

Java - Wprowadzenie do języka

sdacademy.pl

Version 1.1.13, 15-10-2018

Table of Contents

Przedmowa	1
Wstęp	2
JRE - Java Runtime Environment	2
JDK - Java Development Kit	2
Wersja Javy	2
HelloWorld	2
Kompilacja	3
Uruchomienie	3
Archiwum	3
Zadania	6
IDE - Integrated Development Environment	7
IntelliJ	7
Meta-dane	7
Skróty	7
Live Templates	7
Pluginy	8
Zadania	8
Zależności	9
Maven	9
Konwencja ponad konfiguracje	9
pom.xml	9
GAV	10
Semantic Versioning	10
Dependencies	11
Maven Central	12
Cykl życia	12
Fazy	12
.m2	13
Zadania	14
Elementy języka	15
Pakiet (package)	15
Klasa (class)	15
Tworzenie klasy (new)	16
Pole (field)	16
Używanie pól	16
Metoda (method)	16
Wywołanie metody	17
Zmienne (variables)	17

Używanie zmiennych	18
Konstruktor (constructor)	18
Wywołanie konstruktora	19
Modyfikatory dostępu	19
Dziedziczenie (extends)	20
Importowanie (import)	20
Zadania	20
Testowanie	22
JUnit	22
Struktura testu	22
Asercje	23
Cykl życia	24
Zadania	25
Typy danych	27
Typy proste (prymitywne)	27
Typy złożone (obiektowe)	29
Autoboxing	30
Autounboxing	31
Operatory	32
Operatory arytmetyczne	32
Skrócona wersja operatorów	33
Operatory porównawcze	33
Operatory logiczne	34
Inkrementacja	35
Dekrementacja	35
Tablice	36
Tworzenie	36
Odczyt	36
Rozmiar	37
Varargs	37
Zadania	37
Pętle (loops)	39
for	39
while	39
do while	40
for each	41
Zadania	41
Literał (String)	43
Tworzenie	43
Konkatenacja	43
Metody	43

Object	48
toString	48
hashCode	48
equals	48
getClass	49
wait, notify i notifyAll	49
clone	49
finalize	49
Zadania	49
equals i hashCode	51
equals	51
hashCode	52
Kontrakt	53
Zadania	53
Instrukcje warunkowe	54
if	54
if, else	54
if, else if, else	54
switch	55
Zadania	55
OOP (Object Oriented Programming)	57
Obiekt	57
Założenia	57
Elementy statyczne i finalne	59
Elementy statyczne	59
Elementy finalne	61
Elementy statyczne i finalne - stałe	63
Zadania	64
Interfejs	66
Tworzenie	66
Implementowanie	66
Metody domyślne	67
Dziedziczenie	67
Stałe w interfejsie	68
Zadania	68
Klasa abstrakcyjna	70
Metoda abstrakcyjna	70
Klasa abstrakcyjna vs interfejs	71
Zadania	71
Enum	72
Wywołanie	72

Pola w enumeratorze	72
Iteracja	73
Zadania.....	73
Wyjątki	75
Checked exceptions	75
Unchecked exceptions	75
Rzucanie wyjątków (throw new Wyjątek())	75
Obsługa wyjątków (try/catch/finally)	76
Zadania.....	77
JavaDoc	78
Tworzenie	78
Dyrektywy	79
@author	79
@version	80
@since.....	80
@depracted	81
@param	81
@throws.....	82
@link.....	82
@see	82
Zadania.....	83
Adnotacje	84
Tworzenie	84
Używanie	84
Zasięg (target).....	85
Retencja	85
Zadania.....	85
Czas (LocalTime).....	87
Tworzenie	87
Zmiana czasu	87
Elementy składowe	87
Sprawdzanie czasu	88
Zakres czasu	88
Zadania.....	88
Data (LocalDate)	89
Tworzenie	89
Zmiana daty	89
Elementy składowe	89
Sprawdzanie daty	90
Zakres dat	90
Zadania.....	90

Data i czas	91
Kolekcje (Collections)	92
Zbiory (Set)	92
Listy (List)	94
Kolejki (Queue)	95
Mapy (Map)	97
Iterator	99
Collections	99
Dodatkowe zadania	100
Typy generyczne	101
Tworzenie	101
Ograniczenie typów	101
Zadania	102
Optional	103
Tworzenie	103
Odczyt	103
Wartość domyslna	103
Zadania	104
Interfejsy funkcyjne	105
Function <T, R>	105
Consumer <T>	105
Supplier <T>	106
Predicate <T>	107
Własny interfejs funkcyjny	107
Zadania	107
Lambda	109
Składnia	109
Typy	109
Ciało lambdy	109
Zadania	110
Strumienie	111
Tworzenie	111
Operacje pośrednie	111
Operacje terminalne	111
Strumienie a interfejsy funkcyjne	112
Strumienie a kolekcje	112
Method reference	112
Zadania	112
InputOutput (IO)	114
Byte Streams	114
File Input/Output Stream	114

Character Streams	115
Buffered Streams	116
Formatowanie danych	116
Data Streams	119
Object Streams	119
Zadania	120
New InputOutput (NIO)	121
Ścieżka (Path)	121
Pliki (Files)	121
Informacje o folderze/pliku	121
Usuwanie folderu/pliku	122
Kopiowanie folderu/pliku	122
Przenoszenie folderu/pliku	122
Tworzenie plików/folderów	123
Czytanie z pliku	123
Zapisywanie do pliku	123
Zadania	123
Wielowątkowość	125
Main	125
Thread	125
run vs start	125
Runnable	126
Pula wątków	127
Zadania	128
Debugowanie	129
Breakpoint	129
Ramki	129
Wątki	130
Zmienne	130
Wywołanie	131
Skróty	132
Zadania	132
Odpowiedzi	135
Wstęp	135
Zależności	137
Elementy języka	138
Testowanie	141
Typy danych	143
Operatory	151
Tablice	158
Pętle (loops)	160

Literał (String)	162
Object	166
equals i hashCode	167
Instrukcje warunkowe	170
Elementy statyczne	174
Interfejs	177
Klasa abstrakcyjna	181
Enum	185
Wyjątki	187
JavaDoc	189
Adnotacje	190
Czas (LocalTime)	192
Data (LocalDate)	193
Kolekcje (Collections)	194
Typy generyczne	201
Optional	204
Interfejsy funkcyjne	206
Lambda	209
Strumienie	210
InputOutput (IO)	212
New InputOutput (NIO)	216
Wielowątkowość	217
Kolofon	218

Przedmowa



Cześć, nazywam się **Krzysztof Chruściel**. Jestem Java developerem od kilku lat. Stworzyłem tą książkę z myślą o ludziach, którzy chcą zacząć swoją przygodę z programowaniem w języku **Java**. Znajdzicie w niej bardzo szeroki zakres materiału. Na początku zaczniemy od komplikacji 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 dziale 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 komplikacji w folderze znajdują się dwa pliki:

- `Runner.java` - kod źródłowy
- `Runner.class` - skompilowana klasa z kodem bajtowym (więcej o kodzie 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. Wysydasz 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 wersjach:

