się biblioteka do asercji z fluent-assertions:

- FestAssert
- FluentAssertions
- **AssertJ**
- Asserts

Jaką adnotacją oznaczamy testy w bibliotece **JUnit**:

- **@Test**
- @Subject
- @TestRun
- @Runner

Akronim GWT w kontekście testowania oznacza:

- Good Written Tests
- **Given When Then**
- Good When Run

Ile bajtów w pamięci zajmuje typ int:

- 1
- 8
- **4**
- 2

Jaka jest domyślna wartość typu obiektowego:

- 1
- 0
- nie ma
- **null**

Jakim operatorem obliczymy resztę z dzielenia:

- /
- *
- **%** - operator modulo
- +

Jaki będzie wynik dzielenia 17/4.0 w Javie gdy jedna z wartości to double:

- 1
- 4
- **4.25**
- Pojawi się wyjątek

Który z operatorów sprawdza czy dwa obiekty zajmują inne miejsce w pamięci:

- ==
- >>
- **!=**
- =!

Który z operatorów logiczny jest alternatywą:

- ||
- &&
- !

Jaki będzie wynik dla x = 7:

```
(x > 6 && x < 7) || (x >= 7 || x < 8)
```

- **true**
- false

Jaki wynik zostanie zwrócony z metody:

```
int getValue() {
    int x = 10;
    int i = x++;
    i++;
    return --i;
}
```

- 9
- **10**
- 11
- 12

Który zapis odczytuje pierwszą wartość z tablicy:

- int x = tab[1]; - indeksujemy od zera
- int x = tab(1); - nawiasy kwadratowe służą do odczytu wartości
- int x = tab.getValue[0]; - nie ma takiej metody w tablicach
- **int x = tab[0];**

Czy można stworzyć tablicę bez określonego z góry rozmiaru:

- Tak
- **Nie**

Odczytanie wartości na indeksie 6 z tablicy String[] tab = new String[5] zwróci:

- null
- ""
- **wyjątek**
- nic się nie stanie - dostaniemy wyjątek

Czy ten zapis jest poprawny: void method(String ... strings, int value):

- Tak
- **Nie**

Jak można odczytać wielkość tablicy:

- **tab.size** - to jest pole, nie metoda
- tab.length()
- tab.length
- tab.getSize()

Odczytanie wartości na indeksie 2 z tablicy int[] tab = new int[5] zwróci:

- **0** - wartość domyślna dla typu int
- 1
- wyjatek
- nic

Jakie mamy rodzaje pętli:

- **for each**
- loop
- **for**
- while do
- **while**
- enums
- counter
- **do while**

Który rodzaj pętli nie zwraca informacji o indeksie:

- for
- while do
- counter
- **for each**

Jaki będzie wynik pętli:

```
int x = 0;
while(true) {
    if (x = 10) {
        done;
    }
    ++x;
}:
```

- x = 10
- nieskonczona petla
- **kod się nie skompiluje** - nie ma takiego słowa kluczowego jak done
- x = 11

String jest:

- **Niemutowalny**
- Typem prymitywnym

- **Typem obiektowym**
- Przechowuje liczby całkowite

Która z poniższych inicjalizacji przechowywana jest w String Pool:

- String abc = new String("abc");
- **String abc = "abc";**

Operacja łączenia Stringów nazywana jest:

- kotygnacją
- **konkatenacją**
- koniunkcją

Która z poniższych operacji służy do usuwania białych znaków:

- isEmpty()
- removeWhitespaces()
- removeSpaces()
- **trim()**

Jaki będzie wynik operacji: "text".substring(1,3):

- **ex**
- ext
- tex

Która z metod wywoływana jest w momencie usuwania obiektu z pamięci:

- finally()
- afterRemove()
- **finalize()** - od Javy 9 jest oznaczona jako depracted
- drop()

Jaka metoda wywoływana jest "pod spodem" na obiekcie w System.out.print(new Integer(1)):

- hashCode();
- equals();
- append();
- convertToString();
- **toString();**

Standardowa implementacja metody equals sprawdza:

- czy dwa obiekty są takie same

- **czy dwa obiekty zajmują takie same miejsce w pamięci**
- czy dwa obiekty mają taka samą ilość pól

Jes li wartos c metody hashCode jest taka sama x.hashCode() == y.hashCode() to czy x.equals(y) musi zwrócić true:

- Tak
- **Nie**

Metoda equals powinna być:

- **symetryczna**
- konkretna
- kontraktowa
- spójna
- **przechodznia**
- zawrotna
- **zwrotna**

Wynik dla null.equals(null) to:

- true
- false
- **NullPointerException**

Wynik dla x.equals(null) to:

- **false**
- true
- NullPointerException

Czy zapis: switch(true) jest poprawny:

- Tak
- **Nie**

Jak sprawdzić czy liczba jest nieparzysta:

- **x % 2 != 0**
- x / 2 != 0
- x * 2 != 0

Czy można nadpisać niefinalną metodę:

- **Tak**

- Nie

Elementy statyczne należą do:

- instancji
- **klasy**
- kompilatora
- metody

Który z zapisów jest poprawny i zgodny z **konwencją** dla stałych:

- public final static String fieldWithName = "field";
- public static final String FIELDWITHNAME = "field";
- public static String fieldWithName = "field";
- **public static final String FIELD_WITH_NAME = "field"**;

Czy można utworzyć nową instancję klasy abstrakcyjnej:

- Tak
- **Nie**

Czy interfejs może dziedziczyć po innych klasach:

- Tak
- **Nie**

Zakładając że pole int x jest statyczne to czy można je odczytać: SomeClass test = new SomeClass(); int x = test.x:

- **Tak**
- Nie

Do utworzenia klasy abstrakcyjnej i interfejsu korzystamy ze słów kluczowych:

- class i interface
- **abstract class i interface**
- interface i abstract interface
- static class i interface

Adnotacja @Override służy do:

- przeciążania
- **przysłaniania**

Czy można dziedziczyć po finalnej klasie:

- Tak
- **Nie**

Typ wyliczeniowy to:

- class
- abstract class
- **enum**
- counter

Czy interfejs może implementować inne interfejsy:

- Tak
- **Nie**

Czy enum może zawierać pola i metody:

- **Tak**
- Nie

Czy można utworzyć instancje Enum'a:

- Tak
- **Nie**

Czy w interfejsie mogą znajdować się metody z implementacją:

- **Tak**
- Nie

Czy metoda oznaczona jako abstract może mieć ciało:

- Tak
- **Nie**

Czy można utworzyć nową instancję interfejsu:

- Tak
- **Nie**

Czy można utworzyć stałe w interfejsie:

- **Tak**
- Nie

Czy w nieabstrakcyjnej metodzie możemy wywoływać metody abstrakcyjne:

- **Tak**

- Nie

Która metoda zwraca wszystkie wartości enum'a:

- **Enum.values()**
- Enum.enums()
- Enum.tabs()
- Enum.constants()

Czy pola w enumeratorze mogą być przed stałymi:

- Tak
- **Nie**

Aby "rzucić" wyjątkiem musi on dziedziczyć po klasie:

- Exceptions
- ThrowableExceptions
- **Throwable**
- UncheckedExceptions

Blok finally:

- Wywoła się tylko jak wystąpi wyjątek
- **Wywoła się zawsze**
- Wywoła się tylko gdy wyjątek nie wystąpi
- Wywoła się tylko gdy operujemy na plikach

Czy wyjątki które nie dziedziczą po RuntimeException trzeba obsłużyć:

- Tak
- **Nie**

Na jakie sposoby można obsłużyć wyjątki:

- **try/catch**
- try/finally
- **try/catch/finally**
- **throws w deklaracji metody**

Który z operatorów służy do łączenia wyjątków w sekcji catch:

- ||
- &&
- &

- |

Która z adnotacji decyduje gdzie można umieścić adnotację:

- @Retention
- @Place
- **@Target**
- @Destination

Który z poniższych typów oznacza, iż adnotacje będzie można umieścić tylko na klasie:

- METHOD
- CLASS
- FINAL
- **TYPE**
- FIELD
- DEFINITION

Adnotacja @Override wykorzystywana jest:

- **podczas kompilacji**
- w czasie wykonywania programu - runtime
- w bytekodzie

Definicję adnotacji tworzymy poprzez słowo kluczowe:

- @enum
- @class
- @annotation
- **@interface**

Która z kolekcji zachowuje kolejność:

- **LinkedList**
- HashSet
- **ArrayList**
- **LinkedHashSet**

Która z kolekcji nie pozwala na duplikaty:

- **HashSet**
- LinkedList
- TreeSet

- ArrayList

Która ze złożoności algorytmicznej jest najlepsza:

- **O(1)**
- O(log n)
- O(n^2)
- O(n)

Czy TreeSet może przechowywać wartość null:

- Tak
- **Nie**

Czy equals i hashCode jest wymagany do poprawnego działania kolekcji Hash*:

- **Tak**
- Nie

Odczyt wartości z LinkedListy ma złożoność:

- O(1)
- O(n^2)
- **O(n)**
- O(log n)

Czy można odczytać wartość z HashMap'y po indeksie:

- Tak
- **Nie**

Która z poniższych metod pobiera i usuwa pierwszą wartość z kolejki:

- get(0)
- **poll()**
- peek()
- put()

Czy można zrobić tak: List<Integer> list = Arrays.asList(1); list.add(2):

- Tak
- **Nie** - Arrays.asList(1) tworzy listę o stałym rozmiarze

Czy ten zapis: Optional.of(null) jest poprawny:

- Tak

- **Nie** - dostaniemy wyjątek, dla wartości null stosujemy ofNullable

Która z metod Iteratora pobiera kolejną wartość:

- get()
- **next()**
- take()
- iter()

Która z metod sprawdza czy Optional nie jest pusty:

- **isPresent()**
- ifPresent()
- isEmpty()
- exists()

Zwyczajowo w typach generycznych dla elementów dajemy literę:

- K
- **E**
- T
- S
- V

Typy generyczne można ograniczać poprzez słowo kluczowe:

- **extends**
- implements
- more
- generic

Interfejs funkcyjny:

- musi posiadać adnotację @FunctionalInterface
- **musi mieć dokładnie jedną deklarację metody**
- musi być enumeratorem
- musi mieć pola

W jakim pakiecie znajdują się domyślne interfejsy funkcyjne:

- java.util.functions
- java.util.interfaces
- java.lang

- **java.util.function**

Który z interfejsów funkcyjnych nic nie przyjmuje ale coś zwraca:

- Function
- **Supplier**
- Predicate
- Consumer

W którym z domyślnych interfejsów mamy metodę apply():

- **Function**
- Supplier
- Predicate
- Consumer

Który z interfejsów funkcyjnych przyjmuje dowolny typ ale nic nie zwraca:

- Function
- Supplier
- Predicate
- **Consumer**

W którym z domyślnych interfejsów mamy metodę test():

- Function
- Supplier
- **Predicate**
- Consumer

W javadoc taki znak @ to:

- adnotacja
- **dyrektywa**
- opis
- wskazanie klasy

Która z metod to potęgowanie:

- Math.sqrt()
- Math.abs()
- Maths.power()
- **Math.pow()** - statyczna metoda pow od power

Jaki typ interfejsu funkcyjnego reprezentuje ten kod: () → "SDA":

- **Supplier** - nic nie przyjmuje, coś zwraca
- Predicate
- Function
- Consumer

Który z interfejsów funkcyjnych konsumuje dwa elementy:

- BiFunction
- BiSupplier
- **BiConsumer**
- BiPredicate

Czy taki zapis lambda: () → {}; jest poprawny:

- **Tak** - jest to interfejs funkcyjny Runnable
- Nie

Które z operacji zamykają strumień:

- map - tylko mapuje wartości
- filter - tylko filtruje wartości
- **collect** - zamyka strumień i zwraca wartości w formie kolekcji
- **findAny** - zamyka strumień i zwraca wartość jeśli daną znalazł

Czy może być więcej niż jedną operację terminalną w strumieniu:

- Tak
- **Nie**

Który z operatorów zamienia jeden typ na drugi:

- **map**
- filter
- peek
- collect

Który z zapisów to method reference:

- System.out.println()
- System.out::println()
- **System.out::println**
- System.out:println()

W File Input Stream koniec pliku oznaczamy przez:

- **-1**
- 0
- 1
- Exception

W jakiej sekcji należy zamykać pliki:

- catch()
- **finally()**
- try()
- close()

Który mechanizm przekształca strumień danych w tokeny:

- **Scanner**
- Formatter
- BufferedStream
- FileWriter

Kolofon

The CodeCouple.pl Press, Krzysztof Chruściel

Opublikowana dla wszystkich chcących się rozwijać!