JAVA SE 8 PROGRAMMER II

BARTOSZ ANDREATTO

Programista w Banku Pekao S.A z bogatym doświadczeniem w zakresie technologii back-end'owych wykorzystujących wirtualną maszynę Javy. Pasjonat rozwiązań opartych na ekosystemie Spring oraz rozwiązań Oracle. Certyfikowany programista Java. Uwielbia nauczać oraz dzielić się wiedzą.

LinkedIn: https://www.linkedin.com/in/bartosz-andreatto-02b03413a/

e-mail: bandreatto@gmail.com

ZAKRES SZKOLENIA

- Budowa klas w Javie
- Zaawansowane aspekty budowy klas w Javie
- Kolekcje i typy generyczne
- Interfejsy funkcyjne wyrażeń lambda
- Java Stream API
- Wyjątki i asercje
- API dotyczące daty / czasu w Javie SE 8
- Podstawy Java IO
- Plikowe I/O w Javie (NIO.2)
- Wielowątkowość
- JDBC
- Lokalizacja

ZAKRES SZKOLENIA

- Java SE 8 Programmer I Exam Number: 1Z0-808

  https://education.oracle.com/java-se-8-programmer-i/pexam_1Z0-808

  - liczba pytań: **56**

  - czas trwania: 120 minut

  - pytania wielokrotnego wyboru

  - próg: 65%

- Java SE 8 Programmer II Exam Number: 1Z0-809

  https://education.oracle.com/java-se-8-programmer-i/pexam_1Z0-808

  - liczba pytań: **68**

  - czas trwania: 120 minut

  - pytania wielokrotnego wyboru

  - próg: 65%

ORGANIZACYJNIE

- Repozytorium GIT:

  https://github.com/bandreatto/1z0_808_9/tree/slupsk_2022_12

- 28 (**7 x 4**) godzin lekcyjnych
- Przerwy:

  - wtorek: **13:30 – 13:50** (obiad), **15:20 – 15:30**

  - środa – piątek: **09:30 – 09:40**, **11:10 – 11:20**,

    12:50 – 13:20 (obiad)

PRZYGOTOWANIE

- https://enthuware.com/java-certification-mock-exams/oracle-certified-professional/java-se-8-1z0-809
- https://www.whizlabs.com/ocpjp-scjp/
- https://www.coderanch.com/wiki/659977/Ocpjp-Faq
- OCA / OCP Java SE 8 Programmer Certification Kit https://www.amazon.pl/OCA-OCP-Java-Programmer-Certification/dp/1119272092
- OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: https://www.amazon.pl/OCA-Certified-Associate-Programmer-1Z0-808/dp/1118957407
- OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide https://www.amazon.pl/OCP-Certified-Professional-Programmer-1Z0-809/dp/1119067901/
- OCA / OCP Java SE 8 Programmer Practice Tests: Exam 1Z0-808 and Exam 1Z0-809 https://www.amazon.pl/OCA-Java-Programmer-Practice-Tests/dp/111936339X/

ZAKRES SZKOLENIA – DZIEŃ 1

- Budowa klas w Javie

  enkapsulacja; dziedziczenie; modyfikatory dostępu; polimorfizm; metody: toString, equals, hashCode; wzorzec singleton; obiekty immutable; byty statyczne; bloki inicjalizacyjne

- Zaawansowane aspekty budowy klas w Javie

  klasy i metody abstrakcyjne; słowo kluczowe final; klasy wewnętrzne; enumeracje; interfejsy: deklaracja, implementacja, dziedziczenie; adnotacja @Override; tworzenie i użycie wyrażeń lambda

- Kolekcje i typy generyczne

  klasy generyczne; typy: ArrayList, TreeSet, TreeMap, ArrayDeque; interfejsy: Comparator, Comparable; przetwarzanie kolekcji za pomocą strumieni i filtrów; metoda forEach; interfejs Stream; filtrowanie kolekcji z użyciem wyrażeń lambda; referencje do metod

BUDOWA KLAS W JAVIE

Klasy w programowaniu obiektowym służą do opisywania otaczających nas przedmiotów, zdarzeń, czynności, stanów oraz relacji między opisywanymi przedmiotami. Takie typy zdefiniowane za pomocą klas nazywamy **typami złożonymi** lub też **typami referencyjnymi**.

