



Platforma Java

Platformy Technologiczne

Michał Wójcik, Waldemar Korłub, Michał Piotrowski

**Java** - rodzaj kawy pochodzący z wyspy o tej samej nazwie znajdującej się w Indonezji.

**Java Platform** - platforma softwarowa działająca nad platformami hardwarowymi:

- **JVM** - Java Virtual Machine,
- **API** - Application Programming Interface.

**Java Language** - obiektowy język programowania.

**Java Virtual Machine** - (JVM) maszyna wirtualna, środowisko zdolne do wykonywania kodu bajtowego uzyskanego w wyniku kompilacji kodu źródłowego w języku Java.

**Java Development Kit** - (JDK) narzędzia programistyczne dla platformy Java.

**Java Runtime Environment** - (JRE) środowisko uruchomieniowe dla programów przeznaczonych na platformę Java.

**Przenośność** może być różnie rozumiana.

Przenośność aplikacji C++ - raz napisana aplikacja może zostać skompilowana pod konkretną platformę ( sprzęt, system operacyjny) gdzie następnie może być uruchamiana.

Przenośność aplikacji *JavaScript* - raz napisana aplikacja może być wykonywana (interpretowana) na dowolnej platformie o ile jest tam dostępny interpreter (zazwyczaj w ramach przeglądarki internetowej).

Przenośność aplikacji *Java* - raz napisane i skompilowane aplikacja może zostać uruchomiona na dowolnej platformie o ile jest tam dostępna maszyna wirtualna w odpowiedniej wersji.

### Java VM:

- **\*.class** - pliki zawierające kod bajtowy maszyny wirtualnej,
- JIT - Just-In-Time - kompilacja do kodu natywnego w czasie wykonania,
- automatyczne zarządzanie pamięcią - GarbageCollector:
  - przydział pamięci na stercie równie wydajny jak przydział pamięci na stosie,
  - zliczanie referencji,
  - obecność GC nie oznacza, że nie da się spowodować wycieku pamięci.

## Technologie Java:

- **Java Card:**

- aktualna wersja 3.1,
- przeznaczona na karty elektroniczne (smart card),
- np.: karty hotelowe;

- **Java Micro Edition (Java ME):**

- aktualna wersja 8.3,
- przeznaczona na urządzenia mobilne i wbudowane,
- np.: odtwarzacze Blu-ray, czytniki Kindle;

- **Java Standard Edition (Java SE):**

- aktualna wersja 13 (często używane wersje 8 i 11),
- tworzenie aplikacji desktopowych;

- **JavaFX:**

- aktualna wersja 13,
- do wersji 10 część Java SE, od wersji 11 samodzielny moduł,
- tworzenie aplikacji okienkowych;

## Technologie Java:

- **Java Enterprise Edition (Java EE):**

- aktualna wersja 8 (często używana wersja 7),
- tworzenie aplikacji serwerowych (webowych i klasy enterprise);

- **Jakarta Enterprise Edition (Jakarta EE):**

- następca Java EE,
- aktualna wersja 8 (w trakcie rozwoju);

- **Android:**

- przeznaczona na urządzenia mobilne i wbudowane,
- Dalvik, ART.

**Zintegrowane środowiska programistyczne (IDE):**

- IntelliJ IDEA,
- NetBeans,
- Eclipse,
- Android Studio.

**Maszyny wirtualne:**

- HotSpot (Oracle),
- Eclipse OpenJ9 (IBM),
- GraalVM (Oracle),
- Jikes RVM (Research Virtual Machine),
- Dalvik (zastąpiona przez Android Runtime (ART)),
- ...

**Narzędzia dla programistów (SDK):**

- Oracle JDK,
- Oracle Open JDK,
- AdoptOpenJDK,
- Amazon Correto,
- Sapmachine,
- Red Hat OpenJDK,
- Zulu,
- IBM OpenJDK.

## Języki:

- Java,
- Scala,
- Groovy,
- Kotlin,
- JavaScript (Rhino, Nashorn),
- Clojure (Lisp dialect),
- Ruby (JRuby),
- Python (Jython).

## Biblioteki:

- Apache Commons,
- JUnit,
- Mockito,
- ...

## Frameworki:

- Spring Framework,
- JavaServer Faces,
- Play Framework,
- ...

- Cay S. Horstmann, *Java 8. Przewodnik doświadczonego programisty*, Helion, 2015.
- Herbert Schildt, *Java. Kompendium programisty. Wydanie X*, Hellion 2018,
- Herbert Schildt, *Java. Przewodnik dla początkujących. Wydanie VII*, Helion 2018.
- Kathy Sierra and Bert Bates, *Java. Rusz głową! Wydanie II*, Helion, 2011.
- Oracle Corporation, *The Java Tutorials*, <https://docs.oracle.com/javase/tutorial/>.
- Oracle Corporation, *JDK 8 Documentation*, <https://docs.oracle.com/javase/8/>.
- Oracle Corporation, *JDK 11 Documentation*, <https://docs.oracle.com/javase/11/>.
- w3schools, *Java Tutorial*, <https://www.w3schools.com/java/>.



POLITECHNIKA  
GDAŃSKA

Konwencje i uruchomienie

Platformy Technologiczne

Michał Wójcik, Waldemar Korłub, Michał Piotrowski

Main.java:

```
package dev.bluelwolf.example.hello;

public class Main {

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

**Podstawowa aplikacja** składa się z następujących elementów:

- przynajmniej jedna publiczna klasa zdefiniowana w pliku o takiej samej nazwie co klasa,
- statyczna metoda `main` realizująca działanie aplikacji,
- parameter metody `main` w postaci tablicy łańcuchów znaków.

Main.java:

```
package dev.bluewolf.example.hello;

public class Main {

    public final static String WELCOME_TEXT = "Hello World";

    public static void main(String[] args) {
        printWelcomeText();
    }

    private static void printWelcomeText() {
        System.out.println(WELCOME_TEXT);
    }
}
```

Wybrane **konwencje** dla języka Java:

- źródła (klasy) umieszczane w oddzielnych plikach, których nazwy odpowiadają nazwom klas,
- klasy grupowane w pakietach (katalogi),
- jedna deklaracja lub wyrażenie w linii,
- nazwy klas i interfejsów zaczynamy wielkimi literami, każde kolejne słowo zaczyna się od wielkiej litery (**UpperCamelCase**),
- nazwy metod i zmiennych zaczynamy małą literą, kolejne słowa zaczynami wielkimi literami (**lowerCamelCase**),
- nazwy stałych piszemy wyłącznie wielkimi literami, kolejne słowa oddzielamy podkreśleniem (**UPPER\_SNAKE\_CASE**),
- nazwy pakietów powinny składać się wyłącznie z małych liter (odwrócona nazwa domeny).

Main.java:

```
package dev.bluewolf.example.hello;

/**
 * Application main class.
 */
public class Main {

    /**
     * Welcome text.
     */
    public final static String WELCOME_TEXT = "Hello World";

    /**
     * Application main methods.
     *
     * @param args command line arguments
     */
    public static void main(String[] args) {
        printWelcomeText();
    }

    /**
     * Prints welcome text.
     */
    private static void printWelcomeText() {
        System.out.println(WELCOME_TEXT);
    }
}
```

## Dokumentowanie kodu w języku Java:

- javadoc komentowania kodu i generowania dokumentacji,
- generowanie dokumentacji na podstawie odpowiednio sformatowanych komentarzy,
- komentarze na poziomie typów (klas, interfejsów, typów enum), pól i metod,
- opisywanie metod z wykorzystaniem:
  - `@param` - parametr metody,
  - `@return` - wartość zwracana,
  - `@throws` - rzucany wyjątek.

Mage.java:

```
package dev.bluewolf.example.mage.model;

public class Mage {

    public static final int MAX_POWER;

    static {
        MAX_POWER = 20;
    }

    private String name;

    private int power = 0;

    public Mage(String name) {
        this.name = name;
    }

}
```

Wybrane **konwencje** dla języka Java:

- inicjalizacja pól klasy przy deklaracji pól lub:
  - w konstruktorze dla pól obiektu,
  - w sekcji **static** dla pól statycznych.

Mage.java:

```
package dev.bluewolf.example.mage.model;

import java.io.Serializable;

public class Mage implements Serializable {

    private String name;

    public Mage() {
    }

    public Mage(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getInitials() {
        return name.charAt(0);
    }
}
```

**JavaBeans** - komponenty, których można wielokrotnie używać, klasy Java o specjalnej charakterystyce:

- domyślny publiczny konstruktor bez parametrów,
- serializowalne,
- brak publicznych pól,
- dostęp do pól poprzez akcesory (**get** i **set**).

**Właściwości** komponentu JavaBean - charakterystyki obiektu do których ma dostęp programista:

- pozwalają na odczytywanie/zapisywaniu stanu komponentu (wpływają na jego zachowanie),
- klasyfikacja ze względu na poziom dostępu:
  - writable – możliwość odczytu i zapisu (**get** i **set**),
  - readonly – możliwość jedynie odczytu (**get**),
  - hidden – możliwość jedynie zapisu (**set**).

**Narzędzia** dostępne w ramach zestawu narzędzi deweloperskich:

- **java** - uruchomienie maszyny wirtualnej oraz aplikacji,
- **javac** - kompilacja kodu źródłowego do postaci binarnej,
- **javadoc** - wygenerowanie dokumentacji na podstawie odpowiednio sformatowanych komentarzy w kodzie,
- **jar** - wygenerowanie archiwum jar (zip).

Polecenie `javac` pozwala na **skomplikowanie** kodu źródłowego (plik tekstowy `.java`) do formy binarnej (`.class`) który może zostać uruchomiony na maszynie wirtualnej Java.

```
javac Main.java
```

Polecenie **java** pozwala na **uruchomienie** maszyny wirtualnej oraz wskazanej aplikacji.

```
java Main
```

Polecenie **javadoc** pozwala na automatyczne wygenerowanie **dokumentacji** na podstawie odpowiednio sformatowanych komentarzy. Domyślnie dokumentacja jest generowana jako strony HTML.

```
javadoc *.java
```

Zwykłe aplikacje napisane w języku java **dystrybuowane** są w postaci **archiwum jar** (zip). Polecenie **jar** pozwala na wygenerowanie archiwum.

```
jar cfe hello-world.jar Main .
```

```
java -jar hello-world.jar
```



POLITECHNIKA  
GDAŃSKA

Budowanie aplikacji  
Platformy Technologiczne  
Michał Wójcik, Waldemar Korłub

**Automatyzacja budowania oprogramowania** - automatyzacja procesów związanych z budowaniem oprogramowania, m.in.:

- komplikacja,
- testowanie,
- generowanie dokumentacji,
- pakowanie,
- ...

**Narzędzia** automatyzujące budowanie:

- Make,
- Rake,
- Cake,
- MSBuild,
- Ant,
- Maven,
- Gradle.

**Maven** – narzędzie automatyzujące budowę oprogramowania:

- narzędzie cieszące się dużą popularnością w przypadku projektów opartych na platformie Java,
- zorientowane na wtyczki realizujące różne funkcjonalności,
- może odpowiadać za budowanie, testowanie,
- dokumentowanie i wdrażanie projektu,
- konfiguracja w postaci pliku XML,
- nie wymaga żadnego IDE do działania,
- zintegrowany we wszystkich popularnych IDE,
- samodzielne narzędzie możliwe do użycia:
  - z poziomu powłoki,
  - w skryptach lub skryptów,
  - z poziomu automatycznych narzędzi typu (np.: serwer continuous integration).

Narzędzie Maven zakłada konkretną strukturę projektu:

- **pom.xml** - plik konfiguracyjny,
- **src** - źródła projektu,
- **target** - wynik budowania projektu,
- **src/main** - główne źródła projektu,
- **src/test** - źródła testowe,
- **java** - kompilowane pliki **.java**,
- **resources** - nie kompilowane zasoby, np.: style, pliki graficzne i dźwiękowe, itd.

```
-project-name
|--pom.xml
|--src
|---main
|   |--java
|   |--resources
|---test
|   |--java
|   |--resources
|-target
```

```
<?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>dev.bluewolf.examples</groupId>
    <artifactId>example</artifactId>
    <version>1.0.0-SNAPSHOT</version>

</project>
```

Każdy projekt (lub biblioteka) jest opisany przez trzy właściwości:

- groupId - identyfikator grupy projektu, zwykle zawiera identyfikator firmy lub grupy stojącej za projektem,
- artifactId - identyfikator projektu,
- version - wersja.

```
<project ...>
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>
</project>
```

```
<project ...>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
</project>
```

Budowanie oprogramowania z wykorzystaniem narzędzia maven oparte jest na **cyklach** (lifecycles).

Wbudowane cykle życia:

- **default** - umożliwiający budowanie projektu, jego wdrażanie i publikację artefaktów,
- **clean** - umożliwia wyczyszczenie projektu (usunięcie wszystkich plików powstających w cyklu default),
- **site** - umożliwiający wygenerowanie statycznych stron HTML z dokumentacją projektu.

Aby wywołać konkretny cykl:

```
mvn clean
```

Każdy cykl życia składa się z faz(ang. phases).

Główne fazy cyklu **default**:

1. **validate** - sprawdzanie poprawności projektu,
2. **compile** - kompilacja kodu źródłowego,
3. **test** - wykonanie testów jednostkowych,
4. **package** - pakowanie skompilowanych klas i zasobów do archiwum (np.: jar),
5. **verify** - testy integracyjne,
6. **install** - instalacja w lokalnym repozytorium:
  - inne lokalne projekty mogą korzystać z zależności;
7. **deploy** - wgranie do zdalnego repozytorium:
  - inni deweloperzy mogą korzystać z zależności.

Aby zbudować aplikację do określonej fazy:

```
mvn package
```

Każda faza składa się z listy celów (goals):

- wtyczki mogą dodawać własne cele do faz wzmacniając proces budowania aplikacji.

```
$ mvn archetype:generate \
  -DgroupId=dev.bluelwolf.examples \
  -DartifactId=maven-example \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

```
$ mvn package
$ java -classpath target/maven-example-1.0-SNAPSHOT.jar dev.bluelwolf.examples.App
```

**Zależności** projektu:

- biblioteki używane w naszym projekcie często zbudowane są w oparciu o inne biblioteki:
  - te inne biblioteki opierają się na jeszcze innych bibliotekach,
  - zależności mają swoje własne zależności - powstaje drzewo zależności;
- konkretna wersja biblioteki zależy od konkretnych wersji swoich zależności:
  - aktualizacja biblioteki pociąga konieczność aktualizacji całej gałęzi drzewa zależności.

**Zależności** - zewnętrzne biblioteki wymagane przez aplikację:

- deklarowane w `pom.xml`,
- ściągane automatycznie z maven central repo,
- możliwość ustawienia prywatnych repozytoriów.

**Scope** - zasięg zależności:

- `compile` - wymagane w czasie kompilacji oraz w czasie działania aplikacji,
- `provided` - jak wyżej, ale dostarczane przez środowisko wykonawcze, np. serwer aplikacji,
- `runtime` - wymagane w czasie działania aplikacji (ale nie w czasie kompilacji),
- `test` - wymagane w czasie uruchamiania testów jednostkowych,
- `import` - import zależności z innego pliku `pom.xml`,
- `system` - niezalecane.

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.9</version>
    </dependency>
  </dependencies>
</project>
```

```
<project ...>
  <repositories>
    <repository>
      <id>repository.jboss.org</id>
      <name>JBoss - Releases Repository</name>
      <url>https://repository.jboss.org/nexus/content/repositories/releases</url>
    </repository>
  </repositories>
</project>
```

Proces budowania realizowany jest za pomocą **pluginów** realizujących odpowiednie funkcjonalności. Domyślnie maven dodaje m.in. następujące pluginy:

- maven-install-plugin - wgranie pluginu do lokalnego repozytorium,
- maven-compiler-plugin - proces kompilacji,
- maven-surefire-plugin - proces testowania,
- maven-jar-plugin - tworzenie archiwum jar,
- maven-deploy-plugin - wgranie aplikacji do zdalnego repozytorium.

Inne przydatne pluginy:

- maven-javadoc-plugin - wygenerowanie paczki z dokumentacją aplikacji,
- maven-source-plugin - wygenerowanie paczki ze źródłami aplikacji,
- maven-dependency-plugin - skopiowanie wszystkich zależności aplikacji do katalogu target.

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.1.2</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <mainClass>dev.bluewolf.examples.App</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.1.1</version>
        <executions>
          <execution>
            <id>attach-javadocs</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>3.1.0</version>
        <executions>
          <execution>
            <id>attach-sources</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

**Profile** pozwalają utworzenie niezależnych zestawów pluginów. Przykładowo:

default - jedynie komplikacja i testowanie, release - komplikacja, przygotowanie archiwum jar, wygenerowanie dokumentacji oraz archiwum ze wszystkimi elementami aplikacji.