- **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.имл** 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 rozwiązań typu **Live Templates**. **Live Templates** pozwala na definiowanie 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 konwencję 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 (moga 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 umieściłybyśmy 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 znany nam pliku **pom.xml** w sekcji **dependencies** dodajemy nową zależność:

```
<?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

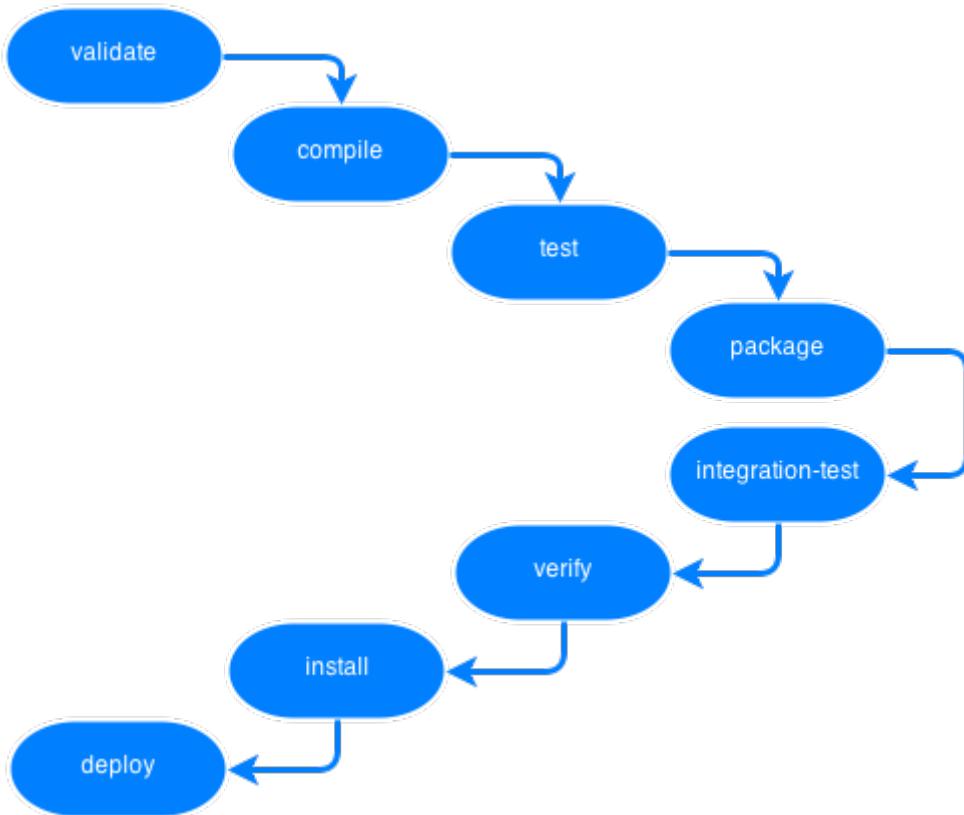
Jeśli poszukujemy artefaktów stworzonych przez innych musimy udać się do repozytorium zdalnego. Repozytorium zdalnym dla projektu **Maven** jest **Maven Central**, które jest dostępne pod adresem: <https://mvnrepository.com/>

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 poza 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 działań 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 (konstruowania) obiektu.

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

W naszym przykładzie:

```
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 pakietu, 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

- Stwórzyc 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 chronionym 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 $10 + \text{oprocentowanie}$)
- 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 $15 + \text{oprocentowanie}$)
- 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 stosuje metodę 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 poprawnej 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 numberOfWorks = maluch.getNumberOfWheels();
```

Then

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

CarTest.java

```
//Then  
assertEquals(numberOfWorks, 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** dostarcza całą game assercji dzięki którym możemy sprawdzić poprawność kodu:

- assertEquals(expected, actual) - **assertEquals(numberOfWorks, 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 **expected** 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 mechanizm 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

```
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

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

```
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 zajmuje **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 zajmuje **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 **-2³¹ do 2³¹⁻¹**. W

pamięci zajmuje **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 '`I`'/'`L`'. W pamięci zajmuje **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 `1000000000000L`) 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 zmienoprzecinkowych. Może przechowywać do 6-7 cyfr po przecinku. W pamięci zajmuje **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 zmienoprzecinkowych. Może on przechowywać więcej liczb w pamięci w stosunku do `float` (około 15 cyfr po przecinku). W pamięci zajmuje **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 zajmuje **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 height; // 4 bajty

} // rozmiar = 4 + 4 = 8 bajtów

class Car {

    int numberOfWheels; // 4 bajty
    Window leftWindow; // 8 bajtów
    Window rightWindow; // 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 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 (+)

Dodawanie wykonuje się operatorem `+`.

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

Odejmowanie (-)

Odejmowanie wykonuje się operatorem `-`.

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

Mnożenie (*)

Mnożenie wykonuje się operatorem `*`.

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

Dzielenie całkowite (/)

Dzielenie wykonuje się operatorem `/`.

- Stwórz test oraz implementację dla operacji dzielenia (`int div(int first, int second)`) dla cyfr 17 i 4

Reszta z dzielenia (modulo) (%)

Reszte z dzielenia wyliczamy przy wykorzystaniu operatora `%`.

- Stwórz test oraz implementację dla operacji modulo (`int mod(int first, int second)`) dla cyfr 17 i 4
- Stwórz test oraz implementację dla operacji modulo (`double mod(double first, int second)`) dla cyfr 17.0 i 4

Skrócona wersja operatorów

Operatory arytmetyczne mogą wystąpić także w wersji skróconej. Pozwala to na oszczędzanie czasu:

```
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
```

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. Czyli czy mają wskaźnik na tę samą referencję:

```
Car maluch = new Car();
Car drugiMaluch = maluch;
maluch == drugiMaluch // true
```

- Stwórz klasę **Comparator**
- Stwórz klasę testową **ComparatorTest** dla klasy **Comparator**
- Napisz test i implementację dla metody **boolean compare(String text, String textToCompare)**

Różny (!=)

Operator **!=** sprawdza czy dwa obiekty są różne. Czyli czy nie mają wskaźnika na tę samą referencję:

```
Car maluch = new Car();
Car drugiMaluch = maluch;
maluch != drugiMaluch // false
```

- W klasie testowej **ComparatorTest** dodaj test dla metody **boolean areDifferent(String text, String textToCompare)**
- W klasie testowej **ComparatorTest** dodaj test dla metody **boolean areDifferent(String text, String textToCompare)** z tym, że przekaż nowe instancje klasy **String**

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.

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

Operatory logiczne

Operatory logiczne służą do łączenia warunków logicznych.

Koniunkcja (AND) (`&&`)

Do łączenia warunków logiczny możemy wykorzystać operator koniunkcji `&&`. Przykładowo chcemy sprawdzić czy liczba jest mniejsza od 100 ale większa od 10:

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

Alternatywa (OR) (`||`)

Operator alternatywy (lub) `||` służy do łączenia warunków. Przykładowo interesuje nas liczba większa od pięć lub mniejsza od 3:

```
int x = 4;  
x > 5 || x > 3 // false true = true
```

Negacja (NOT) (`!`)

Operator negacji `!` służy do zanegowania wartości, czyli zwrócenie wartości przeciwnej:

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

Inkrementacja

Bardzo często wykorzystywany operatorem jest operator **inkrementacji `++`**. Zwiększa on wartość zmiennej o jeden.

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

Operator ten może występować w dwóch wersjach:

- **prefixowej** - zwiększa wartość przed przypisaniem

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

- **postfixowej** - zwiększa wartość zmiennej po przypisaniu

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

- Stwórz test oraz implementację dla operacji dodawania o jeden (`int addOne(int numer)`)

Dekrementacja

Bardzo często wykorzystywany operatorem jest operator **dekrementacji `--`**. Zmniejsza on wartość zmiennej o jeden.

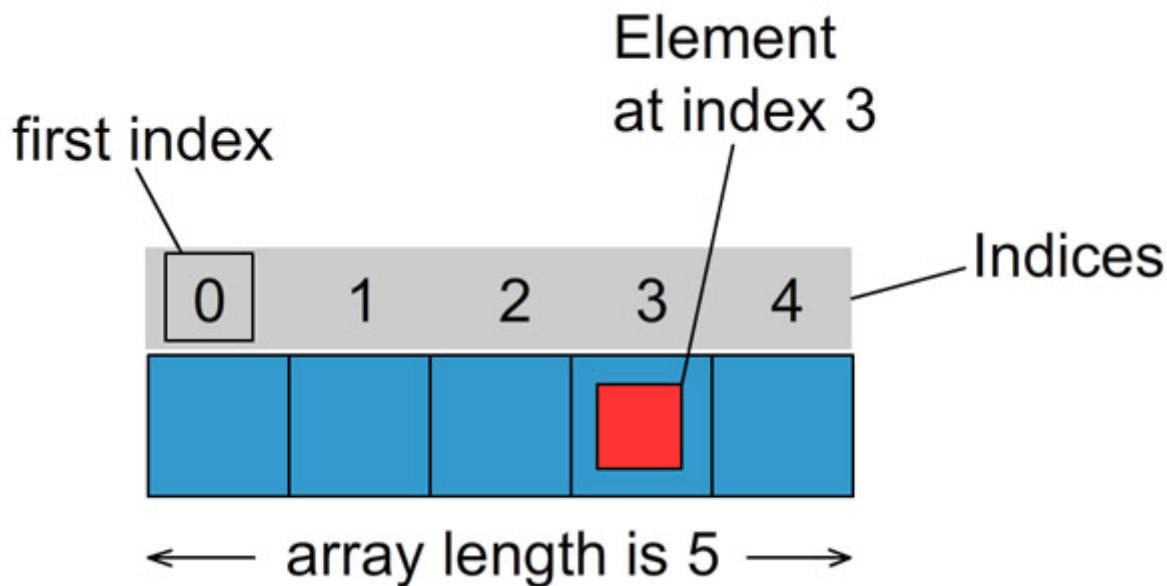
```
int x = 10;  
x--; // 10 - 1 = 9
```