Klasy posiadają dwie podstawowe składowe:

- **pola** – zmienne lub stałe opisujące obiekt danej klasy
- **metody** – operacje, które udostępnia nasza klasa

OOP

Koncepcja **O**bject **O**riented **P**rogramming (OOP) powiązana jest z językami programowania, które korzystają z obiektów. OOP wprowadza i łączy koncepcje, takie jak:

- **dziedziczenie**
- **polimorfizm**
- **enkapsulacja**
- **abstrakcja**
- **kompozycja**

ENKAPSULACJA

Enkapsulacja jest mechanizmem, który umożliwia **ukrywanie danych i metod przed „światem zewnętrznym”**. Enkapsulacja umożliwia udostępnianie tylko tych mechanizmów i danych, które mają być widoczne z zewnątrz klasy. Ukrywanie danych jest realizowane poprzez **modyfikatory dostępu**.

W przykładzie dostęp do danych o pracowniku **został ukryty** dla pól id oraz dateOfBirth. Został ograniczony tylko i wyłącznie do odczytu poprzez metody typu getter.

```java
public class Employee {
    private final String id;
    private final LocalDate dateOfBirth;
    private String firstName;
    private String lastName;

    public Employee(String id, String firstName, String lastName,
                    LocalDate dateOfBirth) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getId() {
        return id;
    }

    public LocalDate getDateOfBirth() {
        return dateOfBirth;
    }
}
```

ENKAPSULACJA

UWAGA: Zawsze powinniśmy chować szczegóły implementacyjne danej klasy przed użytkownikiem końcowym klasy, wykorzystując enkapsulację.

MODYFIKATORY DOSTĘPU

Can access	If that member is `private`?	If that member has default (package private) access?	If that member is `protected`?	If that member is `public`?
Member in the same class	yes	yes	yes	yes
Member in another class in the same package	no	yes	yes	yes
Member in a superclass in a different package	no	no	yes	yes
Method/field in a class (that is not a superclass) in a different package	no	no	no	yes

KOMPOZYCJA

Implementowanie **re-używalnych komponentów**, które skupia się **na składaniu obiektów** nazywa się **kompozycja**. Klasa **agreguje** w sobie składniki innych klas.

```java
public class Computer {
    private Processor processor;
    private Ram ram;

    public Computer(Processor processor, Ram ram) {
        this.processor = processor;
        this.ram = ram;
    }

    public void run() {
        // usage of processor and ram object
    }
}

class Processor {
    private String name;
    private int numberOfCores;

    public Processor(String name, int numberOfCores) {
        this.name = name;
        this.numberOfCores = numberOfCores;
    }
}

class Ram {
    private String name;
    private int size;

    public Ram(String name, int size) {
        this.name = name;
        this.size = size;
    }
}
```