```
<project ...>
  <profiles>
    <profil>
      <id>release</id>
      <build>
        </build>
    </profile>
  </profiles>
</project >
```

Aby wykorzystać profil:

```
mvn -Prelease package
```

- Apache, *Maven in 5 minutes*, <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>.
- Apache, *Maven documentation*, <https://maven.apache.org/guides/getting-started/index.html>.



POLITECHNIKA  
GDAŃSKA

Kolekcje

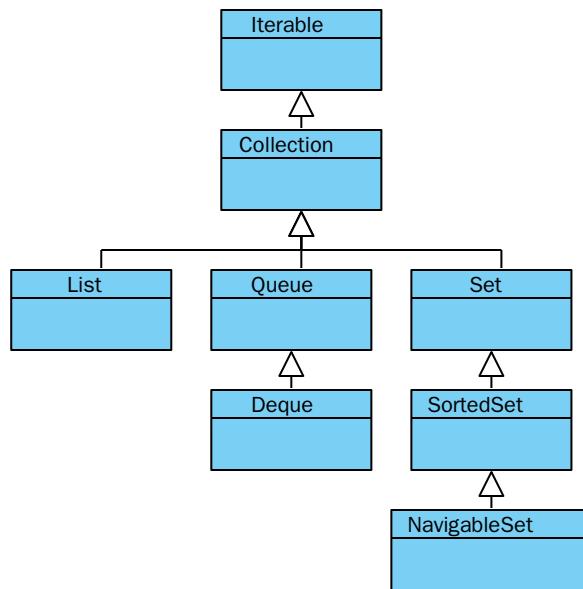
Platformy Technologiczne

Michał Wójcik, Waldemar Korłub, Michał Piotrowski

**Kolekcja** - kontener, obiekt grupujący elementy.

**Collection framework** - narzędzia służące reprezentacji i manipulacji kolekcjami:

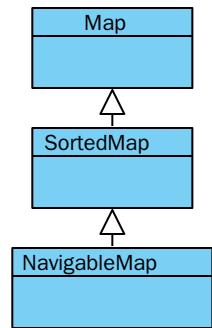
- interfejsy - abstrakcyjne typy reprezentujące kolekcje,
- implementacje - konkretne implementacje interfejsów,
- algorithms - statyczne metody udostępniające użyteczne funkcjonalności (np.: sortowanie).



Relacje pomiędzy interfejsami definiującymi rodzaje kolekcji.

**Rodzaje kolekcji** (interfejsy):

- **Iterable** - deklaruje możliwość iterowania po elementach, pobranie iteratora lub wykorzystanie w pętlach **for-each**,
- **Collection** - najbardziej ogólny rodzaj kolekcji, brak ograniczeń typu powtarzalność elementów lub kolejność,
- **Set** - nie zawiera dwóch lub więcej takich samych elementów, brak zapewnienia kolejności występowania elementów,
- **SortedSet** - automatyczne sortowanie wstawianych elementów,
- **NavigableSet** - możliwość pobierania elementów "większych"/"mniejszych" niż wskazane,
- **List** - sekwencja elementów, może zawierać takie same elementy,
- **Queue** - kolejka FIFO,
- **Deque** - kolejka FIFO, LIFO, lista dwukierunkowa.



Relacje pomiędzy interfejsami definiującymi rodzaje map (kolekcji).

## Rodzaje kolekcji (interfejsy):

- **Map** - ograniczenia na klucze takie jak na elementy w **Set**, nie zawiera dwóch lub więcej takich samych kluczy, brak zapewnienia kolejności występowania kluczy,
- **SortedMap** - ograniczenia na klucze takie jak na elementy w **SortedSet**, automatyczne sortowanie wstawianych kluczy,
- **NavigableMap** - ograniczenia na klucze takie jak na elementy w **NavigableSet**, możliwość pobierania kluczy "większych"/"mniejszych" niż wskazane,

**Operacje** na pojedynczych elementach w **Collection**:

- `boolean add(Object o)` - oddanie pojedynczego elementu,
- `boolean remove(Object o)` - usunięcie pojedynczego elementu,
- `boolean contains(Object o)` - sprawdzenie czy element znajduje się w kolekcji.

```
Collection list = new ArrayList();
list.add("woof");
System.out.println(list.contains("woof")); // true
list.remove("woof");
System.out.println(list.contains("woof")); // false
```

Operacje na pojedynczych elementach w `Map`:

- `boolean put(Object key, Object value)` - dodanie pojedynczego elementu,
- `Object get(Object key)` - pobranie wartości,
- `Object remove(Object key)` - usunięcie pojedynczego elementu,
- `boolean containsKey(Object o)` - sprawdzenie czy klucz znajduje się w kolekcji,
- `boolean containsValue(Object o)` - sprawdzenie czy wartość znajduje się w kolekcji.

```
Map map = new HashMap();
map.put("wolf", 2);
System.out.println(map.containsKey("wolf")); // true
System.out.println(map.containsValue(2)); // true
```

```
Collection list = new ArrayList();
list.add("woof");

for (Object o : list) {
    System.out.println(o);
}
```

Możliwość przeglądania kolekcji za pomocą pętli **for-each**. Podczas przeglądania kolekcji nie można jej modyfikować (dodawanie/usuwanie elementów, sortowanie, itd).

```
Collection list = new ArrayList();
list.add("woof");

for (Iterator it = list.iterator(); it.hasNext();) {
    Object o = it.next();
    it.remove();
}
```

Możliwość pobrania iteratora z kolekcji:

- `boolean hasNext()` - sprawdza czy jest jeszcze coś do przejrzenia,
- `Object next()` - przesuwa iterator i zwraca obiekt,
- `void remove()` - usuwa z kolekcji obiekt, na który wskazuje iterator.

```
Map map = new HashMap();
map.put("wolf", 2);

for (Object k : map.keySet()) {
    System.out.println(k + " " + map.get(k));
}
```

Brak możliwości przeglądania mapy bezpośrednio za pomocą pętli **for-each**.  
Możliwość skorzystania z metod:

- **Set keySet()** - zwraca zbiór kluczy,
- **Collection values()** - zwraca kolekcję wartości.

**Operacje zbiorcze** (bulk operations) - operacje na całej kolekcji:

- `boolean containsAll(Collection collection)`,
- `boolean addAll(Collection collection)`,
- `boolean removeAll(Collection collection)`,
- `boolean retainAll(Collection collection)` - usuwa elementy, których nie ma w kolekcji podanej jako parametr,
- `void clear()`,
- `int size()`,
- `boolean isEmpty()`.

**Konwersja na tablice:**

- `Object[] toArray()` - zwraca kolekcję jako tablicę typu `Object`,
- `T[] toArray(new T[0])` - zwraca kolekcję jako tablicę typu `T`.

**Operacje zbiorcze** (bulk operations) - operacje na całej mapie:

- `boolean putAll(Map map)`, jako paramter,
- `void clear()`,
- `int size()`,
- `boolean isEmpty()`.

```
Collection objects = new ArrayList();
objects.add("woof");

for (Object o : objects) {
    String s = (String) o;
}

objects.add(new Integer(2)); // ok
```

```
Collection<String> strings = new ArrayList<>();
strings.add("woof");

for (String s : strings) {
}

strings.add(new Integer(2)); // compilation error
```

```
Map objects = new HashMap();
objects.put("woof", 2);

for (Object k : objects.keySet()) {
    String s = (String) k;
    Integer v = (Integer) objects.get(k);
}

objects.put(new Integer(2), "bark");
```

```
Map<String, Integer> strings = new HashMap<>();
strings.put("woof", 2);

for (String k : strings.keySet()) {
    Integer v = strings.get(k);
}

strings.put(new Integer(2), "bark"); // compilation error
```

## Implementacje kolekcji (klasy):

- **Set:**
  - **HashSet** - implementacja mapy z haszowaniem, brak sortowania i stałej kolejności elementów,
  - **LinkedHashSet** - implementacja mapy z haszowaniem i listy dwukierunkowej, zapewnia kolejność elementów;
- **NavigableSet:**
  - **TreeSet** - implementacja struktury drzewiastej, zapewnia sortownie.

## Implementacje kolekcji (klasy):

- **List:**
  - `ArrayList` - implementacja dynamicznie rozszerzalnej tablicy,
  - `Vector` - implementacja dynamicznie rozszerzalnej tablicy (thread-save),
  - `LinkedList` - implementacja listy dwukierunkowej;
- **Queue:**
  - `PriorityQueue` - implementacja kopca, zapewnia sortowanie;
- **Deque:**
  - `LinkedList` - implementacja listy dwukierunkowej.

## Implementacje kolekcji (klasy):

- **Map**:
  - **HashMap** - implementacja mapy z haszowaniem, brak sortowania i stałej kolejności kluczy,
  - **LinkedHashMap** - implementacja mapy z haszowaniem i listy dwukierunkowej, zapewnia kolejność kluczy;
- **NavigableMap**:
  - **TreeMap** - implementacja struktury drzewiastej, zapewnia sortowanie kluczy.

Wykorzystanie metody `Object#equals`:

- `ArrayList`, `Vector`, `LinkedList`, `HashSet`, `LinkedHashSet`

Wykorzystanie metody `Object#hashCode`:

- `HashSet`, `LinkedHashSet`

Wykorzystanie metody `Comparable#compareTo`:

- `TreeSet`, `PriorityQueue`

Wykorzystanie metody `Comparator#compare`:

- `TreeSet`, `PriorityQueue`

Wykorzystanie metody `Object#equals`:

- `HashMap`, `LinkedHashMap`.

Wykorzystanie metody `Object#hashCode`:

- `HashMap`, `LinkedHashMap`.

Wykorzystanie metody `Comparable#compareTo`:

- `TreeMap`.

Wykorzystanie metody `Comparator#compare`:

- `TreeMap`.

Wykorzystanie metody `Object#equals`:

- `HashMap`, `LinkedHashMap`, `TreeMap`.

```
String s1 = new String("woof");
String s2 = new String("woof");

System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2)); //true
```

Metoda **equals** służy do porównywania obiektów po ich wartości i powinna być zaimplementowana we wszystkich klasach których obiekty mogłyby być porównywane.

```
Object o1 = new Object();
Object o2 = new Object();

System.out.println(o1.equals(o2));
```

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

Domyślna implementacja **equals** porównuje obiekty za pomocą operatora **==**.

```
public class Wolf {  
  
    private String name;  
  
    private int age;  
  
    @Override  
    public boolean equals(Object other) {  
        //...  
    }  
}
```

```
public boolean equals(Object other) {  
    if (this == other) {  
        return true; //very same object  
    }  
  
    if (other == null || getClass() != other.getClass()) {  
        return false; //other is null or final classes are different  
    }  
  
    Wolf wolf = (Wolf) other;  
  
    if (age != wolf.age) {  
        return false; //check each field  
    }  
  
    if (name != null ? !name.equals(wolf.name) : wolf.name != null) {  
        return false; //check each field  
    }  
    return true;  
}
```

```
public boolean equals(Object other) {  
    if (this == other) {  
        return true;  
    }  
  
    if (other == null || getClass() != other.getClass()) {  
        return false;  
    }  
  
    Wolf wolf = (Wolf) other;  
  
    return age == wolf.age && Objects.equals(name, wolf.name);  
}
```

Możliwość użycia statycznej metody `Objects.equals` poprawnie sprawdzającej wartości `null`.

Metoda **hashCode**:

- zwraca hash (skrót) obiektu,
- wykorzystywana na potrzeby struktur tablic z haszowaniem,
- zwraca liczbę całkowitą z określonego zakresu.

```
public class Object {  
    public native int hashCode();  
}
```

Domyślna implementacja generuje wartość na podstawie referencji i jest zaimplementowana natywnie w maszynie wirtualnej.

```
String s1 = new String("wolf");
String s2 = new String("wolf");

System.out.println(s1.equals(s2)); //true
System.out.println(s1.hashCode() == s2.hashCode()); //true
```

Implementacja metod `equals` i `hashCode` musi być spójna:

- jeśli metoda `equals` zwraca `true` to wartości `hashCode` muszą być takie same,
- jeśli metoda `equals` zwraca `false` to wartości `hashCode` powinny (nie muszą) być różne.

Funkcje skrótu ponieważ z zasady opisują większą ilość danych w postaci krótszego skrótu są kolizyjne.

```
public int hashCode() {  
    int result = name != null ? name.hashCode() : 0;  
    result = 31 * result + age;  
    return result;  
}
```

Przy generowaniu skrótu:

- zaleca się korzystać z metod **hashCode** już zaimplementowanych w innych obiektach,
- wartość typów prostych zaleca się przemnożyć przez dobraną liczbę całkowitą (najlepiej pierwszą).

```
public int hashCode() {  
    return Objects.hash(name, age);  
}
```

Możliwość użycia statycznej metody `Objects.hashCode` poprawnie sprawdzającej wartości `null`.

Interfejs **Comparable** służy zdefiniowaniu naturalnego porządku:

- deklaruje jedną metodę `int compareTo(Object other)`:
  - zwraca wartość `0` gdy obiekty są takie same,
  - zwraca wartość dodatnią gdy obiekt, z którego wywołana jest metoda jest "większy",
  - zwraca wartość ujemną gdy obiekt, z którego wywołana jest metoda jest "mniejszy";
- obiekt sam "wie" jak ma być porównywany z innymi,
- wykorzystywany przy strukturach drzewiastych i sortowaniu.

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

Interfejs **Comparable** jest zdefiniowany jako typ generyczny.

```
String wolf = "wolf";
String dog = "dog";

System.out.println(wolf.compareTo(dog)); // 19
System.out.println(dog.compareTo(wolf)); // -19
```

Wiele klas standardowo dostępnym na platformie Java ma już zaimplementowany interfejs **Comparable**.

```
public class Wolf implements Comparable<Wolf> {

    private String name;

    private int age;

    public Wolf(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Wolf other) {
        int ret = name.compareTo(other.name);

        if (ret == 0) {
            ret = age - other.age;
        }

        return ret;
    }
}
```

```
Wolf w1 = new Wolf("Kiba", 6);
Wolf w2 = new Wolf("Toboe", 2);
System.out.println(w1.compareTo(w2)); // -9
```

Interfejs **Comparator** służy zdefiniowaniu dodatkowego (innego niż naturalny) porządku:

- deklaruje jedną metodę `int compare(Object o1, Object o2)`:
  - zwraca wartość `0` gdy obiekty są takie same,
  - zwraca wartość dodatnią gdy pierwszy jest "większy",
  - zwraca wartość ujemną gdy pierwszy obiekt jest "mniejszy";
- wykorzystywany przy strukturach drzewiastych i sortowaniu.

```
public interface Comparator<T> {  
    int compare(T t1, T t2);  
}
```

Interfejs **Comparator** jest zdefiniowany jako typ generyczny.

```
public class WolfComparator implements Comparator<Wolf> {  
  
    @Override  
    public int compare(Wolf w1, Wolf w2) {  
        int ret = w1.getAge() - w2.getAge();  
        return ret;  
    }  
}
```

```
Wolf w1 = new Wolf("Kiba", 6);
Wolf w2 = new Wolf("Toboe", 2);

WolfComparator comparator = new WolfComparator();

System.out.println(comparator.compare(w1, w2)); // 4
System.out.println(comparator.compare(w2, w1)); // -4
```

Metoda **toString** zwraca reprezentację tekstową obiektu, na którym została wywołana.

```
public class Object {  
  
    public String toString() {  
        return getClass().getName() + "@"  
            + Integer.toHexString(hashCode());  
    }  
  
}
```

Domyślna implementacja bazuje na nazwie klasy oraz wartości zwróonej przez funkcję skrótu. Podobnie jak metody **equals** i **hashCode** bazuje na adresie referencji.