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

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

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-początkowe; 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

```

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 zwiększonimi wartościami o jeden

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

- Stworzyć metodę testową dla klasy `LoopExample` sprawdzający metodę `int[] fillDoWhile(int[] tab)`

Literał (String)

Ten typ danych pojawił się już w poprzednich Ćwiczeniach. 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 zwany **String Pool**. **String** jest typem, który jest **niemutowalny**. Oznacza to, iż każda modyfikacja powoduje powstanie nowego literala!

Tworzenie

String jest typem obiektowym 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 obiektowym 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 zwiększenia 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 znajduje 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. Domyślana 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*). Domyślana 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 nadisaną 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 umieszczania 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 domyślana  
// implementacja hashCode zwraca hex(adres-w-pamięci)  
    // logika  
}
```



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

Nadpisana metoda **hashCode** "pod spodem" wykorzysty whole najczesciej mnozenie przez liczby pierwsze:

```
@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("Podajeś 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("Podajeś liczbę 10!");  
} else {  
    System.out.println("Nie podajeś 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("Podajeś liczbę 1!");  
} else if (liczbaDoSprawdzenia == 10) {  
    System.out.println("Podajeś liczbę 10!");  
} else if (liczbaDoSprawdzenia == 100) {  
    System.out.println("Podajeś liczbę 100!");  
} else if (liczbaDoSprawdzenia === 1000) {  
    System.out.println("Podajeś 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("Podajeś liczbę 1!");
        break;
    case 10:
        System.out.println("Podajeś liczbę 10!");
        break;
    case 100:
        System.out.println("Podajeś liczbę 100!");
        break;
    case 1000:
        System.out.println("Podajeś 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 zwrócić 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
```

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** określającą 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 rozwiązyania problemu i zwiększenie jego ogólności.

Hermetryzacja

Hermetryzacja 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 tą, której nie ma obiekt ogólniejszy.

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 statycznego 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; // compilation 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 { // compilation 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() { // compilation 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:

```
final class FinalField {  
  
    final int value = 10;  
    final int secondValue;  
    final int thirdValue; // compilation error  
  
    FinalClass (int secondValue) {  
        this.secondValue = secondValue;  
    }  
  
}
```

Elementy finalne - zmienna

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

```
final class FinalVariable {  
  
    void someMethod() {  
        final int value = 20;  
        value = 30; // compilation error  
    }  
  
}
```

Elementy finalne - parametr

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

```
final class FinalParameter {  
  
    void someMethod(final int value) {  
        value = 20; // compilation 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

- Utwórz 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 stałą) 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 interfejs, 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 {  
  
}
```

```
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:

```
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:

```
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

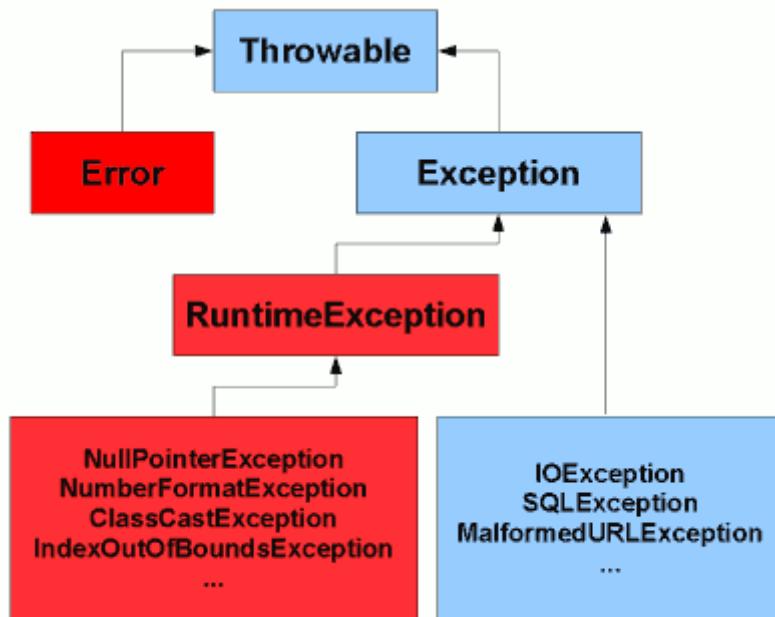
- Stworzyć nowy projekt **Maven** z nazwą `enum-example`
- Utwórz enumerator `WeekDay.java`, który zawiera wszystkie dni tygodnia
- Utwórz klasę `DayPrinter.java` z metodą `void printDayBy(WeekDay weekDay)`, która wypisze

(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 enumeratorem **Month**

Wyjątki

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



Checked exceptions

Są to wyjątki, które trzeba obsługiwać (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**):

```
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 wyjątków, jednakże należy pamiętać o hierarchii "od szczegółu do ogólnego". 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

- Stwórz 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

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

@author

Dyrektwa **@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 uzupełnić 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 opcjonalnyParamter(); //Możemy definiować opcjonalne parametry
    String parametr() default "Value";

}
```

Używanie

Umieszczanie **adnotacji**:

AnnotationExample.java

```
class AnnotationExample {

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

}
```

Z **adnotacją** 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` - umieszczone w skompilowanej klasie (wykorzystywana do modyfikacji byte kodu. Jest to domyślna retencja)
- `RetentionPolicy.RUNTIME` - dostępne w trakcie działania programu (refleksja)
- `RetentionPolicy.SOURCE` - usuwane przez kompilator w trakcie komplikacji

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 parametru `name` i `surname`

Wyszukiwanie adnotacji

- Stwórz 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 dodawania 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 dodawania 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

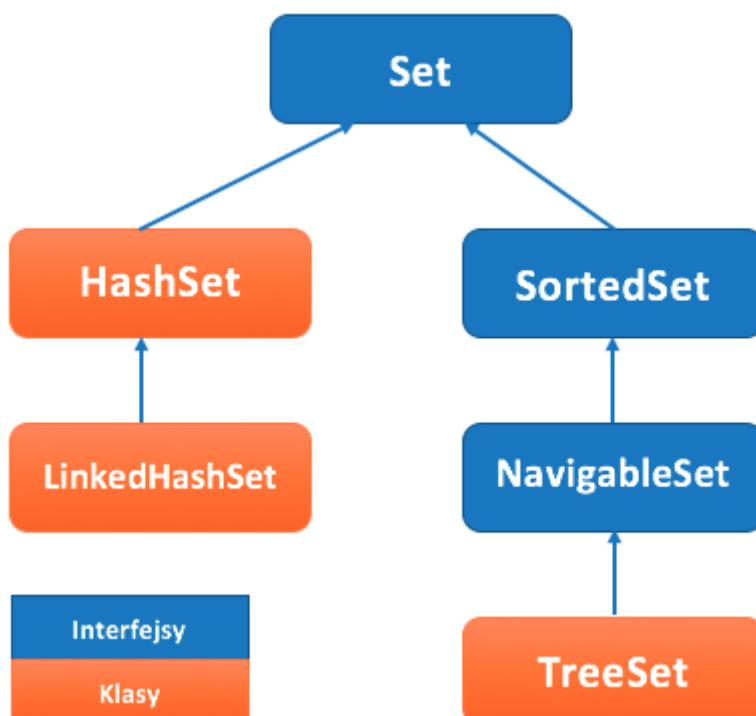
Jeśli chcemy pobrać jednocześnie datę i czas korzystamy z klasy `LocalDateTime`, która jest kombinacją `LocalDate` i `LocalTime`.

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 tablicy to niestety co jakiś czas musielibyśmy **tworzyć nową**. Ponadto, należało by pamiętać ostatnio użyty indeks. To utrudnione wykorzystywanie tablic doprowadziło do postawienia framework'u **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 zachowuje 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]
```

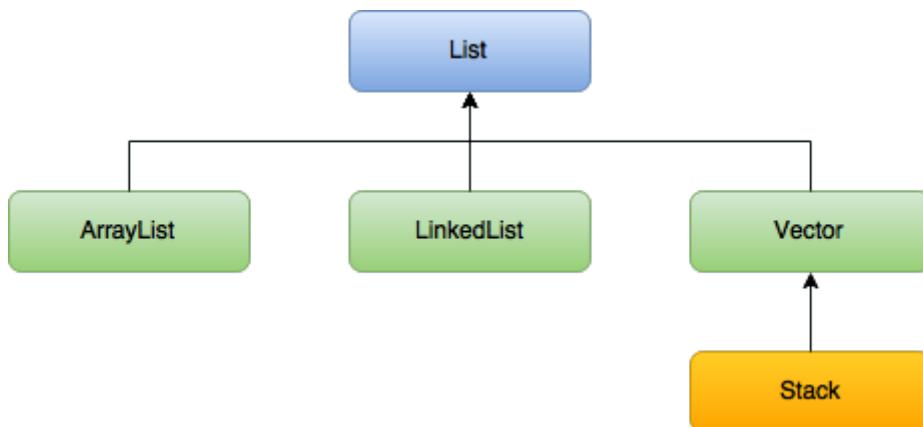
Szybkość działania kolekcji **TreeSet** określa się mianem $O(\log(n))$.