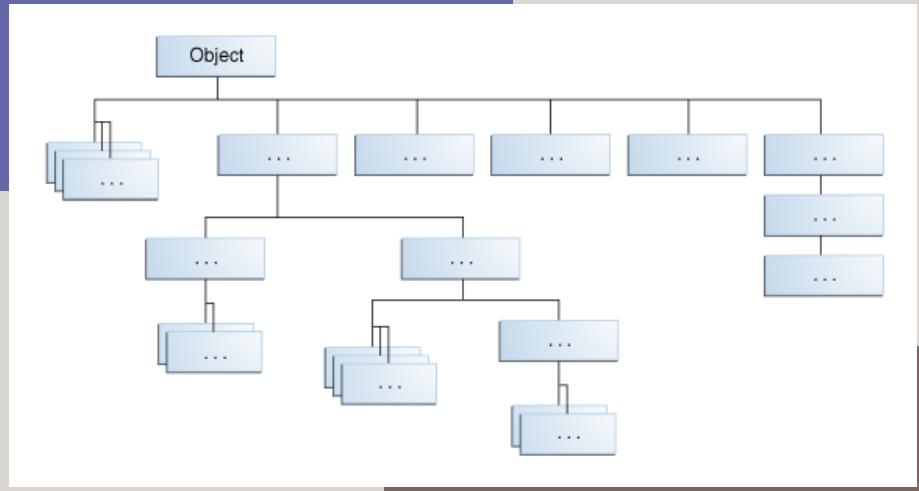
DZIEDZICZENIE

Jednym z najważniejszych **filarów OOP** jest **dziedziczenie klas**. Mechanizm ten umożliwia **współdzielenie zachowań pomiędzy klasami**. Klasa może dziedziczyć po innej klasie, co oznacza, że oprócz dotychczasowej funkcjonalności, może definiować nowe zachowania i właściwości. Dziedziczenie w Javie jest realizowane w oparciu o słówko kluczowe **extends**, które umieszczamy po nazwie deklarowanej klasy.

```java
public class Car {
    public void turnOnEngine(){
        System.out.println("Turn on engine!");
    }
}

public class SportCar extends Car {
    // klasa SportCar dziedziczy po klasie Car
    public void drive() {
        turnOnEngine();
        System.out.println("I'm driving!");
    }
}
```

HIERARCHICZNOŚĆ DZIEDZICZENIA

W Javie **wszystkie klasy dziedziczą po klasie Object**, pochodzącej z pakietu **java.lang**. Zawiera ona wspólną logikę i implementację **dla każdej klasy pochodnej**, która istnieje w ramach platformy, bądź będzie dopiero powstawać.



HIERARCHICZNOŚĆ DZIEDZICZENIA

Właściwości klasy pochodnej:

- dziedziczy **wszystkie metody i pola klasy bazowej**
- może rozszerzać **tylko co najwyżej jedną klasę**
- ma dostęp do wszystkich pól i metod, które zostały zdefiniowane **z wykorzystaniem modyfikatorów dostępu public i protected**
- ma dostęp do elementów z domyślnym modyfikatorem dostępu (**package-private**), jeżeli klasa bazowa znajduje się w tym samym pakiecie

HIERARCHICZNOŚĆ DZIEDZICZENIA

Klasa taka może bezpośrednio korzystać z możliwości klasy bazowej, tzn. bez żadnych odwołań **może np. wywoływać metody, do których ma dostęp**. Ponadto nadal może „zachowywać" się, jak zwykły obiekt, tzn. **może definiować nowe metody, pola czy stałe**.

Oprócz tego ma możliwość **nadpisania lub rozszerzenia** metody z klasy bazowej, wykorzystując **odpowiednie adnotacje**. W celu rozszerzenia możliwości konstruktora lub metody bazowej, musimy wykorzystać słowo kluczowe **super**, które umożliwia wywołanie logiki z klasy bazowej.

HIERARCHICZNOŚĆ DZIEDZICZENIA

```java
public class Computer {
    private String cpu;
    private String ram;
    private String gpu;

    public Computer(String cpu, String ram, String gpu) {
        this.cpu = cpu;
        this.ram = ram;
        this.gpu = gpu;
    }

    public void configure() {
        System.out.println("Booting ... ");
        System.out.println("Configure cpu: " + cpu);
        System.out.println("Configure ram: " + ram);
        System.out.println("Configure gpu: " + gpu);
    }
}

public class Laptop extends Computer {
    // Computer jest klasą bazową klasy Laptop
    private int battery;

    public Laptop(String cpu, String ram, String gpu, int battery) {
        // w przypadku klasy pochodnej, wywołanie domyślnego konstruktora
        // klasy bazowej jest wymagane
        super(cpu, ram, gpu);
        this.battery = battery;
    }

    @Override // (1)
    public void configure() {
        // wywołujemy funkcjonalność z klasy bazowej
        super.configure();
        System.out.println("Configure battery: " + battery);
    }
}
```