```
public class Wolf {  
  
    private String name;  
  
    private int age;  
  
    public Wolf(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return "Wolf{" + "name='" + name + "'" + ", age=" + age + "}";  
    }  
}
```

```
Wolf wolf = new Wolf("Hige", 4);

System.out.println(wolf.toString()); //Wolf{name='Hige', age=4}
System.out.println(wolf); //Wolf{name='Hige', age=4}
```

Metoda **toString** jest wywoływana automatycznie m.in. przez metodę  
**System.out.println**.



Wątki  
Platformy Technologiczne  
Michał Wójcik

**Proces** - pojedyncza instancja wykonywanego programu. Każdy proces posiada własne:

- unikatowy numer identyfikujący (PID - process identifier),
- środowisko wykonania,
- skończony zbiór zasobów,
- przestrzeń w pamięci.

**Zasoby** przydzielone procesowi:

- czas procesora,
- pamięć,
- dostęp do urządzeń I/O,
- pliki.

Zarządzaniem procesami zajmuje się **jądro** systemu operacyjnego.

Sposoby **wykonywania** procesów:

- sekwencyjnie - poszczególne akcje procesu są wykonywane jedna po drugiej, kolejna akcja rozpoczyna się po zakończeniu poprzedniej,
- współbieżnie - kolejna akcja może rozpocząć się przed zakończeniem poprzedniej, akcje mogą być wykonywane w tym samym czasie lub z przeplotem (procesor przełącza się pomiędzy różnymi akcjami),
- równolegle - akcje są wykonywane w tym samym czasie na kilku procesorach (lub rdzeniach).

**Wątki** - części programu wykonywane współbieżnie (niekoniecznie równolegle):

- istnieje wewnątrz procesu,
- i współdzielą między sobą zasoby procesu z wyjątkiem czasu procesora.

Przykłady **wykorzystania wątków**:

- wykonywanie tych samych obliczeń na różnych i niezależnych elementach dużego zbioru,
- wykonywanie skomplikowanych akcji w tle bez blokowania interfejsu użytkownika.

**Sekcja krytyczna** - fragment kodu, w którym korzysta się z zasobu nie obsługującego dostępu przez wiele wątków na raz.

Przykłady **sekcji krytycznej**:

- zmiana wartości globalnej zmiennej przez kilka wątków,
- zapisywanie/czytanie do/z jednego zasobu (pliku) przez wiele wątków,
- manipulacja jednym zbiorem (tablica, kolekcja) przez wiele wątków.

Dwa sposoby **tworzenia wątków**

- implementacja interfejsu **Runnable** (zalecana):
  - pozwala na dziedziczenie po innych klasach,
  - skupia się na zdefiniowaniu pracy do wykonania a nie mechanizmie utworzenia nowego wątku;
- dziedziczenie po klasie **Thread** (niezalecane):
  - obiekty **Thread** służą do utworzenia nowego wątku w maszynie wirtualnej.

```
public class Main {  
  
    public static void main(String[] args) {  
        System.out.println("Welcome to the main thread");  
    }  
  
}
```

Metoda **main** jest przetwarzana przez główny wątek aplikacji tworzony wraz z procesem.

```
public class Main {  
  
    public static void main(String[] args) {  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException ex) {  
  
        }  
        System.out.println("Welcome to the main thread");  
    }  
  
}
```

```
public class GoodHowler implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("woof woof");  
    }  
  
}
```

```
Thread thread = new Thread(new GoodHowler());  
thread.start();  
System.out.println("Right after starting thread!");
```

```
public class BadHowler extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("woof woof");  
    }  
  
}
```

```
Thread thread = new BadHowler();  
thread.start();
```

```
public class Resource {  
  
    public synchronized void m1() {  
    }  
  
    public synchronized void m2() {  
    }  
  
}
```

Synchronizowane metody nie mogą być wywołane przez więcej niż jeden wątek z poziomu tego samego obiektu w tym samym czasie.

Wszystkie synchronizowane metody są traktowane jako jedna sekcja krytyczna.

```
public class Resource {  
  
    public synchronized static void m1() {  
    }  
  
    public synchronized static void m2() {  
    }  
  
}
```

Synchronizowane metody statyczne nie mogą być wywołane przez więcej niż jeden wątek z poziomu tej samej klasy w tym samym czasie.

Wszystkie synchronizowane metody są traktowane jako jedna sekcja krytyczna.

```
public class Resource {  
  
    public void m1() {  
        synchronized (this) {  
  
        }  
    }  
}
```

Synchronizowane fragmenty kodu nie mogą być przetwarzane przez więcej niż jeden wątek. Synchronizowanie bloków wymaga podania obiektu wykorzystywanego jako rygiel (lock). Do synchronizacji wykorzystywana jest instancja obiektu w pamięci.

Synchronizacja całych metod jest tym samym co synchronizacja bloku całego ciała metody z podaniem obiektu **this**.

```
public class LocalSynchronizedResource {  
  
    private final Object lock = new Object();  
  
    public void m1() {  
        synchronized (lock) {//Resource objects are independent.  
    }  
}  
}
```

```
public class GlobalSynchronizedResource {  
  
    private static final Object lock = new Object();  
  
    public void m1() {  
        synchronized (lock) {//Resource objects synchronized globally.  
    }  
}  
}
```

```
public class Sleeper implements Runnable {  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(1000 * 60 * 60);  
        } catch (InterruptedException ex) {  
        }  
    }  
}
```

```
Sleeper sleeper = new Sleeper();  
Thread thread = new Thread(new Sleeper());  
thread.start();  
  
thread.interrupt();
```

```
public class Sleeper implements Runnable {  
  
    @Override  
    public void run() {  
        while (!Thread.isInterrupted()) {  
            try {  
                Thread.sleep(1000 * 60);  
            } catch (InterruptedException ex) {  
                break;  
            }  
        }  
    }  
}
```

Metoda `Thread.sleep(long ms)`:

- pozwala na uśpienie wątku na określony czas,
- brak pewności ile dokładnie wątek będzie uśpiony,
- możliwe wybudzenie za pomocą metody `Thread#interrupt()`,
- wybudzenie powoduje rzucenie wyjątku `InterruptedException`,
- możliwość sprawdzenia próby wybudzenia przez metodę `Thread.isInterrupted()`,
- pozostaje w sekcji krytycznej.

Uwaga: zarówno złapanie wyjątku `InterruptedException` jak i sprawdzenie stanu metodą `isInterrupted()` konsumuje zdarzenie, powtórne wywołanie metody `isInterrupted()` zwróci wartość `false`.

```
public class Worker implements Runnable {  
  
    int result = 0;  
  
    public Worker(int value) {  
        result = value;  
    }  
  
    @Override  
    public void run() {  
        result *= 2; //Epic computations.  
    }  
  
    public int getResult() {  
        return result;  
    }  
}
```

```
Worker worker = new Worker();  
Thread thread = new Thread(worker);  
thread.start();  
//...  
thread.join();  
System.out.println(worker.getResult());
```

Metoda `Thread#join` pozwala na oczekiwania na zakończenie danego wątku.

```
public class Resource {  
  
    private String value = null;  
  
    public synchronized String take() throws InterruptedException {  
        while (value == null) {  
            wait(); //Wait, maybe someone will put sth here.  
        }  
        String ret = value;  
        value = null;  
        return ret;  
    }  
  
    public synchronized void put(String value) {  
        this.value = value;  
        notifyAll();  
    }  
}
```

```
public class Producer implements Runnable {  
  
    private Resource resource;  
  
    public Producer(Resource resource) {  
        this.resource = resource;  
    }  
  
    @Override  
    public void run() {  
        resource.put("woof");  
    }  
}
```

```
public class Consumer implements Runnable {  
  
    private Resource resource;  
  
    public Consumer(Resource resource) {  
        this.resource = resource;  
    }  
  
    @Override  
    public void run() {  
        try {  
            System.out.println(resource.take());  
        } catch (InterruptedException ex);  
    }  
}
```

Metoda `Object#wait`:

- pozwala czekać na jakiś zasób,
- nie blokuje sekcji krytycznej,
- musi być wywołana na tym samym obiekcie na którym kod jest synchronizowany.

Metody `Object#notify` i `Object#notifyAll`:

- mogą zostać użyte do powiadomienia jednego lub kilku wątków o dostępności zasobu,
- wybudzają wątki uśpione metodą `wait`,
- muszą zostać wywołane na tym samym obiekcie na którym kod jest synchronizowany.

Synchronizowane kolekcje (dekoratory na stworzone wcześniej instancje) można uzyskać za pomocą metody wytwórczych z klasy **Collections**:

- `synchronizedCollection`,
- `synchronizedSet`,
- `synchronizedSortedSet`,
- `synchronizedNavigableSet`,
- `synchronizedList`,
- `synchronizedMap`,
- `synchronizedSortedMap`,
- `synchronizedNavigableMap`,

```
List<String> list = new ArrayList<>();
List<String> synchroList = Collections.synchronizedList(list);
```



POLITECHNIKA  
GDAŃSKA

Obsługa Wejścia / Wyjścia  
Platformy Technologiczne  
Michał Wójcik

**Strumienie** - ciągi danych z/do których dane mogą być pobierane/dodawane:

- **Byte streams** - strumienie bajtowe, czytanie/pisanie bajtami,
- **Character streams** - strumienie znakowe, czytanie/pisanie znakami.

Istnieje wiele rodzajów strumieni:

- do/z pliku,
- do/z gniazdką sieciowego,
- do/z pamięci.

Bazowe klasy abstrakcyjne:

- **Byte streams:**
  - `InputStream`,
  - `OutputStream`;
- **Character streams:**
  - `Reader`,
  - `Writer`.

Nie definiują źródła/celu danych a jedynie ich format (znaki lub bajty).

Metody klasy `InputStream`:

- `int read()` - zwraca jeden bajt,
- `int read(byte[] buffer)` - wczytuje bajty do tablicy i zwraca ich liczbę,
- `int read(byte[] buffer, int offset, int length)` - dodatkowe ograniczenia,
- `int available()` - estymacja liczby bajtów pozostałych do odczytania,
- `void close()` - zamyka strumień.

Metody klasy `OutputStream`:

- `void write(int b)` - zapisuje bajt,
- `void write(byte[] buffer)` - zapisuje bajty z tablicy,
- `void write(byte[] buffer, int offset, int length)` - dodatkowe ograniczenia,
- `void close()` - zamyka strumień.

Metody klasy **Reader**:

- `int read()` - zwraca numer znaku,
- `int read(char[] buffer)` - wczytuje znaki do tablicy i zwraca ich liczbę,
- `int read(char[] buffer, int offset, int length)` - dodatkowe ograniczenia,
- `boolean ready()` - czy zostało coś do przeczytania,
- `void close()` - zamyka strumień.

Metody klasy `Writer`:

- `void write(int c)` - zapisuje znak o podanym numerze,
- `void write(String text)` - zapisuje podany ciąg znaków,
- `void write(String text, int offset, int length)` - dodatkowe ograniczenia,
- `void write(char[] buffer)` - zapisuje znaki z tablicy,
- `void write(char[] buffer, int offset, int length)` - dodatkowe ograniczenia,
- `void close()` - zamyka strumień.

Konkretnie klasy dla obsługi plików:

- **Byte streams:**
  - `FileInputStream`,
  - `FileOutputStream`;
- **Character streams:**
  - `FileReader`,
  - `FileWriter`.

```
try {
    Writer writer = new FileWriter("/tmp/output.txt");
    writer.write("Hello World");
    writer.close();
} catch (IOException e) {

}
```

```
try {
    Reader reader = new FileReader("/tmp/output.txt");

    char[] buffer = new char[11];
    reader.read(buffer);
    System.out.println(buffer);

    reader.close();
} catch (IOException ex) {

}
```

```
try {
    OutputStream os = new FileOutputStream("/tmp/output");
    os.write("Hello World".getBytes());
    os.close();
} catch (IOException ex) {

}
```

```
try {
    InputStream is = new FileInputStream("/tmp/output");

    byte[] buffer = new byte[11];
    is.read(buffer);
    System.out.println(new String(buffer));

    is.close();
} catch (IOException ex) {

}
```

Pliki tekstowe to pliki binarne które można łatwo zinterpretować.

```
try {
    OutputStream os = new FileOutputStream("/tmp/output");
    byte[] buffer = {2, 32, 124, 42, 12, 32, 42, 23, 42, 12, 4};
    os.write(buffer);
    os.close();
} catch (IOException ex) {
}
```

Danych, które nie reprezentują tekstu nie można łatwo odczytać. Można skorzystać z narzędzi:

- Okteta <https://www.kde.org/applications/utilities/okteta/>
- HxD <https://mh-nexus.de/en/hxd/>
- HexEd.it <https://hexed.it/>

```
OutputStream os;
try {
    os = new FileOutputStream("/tmp/output");
    os.write(12);
} catch (IOException) {

} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException) {

    }
}
}
```

```
try (OutputStream os = new FileOutputStream("/tmp/output")) {  
    os.write(12);  
} catch (IOException ex) {  
}
```

Składnia **try-with-resource** pozwala na zdefiniowanie w ramach bloku **try** zmiennej której klasa implementuje interfejs **Autocloseable**. Powoduje to wygenerowanie poprawnego mechanizmu zamknięcia stworzonego obiektu.

**Strumienie buforowane** - usprawnienie procesu odczytu/zapisu danych:

- zmniejszenie odwołań do natywnego API,
- zrealizowane w formie dekoratorów,
- nie definiują źródła/celu danych a jedynie dodają mechanizm buforowania,
- dane czytane/zapisywane większymi partiami,
- udostępniają oryginalne metody strumieni (zoptymalizowane),
- udostępniają dodatkowe metody.

## Strumienie buforowane:

- **Byte streams:**
  - `BufferedInputStream`,
  - `BufferedOutputStream`;
- **Character streams:**
  - `BufferedReader`,
  - `BufferedWriter`.

```
try (OutputStream os = new FileOutputStream("/tmp/test");
    BufferedOutputStream bos = new BufferedOutputStream(os)) {
    bos.write(12);
} catch (IOException ex) {
```

Strumień buforowany należy nałożyć na konkretny strumień.

Metody klasy `BufferedOutputStream`:

- `void flush()` - opróżnienie bufora.

Metody klasy `BufferedWriter`:

- `void flush()` - opróżnienie bufora,
- `void newLine()` - wstawienie znaku nowej linii.

Metody klasy `BufferedReader`:

- `String readLine()` - wczytanie całej linii tekstu.

**Serializacja** - konwersja obiektu do strumienia bajtów.

Strumienie pozwalające na serializację:

- `ObjectInputStream`,
- `ObjectOutputStream`,
- zrealizowane w formie dekoratorów,
- nie definiują źródła ani celu danych a jedynie możliwość serializacji/deserializacji.

Konfiguracja procesu serializacji:

- serializacja pozwala na konwersję całych obiektów na strumienie bajtów,
- obiekt musi implementować interfejs **Serializable**,
- pola z modyfikatorem **transient** nie podlegają serializacji,
- klasy powinny deklarować wersję klasy za pomocą pola **static final long serialVersionUID**.

```
try (OutputStream os = new FileOutputStream("/tmp/object");
     ObjectOutputStream oos = new ObjectOutputStream()) {
    oos.writeDouble(3.3);
    oos.writeObject("the blue wolf");
} catch (IOException ex) {

}
```

```
try (InputStream is = new FileInputStream("/tmp/object");
     ObjectInputStream ois = new ObjectInputStream(is)) {
    System.out.println(ois.readDouble());
    System.out.println(ois.readObject());
} catch (IOException | ClassNotFoundException ex) {

}
```

```
public class Box implements Serializable {  
  
    static final long serialVersionUID = 13;  
  
    private String label;  
  
    private double capacity;  
  
    private transient String placement;  
  
    //getter and setters  
  
}
```

```
try (OutputStream os = new FileOutputStream("/tmp/object");
     ObjectOutputStream oos = new ObjectOutputStream(os)) {
    Box box = new Box();
    box.setLabel("my box");
    box.setCapacity(22);
    box.setPlacement("top");
    oos.writeObject(box);
} catch (IOException ex) {
}
```

```
try (InputStream is = new FileInputStream("/tmp/object");
     ObjectInputStream ois = new ObjectInputStream(is)) {
    Box box = (Box) ois.readObject();
    System.out.println(box.getLabel()); // my box
    System.out.println(box.getCapacity()); // 22
    System.out.println(box.getPlacement()); // null
} catch (IOException | ClassNotFoundException ex) {
}
```



POLITECHNIKA  
GDAŃSKA

Gniazdka sieciowe  
Platformy Technologiczne  
Michał Wójcik

Klasa **InetAddress** pozwala na zapisanie adresu IP (Internet Protocol) definiującego urządzenie sieciowe.

Pobranie lokalnego/zdalnego adresu:

```
try {
    InetAddress address = InetAddress.getLocalHost();
    //    InetAddress address = InetAddress.getByName("google.com");
    System.out.println(address.getHostName());
    System.out.println(address.getHostAddress());
} catch (UnknownHostException ex) {
    System.err.println(ex);
}
```

Pobieranie kilku wartości (adresów) dla jednej nazwy:

```
try {
    for (InetAddress address: InetAddress.getAllByName("google.com")) {
        System.out.println(address.getHostName());
        System.out.println(address.getHostAddress());
        System.out.println();
    }
} catch (UnknownHostException ex) {
    System.err.println(ex);
}
```

**URL** - adres definiujący adres zasobu w sieci. Składa się z elementów:

- protokół (np.: http, https, ftp),
- opcjonalna nazwa użytkownika,
- opcjonalne hasło jeśli wymagane do uwierzytelniania,
- adres hosta,
- opcjonalny port (wartości domyślne zdefiniowane dla protokołów),
- ścieżka do pliku (zasobu),
- opcjonalne parametry żądania,
- identyfikator fragmentu.