Zadania

- Stwórz nowy projekt **Maven** o nazwie **collection-example**
- Stwórz klasę **Car** z jednym polem **final int value** ustawianym w konstruktorze
- Stwórz klasę testową **SetTest**, w której przetestujesz metodę z interfejsu **Set**:
 - **add** (spróbuj dodać kilka takich samych instancji klasy **Car**)
 - nadpisz **equals** i **hashCode** w klasie **Car**
 - przetestuj ponowanie metody **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" implementuje listę dwukierunkową. Odczyt danych odbywa się w czasie $O(n)$ natomiast wstawianie elementów w czasie $O(1)$:

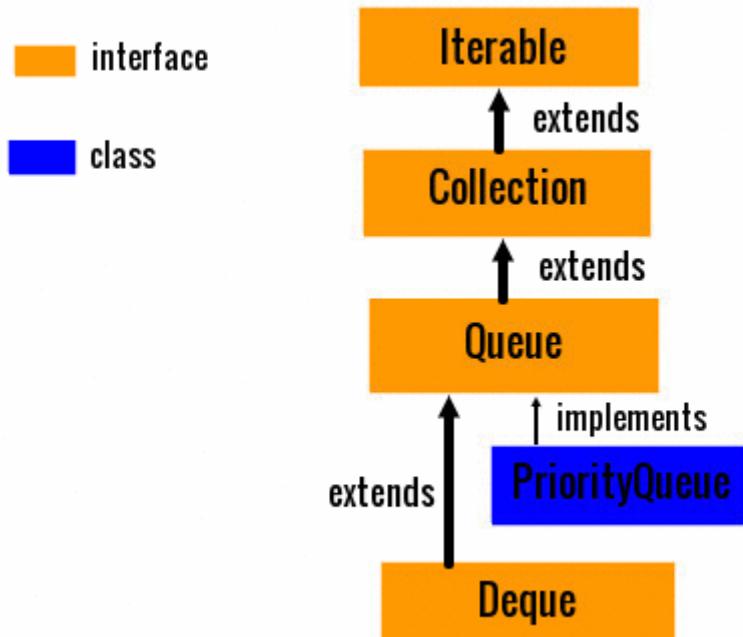
```
LinkedList<String> linkedList = new LinkedList<>();
```

Zadania

- Wykorzystaj projekt stworzony w zadaniach ze zbiorami
- Stwórz klasę testową `ListTest`, w której przetestujesz metody z interfejsu `List`:
 - `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:

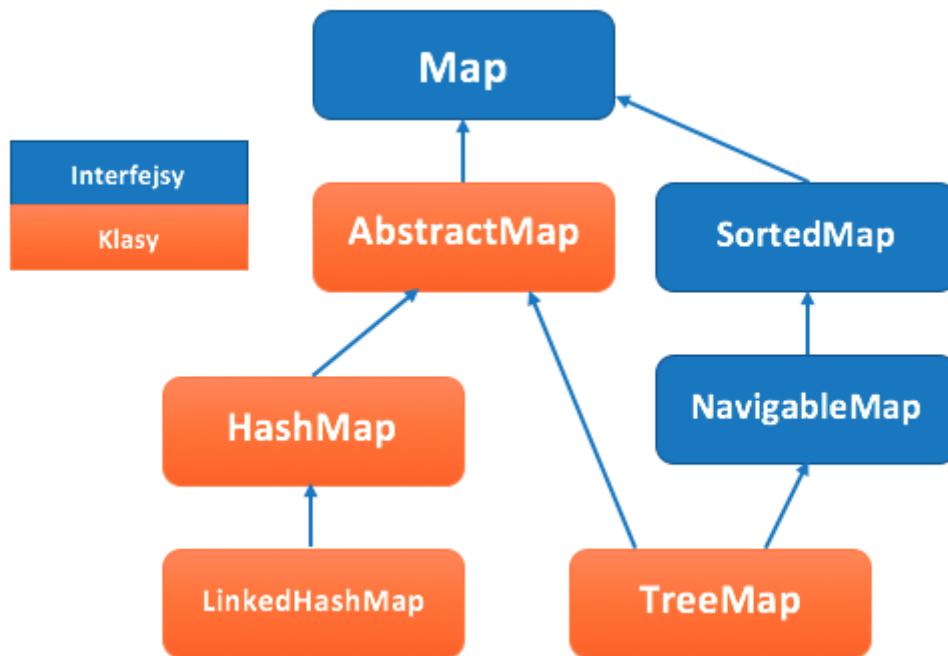
```

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:

```
Map<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 **synchronizowane**. 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 zadaniach ze zbiorami
- Stwórz klasę testową `MapTest`, w której przetestujesz metody z interfejsu `Map`:
 - `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 związaną 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 zadaniach ze zbiorami
- Stwórz klasę testową `CollectionsTest`, w której przetestujesz metody z klasy `Collections`:
 - `frequency`
 - `max`
 - `min`
 - `reverse`

Dodatkowe zadania

- * Stwórz klasę testową, w której przetestujesz `PriorityQueue`
- * Stwórz klasę testową, w której przetestujesz `LinkedHashSet`
- * Stwórz klasę testową, w której przetestujesz `LinkedHashMap`
- * Stwórz klasę testową, w której przetestujesz `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 **rezywane**. 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 parametru. 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**:

```
interface Car {  
    void takeOffWhell();  
}
```

```
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**, która posiada metodę abstrakcyjną **void prepare()** (w klasach nadpisujących niech wypisują pole **name**)
- Klasa **Food** powinna przyjmować i ustawiać dwa pola **protected final String name** i **protected final String weight** w konstruktorze
- 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 klasę 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 wartością 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 predykt sprawdzający czy podana liczba jest parzysta

- Stworzyć supplier'a zwracający losową liczbę całkowitą (sprawdź jak pobrać liczbę losową w **Javie**)
- Stworzyć consumer'a wypisującego podaną liczbę
- Stworzyć funkcję, która przyjmuje liczbę całkowitą i podnosi ją do potęgi o tą samą wartość co podana ($4 \rightarrow 4^4$) (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 przypomnimy sobie jak wygląda **interfejs funkcyjny supplier** okaże się, że nie przyjmuje żadnego parametru (dla tego parametr w **lambdzie** jest opcjonalny):

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

Typy

Składnia wyrażenia **lambda** może zaciemniać typ wprowadzonych parametrów, dla tego 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 "inný dzień";
};

() -> {};
```

Zadania

- Stwórz nowy projekt **Maven** o nazwie **lambda-example**
- Utwórz nowy interfejs funkcyjny **StringSupplier** z jedną metodą **String string()**;
- Utwórz nową klasę **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 **StringSupplier**

Strumienie

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

Tworzenie

Strumienie można tworzyć na kilka sposobów. 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 framework'u **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.

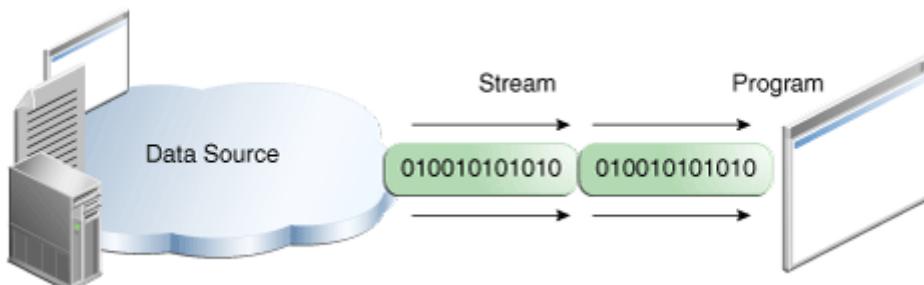
Zadania

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

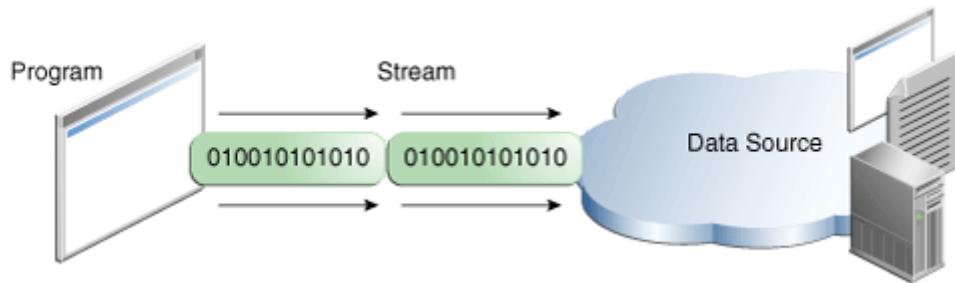
-
- "fifth"
 - z utworzonego strumienia zwróć słowa dłuższe niż 5 znaków
 - oraz zwróć je w formacie UPPERCASE
- stwórz strumień z pięcioma elementami typu **String**:
- "first"
 - "second"
 - "third"
 - "fourth"
 - "fifth"
- zwróć pierwszy element który ma więcej niż 7 znaków
- stwórz strumień z pięcioma 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 podstawowym 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 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 zamknięty, wywołamy metodę **flush** lub bufor się przepłni.