```
protocol://user:password@host:port/path?query#ref
```

```
try {
    URL url = new URL("http://user:pass@host:8080/path?p1=v1&p2=v2#ref");
    System.out.println(url.getProtocol()); // http
    System.out.println(url.getHost()); // host
    System.out.println(url.getPort()); // 8080
    System.out.println(url.getDefaultPort()); // 80
    System.out.println(url.getFile()); // /path?p1=v1&p2=v2
    System.out.println(url.getPath()); // /path
    System.out.println(url.getQuery()); // p1=v1&p2=v2
    System.out.println(url.getRef()); // ref
    System.out.println(url.getAuthority()); // user:pass@host:8080
    System.out.println(url.getUserInfo()); // /user:pass
}catch (MalformedURLException ex) {
    System.err.println(ex);
}
```

Otwieranie połączenia za pomocą klasy `URL`:

```
try {
    URL url = new URL("...");
    try {
        URLConnection connection = url.openConnection();
        try (InputStream is = connection.getInputStream()) {
            //...
        }
    } catch (IOException ex) {
        System.err.println(ex);
    }
} catch (MalformedURLException ex) {
    System.err.println(ex);
}
```

**Gniazdko sieciowe** - podstawowy mechanizm komunikacji dostępny w aplikacjach.

Reprezentuje dwukierunkowy (nadawanie i odbieranie danych) punkt końcowy połączenia.

Otwarcie gniazdka.

```
try {
    InetAddress address = InetAddress.getByName("wp.pl");
    try (Socket socket = new Socket(address, 8)) {
        System.out.println("working");
    }
} catch (IOException ex) {
    System.err.println(ex);
}
```

Klasa **Socket** - podstawowy element w komunikacji sieciowej dla technologii Java.

Reprezentuje punkt końcowy połączenia.

Nasłuchiwanie na połączenia:

```
try (ServerSocket server = new ServerSocket(9797)) {  
    try (Socket socket = server.accept()) {  
        }  
    } catch (IOException ex) {  
        System.err.println(ex);  
    }  
}
```

Klasa **ServerSocket** - reprezentuje serwer:

- pozwala na akceptowanie przychodzących połączeń,
- zwraca nowy obiekt **Socket** będący końcem zestawionego połączenia.

Nawiązywanie połączenia z serwerem:

```
try (Socket client = new Socket("localhost", 9797)) {  
} catch (IOException ex) {  
    System.err.println(ex);  
}
```

```
try (Socket client = ...) {  
    try (InputStream is = client.getInputStream();  
        OutputStream os = client.getOutputStream()) {  
  
    }  
} catch (IOException ex) {  
    System.err.println(ex);  
}
```

Obiekt **Socket** zwraca podstawowe strumienie bajtowe, na które może być nałożony dowolny dekorator.



POLITECHNIKA  
GDAŃSKA

JavaFX

Platformy Technologiczne

Michał Wójcik

**JavaFX** – platforma rozwijająca platformy klienckie Java w celu umożliwienia łatwego tworzenia i wdrażania aplikacji desktopowych, mobilnych oraz przeznaczonych na urządzenia wbudowane.

### **Wykorzystanie** JavaFX:

- Java 8 - dostarczana z dystrybucją Oracle JRE, JDK,
- Java 11 - zewnętrzna zależność.

```
<dependencies>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>11</version>
    </dependency>
</dependencies>
```

```
<plugins>
    <plugin>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-maven-plugin</artifactId>
        <version>0.0.3</version>
        <configuration>
            <mainClass>dev.bluewolf.javafx.HelloFx</mainClass>
        </configuration>
    </plugin>
</plugins>
```

```
$ mvn clean javafx:run
```

## Uruchomienie aplikacji:

- główna klasa dziedziczy po `Application`,
- metoda `main` wywołuje metodę `launch` zdefiniowaną w klasie `Application`,
- nadpisanie metody `Application#start(Stage)` tworzącej scenę,
- obiekt `Stage` reprezentuje okno aplikacji.

```
package dev.bluelwolf.javafx;

import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFx extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.show();
    }
}
```

**Glass Windowing Toolkit** – jest warstwą zależną od platformy, która łączy platformę JavaFX z natywnym systemem:

- dostarcza natywne operacje zarządzania oknami,
- zarządza kolejką zdarzeń, wykorzystuje systemową natywną kolejkę zdarzeń do kolejkowania wątków,
- działa w tym samym wątku co aplikacja JavaFX.

**Prism** – obsługuje proces renderowania zawartości okien, może działać zarówno softwarowo jak i sprzętowo:

- DirectX dla systemów Windows,
- OpenGL dla Mac, Linux, Embedded,
- Java2D gdy wsparcie sprzętowe nie jest dostępne.

**Quantum Toolkit** – łączy Prim i Glass Windowing Toolkit i udostępnia wyższej warstwie, zarządza regułami obsługi wątków związanymi z renderowaniem i obsługa zdarzeń.

**JavaFX application thread** – główny wątek aplikacji, wszystkie dostęp do wyświetlonej sceny muszą być realizowane w tym wątku.

**Prism render thread** – renderowanie poza obsługą zdarzeń, pozwala na renderowanie ramki N gdy przetwarzana jest ramka N+1.

**Media thread** – działa w tle, synchronizuje ostatnie ramki za pomocą scene graph wykorzystując JavaFX application thread.

## Kontrolki UI:

- realizowane jako węzły sceny,
- wygląd modyfikowany poprzez CSS.

Dostępne w **pakiecie javafx.scene.control**:

- Label,
- Button, Radio Button, Toggle Button,
- Checkbox, Choice Box,
- Text Field, Password Field,
- Scroll Bar, Scroll Pane,
- List View, Table View, Tree View, Tree Table View,
- Combo Box,
- Separator,
- Slider,
- Progress Bar, Progress
- Indicator,
- Hyperlink,
- HTML Editor,
- Tooltip,
- Titled Pane, Accordion,
- Menu,
- Color Picker, Date Picker,
- Pagination Control,
- File Chooser.

**Kontenery** pozwalają na wygodne i dynamiczne układanie kontrolek UI na scenie:

- **BorderPane** – regiony: top, bottom ,right, left, center,
- **HBox** – jeden rząd poziomo,
- **VBox** – jeden rząd pionowo,
- **StackPane** – stos, tworzenie skomplikowanych elementów poprzez składanie,
- **GridPane** – siatka,
- **FlowPane** – następujące po sobie elementy poziomo lub pionowo,
- **TilePane** – siata o elementach w równym rozmiarze,
- **AnchorPane** – zakotwiczenie elementów: top, bottom, left, right, center.

Obiekt **FileChooser** pozwala na wybranie pliku za pomocą natywnego mechanizmu systemu operacyjnego.

```
FileChooser chooser = new FileChooser();
```

Wczytanie pliku:

```
File file chooser.showOpenDialog(Window stage);
try (FileReader fr = new FileReader(file)) {
} catch (IOException ex) {
}
```

Zapisanie pliku:

```
File file chooser.showSaveDialog(Window stage); //allows to create file
try (FileWriter fw = new FileWriter(file)) {
} catch (IOException ex) {
}
```

Otwarcie kilku plików:

```
List<File> files = chooser.showOpenMultipleDialog(stage);
```

**FXML** – pozwala na definiowanie interfejsu użytkownika: oparty na XML,

- nie komplikowany,
- możliwość tworzenia interfejsu dynamicznie,
- możliwość wykorzystania stylów CSS.

### Zależności dla FXML:

- Java 8 - dostarczany wraz z Oracle JRE/JDK,
- Java 11 - zewnętrza zależność.

```
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>11</version>
</dependency>
```

W celu przygotowania widoku należy przygotować następujące elementy:

- plik `.fxml` z definicją widoku,
- implementację interfejsu `Initializable` dostarczającego dane do widoku i realizującego akcje,
- pliki `.properties` z zasobami wielojęzykowymi.

Plik `/dev/bluewolf/example/view/board_view.fxml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.*?>

<?import javafx.scene.control.Button?>
<BorderPane xmlns="http://javafx.com/javafx"
             xmlns:fx="http://javafx.com/fxml"
             fx:controller="dev.bluewolf.example.view.BoardView"
             prefHeight="400.0" prefWidth="600.0">
    <center>
        <Button fx:id="helloButton" onAction="#hello" text="%hello"/>
    </center>
</BorderPane>
```

Plik `/dev/bluewolf/example/view/BoardView.java`:

```
package dev.bluewolf.example.view;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;

import java.net.URL;
import java.util.ResourceBundle;

public class BoardView implements Initializable {

    @FXML
    private Button helloButton;

    private ResourceBundle rb;

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        rb = resourceBundle;
    }

    @FXML
    public void hello(ActionEvent event) {
        System.out.println(rb.getString("hello"));
    }
}
```

Plik `/dev/bluewolf/example/view/message/board_view_en.properties`:

```
hello = Hello!
```

Plik `/dev/bluewolf/example/view/message/board_view_pl.properties`:

```
hello = Cześć!
```

Uruchomienie:

```
@Override
public void start(Stage stage) throws IOException {
    URL url = getClass().getResource("/dev/bluewolf/example/view/board_view.fxml");
    ResourceBundle rb = ResourceBundle
        .getBundle("dev.bluewolf.example.view.message.board_view");
    Parent root = FXMLLoader.load(url, rb);

    Scene scene = new Scene(root, 500, 400, Color.BLACK);
    stage.setScene(scene);
    stage.show();
}
```

Konwencja JavaBeans:

```
public class Person {  
  
    private int age ;  
  
    public Person() {  
        age = 0;  
    }  
  
    public final void setAge(int age) {  
        this.age = age ;  
    }  
  
    public final int getAge() {  
        return age ;  
    }  
}
```

Nowa konwencja JavaFX JavaBeans:

```
public class Person {  
  
    private IntegerProperty age = new SimpleIntegerProperty();  
  
    public Person() {  
        age.set(0);  
    }  
  
    public final void setAge(int age) {  
        this.age.set(age);  
    }  
  
    public final int getAge() {  
        return age.get();  
    }  
  
    public IntegerProperty ageProperty() {  
        return age;  
    }  
}
```

Reagowanie na zmianę własności:

```
Person p = new Person();

p.ageProperty().addListener(new ChangeListener() {

    @Override
    public void changed(ObservableValue o, Object oldVal, Object newVal) {
        }

});
```

JavaFX wprowadza rozszerzenie dla znanych do tej pory standardowych kolekcji (`javafx.collections`):

- interfejsy:
  - `ObservableList` – lista pozwalająca na śledzenie zmian,
  - `ListChangeListener` – otrzymuje notyfikacje o zmianach na liście,
  - `ObservableMap` – mapa pozwalająca na śledzenie zmian,
  - `MapChangeListener` – otrzymuje notyfikacje o zmianach w mapie;
- klasy:
  - `FXCollections` – kopia `java.util.Collections`,
  - `ListChangeListener.Change` – zmiany na liście,
  - `MapChangeListener.Change` – zmiany w mapie.

```
public class Counter extends Task<Void> {  
  
    @Override  
    protected Void call() throws Exception {  
        for (int i = 0; i < 100; i++) {  
            updateProgress(i, 100);  
            updateMessage("Processing " + i);  
        }  
        return null;  
    }  
}
```

```
new Thread(new Counter()).start();
```

```
@FXML  
private Label label;  
  
@FXML  
private ProgressBar bar;  
  
@Override  
public void initialize(URL location, ResourceBundle resources) {  
    Task<Void> task = new Counter();  
    label.textProperty().bind(task.messageProperty());  
    bar.progressProperty().bind(task.progressProperty());  
}
```

```
<TableView fx:id="table">
    <columns>
        <TableColumn text="%message">
            <cellValueFactory>
                <PropertyValueFactory property="message"/>
            </cellValueFactory>
        </TableColumn>
        <TableColumn fx:id="progressColumn" text="%progress">
            <cellValueFactory>
                <PropertyValueFactory property="progress"/>
            </cellValueFactory>
        </TableColumn>
    </columns>
</TableView>
```

```
@FXML
private TableView<Counter> table;

@FXML
private TableColumn<Counter, Double> progressColumn;

private ObservableList<Counter> counters = FXCollections.observableArrayList();

@Override
public void initialize(URL location, ResourceBundle resources) {
    table.setItems(counters);
    progressColumn.setCellFactory(ProgressBarTableCell.<Counter>forTableColumn());
}
```

- Oracle Corporation, *The Java Tutorials*, <https://docs.oracle.com/javase/tutorial/>.
- Oracle Corporation, *Java Platform, Standard Edition (Java SE) 8, Client Technologies*,  
<https://docs.oracle.com/javase/8/javase-clienttechnologies.htm>.
- *Getting Started with JavaFX 13*, <https://openjfx.io/openjfx-docs/>.
- *JavaFX 11 API documentation*, <https://openjfx.io/javadoc/11/>.



POLITECHNIKA  
GDAŃSKA

JPA

Platformy Technologiczne

Michał Wójcik

## Java Persistence API (JPA):

- specyfikacja dla bibliotek mapowania obiektowo-relacyjnego,
- popularne implementacje:
  - Hibernate,
  - EclipseLink;
- wykorzystywana w wielu frameworkach i środowiskach:
  - Java SE, Java EE, Spring Framework, Play Framework;
- nie wymaga budowania skomplikowanych obiektów DAO (ang. Data Access Object),
- obsługuje transakcje ACID (Atomicity, Consistency, Isolation, Durability),
- niezależna od dostawcy bazy danych:
  - sterowniki JDBC dla wszystkich popularnych baz,
  - przy prostszych przypadkach możliwość uniknięcia zabawy z SQL.

JPA jest jedynie standardem, istnieje kilka implementacji:

- Hibernate, od Red Hat (serwer WildFly),
- Toplink od Oracle,
- OpenJPA od Apache Software Fundation,
- EclipseLink – implementacja referencyjna od Eclipse Fundation (Eclipse IDE).

## Klasy encyjne (entity classes):

- klasy odwzorowywane na table przechowywane w bazie danych;
- zwykłe klasy POJO (Plain Old Java Object),
- pola klasy nie mogą być publiczne,
- każdy obiekt encyjny ma jednoznacznie identyfikujący go klucz główny:
  - złożone klucze główne są reprezentowane przez oddzielne klasy, które muszą implementować metody `hashCode()` i `equals()`;
- oznaczone za pomocą adnotacji.

**Adnotacje** na poziomie **klasy**:

- `@Entity` – oznaczenie klasy jako encyjnej (wymagana),
- `@Table` – właściwości tabeli w bazie danych, np.:
  - `name` – nazwa tabeli,
  - `indexes` – dodatkowe indeksy (domyślny indeks dla klucza głównego).

### Adnotacje na poziomie pól:

- **@Id** – oznaczenie klucza głównego,
- **@GeneratedValue** – automatyczne generowanie wartości klucza głównego,
- **@Column** – własności kolumny w bazie danych, np.:
  - **name** – nazwa kolumny,
  - **nullable** – czy pole jest wymagane,
  - **unique** – czy kolumna przechowuje unikalne wartości,
  - **updatable** – czy wartość w kolumnie można modyfikować po zapisaniu nowego wiersza w tabeli;
- **@Temporal** – wymagane dla typów **Date** i **Calendar**:
  - umożliwia wykorzystanie bazodanowych typów do przechowywania daty/czasu, jeśli baza je oferuje;
- **@Transient** – oznaczenie pól klasy, które mają zostać pominięte przy mapowaniu obiektowo-relacyjnym.

```
@NoArgsConstructor  
@EqualsAndHashCode  
@ToString  
@Entity  
public class Book {  
  
    @Getter  
    @Id  
    @GeneratedValue  
    private Integer id;  
  
    @Getter  
    @Setter  
    private String title;  
  
    @Getter  
    @Setter  
    @Column(unique = true)  
    private String isbn;  
  
}
```

Atrybut **strategy** adnotacji **@GeneratedValue** określa sposób generowania wartości kluczy głównych:

- **GenerationType.IDENTITY**
  - MySQL: `AUTO_INCREMENT`,
  - PostgreSQL: `SERIAL`,
  - MS SQL: `IDENTITY(1,1)`;
- **GenerationType.SEQUENCE**:
  - wartości generowane przez sekwencję bazodanową,
  - np. Oracle Database: `CREATE SEQUENCE invoice_seq START WITH 1;`
- **GenerationType.TABLE**:
  - wartości generowane na podstawie pomocniczej tabeli;
- **GenerationType.AUTO**:
  - implementacja JPA wybiera strategię w zależności od bazy danych.

**Związki** pomiędzy tabelami w bazie danych można odzwierciedlić jako **powiązania** między encjami:

- jednokierunkowe – jedna z klas encyjnych zawiera odwołanie do drugiej,
- dwukierunkowe – obie klasy encyjne posiadają wzajemne odwołania,

**Adnotacje** definiujące **powiązania**:

- `@OneToOne`,
- `@OneToMany`,
- `@ManyToOne`
- `@ManyToMany`.

Wybrane właściwości adnotacji:

- `mappedBy` - oznaczenie pola będącego właściwicielem związku,
- `cascade` - operacje kaskadowe na elementach zależnych od siebie.

```
@NoArgsConstructor
@EqualsAndHashCode
@ToString
@Entity
public class Employee {

    @Getter
    @Id
    @GeneratedValue
    private Integer id;

    @Getter
    @Setter
    private String name;

    @Getter
    @Setter
    private String surname;

    @Getter
    @Setter
    @ManyToOne
    private Department department;

}
```

```
@NoArgsConstructor
@EqualsAndHashCode
@ToString
@Entity
public class Department {

    @Getter
    @Id
    @GeneratedValue
    private Integer id;

    @Getter
    @Setter
    private String name;

}
```

## Związki dwukierunkowe:

- w relacyjnej bazie danych nie występuje koncepcja związku dwukierunkowego:
  - związki modelowane przy użyciu kluczy obcych,
- w obiektowym języku programowania dwukierunkowe związki są częstym rozwiązaniem,
- obecność atrybutu **mappedBy** wskazuje, że pole reprezentuje drugą stronę związku modelowanego przez inne pole,
- strona bez **mappedBy** jest stroną właściwą – kontroluje związek między encjami.

```
@NoArgsConstructor
@EqualsAndHashCode
@ToString
@Entity
public class Employee {

    @Getter
    @Id
    @GeneratedValue
    private Integer id;

    @Getter
    @Setter
    private String name;

    @Getter
    @Setter
    private String surname;

    @Getter
    @Setter
    @ManyToOne
    private Department department;

}
```

```
@NoArgsConstructor
@EqualsAndHashCode
@ToString
@Entity
public class Department {

    @Getter
    @Id
    @GeneratedValue
    private Integer id;

    @Getter
    @Setter
    private String name;

    @Getter
    @OneToMany(mappedBy = "department")
    private List<Employee> employees;

}
```

```
@NoArgsConstructor
@EqualsAndHashCode
@ToString
@Entity
public class Employee {

    @Getter
    @Id
    @GeneratedValue
    private Integer id;

    @Getter
    @Setter
    private String name;

    @Getter
    @Setter
    private String surname;

    @Getter
    @Setter
    @ManyToOne
    private Employee superior;

    @Getter
    @OneToMany(mappedBy = "superior")
    private List<Employee> subordinates;

}
```

## Jednostka trwałości (Persistence Unit):

- grupuje klasy encyjne mapowane na tabele w tej samej bazie danych,
- aplikacja może definiować kilka jednostek trwałości (używać kilku baz danych),
- określa sposób połączenia z bazą danych:
  - nazwa hosta, port, nazwa bazy, login, hasło, sterownik JDBC do połączenia,
  - źródło danych (ang. DataSource) dostarczane przez środowisko wykonawcze (np. serwer aplikacji);
- standardowo konfigurowana w pliku persistence.xml (w katalogu META-INF projektu),
- frameworki mogą udostępniać alternatywne sposoby konfiguracji, definiować ustawienia domyślne
  - np. Spring Framework: application.properties + domyślna jednostka trwałości obejmująca wszystkie klasy encyjne projektu.

**Kontekst** (ang. Persistence Context):

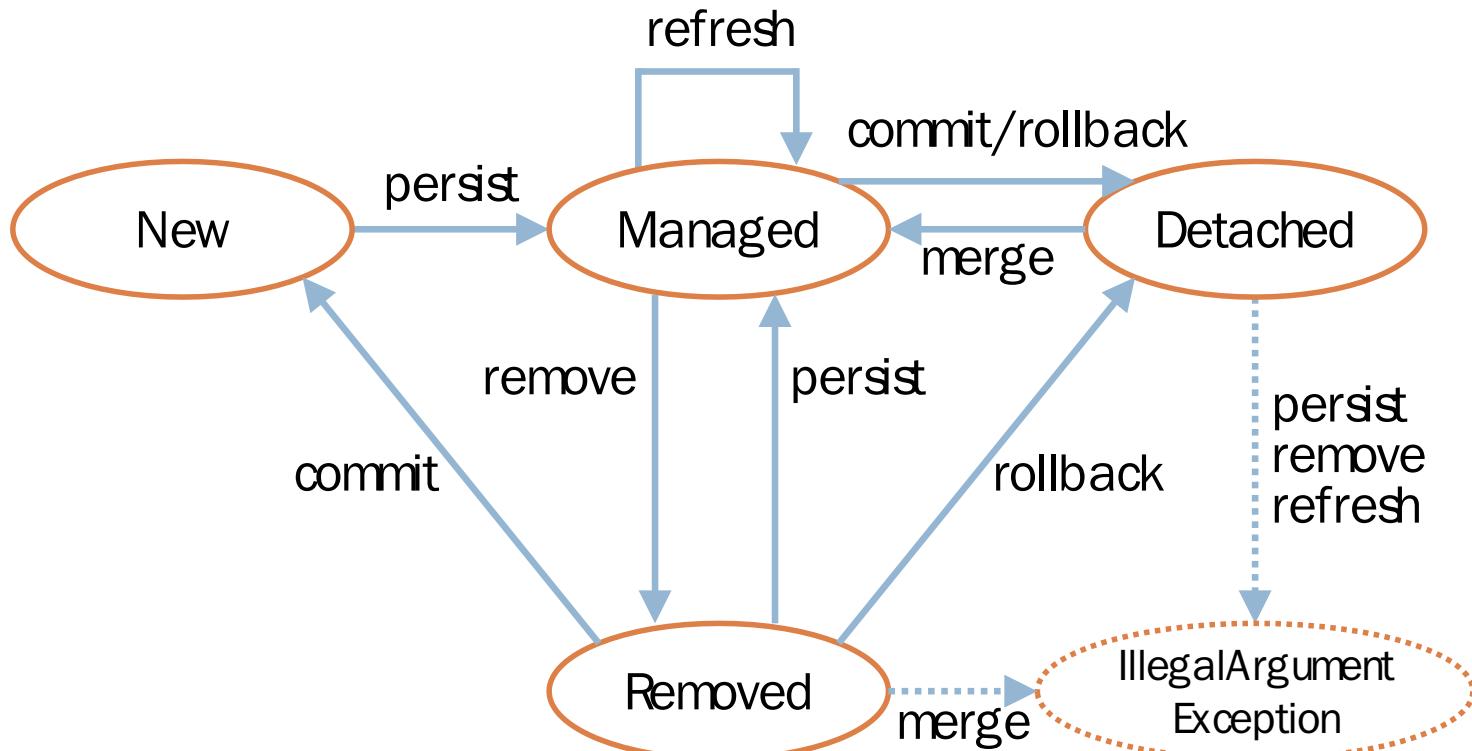
- zbiór zarządzanych (ang. managed) obiektów encyjnych,
- zmiany w zarządzanych obiektach encyjnych są śledzone i zapisywane do bazy danych po zatwierdzeniu transakcji.

**Tożsamość bazodanowa** (ang. Persistence Identity):

- tożsamość obiektu encyjnego wyrażona identyfikatorem powiązany z istniejącym kluczem głównym w bazie danych.

stan encji	związana z Persistence Context	Posiada Persistent Identity
new	✗	✗
managed	✓	✓
detached	✗	✓
removed	✓	✓

Stan *removed* - przeznaczony do usunięcia przy zatwierdzeniu transakcji (obiekt istnieje w bazie danych dopóki transakcja nie zakończy się sukcesem).



**Entity Manager** - udostępnia podstawowe operacje zarządzania encjami:

- wywoływane operacje zmieniają stan obiektu encyjnego (new/managed/detached/removed),
- `void persist(Object o)` – zapis do bazy danych (new  $\Rightarrow$  managed),
- `<T> T merge(T entity)` – detached  $\Rightarrow$  managed
- `void remove(Object o)` – usunięcie encji (managed  $\Rightarrow$  removed)
- `void refresh(Object o)` – aktualizuje stan obiektu encyjnego na podstawie bazy,
- `<T> T find(Class<T> entityClass, Object key)` – wyszukiwanie na podstawie klucza głównego,
- `EntityTransaction getTransaction()` – zwraca obiekt transakcji:
  - większość frameworków umożliwia automatyczne zarządzanie transakcjami.

**Entity Transaction:**

- `begin()` – rozpoczyna transakcję,
- `commit()` – kończy transakcję, zapisuje zmiany do bazy, odłącza zarządzane obiekty encyjne,
- `rollback()` – wycofuje transakcję, odłącza zarządzane obiekty encyjne.

Sposoby **pozyskania** obiektu **EntityManager** w Java SE:

- poprzez fabrykę,
- fabryka może być wykorzystywana przez wiele wątków,
- instancja **EntityManager** może być bezpiecznie używana tylko przez jeden wątek.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("booksPu");
EntityManager em = factory.createEntityManager();
```

Przykładowa klasa:

```
@NoArgsConstructor  
@EqualsAndHashCode  
@ToString  
@Entity  
public class Book {  
  
    @Getter  
    @Setter  
    private String title;  
  
    @Getter  
    @Setter  
    @Id  
    @Column(unique = true)  
    private String isbn;  
  
}
```

Uzyskanie obiektu fabryki:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("testPU");
EntityManager em;
```

Zapisanie nowego obiektu w transakcji:

```
em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(new Book("Ruda Sfora", "1"));
em.getTransaction().commit();
em.close();
```

Wyszukanie obiektu w bazie:

```
em = emf.createEntityManager();
Book wolf = em.find(Book.class, "1");
System.out.println(wolf);
em.close();
```

Zamknięcie fabryki na koniec programu:

```
emf.close();
```

## Java Persistence Query Language (JPQL):

- obiektowo zorientowany język zapytań o składni nawiązującej do języka SQL,
- opiera się na zdefiniowanym w projekcie modelu klas encyjnych,
- umożliwia odpytywanie bazy danych (**SELECT**), aktualizację (**UPDATE**) i usuwanie (**DELETE**) encji,
- zapytania automatycznie tłumaczone na język SQL:
  - z uwzględnieniem automatycznych złączeń tabel w czasie trawersowania związków między encjami.

```
@Entity
public class Product {
    @Id
    UUID id = UUID.randomUUID();
    String name;
    String description;
    Integer price;
    Integer amount;
}
```

Składnia zapytania **SELECT**:

```
select_statement ::= select_clause from_clause
                  [where_clause]
                  [group_by_clause]
                  [having_clause]
                  [orderby_clause]
```

Pobranie wszystkich produktów:

```
SELECT p FROM Product p
```

Pobranie produktów o określonej nazwie:

```
SELECT p FROM Product p WHERE p.name = :name
```

Pobranie produktów o nazwie zgodnej z wyrażeniem regularnym – operator `LIKE`:

```
SELECT p FROM Product p WHERE p.name LIKE :name
```

Wyszukanie produktów, których opis zawiera frazę VT-x w dowolnym miejscu:

```
SELECT p FROM Product p WHERE p.description LIKE '% VT-x %'
```

Bez uwzględniania wielkości znaków:

```
SELECT p FROM Product p WHERE LOWER(p.name) LIKE LOWER('%Laptop%')
```

Pobranie produktów, których stan jest niski (poniżej 5 sztuk), posortowanych według nazw:

```
SELECT p FROM Product p WHERE p.amount < 5 ORDER BY p.name
```

Wywołanie zapytań bez parametrów:

```
String queryString = "SELECT p FROM Product p";
Query query = em.createQuery(queryString, Product.class);
List<Product> products = query.getResultList();
```

Wywołanie zapytań z parametrami:

```
String queryString = "SELECT p FROM Product p WHERE p.name LIKE :name";
Query query = em.createQuery(queryString, Product.class);
query.setParameter("name", "Laptop");
List<Product> products = query.getResultList();
```

Zliczanie:

```
String queryString = "SELECT COUNT(p) FROM Product p WHERE p.name LIKE :name";
Query query = em.createQuery(queryString, Long.class);
query.setParameter("name", "Laptop");
Long count = query.getSingleResult();;
```

OFFSET oraz LIMIT:

```
String queryString = "SELECT p FROM Product p";
Query query = em.createQuery(queryString, Product.class);
query.setFirstResult(21);
query.setMaxResults(10);
List<Product> products = query.getResultList();
```

Fluent syntax:

```
em.createQuery("SELECT p FROM Product p WHERE p.name LIKE :name", Product.class)
    .setFirstResult(21);
    .setMaxResults(10);
    .setParameter("name", "Laptop");
    .getResultList();
```

Dla zapytań UPDATE oraz DELETE:

```
String queryString = "DELETE FROM Product p";
Query query = em.createQuery(queryString);
int changedRows = query.executeUpdate();
```

Zapytania używane w wielu miejscach aplikacji można zdefiniować przy użyciu adnotacji `@NamedQuery`:

```
@NamedQueries({  
    @NamedQuery(name = "Product.findAll",  
        query = "SELECT p FROM Product p"),  
    @NamedQuery(name = "Product.findByName",  
        query = "SELECT p FROM Product p WHERE p.name = :name")  
})  
@Entity  
public class Product {  
    //...  
}
```

```
Query query = em.createNamedQuery("Product.findAll");
```

- Java Persistence, [https://en.wikibooks.org/wiki/Java\\_Persistence](https://en.wikibooks.org/wiki/Java_Persistence).
- RedHat, Hibernate ORC Documentation,  
<https://hibernate.org/orm/documentation/5.4/>.



Aplikacje Webowe w Spring Framework  
Platformy Technologiczne  
Michał Wójcik, Waldemar Korłub

Przed erą zaawansowanych urządzeń osobistych (np.: smartphone, smartwatch, itd) aplikacja serwerowa typowo generowała dokumenty HTML które następnie były renderowane przez przeglądarki.

Współcześnie klient w przeglądarce jest jednym z wielu kanałów dostępu do usług na serwerze:

- aplikacje mobilne – urządzenia typu smartphone, tablet, smartwatch,
- oprogramowanie w urządzeniach typu Smart TV, Smart Car,
- inteligentni asystenci: Alexa, Siri.

Nowe aplikacje klienckie wymagają programistycznego API zamiast dokumentów HTML.

Skoro serwer i tak musi udostępniać API realizujące logikę biznesową, to klient w przeglądarce również mógłby z niego korzystać. Wykorzystanie tego samego API na serwerze dla wszystkich aplikacji klienckich (aplikacje mobilne, strony internetowe, itd) oznacza łatwiejszą konstrukcję serwera.

Aplikacje klienckie w przeglądarce (web front-end) wykorzystują frameworki oparte na języku JavaScript:

- Angular(Google),
- React(Facebook),
- Vue.js,
- Backbone.js
- ...