Formatowanie danych

Operowanie na odczytywanych bajtach może być trudnym zagadnieniem. Aby ułatwić ten proces powstały dwa mechanizmy, skanery przekształcające strumień danych w tokeny i formatory 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 sformatowanych w standardowy sposób:

Printer.java

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

println

Metoda `println` służy do wyświetlnia w nowej linii strumienia danych sformatowanych 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`.

```
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

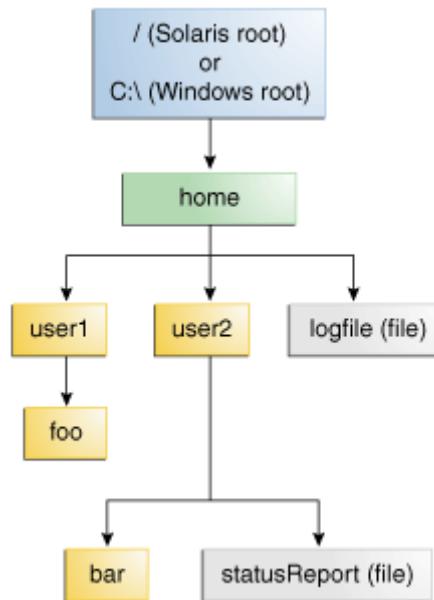
- Stworzyć nowy projekt **Maven** z nazwą **io-example**
- Utworzyć klasę **Runner** z **psvm**
- Stworzyć nowy plik **file.txt** w folderze projektu i dodać do niego kilka linii tekstu
- W metodzie **main** wypisać bajty z pliku na konsoli korzystając z **FileInputStream**
- W metodzie **main** skopiować 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 pathShortcutFor = FileSystems.getDefault().getPath("some/path");
```

Pliki (Files)

Kolejną ważną 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 ułatwiających 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 rekursively (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 zrównoleglić naszą pracę. Zrównoleglenie pracy możemy osiągnąć poprzez stworzenie nowego **wątku**. Uruchomiony program nazywany jest **procesem**, natomiast proces ma uruchomione **wątki** (zadania).

Main

Główny wątek programu nazywany jest wąkiem **main**. Jeśli chcemy pobrać aktualny wątek 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. Jeśli chcemy stworzyć nowy wątek tworzymy nową klasę, która dziedziczy po **Thread** lub tworzymy jej nową instancję:

Fred.java

```
class Fred extends Thread {  
  
    @Override  
    public void run() {  
        //logic  
    }  
}
```

run vs start



O różnicę pomiędzy tymi metodami pytają na rozmowach kwalifikacyjnych!

W kontekście klasy **Thread** pojawiają się dwie metody **run** i **start**. Choć z pozoru wyglądają bardzo podobnie mają inne przeznaczenie. W metodzie **run** umieszczamy logikę, która będzie wykonana w nowym wątku. Natomiast metoda **start** uruchamia nowy wątek:

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 interfejsu tego korzystamy wtedy, kiedy chcemy utworzyć nowe zadanie:

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:

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);
```



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
- Stworzyć nową klasę implementującą interfejs **Runnable** i dodać w niej logikę wypisania nazwy wątku
- W metodzie **main** uruchomić nowo stworzony wątek
- W metodzie **main** stworzyć nową pulę **10 wątków**
- Na nowo utworzonej puli uruchomić wcześniej utworzone zadanie **Runnable**

Debugowanie

Debugowanie jest bardzo przydatną (jak nieobowiązkową) umiejętnością każdego programisty. Jest to proces śledzenia kodu w celu wykrycia **błędu** lub **zrozumienia algorytmu**. Podczas tego procesu możemy podstawać 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 miejscach w kodzie:

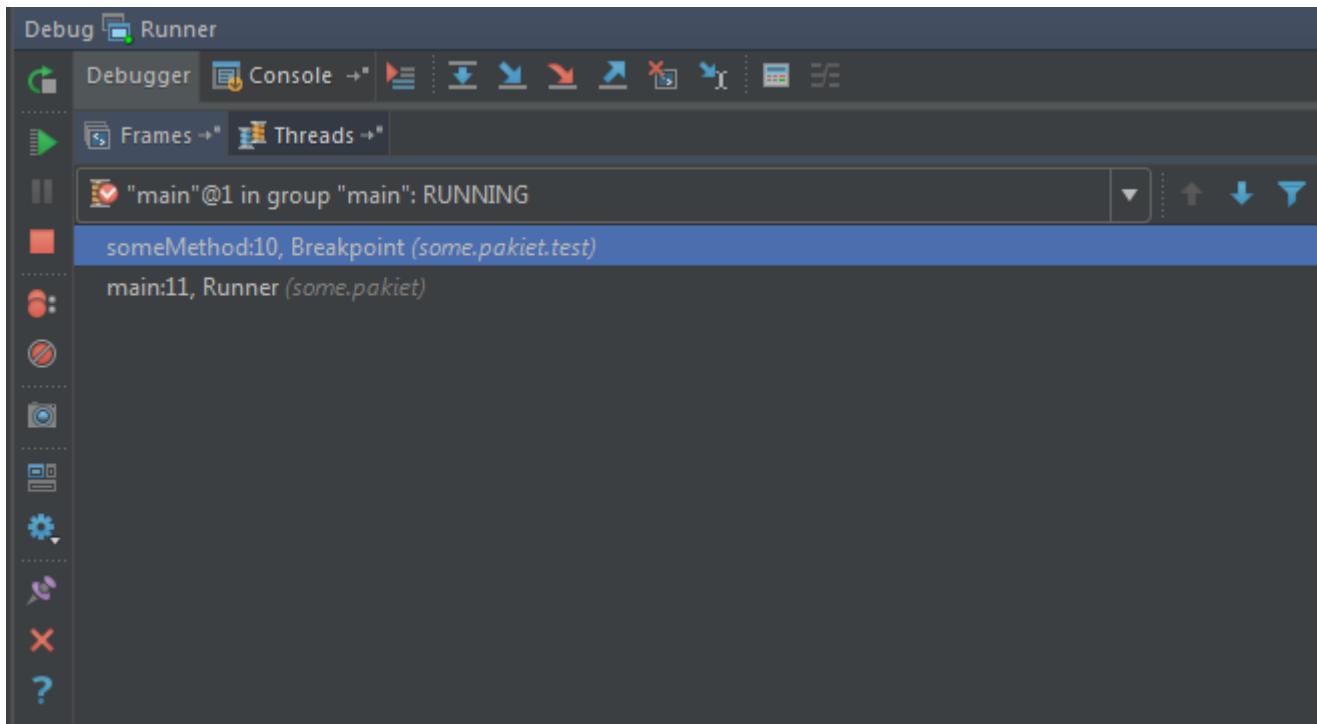
```
3
4 ● class Breakpoint {
5
6 ○ - int field;
7
8 ○ + void someMethod() {
9
10 ● System.out.println("Some Line");
11
12 }
13
14 }
```

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

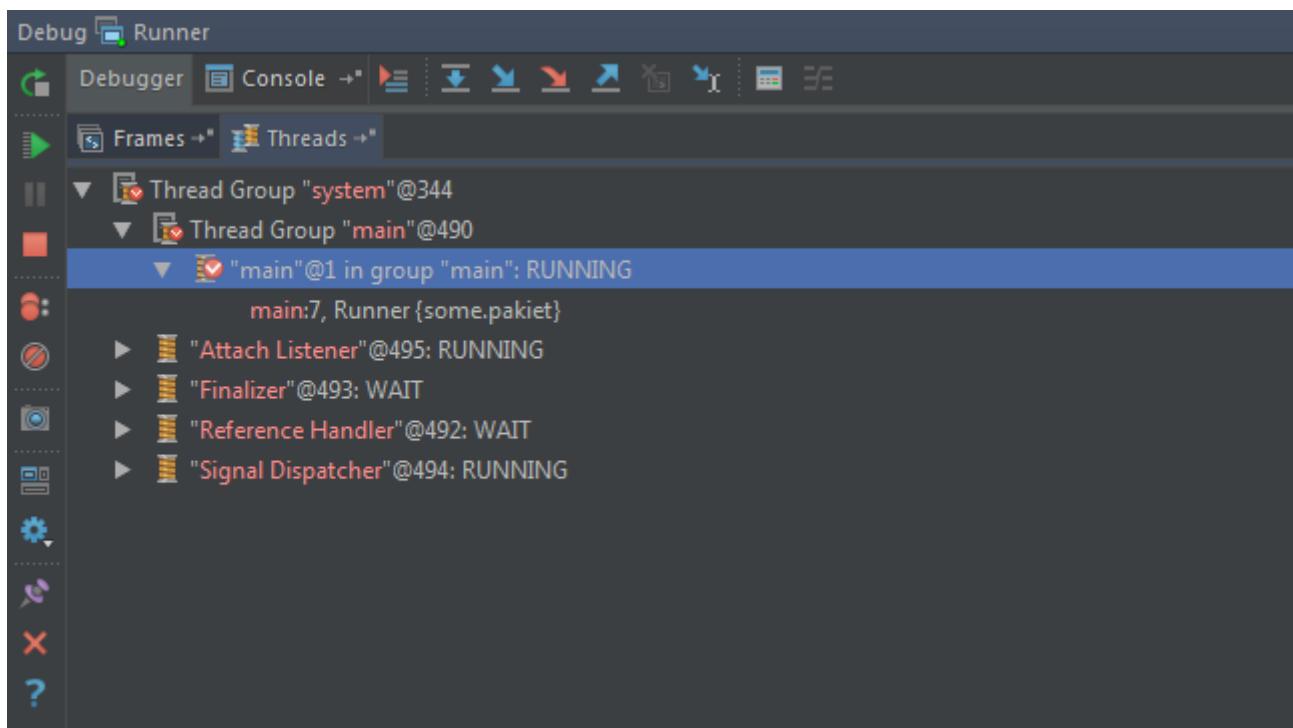
Przy każdym wywołaniu kodu "odkładana" jest ramka na stos. W widoku poniżej możemy sprawdzić cały stos wywołań:



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

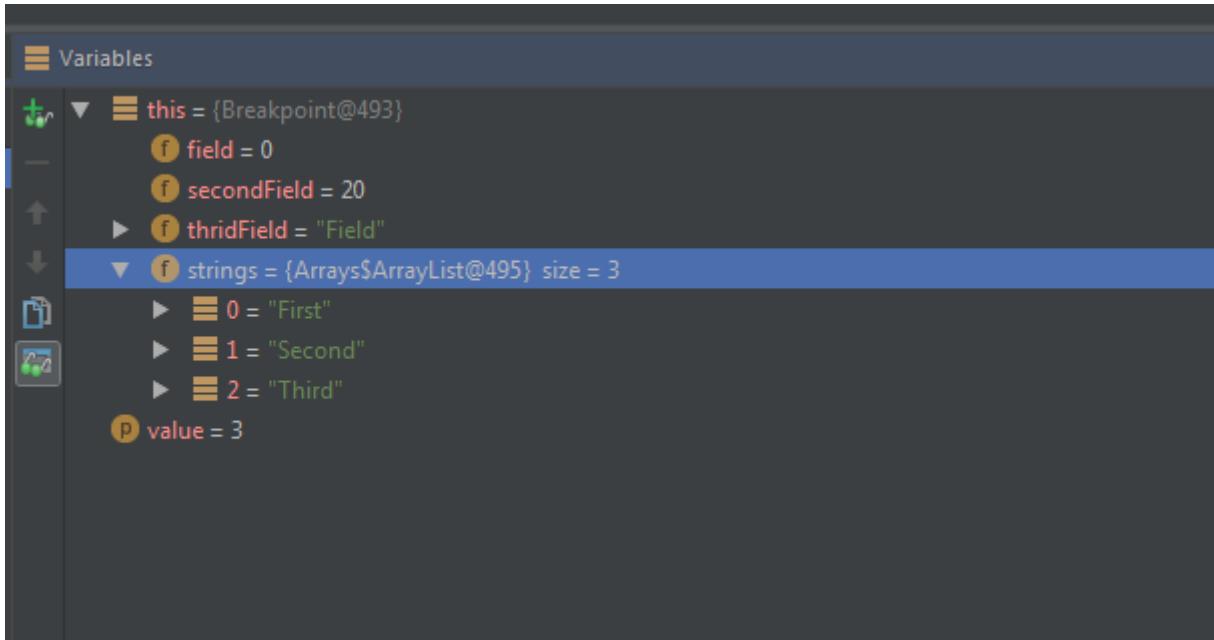
Wątki

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



Zmienne

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

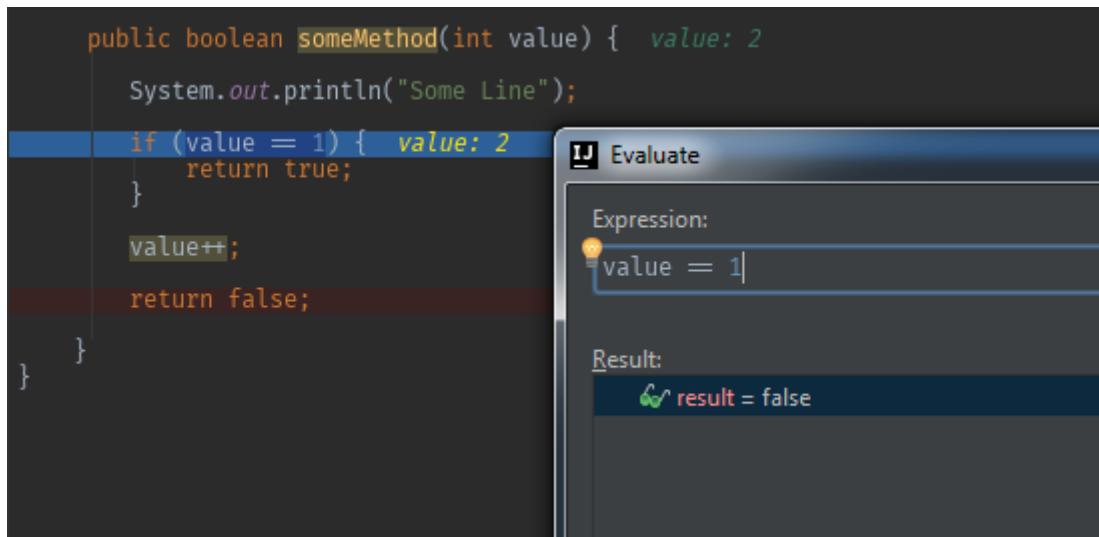


Ponadto, IntelliJ w kodzie pokazuje nam aktualne wartości:

```
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 pewien fragmentu kodu wystarczy, że go zaznaczmy 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 **Runner** z **psvm**:

Runner.java

```
class Runner {

    public static void main(String[] args) throws InterruptedException {
        new Debug().divisible(7);
        new Job().start();
        while(true) {
            Thread.sleep(5000);
            System.out.println("Janusz");
        }
    }
}
```

- 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;  
        }  
    }  
}
```

- 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

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`:

```
package pl.sda.bank;

public class Bank {

    protected String imionDluznikow() {
        return "Jan" + "Tomasz";
    }

}
```

Klasa `BankPKO`:

```
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`:

```
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**:

```
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**:

```
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`:

```
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 numberOfWorks = maluch.getNumberOfWheels();  
        // Then  
        assertThat(numberOfWorks).isEqualTo(4);  
    }  
}
```

Typy danych

Typy proste (prymitywne)

Testy dla klasy `PrimitiveTypes`:

PrimitiveTypesTest.java

```
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(1000000000000L);
}

@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 doubleDefault = primitiveTypes.doubleDefault;
    // Then
    assertThat(doubleDefault).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`:

```
class PrimitiveTypes {  
  
    byte byteDefault;  
    byte byteExample = 100;  
  
    short shortDefault;  
    short shortExample = 100;  
  
    int intDefault;  
    int intExample = 100;  
  
    long longDefault;  
    long longExample = 1000000000000L;  
  
    float floatDefault;  
    float floatExample = 1.0f;  
  
    double doubleDefault;  
    double doubleExample = 1.0;  
  
    boolean booleanDefault;  
    boolean booleanExample = true;  
  
    char defaultChar;  
    char charExample = 'A';  
  
}
```

Typy złożone (obiektowe)

Testy dla klasy **ObjectTypes**:

ObjectTypesTest.java

```
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);  
    }  
}
```

ObjectTypes.java

Implementacja klasy **ObjectTypes**:

```
class ObjectTypes {  
  
    String stringExample = "text";  
    String stringNull;  
    String stringNewExample = new String("text");  
  
    Integer integerExample = new Integer(1);  
  
}
```

Autoboxing

Testy dla klasy Autoboxing:

```
class AutoboxingTest {  
  
    @Test  
    void shouldReturnAutoboxedValue() {  
        // Given  
        Autoboxing autoboxing = new Autoboxing();  
        // When  
        Integer result = autoboxing.autoboxingExample;  
        // Then  
        assertThat(result).isEqualTo(1);  
    }  
  
}
```

Implementacja klasy Autoboxing:

```
class Autoboxing {  
  
    Integer autoboxingExample = 1;  
  
}
```

Autounboxing

Testy dla klasy Autounboxing:

```
class AutounboxingTest {  
  
    @Test  
    void shouldReturnAutounboxedValue() {  
        // Given  
        Autounboxing autounboxing = new Autounboxing();  
        // When  
        int result = autounboxing.autounboxingExample;  
        // Then  
        assertThat(result).isEqualTo(1);  
    }  
  
}
```

Implementacja klasy **Autounboxing**:

```
class Autounboxing {  
  
    int autounboxingExample = new Integer(1);  
  
}
```

Operatory

Operatory arytmetyczne

Testy dla klasy `Calculator`:

CalculatorTest.java

```
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 shouldMultTwoNumbers() {
        // 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(20, 10);
        // Then
        assertThat(resultToCheck).isEqualTo(2);
    }
}
```

```
@Test
void shouldModTwoNumbers() {
    // Given
    Calculator calculator = new Calculator();
    // When
    int resultToCheck = calculator.mod(17, 4);
    // Then
    assertThat(resultToCheck).isEqualTo(1);
}

@Test
void shouldModOnTwoDoubleNumbers() {
    // Given
    Calculator calculator = new Calculator();
    // When
    double resultToCheck = calculator.mod(17.0, 4);
    // Then
    assertThat(resultToCheck).isEqualTo(1.0);
}

}
```

Implementacja klasy **Calculator**:

```
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;  
    }  
  
    int mod(int first, int second) {  
        return first % second;  
    }  
  
    double mod(double first, int second) {  
        return first % second;  
    }  
  
}
```

Operatory porównawcze

Testy dla klasy **Comparator**:

```
class ComparatorTest {  
  
    @Test  
    void shouldReturnTrueWhenTwoTextAreSameInMemory() {  
        // Given  
        String text = "text";  
        String textToCompare = "text";  
  
        Comparator comparator = new Comparator();  
        // When  
        boolean result = comparator.compare(text, textToCompare);  
        // Then  
        assertThat(result).isTrue();  
    }  
}
```

```

@Test
void shouldReturnFalseWhenTwoTextAreNotSameInMemory() {
    // Given
    String text = "text";
    String textToCompare = new String("text");

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.compare(text, textToCompare);
    // Then
    assertThat(result).isFalse();
}

@Test
void shouldReturnTrueWhenTwoTextsAreDifferent() {
    // Given
    String text = "text";
    String textToCompare = "textToCompare";

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.areDifferent(text, textToCompare);
    // Then
    assertThat(result).isTrue();
}

@Test
void shouldReturnFalseWhenTwoTextsAreSame() {
    // Given
    String text = "text";
    String textToCompare = "text";

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.areDifferent(text, textToCompare);
    // Then
    assertThat(result).isFalse();
}

@Test
void shouldReturnTrueWhenNumberIsLower() {
    int number = 1;
    int numberToCompare = 2;

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.isLower(number, numberToCompare);
    // Then
    assertThat(result).isTrue();
}

```

```

@Test
void shouldReturnFalseWhenNumberIsNotLower() {
    int number = 2;
    int numberToCompare = 1;

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.isLower(number, numberToCompare);
    // Then
    assertThat(result).isFalse();
}

@Test
void shouldReturnFalseWhenNumberIsNotGreater() {
    int number = 1;
    int numberToCompare = 10;

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.isGreater(number, numberToCompare);
    // Then
    assertThat(result).isFalse();
}

@Test
void shouldReturnTrueWhenNumberIsGreater() {
    int number = 10;
    int numberToCompare = 1;

    Comparator comparator = new Comparator();
    // When
    boolean result = comparator.isGreater(number, numberToCompare);
    // Then
    assertThat(result).isTrue();
}

}

```

Implementacja klasy `Comparator`:

```

class Comparator {

    boolean compare(String text, String textToCompare) {
        return text == textToCompare;
    }

    boolean areDifferent(String text, String textToCompare) {
        return text != textToCompare;
    }

    boolean isLower(int number, int numberToCompare) {
        return number < numberToCompare;
    }

    boolean isGreater(int number, int numberToCompare) {
        return number > numberToCompare;
    }

}

```

Inkrementacja

```

int addOne(int number) {
    return ++number;
}

@Test
void shouldAddOne() {
    // Given
    Calculator calculator = new Calculator();
    // When
    int result = calculator.addOne(10);
    // Then
    assertThat(result).isEqualTo(11);
}

```

Dekrementacja

```
int subOne(int number) {
    return --number;
}

@Test
void shouldAddOne() {
    // Given
    Calculator calculator = new Calculator();
    // When
    int result = calculator.subOne(10);
    // Then
    assertThat(result).isEqualTo(9);
}
```

Tablice

Testy dla klasy `ArrayExample`:

`ArrayExampleTest.java`

```
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

```

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

```
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](#):

```
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;
    }

}
```

Literał (String)

Testy dla klasy `StringExample`:

`StringExampleTest.java`

```
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

```

class StringExample {

    String concat(String first, String second) {
        return first + second;
    }

}

```

Object

Implementacja klasy `Runner`:

Runner.java

```
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

```
class ObjectExample {  
  
}
```

Implementacja klasy `ToStringObjectExample`:

ToStringObjectExample.java

```
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

```
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

```
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

```
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

```
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

```
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

```
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`

```
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
    assertEquals(monthName).isEqualToIgnoringCase("Taki miesiąc nie istnieje!");
}

}
```

Implementacja klasy **Condition**:

Condition.java

```
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

```
public class Runner {  
  
    public static void main(String[] args) {  
        Condition condition = new Condition();  
        condition.getMonthNamesBy(5);  
    }  
  
}
```

Elementy statyczne

Test dla klasy `MonthConstants`:

MonthConstantsTest.java

```
class MonthConstantsTest {  
  
    @Test  
    void shouldReturnCorrectMonth() {  
        // When  
        String monthName = MonthConstants.getMonthNameBy(5);  
        // Then  
        assertThat(monthName).isEqualTo("Maj");  
    }  
  
}
```

Implementacja klasy `MonthConstants`:

MonthConstants.java

```
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 "Błęd";  
        }  
    }  
}
```

Implementacja klasy **Runner**:

Runner.java

```
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

```
interface Vehicle {  
  
    void drive();  
  
}
```

Implementacja interfejsu `Payable.java`:

Payable.java

```
interface Payable {  
  
    void pay(int quantity);  
  
}
```

Implementacja klasy `Bus.java`:

Bus.java

```
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

```
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

```
class Car implements Vehicle {  
  
    @Override  
    public void drive() {  
        System.out.println("Drive by car");  
    }  
}
```

Implementacja klasy **Person.java**:

Person.java

```
class Person {  
  
    void driveBy(Vehicle vehicle) {  
        vehicle.drive();  
    }  
  
    void buyTicketsFor(Payable payable, int quantity) {  
        payable.pay(quantity);  
    }  
}
```

Implementacja klasy **Runner.java**:

Runner.java

```
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

```
interface JavaProgrammer {  
  
    void typeJava();  
  
}
```

Implementacja interfejsu *TableSoccerPlayer.java*:

TableSoccerPlayer.java

```
interface TableSoccerPlayer {  
  
    void playTableSoccer();  
  
}
```

Implementacja interfejsu *AwesomeProgrammer.java*:

AwesomeProgrammer.java

```
interface AwesomeProgrammer extends TableSoccerPlayer, JavaProgrammer {  
  
    void drinkCoffe();  
  
}
```

Implementacja klasy `Programmer.java`:

Programmer.java

```
class Programmer implements AwesomeProgrammer {  
  
    @Override  
    public void drinkCoffe() {  
        System.out.println("Drink coffe");  
    }  
  
    @Override  
    public void typeJava() {  
        System.out.println("Type java");  
    }  
  
    @Override  
    public void playTableSoccer() {  
        System.out.println("Soccer");  
    }  
}
```

Implementacja klasy `Runner.java`:

Runner.java