**Usługi sieciowe** - aplikacje klienckie i serwerowe komunikujące się ze sobą poprzez WWW z wykorzystaniem protokołu HTTP:

- interoperacyjność pomiędzy aplikacjami uruchomionymi na różnych platformach i frameworkach,
- opisy przetwarzalne przez aplikacje,
- wykorzystanie XML lub JSON,
- luźno powiązane usługi można łączyć w celu uzyskania skomplikowanych operacji.

## Podstawowe założenia:

- bezstanowe, interakcja powinna być odporna na restart serwera,
- usługi buforowania serwera aplikacji oraz innych elementów mogą być wykorzystane w celu usprawnienia wydajności, o ile elementy zwracane przez usługę nie są generowane dynamicznie i mogą być zbuforowane,
- możliwy opis za pomocą WADL lub Swagger,
- producent i konsument muszą obsługiwać ten sam kontekst i przesyłaną zawartość,
- mały narzut na dane, idealny dla urządzeń z ograniczonymi zasobami,
- często wykorzystywane w połączeniu z technologią AJAX.

**Zasoby** powinny być ułożone w hierarchię (rzeczowniki zamiast czasowników):

- `api/books` – wszystkie książki,
- `api/books/1` – książka o indeksie 1,
- `api/books/1/authors` – autorzy konkretnej książki,
- `api/books/1/authors/1` – autor konkretnej książki książki o indeksie 1.

Zamiast indeksów można użyć innych wartości unikatowo opisujących dany element:

- kolejny indeks,
- klucz domenowy (np.: isbn dla książek),
- generowany UUID.

### Typy parametrów:

- path param – parametr będący elementem adresu, wskazuje na zasoby,
- query param – parametr doklejany do adresu (po znaku ?), doprecyzowują zapytanie użytkownika (aplikacji klienckiej):
  - stronicowanie: `api/books?offset=10&limit=5` – pobranie pięciu książek począwszy od 10.
  - filtrowanie: `api/books?unread=true` - pobranie tylko nieprzeczytanych książek,
  - sortowanie: `api/books?sort=latest` - pobranie książek posortowanych po dacie wydania.

Zarządzanie zasobami poprzez operacje HTTP:

- PUT - aktualizacja zasobu,
- GET - pobranie aktualnego stanu zasobu,
- POST - stworzenie nowego zasobu,
- DELETE - usunięcie zasobu.

operacja	zasób	efekt
GET	api/books	pobranie wszystkich książek
GET	api/books/1	pobranie konkretnej książki
POST	/api/books	dodanie nowej książki
POST	/api/books/1	nie stosuje się
PUT	/api/books	zastąpienie kolekcji, zazwyczaj nie stosuje się
PUT	/api/books/1	aktualizacja książki
DELETE	/api/books	usunięcie kolekcji
DELETE	/api/books/1	usunięcie książki

operacja	REST	SQL
create	POST	INSERT
read	GET	SELECT
update	PUT	UPDATE
delete	DELETE	DELETE

Wyniki sygnalizowane kodami odpowiedzi HTTP.

Żądanie	zasób	przykładowy kod błędu
GET	api/books	200 OK
GET	api/books/1	200 OK, 404 Not found
POST	/api/books	201 Created (+location), 403 Forbidden
POST	/api/books/1	405 Method not allowed
PUT	/api/books	405 Method not allowed
PUT	/api/books/1	200 OK, 401 Unauthorized
DELETE	/api/books	200 OK, 403 Forbidden
DELETE	/api/books/1	200 OK, 403 Forbidden, 404 Not found

Popularne narzędzia pracy z REST API:

- Postman:
  - aplikacja Chrome (do pobrania z Chrome Web Store),
  - aplikacja standalone(do pobrania ze strony producenta);
- SoapUI - Standalone.

Termin Spring odnosi się do całej rodziny projektów zbudowanych na projekcie Spring Framework:

- Spring Boot,
- Spring Framework,
- Spring Data,
- Spring Cloud,
- Spring Cloud Data Flow,
- Spring Security,
- Spring Session,
- Spring Integration,
- Spring HATEOAS,
- Spring REST Docs,
- Spring Batch,
- Spring AMQP,
- Spring for Android,
- Spring CredHub,
- Spring Flo,
- Spring for Apache Kafka,
- Spring LDAP,
- Spring Mobile,
- Spring Roo,
- Spring Shell,
- Spring StateMachine,
- Spring Vault,
- Spring Web Flow,
- Spring Web Services.

**Spring Framework** - framework pozwalający na stworzenie złożonych aplikacji webowych oraz klasy enterprise uruchamianych na maszynie wirtualnej Java:

- wspiera automatyczne wstrzykiwanie zależności,
- zawiera framework Spring Web MVC,
- ...

**Spring Web MVC** - framework webowy zbudowany na podstawie specyfikacji Servlet będący elementem Spring Framework:

- implementacja modelu **M**odel-**V**iew-**C**ontroller,
- wspiera implementację usług typu REST,
- ...

**Spring Boot** - mechanizm ułatwiający tworzenie aplikacji opartych na platformie Spring:

- ułatwia konfigurację projektu,
- dopasowuje kompatybilne elementy platformy (zarządzanie wersjami),
- ułatwia budowanie wersji dystrybucyjnej:
  - aplikacje webowe wymagają uruchomienia w ramach serwera webowego w celu obsługi żądań HTTP,
  - pozwala na wykorzystanie wbudowanego (embedded) serwera Tomcat konfigurowane z poziomu projektu.

Zamiast tworzyć projekt od podstaw i samemu dodawać zależności można:

1. skorzystać z Spring Initializr: <http://start.spring.io/>,
2. określić GroupId i ArtifactId,
3. wybrać odpowiednie moduły (zależności):
  - Web,
  - JPA,
  - Derby (baza danych),
  - ...
4. wygenerować projekt,
5. pobrać archiwum ZIP i je rozpakować,
6. otworzyć projekt w ulubionym IDE.

Punkt wejścia aplikacji:

```
@SpringBootApplication
public class BookshopApplication {
    public static void main(String[] args) {
        SpringApplication.run(BookshopApplication.class, args);
    }
}
```

umożliwia uruchomienie aplikacji za pomocą:

```
java -jar app.jar
```

Uruchamia wbudowany serwer Tomcat i wdraża aplikację.

```
@RestController
@RequestMapping("/shop")
public class ShopController {

    private OrderService service;

    public ShopController(OrderService service) {
        //...
    }

    @GetMapping("/orders")
    public List<Order> listOrders(@RequestParam(required = false) String sort) {
        //...
    }

    @GetMapping("/orders/{id}")
    public ResponseEntity<Order> getOrder(@PathVariable UUID id) {
        //...
    }

    @PostMapping("/orders")
    public ResponseEntity<Void> addOrder(@RequestBody Order order) {
        //...
    }
}
```

Powyzsza konfiguracja pozwala na wysłanie następujących żądań HTTP (zakładając że serwer został uruchomiony na domyślnym porcie 8080):

- GET na `localhost:8080/shop/orders` - zwróci wszystkie zamówienia,
- GET na `localhost:8080/shop/orders?sort=asc` - zwróci wszystkie zamówienia posortowane rosnąco,
- GET na `localhost:8080/shop/orders/9eaee866-6eb3-11ea-bc55-0242ac130003` - zwróci zamówienie o konkretnym id,
- POST na `localhost:8080/shop/orders` - dodanie nowego zamówienia.

Użyte adnotacje:

- **@RestController** - zarejestrowanie klasy jako kontrolera dla usług typu REST, instancja klasy będzie tworzona automatycznie podczas obsługi żądania,
- **@RequestMapping** - rejestracja bazowego adresu dla wszystkich metod w klasie,
- **@GetMapping** - obsługa żądania HTTP typu GET,
- **@PostMapping** - obsługa żądania HTTP typu POST,
- **@PathVariable** - odwzorowanie parametru będącego elementem adresu na argument metody,
- **@RequestBody** - odwzorowanie ciała żądania (domyślnie z formatu JSON) na obiekt.

```
@GetMapping("/orders")
public List<Order> listOrders(@RequestParam(required = false) String sort) {
    List<Order> orders;
    if (sort != null) {//optional query param
        orders = service.findAllSorted(sort);
    } else {
        orders = service.findAll();
    }
    return orders; //HTTP 200 code with body automatically converted to JSON array
}
```

```
@GetMapping("/orders/{id}")
public ResponseEntity<Order> getOrder(@PathVariableUUID id) {
    Order order = service.find(id);
    if (order == null) {
        return ResponseEntity.notFound().build(); //HTTP 404 error code
    } else {
        return ResponseEntity.ok(order); //HTTP 200 code with body in JSON format
    }
}
```

Ciało odpowiedzi:

- automatycznie budowanie (domyślnie do formatu JSON) na podstawie obiektu zwróconego przez metodę,
- za pomocą klasy **ResponseEntity** pozwalającej na sterowanie wszystkimi aspektami odpowiedzi HTTP (kod, ciało, nagłówki).

**Logika biznesowa:**

- powinna znajdować się poza kontrolerem,
- powinna być niezależna od protokołu HTTP.

Za przedstawienie wyników logiki biznesowej w kategoriach protokołu HTTP odpowiada kontroler, np. nie znaleziono produktu o podanym id:

- w warstwie biznesowej: `null`,
- w kontrolerze: odpowiedź `404`.

Deklaracja klasy realizującej logikę biznesową:

```
@Service
public class OrdersService{

    final EntityManager em;

    public OrdersService(EntityManager em) {
        this.em = em;
    }

    //logic for orders support
}
```

Dostarczenie serwisu do kontrolera:

```
@RestController
@RequestMapping("/shop")
public class ShopController {

    private OrderService service;

    public ShopController(OrderService service) {
        this.service = service;
    }

}
```

Komponent nie pozyskuje samodzielnie swoich zależności – zamiast tego zależności są dostarczane z zewnątrz:

- przez parametry konstruktora,
- przez metody **set**,
- bezpośrednio do pól obiektu.

Spring oferuje mechanizm wstrzykiwania zależności m.in. do serwisów i kontrolerów.

Za budowanie zależności i ich dostarczanie odpowiada framework Spring:

- Spring skanuje wszystkie klasy w projekcie w poszukiwaniu adnotacji `@Service` / `@Controller` / `@Component` itd,
- odnalezione komponenty mogą być wstrzykiwane jako zależności do innych komponentów,
- gdy potrzebna jest nowa instancja zależności Spring buduje ją wstrzykując najpierw jej własne zależności,
- jeśli występuje kilka konstruktorów, adnotacja `@Autowired` wskazuje, którego z nich ma użyć Spring.

Konfiguracja w pliku `application.properties`:

```
#Connection string
spring.datasource.url=jdbc:derby://localhost:1527/db

#Sterownik dla bazy danych i dialekt jazyka SQL
spring.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver40
spring.jpa.database-platform=org.hibernate.dialect.DerbyTenSevenDialect

#Nazwa uzytkownika bazodanowego i haslo
spring.datasource.username=db_user
spring.datasource.password=db_pass

#Generowanie tabel w bazie danych na podstawie modelu encji
spring.jpa.hibernate.ddl-auto=update
```

Instancje klasy `EntityManager` wstrzykiwane jako zależności komponentów.

Użycia transakcji wymagają wszystkie operacje modyfikujące stan w bazie danych, np.:

```
@Transactional  
public void save(Order order) {  
    if(/*new order*/) {  
        em.persist(order);  
    } else{  
        em.merge(order);  
    }  
}
```

Metody oznaczone adnotacją `@Transactional` wykonywane w ramach transakcji.

Wycofanie transakcji (ang. rollback) następuje w reakcji na wyjątki RuntimeException, np.:

```
public class OutOfStockException extends RuntimeException {  
}
```

```
@Transactional  
public void placeOrder(Order order) {  
    for (Book orderBook : order.getBooks()) {  
        Book book = em.find(Book.class, orderBook.getId());  
        if (book.getAmount() < 1) {  
            throw new OutOfStockException();  
        } else {  
            int newAmount = book.getAmount() -1;  
            book.setAmount(newAmount); //managed object  
        }  
    }  
    em.persist(order);  
}
```

Stan (liczba) książek dla każdej książki z zamówienia zostanie zaktualizowana na bazie w wyniku zakończenia transakcji. W przypadku przerwania transakcji, żadne zmiany nie zostaną zapisane.

- Spring Guides, <https://www.baeldung.com/spring-tutorial>.
- Baeldung, Spring Tutorial, <https://www.baeldung.com/spring-tutorial>.



Testowanie oprogramowania  
Platformy Technologiczne  
Michał Wójcik

**Testowanie oprogramowania** – jeden z procesów wytwarzania oprogramowania.

Testowanie ma na **celu**:

- *weryfikację oprogramowania* - czy wytwarzane oprogramowanie jest zgodne ze specyfikacją.
- *validacja oprogramowania* - czy oprogramowanie jest zgodne z oczekiwaniami użytkownika.

Testowanie oprogramowania może być wdrożone w dowolnym momencie wytwarzania oprogramowania (w zależności od stosowanej metody).

## Poziomy testowania:

- testy jednostkowe – testowanie kodu na poziomie klas
- testy *integracyjne* – testowanie aplikacji jako całości oraz integracji z innymi elementami
- testy systemowe – testy przeprowadzane na kompletnie zintegrowanym systemie,
- testy *akceptacyjne* – w których możemy wyróżnić również specyficzne dla oprogramowania komercyjnego testy alfa i testy beta

Istnieją również inne podziały ze względu na umiejscowienie lub przeprowadzanie testów.

**Ograniczenia** testowania:

- programu nie da się przetestować całkowicie,
- testy nie udowodnią braku błędów,
- bardzo często specyfikacja produktu nie jest gotowa co utrudnia testowanie,
- często błędy nie są poprawiane,
- błędy zawsze będą występować w aplikacji.

## Testowanie ręczne:

- powolne,
- kosztowne,
- nudne,
- powtarzalność trudna do uzyskania.

## Testowanie automatyczne:

- szybkie,
- powtarzalne,
- wykonywane podczas budowania aplikacji,
- ułatwia reagowanie na zmiany.

## Testy jednostkowe (unit tests):

- weryfikacja poprawności działania pojedynczych elementów aplikacji, testy automatyczne,
- wykonywane automatycznie,
- przygotowanie testu polega na napisaniu kodu testującego sprawdzającego kod aplikacji,
- mogą być realizowane podczas procesu budowania aplikacji,
- dokumentują oczekiwane zachowanie testowanego komponentu,
- fakt, że każdy komponent działa poprawnie nie znaczy że całość działa poprawnie.

## Testowane zagadnienia:

- **happy path** - (poprawne przebiegi) normalna ścieżka przetwarzania,
- **corner case** - (przypadki graniczne) krańce przedziałów dozwolonych wartości:
  - np.: min/max;
- **unhappy path** - przypadki wystąpienia błędów,
  - np błędne dane wejściowe.

Wybrane **biblioteki**:

- wykonywanie testów:
  - JUnit (najpopularniejsza),
  - TestNG;
- assercje:
  - AssertJ,
  - Hamcrest;
- mokowanie / atrapy:
  - Mockito,
  - PowerMockito,
  - EasyMock.

**Kod testujący:**

- aplikacje testujemy z wykorzystaniem nowych klas w języku Java które zawierają kod testujący,
- dane wejściowe przygotowujemy z wykorzystaniem bibliotek tworzących *atrapę* obiektów,
- wyniki działania sprawdzamy z wykorzystaniem *asercji*.

```
-project-name
|--pom.xml
|--src
|---main
|----java
|----resources
|---test <--right here
|----java
|----resources
|--target
```

Testy nie wchodzą w część aplikacji dystrybuowanej do klienta. W przypadku projektów typu Maven, klasy testowe jak i zasoby testowe powinny znajdować się w odpowiednim katalogu, który jest pomijany podczas przygotowania paczki do dystrybucji.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Zależności wykorzystywane wyłącznie podczas testów powinny znajdować się w odpowiednim zasięgu.

Uruchomienie testów jest możliwe za pomocą odpowiedniej fazy domyślnego cyklu narzędzia Maven:

```
mvn test
```

lub z poziomu IDE uruchamiając testy jako całość, pojedynczą klasę testową lub pojedynczy test.

Klasa testowana:

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a - b;  
    }  
  
}
```

Klasa testująca:

```
public class CalculatorTest {  
  
    @Test  
    public int add() {  
        Calculator calc = new Calculator();  
        int result = calc.add(2, 2);  
        assert result == 4 : "should be 4";  
    }  
  
}
```

Wykorzystanie słowa kluczowego **assert** powoduje rzucenie błędu **AssertionError** w sytuacji gdy nie jest spełniony warunek logiczny. Opcjonalnie można ustawić wiadomość jaka zostanie przypisana do rzuconego błędu.

Kod testowy w metodzie możemy zorganizować z wykorzystaniem zasady **GWT**:

- **Given** - przygotowujemy dane wejściowe oraz atrapy,
- **When** – wywołanie kodu który testujemy,
- **Then** – zweryfikowanie działania metody oraz jej wyniku.

```
@Test
public void add() {
    //Given:
    Calculator calc = new Calculator ();
    //When:
    int result = calc.add(2, 2);
    //Then:
    assert result == 4 : "should be 4";
}
```

Komentarze w kodzie można pominąć zachowując odpowiednie formatowanie.

Kod testowy w metodzie możemy zorganizować z wykorzystaniem zasady **AAA**:

- **Arrange** - przygotowujemy dane wejściowe oraz atrapy,
- **Act** – wywołanie kodu który testujemy,
- **Assert** – zweryfikowanie działania metody oraz jej wyniku.

```
@Test
public void add() {
    //Arrange:
    Calculator calc = new Calculator ();
    //Act:
    int result = calc.add(2, 2);
    //Assert:
    assert result == 4 : "should be 4";
}
```

Komentarze w kodzie można pominąć zachowując odpowiednie formatowanie.

## Nazewnictwo testów:

- klasa testująca zazwyczaj nazywa się tak samo jak klasa testowana z przyrostkiem **Test**,
- metody testujące nazywają się podobnie do metod testowych, przy czym nazwa zawiera również informacje na temat przypadku jaki jest testowany,
- metoda testująca powinna testować tylko jeden przypadek, każda ścieżka w kodzie powinna być testowana przez osobną metodę.

Istnieje wiele konwencji nazewnictwa testów np:

- **MethodName\_StateUnderTest\_ExpectedBehavior**
  - **isRunning\_EngineStarted\_True**
- **Should\_ExpectedBehavior\_When\_StateUnderTest**
  - **Should\_ReturnTrue\_When\_EngineIsStarted**

Inne popularne sposoby nazewnictwa: <https://dzone.com/articles/7-popular-unit-test-naming>.

**Dobre praktyki:**

- jedna metoda testowa testuje tylko jeden przypadek,
- testy nie powinny zależeć od siebie nawzajem,
- metody testowe powinny być proste i czytelne:
  - jeśli nie jest to konieczne warto unikać konstrukcji warunkowych i pętli
- testy nie powinny bazować na stanie środowiska, w którym wykonuje się aplikacja.

**Test driven development** – zwinna metodyka wytwarzania oprogramowania polegająca na wielokrotnym powtarzaniu następujących kroków:

- **dodanie testu** – przygotowanie metod testujących na podstawie specyfikacji,
- **uruchomienie wszystkich testów** – nowy test powinien się nie powieść co oznacza, że testowana jest funkcjonalność, której jeszcze nie ma,
- **przygotowanie kodu przechodzącego test** – kod nie musi być dobry, powinien jedynie działać poprawnie,
- **uruchomienie wszystkich testów** – jeśli nadal występują błędy kod musi być poprawiony,
- **refaktoryzacja kodu** – poprawienie kodu,
- **uruchomienie wszystkich testów** – po refaktoryzacji sprawdzamy czy testy nadal przechodzą.

Klasy **JUnit4** możemy znaleźć w pakiecie **org.junit.\***

Junit 4	Opis
<code>@Test</code>	Identyfikuje metodę jako metodę testową.
<code>@Before</code>	Identyfikuje metodę która będzie wykonywana przed każdym testem. Przygotowanie środowiska np zainicjalizowanie klasy.
<code>@After</code>	Identyfikuje metodę która będzie wykonywana po każdym teście. Posprzątanie po teście np usunięcie plików z dysku.
<code>@BeforeClass</code>	Podobnie jak <code>@Before</code> ale metoda wykonywana jest tylko raz dla klasy testowej.
<code>@AfterClass</code>	Podobnie jak <code>@After</code> ale metoda wykonywana jest tylko raz dla całej klasy testowej.
<code>@Ignore</code>	Umieszczony nad klasą testową wyłącza test.
<code>@Test(expected=Exception.class)</code>	Generuje błąd gdy nasza metoda testowa nie zwraca wskazanego wyjątku.
<code>@Test(timeout=100)</code>	Generuje błąd gdy test wykonuje się dłużej niż zadana wartość.

JUnit posiada bardzo prosty wbudowany mechanizm sprawdzania wyników:

- `assertEquals` – sprawdzenie czy obiekty są takie same, `assertNotEquals`,
- `assertNull` – sprawdzenie czy zmienna wskazuje na null, `assertNotNull`,
- `assertArrayEquals` – sprawdzenie czy tablice są takie same,
- `assertThat` – sprawdzenie warunku (Hamcrest) podanego jako argument.

```
import static org.junit.Assert.assertEquals;
```

```
String expected = "woof";
String actual = "woof";
assertEquals("Should be equal", expected, actual);
```

Bardziej rozbudowany mechanizm do sprawdzania wyników dostarcza biblioteka **Hamcrest**. Biblioteka ta jest dostarczana wraz z **JUnit4** jako zależność.

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.Every.everyItem;
```

```
boolean a = false;
boolean b = true;
assertThat(a, equalTo(b));
assertThat(a, is(equalTo(b)));
assertThat(a, is(b));
assertThat(List.of("fu", "ba", "be"), everyItem(containsString("b")));
```

Biblioteka o bardzo dużych możliwościach, niestety wykorzystanie niektórych implementacji **Matcher** wymaga sporej wiedzy i doświadczenia.

Więcej informacji: <http://hamcrest.org/JavaHamcrest/>.

**AssertJ** to w tej chwili jedna z najpopularniejszych bibliotek do weryfikacji wyników.

Bardzo łatwa w wykorzystaniu, prosty intuicyjny interfejs.

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.13.2</version>
    <scope>test</scope>
</dependency>
```

Więcej informacji na: <http://joel-costigliola.github.io/assertj/index.html>.

```
import static org.assertj.core.api.Assertions.*;
```

```
assertThat(frodo.getName()).isEqualTo("Frodo");
```

```
assertThat(frodo).isNotEqualTo(sauron);
```

```
assertThat(frodo.getName()).startsWith("Fro")
    .endsWith("do")
    .isEqualToIgnoringCase("frodo");
```

```
assertThat(fellowshipOfTheRing).hasSize(9)
    .contains(frodo, sam)
    .doesNotContain(sauron);
```

```
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(33);
```

```
assertThat(fellowshipOfTheRing).extracting("name")
    .contains("Boromir", "Gandalf", "Frodo", "Legolas");
```

```
import static org.assertj.core.api.Assertions.*;
```

```
assertThatThrownBy(() -> {
    throw new Exception("boom!");
}).hasMessage("boom!");
```

```
Throwable thrown = catchThrowable(() -> {
    throw new Exception("boom!");
});
assertThat(thrown).hasMessageContaining("boom");
```

**Atrapy** (Mocks) – podmienianie nietestowanego obiektu na przygotowaną atrapę:

- służy testowaniu metod korzystających z nietestowanych elementów:
  - strumienie do zewnętrznych zasobów,
  - połączenie z bazą danych,
  - połączenie ze zdalnym serwerem,
- specjalnie zbudowane obiekty realizujące zaimplementowane (przewidywalne) zachowanie,
- popularne biblioteki:
  - **EasyMock**,
  - **Mockito**,
  - **PowerMockito** (**Mockito** na sterydach).

W przykładach będziemy korzystać z bibliotek **Mockito** oraz **PowerMockito**.

Więcej informacji oraz przykładów dla **Mockito** na:

<http://static.javadoc.io/org.mockito/mockito-core/2.23.4/org/mockito/Mockito.html>.

Więcej informacji oraz przykładów dla **PowerMockito** na:

<https://github.com/powermock/powermock/wiki>.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.1.0</version>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-module-junit4</artifactId>
    <version>2.0.4</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-api-mockito2</artifactId>
    <version>2.0.4</version>
    <scope>test</scope>
</dependency>
```

Nie testowany interfejs:

```
public interface DataSource {  
    String next();  
}
```

Testowana metoda (do działania wymaga instancji interfejsu):

```
public class DataReader {  
  
    public List<String> readData(DataSource source) {  
        List<String> data = new ArrayList<>();  
        String line;  
        while ((line = source.next()) != null) {  
            data.add(line);  
        }  
        return data;  
    }  
}
```

Testująca metoda:

```
@Test
public void readData() {
    DataSource source = Mockito.mock(DataSource.class);
    Mockito.when(source.next()).thenReturn("woof").thenReturn("woof").thenReturn(null);
    DataReader reader = new DataReader();

    List<String> actual = reader.readData(source);

    assertThat(actual).containsExactly("woof", "woof");
}
```

Aby nie testować kilku rzeczy na raz (metody `readData` i implementacji `DataSource`) wykorzystywana jest atrapa o przewidywalnym zachowaniu.

```
import org.mockito.Mockito;
```

```
DataSource source = Mockito.mock(DataSource.class);
Mockito.when(source.next()).thenReturn("woof").thenReturn("woof").thenReturn(null);
```

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
```

```
DataSource source = mock(DataSource.class);
when(source.next()).thenReturn("woof").thenReturn("woof").thenReturn(null);
```

```
import org.powermock.api.mockito.PowerMockito;
```

```
DataSource source = PowerMockito.mock(DataSource.class);
PowerMockito.when(source.next()).thenReturn("woof").thenReturn("woof").thenReturn(null);
```

```
import static org.powermock.api.mockito.PowerMockito.mock;
import static org.powermock.api.mockito.PowerMockito.when;
```

```
DataSource source = mock(DataSource.class);
when(source.next()).thenReturn("woof").thenReturn("woof").thenReturn(null);
```

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
```

```
@Test
public void readData() {
    DataSource source = mock(DataSource.class);
    when(source.next()).thenReturn("woof").thenReturn("woof").thenReturn(null);
    DataReader reader = new DataReader();

    List<String> actual = reader.readData(source);

    assertThat(actual).containsExactly("woof", "woof");
    verify(source, times(3)).next();
}
```

Oprócz stworzenia atrapy mamy również możliwość sprawdzenia czy nasze atrapy zostały wywołane w poprawny sposób np. Czy parametry jakie zostały przekazane do naszych atrap spełniają nasze wymagania.

Do sprawdzenia wykorzystujemy metodę **Mockito.verify**.

Nietestowany interfejs:

```
public interface DataWriter {  
    void write(String value);  
}
```

Klasa modelowa:

```
@AllArgsConstructor  
@Getter  
public class Elf {  
    private String name;  
    private int age;  
}
```

Testowana klasa:

```
@AllArgsConstructor  
public class ElfWriter {  
  
    private DataWriter writer;  
  
    public void write(Elf elf) {  
        writer.write(elf.getName());  
        writer.write(Integer.toString(elf.getAge()));  
    }  
}
```

Metoda testująca:

```
@Test
public void writeElf() {
    DataWriter writer = mock(DataWriter.class);
    ElfWriter elfWriter = new ElfWriter(writer);
    Elf elf = new Elf("Legolas", 120);

    elfWriter.write(elf);

    verify(writer).write(eq("Legolas"));
    verify(writer).write(eq("120"));
}
```

```
public class ElfWriterTest {  
  
    @Mock  
    private DataWriter writer;  
  
    @InjectMocks  
    private ElfWriter elfWriter;  
  
    @Before  
    public void before() throws Exception {  
        MockitoAnnotations.initMocks(this);  
    }  
  
    @Test  
    public void writeElf() {  
        Elf elf = new Elf("Legolas", 120);  
  
        elfWriter.write(elf);  
  
        verify(writer).write(eq("Legolas"));  
        verify(writer).write(eq("120"));  
    }  
}
```

```
@Test
public void writeElf() {
    DataWriter writer = mock(DataWriter.class);
    ElfWriter elfWriter = new ElfWriter(writer);
    Elf elf = new Elf("Legolas", 120);

    elfWriter.write(elf);

    ArgumentCaptor<String> argument = ArgumentCaptor.forClass(String.class);
    verify(writer, times(2)).write(argument.capture());
    assertThat(argument.getAllValues()).containsExactly("Legolas", "120");
}
```

Obiekt **Captor** wykorzystujemy do przechwycenia parametry przekazanego do naszej atrapy aby następnie wykonać dodatkowe sprawdzenie.

Klasa testowana:

```
public class DataReader {  
    public List<String> readData(File file) {  
        List<String> data = new ArrayList<>();  
        String line;  
        try (FileReader fr = new FileReader(file);  
             BufferedReader br = new BufferedReader(fr)) {  
            while ((line = br.readLine()) != null) {  
                data.add(line);  
            }  
        } catch (IOException ex) {  
        }  
        return data;  
    }  
}
```

Metoda testująca:

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
import static org.powermock.api.mockito.PowerMockito.whenNew;

@RunWith(PowerMockRunner.class)
public class DataReaderTest {

    @Test
    @PrepareForTest(DataReader.class)
    public void writeElf() throws Exception {
        File file = new File("");
        FileReader fr = mock(FileReader.class);
        BufferedReader br = mock(BufferedReader.class);
        whenNew(FileReader.class).withArguments(file).thenReturn(fr);
        whenNew(BufferedReader.class).withArguments(fr).thenReturn(br);
        when(br.readLine()).thenReturn("Woof").thenReturn("woof").thenReturn(null);
        DataReader reader = new DataReader();

        reader.readData(file);

        verify(br, times(3)).readLine();
    }
}
```

- JUnit 4 wiki, <https://github.com/junit-team/junit4/wiki>.
- JUnit 4.12 API, <https://junit.org/junit4/javadoc/4.12/>.
- Hamcrest tutorial, <http://hamcrest.org/JavaHamcrest/tutorial>.
- Hamcrest 1.3 API, <http://hamcrest.org/JavaHamcrest/javadoc/1.3/>.
- AssertJ One minute starting guide, <https://joel-costigliola.github.io/assertj/assertj-core-quick-start.html>.
- AssertJ Core features highlight, <https://joel-costigliola.github.io/assertj/assertj-core-features-highlight.html>.
- AssertJ 3 API, <http://joel-costigliola.github.io/assertj/core-8/api/index.html>.
- Mockito 3.1.0 API, <https://static.javadoc.io/org.mockito/mockito-core/3.1.0/org/mockito/Mockito.html>.
- PowerMock Wiki, <https://github.com/powermock/powermock/wiki>



System kontroli wersji GIT  
Platformy Technologiczne  
Michał Wójcik

**System kontroli wersji** - (VCS, version/revision control system) pozwala na

- śledzenie zmian w kodzie źródłowym,
- łączenie zmian w poszczególnych plikach wprowadzonych przez wiele osób,
- przeglądanie i cofanie zmian.

Systemy kontroli wersji mogą być oparte na różnych **architekturach**:

- **lokalne** - zmiany zapisywane są tylko na lokalnym komputerze:
  - RCS;
- **scentralizowane** - zmiany zapisywane są na serwerze (klient-serwer):
  - CVS,
  - Subversion,
- **rozproszone** - zmiany zapisywane zarówno lokalnie jak i na serwerze:
  - Git,
  - Mercurial.

## Git - rozproszony system kontroli wersji:

- przechowuje migawki plików zamiast zbioru zmian (delty) jak większość systemów VSC,
- możliwość zatwierdzania zmian (commit) do lokalnej historii bez konieczności połączenia z serwerem,
- stworzony m.in. przez Linusa Torvaldsa,
- główne założenia:
  - szybkość,
  - prosta konstrukcja,
  - silne wsparcie dla nieliniowego rozwoju,
  - pełne rozproszenie,
  - wydajna obsługa dużych projektów.

Na rynku istnieje wiele dostawców **zdalnych repozytoriów**:

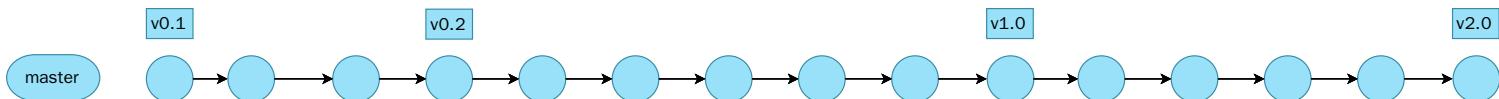
- GitHub <https://github.com/>
- GitLab <https://about.gitlab.com/>
- Bitbucket <https://bitbucket.org/>

```
$ git help
```

```
$ git config --global user.name "ookami"  
$ git config --global user.email "ookami@example.com"  
$ git config --global core.editor "vim"  
$ git config --list
```

Polecenia git:

- `help` - wyświetla pomoc,
- `config <name> <value>` - zarządzanie konfiguracją.