```
public class Runner {  
  
    public static void main(String[] args) {  
        Programmer programmer = new Programmer();  
        programmer.typeJava();  
        programmer.playTableSoccer();  
        programmer.drinkCoffe();  
    }  
}
```

Klasa abstrakcyjna

Implementacja klasy `Drink.java`:

Drink.java

```
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

```
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

```
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

```
class SexOnTheBeach extends Drink {  
  
    @Override  
    void showName() {  
        System.out.println("SexOnTheBeach");  
    }  
  
    @Override  
    void addWater() {  
        System.out.println("250ml");  
    }  
  
    @Override  
    void addAlcohol() {  
        System.out.println("100ml");  
    }  
  
    @Override  
    void addJuice() {  
        System.out.println("150ml");  
    }  
  
    @Override  
    void addIce() {  
        System.out.println("Yes");  
    }  
}
```

Implementacja klasy *Runner.java*:

Runner.java

```
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

```
enum WeekDay {  
  
    MONDAY,  
    TUESDAY,  
    WEDNESDAY,  
    THURSDAY,  
    FRIDAY,  
    SATURDAY,  
    SUNDAY  
  
}
```

Implementacja klasy `DayPrinter.java`:

DayPrinter.java

```
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

```
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

```
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

```
public class Runner {  
  
    public static void main(String[] args) {  
        ExceptionExamples exceptionExamples = new ExceptionExamples();  
        exceptionExamples.catchExample();  
    }  
  
}
```

Implementacja klasy **CheckedException**:

CheckedException.java

```
class CheckedException extends Exception {  
}
```

Implementacja klasy **UncheckedException**:

UncheckedException.java

```
class UncheckedException extends RuntimeException {  
}
```

Implementacja klasy **ExceptionExamples**:

```
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

```
public class Annotation extends Parent {  
  
    @Override  
    public boolean equals(Object obj) {  
        return super.equals(obj);  
    }  
  
    @Override  
    void removeOverrideAnnotation() {  
  
    }  
}
```

Implementacja klasy `Parent.java`:

Parent.java

```
class Parent {  
  
    void removeOverrideAnnotation() {  
  
    }  
  
}
```

Implementacja adnotacji `Author.java`:

Author.java

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
@interface Author {  
  
    String name() default "Jan";  
    String surname() default "Nowak";  
  
}
```

Implementacja klasy `AnnotationExample.java`:

AnnotationExample.java

```
class AnnotationExample {  
  
    @Author  
    void someMethod() {  
  
    }  
  
    @Author(name = "michal", surname = "nie nowak ;)")  
    void someMethodOne() {  
  
    }  
  
    void someSecondMethod() {  
  
    }  
  
}
```

Implementacja klasy *Runner.java*:

Runner.java

```
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

```
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

```
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 **Set**:

SetTest

```
class SetTest {  
  
    @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 **List**:

ListTest

```
class ListTest {  
  
    @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 **Map**:

MapTest

```
class MapTest {
```

```

@Test
void shouldPutNewElement() {
    // Given
    Map<String, String> map = new HashMap<>();
    // When
    map.put("key", "value");
    // Then
    assertThat(map.get("key")).isEqualTo("value");
}

@Test
void shouldPutAllNewElements() {
    // Given
    Map<String, String> map = new HashMap<>();
    Map<String, String> mMap = new HashMap<>();
    map.put("key", "value");
    // When
    map.putAll(mMap);
    // Then
    assertThat(map.get("key")).isEqualTo("value");
}

@Test
void shouldReturnTrueWhenMapContainsKey() {
    // Given
    Map<String, String> map = new HashMap<>();
    map.put("key", "value");
    // When
    boolean containsKey = map.containsKey("key");
    // Then
    assertThat(containsKey).isTrue();
}

@Test
void shouldReturnTrueWhenMapContainsValue() {
    // Given
    Map<String, String> map = new HashMap<>();
    map.put("key", "value");
    // When
    boolean containsValue = map.containsValue("value");
    // Then
    assertThat(containsValue).isTrue();
}

@Test
void shouldReturnRemovedElement() {
    // Given
    Map<String, String> map = new HashMap<>();
    map.put("key", "value");
    // When

```

```
String value = map.remove("key");
// Then
assertThat(value).isEqualTo("value");
}

@Test
void shouldReturnGetElement() {
    // Given
    Map<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

```
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  
        Map<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(Map<Integer, String> treeMap) {  
        for (Map.Entry<Integer, String> entry : treeMap.entrySet()) {  
            return entry.getKey();  
        }  
        return 0;  
    }  
}
```

Testy dla klasy **Collections**:

CollectionsTest

```
class CollectionsTest {  
  
    @Test  
    void shouldReturnCorrectFrequency() {  
        // Given
```

```

List<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
    List<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
    List<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
    List<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

```
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

```
class Nudle extends Food {  
  
    protected Nudle(String name, String weight) {  
        super(name, weight);  
    }  
  
    void prepare() {  
        System.out.println(name);  
    }  
}
```

Implementacja klasy `Cabbage`:

Cabbage.java

```
class Cabbage extends Food {  
  
    protected Cabbage(String name, String weight) {  
        super(name, weight);  
    }  
  
    void prepare() {  
        System.out.println(name);  
    }  
}
```

Implementacja klasy **Beef**:

Beef.java

```
class Beef extends Food {  
  
    protected Beef(String name, String weight) {  
        super(name, weight);  
    }  
  
    void prepare() {  
        System.out.println(name);  
    }  
}
```

Implementacja klasy **Chef**:

Chef.java

```
class Chef<T extends Food> {  
  
    void prepareMeal(T foodToPrepare) {  
        foodToPrepare.prepare();  
    }  
}
```

Implementacja klasy **Runner**:

```
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

```
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

```
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

```
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

```
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

```
class IsEvenPredicate implements Predicate<Integer> {  
  
    public boolean test(Integer integer) {  
        return integer % 2 == 0;  
    }  
}
```

Implementacja klasy `RandomSupplier`:

RandomSupplier.java

```
class RandomSupplier implements Supplier<Integer> {  
    @Override  
    public Integer get() {  
        return new Random().nextInt();  
    }  
}
```

Implementacja klasy **NumberConsumer**:

NumberConsumer.java

```
class NumberConsumer implements Consumer<Integer> {  
  
    @Override  
    public void accept(Integer integer) {  
        System.out.println(integer);  
    }  
}
```

Implementacja klasy **PowerFunction**:

PowerFunction.java

```
class PowerFunction implements Function<Integer, Double> {  
    @Override  
    public Double apply(Integer integer) {  
        return Math.pow(integer, integer);  
    }  
}
```

Testy do klasy **IsOdd**:

IsOddTest.java

```
public class IsOddTest {

    @Test
    void shouldReturnFalseWhenValueIsNotOdd() {
        // Given
        IsOdd isOddChecker = new IsOdd();
        // When
        boolean isOdd = isOddChecker.isOdd(2);
        // Then
        assertThat(isOdd).isFalse();
    }

    @Test
    void shouldReturnFalseWhenValueIsOdd() {
        // Given
        IsOdd isOddChecker = new IsOdd();
        // When
        boolean isOdd = isOddChecker.isOdd(3);
        // Then
        assertThat(isOdd).isTrue();
    }

}
```

Implementacja interfejsu **IsOddPredicate**:

IsOddPredicate.java

```
@FunctionalInterface
interface IsOddPredicate {

    boolean isOdd(int numberToCheck);

}
```

Implementacja klasy **IsOdd**:

IsOdd.java

```
class IsOdd implements IsOddPredicate {
    @Override
    public boolean isOdd(int numberToCheck) {
        return numberToCheck % 2 != 0;
    }
}
```

Lambda

Implementacja interfejsu `StringSupplier.java`:

`StringSupplier.java`

```
interface StringSupplier {  
  
    String string();  
  
}
```

Implementacja klasy `Runner.java`:

`Runner.java`

```
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

```
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`

```
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();  
        }  
    }  
}
```

```

        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`

```
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

```
class Runner {  
  
    public static void main(String[] args) {  
        System.out.println(Thread.currentThread().getName());  
        new NewThread().start();  
        new Thread(new NewRunnable()).start();  
  
        ExecutorService threadPool = Executors.newFixedThreadPool(10);  
  
        for (int i = 0; i < 10; i++) {  
            threadPool.submit(new NewRunnable());  
        }  
    }  
}
```

Implementacja klasy `NewThread`:

NewThread.java

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

Implementacja klasy `NewRunnable`:

NewRunnable.java

```
class NewRunnable implements Runnable {  
    @Override  
    public void run() {  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```

Kolofon

The CodeCouple.pl Press, Krzysztof Chruściel

© 2018 przez The CodeCouple.pl Press

Opublikowana dla wszystkich chcących się rozwijać!