```
$ mkdir woof-world
$ cd woof-world
$ touch README.md
$ vim README.md
$ git init
$ git add README.md
$ git commit -m "Initial commit"
```

```
$ git status
```

Polecenia git:

- **init** - inicjalizacja pustego lokalnego repozytorium (tworzy katalog **.git**),
- **status** - wyświetla stan plików,
- **add <file>** - dodanie pliku do indeksu,
- **commit** - wgranie zmian do repozytorium.

Uwaga: Pierwsze zatwierdzenie zmian domyślnie tworzy gałąź (branch) o nazwie **master**.

```
$ touch hello.sh
$ vim hello.sh
$ git add hello.sh
$ git commit -m "Script created."
```

```
$ git log
$ git checkout 4899b18d477b8eb0ae303539bea3d0b3db34eef6
$ git checkout master
```

Polecenia git:

- **log** - wyświetla historię zmian,
- **checkout <commit\_id>** - przywraca pliki kopi roboczej, należy podać identyfikator zatwierdzonych zmian,
- **checkout <bbranch\_name>** - przełączenie gałęzi, należy podać nazwę gałęzi.

Uwaga: Po przełączeniu kopi roboczej repozytorium znajduje się w stanie **HEAD detached**. Nie należy w tej sytuacji dokonywać zmian i wykonywać operacji **commit**. W celu kontynuowania pracy najłatwiej przełączyć gałąź.

Uwaga: Przełączenie kopi roboczej może powodować utratę niezatwierdzonych zmian.

```
$ git remote add origin https://bitbucket.org/ookami/woof-world/  
$ git push origin master
```

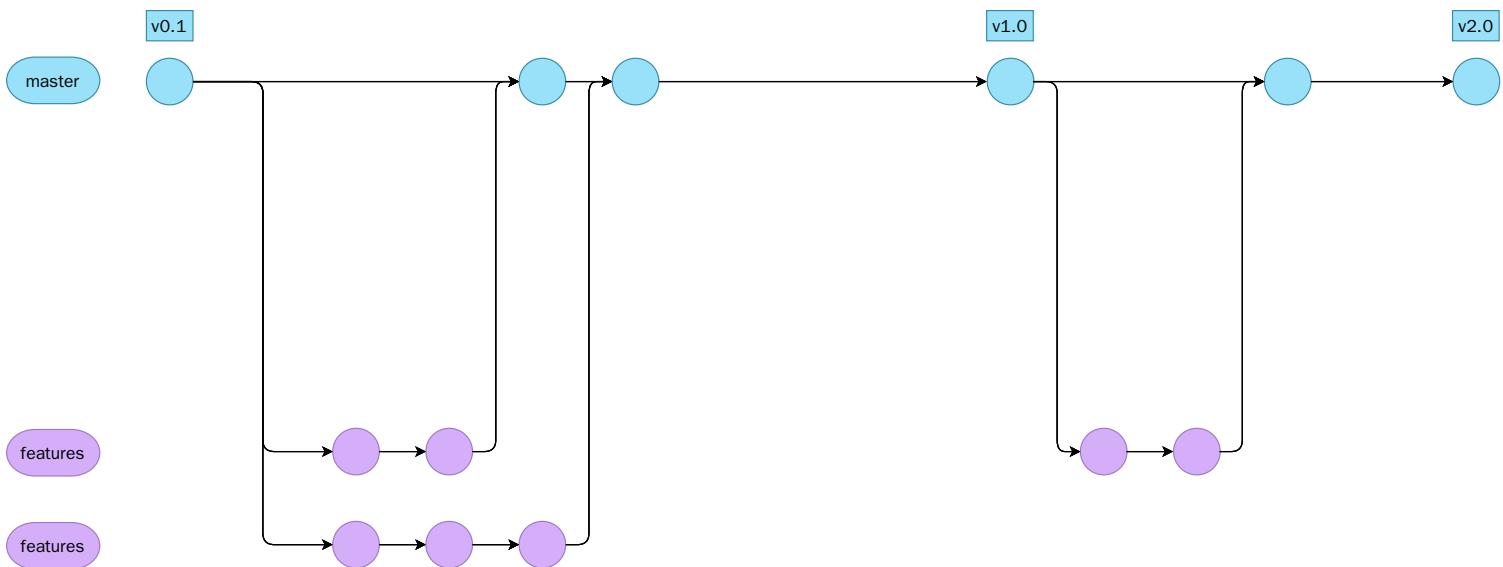
```
$ git push --set-upstream origin master
```

```
$ git remote -v
```

Uwaga: W celu wgrania lokalnego repozytorium do zdalnego należy je wpierw stworzyć.

Polecenia git:

- **remote** - wyświetla zdalne repozytoria:
  - **add <name> <url>** - dodaje zdalne repozytorium;
- **push** - wgranie aktualnej gałęzi do domyślnego repozytorium zdalnego:
  - **push <remote\_name> <branch\_name>** - wgranie konkretnej gałęzi do konkretnego repozytorium zdalnego.



Pojedyncze zadania powinno wykonywać się na innej gałęzi, nazywanej zazwyczaj **feature branch**.

```
$ git checkout -b "feature-1"
```

Proponowane nazewnictwo **feature-<task-id>**.

```
$ git branch -a
```

Polecenia git:

- **branch** - wyświetla informacje o gałęzi.

```
$ vim script.sh
$ git add script.sh
$ git commit -m "Task done"
$ git push origin feature-1
```

```
$ git checkout master  
$ git merge feature-1
```

Polecenia git:

- `merge <branch_name>` - wgrywa zmiany z wybranej gałęzi do aktualnej.

```
$ git checkout -b "feature-2"  
$ vim script.sh  
$ git commit -m "Task done"
```

```
$ git checkout -b "feature-3"  
$ vim script.sh  
$ git commit -m "Task done"
```

W sytuacji gdy równolegle wprowadzono zmiany na tych samych plikach w różnych gałęziach może dojść do konfliktów podczas łączenia.

```
$ git check master  
$ git merge feature-2  
$ git merge feature-3  
Auto-merging test  
CONFLICT (content): Merge conflict in test  
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:  script.sh

no changes added to commit (use "git add" and/or "git commit -a")
```

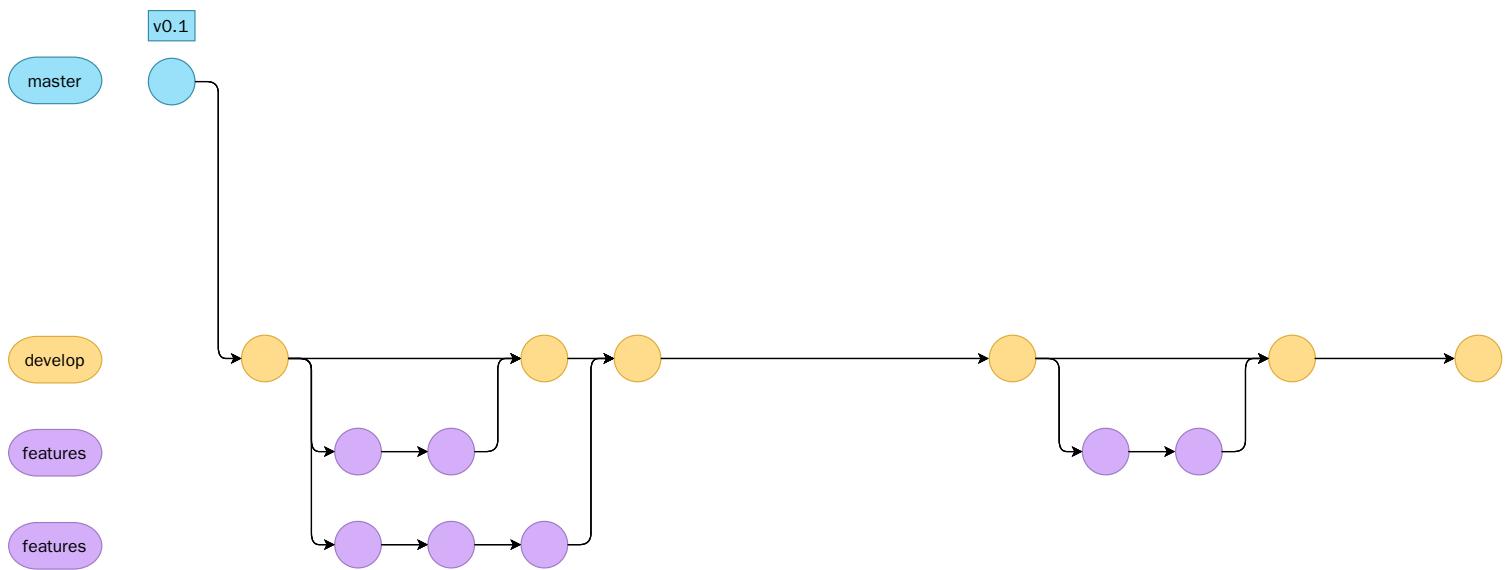
```
$ vim script.sh
```

```
#!/bin/env bash
<<<<< HEAD
echo "Hello World!"
=====
echo "Hello World?"
>>>>> feature-3
```

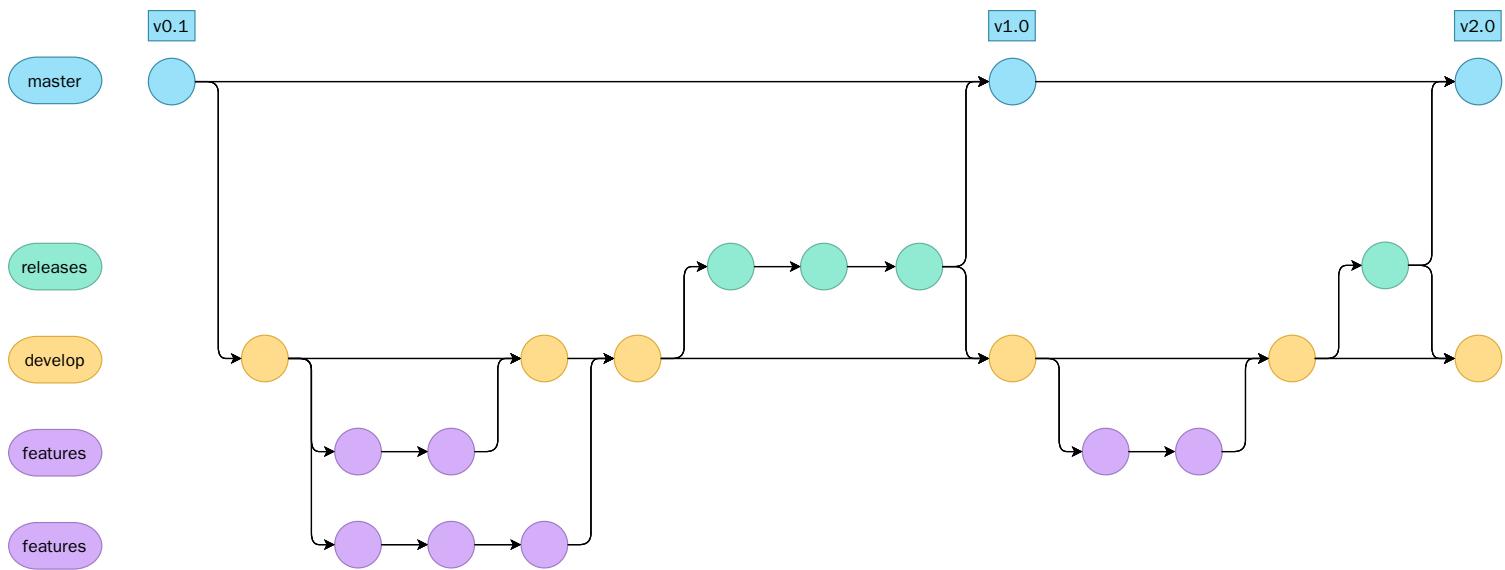
Po nieudanym scalaniu do pliku zostają dodane znaczniki:

- <<<<< <b>branch\_name</b>> - początek konfliktu w aktualnej gałęzi,
- ===== - granica między konfliktami,
- >>>>> <b>branch\_name</b>> - koniec konfliktu w wybranej gałęzi.

```
$ git add script.sh
$ git commit
```

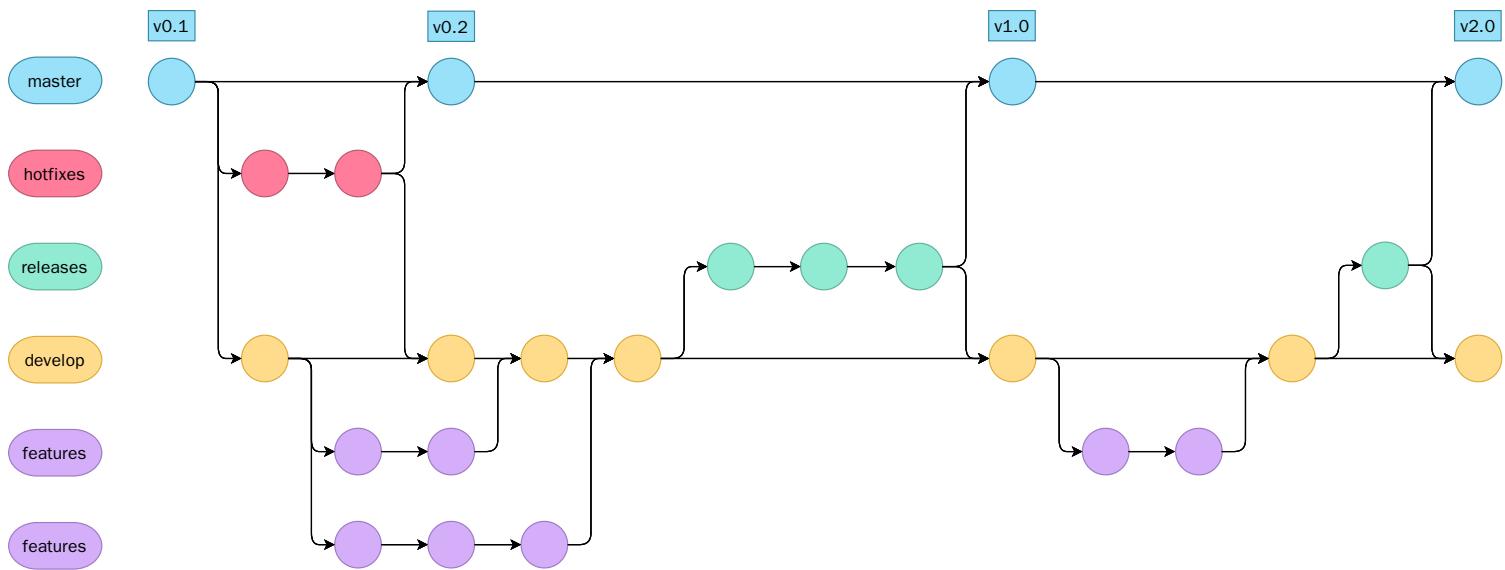


Zgodnie z założeniami Git Flow należy wydzielić gałąź **develop**, która będzie służyła jako baza dla gałęzi typu **feature**.



Przed wydaniem produktu należy utworzyć gałąź `release` gdzie produkt jest ostatecznie przygotowywany do wydania.

Proponowane nazwy: `rc-<release_version>`.



W przypadku konieczności wprowadzenia nagłych i pilnych poprawek do wydań należy skorzystać z gałęzi **hotfix**.

Proponowane nazwy: **hotfix-<task\_id>**.

**Tagowanie** - etykietowanie istotnych miejsc w historii.

```
$ git tag -a v1.0.0 -m "Releasing v1.0.0"  
$ git tag  
$ git push <remote_name> <tag_name>
```

Polecenia git:

- `tag` - lista etykiet,
- `tag <name>` - stworzenie etykiety.

- Scott Chacon, Ben Straub, *Pro Git Book*, <https://git-scm.com/book/pl/v2/>
- Atlassian, *Become a git guru*, <https://www.atlassian.com/git/tutorials>