KOMPOZYCJA A DZIEDZICZENIE

Dziedziczenie klas **definiowane jest statycznie**, co skutkuje tym, że nie ma **możliwości zmiany implementacji w czasie wykonywania aplikacji**. Implementacja podklas zależna jest od implementacji klasy bazowej, więc zmiany w klasie bazowej często wymuszają również zmiany w podklasach. **Rozbudowane hierarchie dziedziczenia wpływają także negatywnie na testowanie kodu** oraz analizę struktury implementacyjnej. Dzięki wykorzystaniu kompozycji możemy dynamicznie zmieniać implementację (m.in. dzięki poliformizmie), bez potrzeby modyfikacji komponentów zagregowanych i na odwrót.

POLIMORFIZM

Mechanizm nazywany inaczej **wielopostaciowością**. Polega on głównie na tym, że programista korzystający z jakiegoś obiektu, **nie musi wiedzieć, czy jego funkcjonalność pochodzi z klasy będącej typem danego obiektu, czy jakiejś innej klasy, która implementuje lub dziedziczy po interfejsie lub klasie bazowej**. Innymi słowy, jest to zdolność obiektów do zwracania różnych odpowiedzi na to samo żądanie.

Polimorfizm jest bardzo często wykorzystywany **podczas pracy z kolekcjami**

```java
public class MainClass {
    public static void main(String[] args) {
        List<Integer> ints = new ArrayList<>(); // programista pracuje na
        // INTERFEJSIE, wykorzystując pewną implementację
        ints.add(1);
        ints.add(7);
        for (int i = 0; i < ints.size(); i++) {
            System.out.println(ints.get(i));
        }
    }
}
```

POLIMORFIZM

W kolejnym przykładzie definiujemy pewną hierarchię - **VodPlayer** i jej trzy klasy pochodne, **NetflixPlayer**, **HBOGoPlayer** i **DefaultPlayer**. W klasie **Android TV** wykorzystujemy polimorfizm pracując na abstrakcji, której implementacja jest wybierana na podstawie argumentów wejściowych aplikacji.

```java
public class VodPlayer {
    public void play(String title) {
        System.out.println("Playing " + title);
    }
}

public class NetflixPlayer extends VodPlayer {
    @Override
    public void play(final String title) {
        System.out.println("Playing " + title + " on Netflix");
    }
}

public class HBOGoPlayer extends VodPlayer {
    @Override
    public void play(final String title) {
        System.out.println("Playing " + title + " on HBO");
    }
}

public class DefaultPlayer extends VodPlayer {
    @Override
    public void play(final String title) {
        System.out.println("Playing " + title + " on default player");
    }
}

public class AndroidTV {
    public static void main(String[] args) {
        final String player = args[0];
        VodPlayer vodPlayer;
        if (player.equals("Netflix")) {
            vodPlayer = new NetflixPlayer();
        } else if (player.equals("HBO")) {
            vodPlayer = new HBOGoPlayer();
        } else {
            vodPlayer = new DefaultPlayer();
        }
        playEpisode(vodPlayer, "GOT_S1E1");
    }

    static void playEpisode(VodPlayer vodPlayer, String title) {
        // nie wiemy z jaką implementacją mamy do czynienia
        vodPlayer.play(title);
    }
}
```

INSTANCEOF

Jeżeli chcielibyśmy **sprawdzić, czy instancja danej klasy jest danego typu**, a następnie go wykorzystać możemy zrobić to w dwóch krokach, wykorzystując operator **instanceof**.

```
class HeavyAnimal { }
class Hippo extends HeavyAnimal { }
class Elephant extends HeavyAnimal { }
```

```
HeavyAnimal hippo = new Hippo();
boolean b1 = hippo instanceof Hippo;          // true
boolean b2 = hippo instanceof HeavyAnimal;    // true
boolean b3 = hippo instanceof Elephant;       // false
```

```
HeavyAnimal hippo = new Hippo();
boolean b4 = hippo instanceof Object;      // true
Hippo nullHippo = null;
boolean b5 = nullHippo instanceof Object;  // false
```

```
Hippo anotherHippo = new Hippo();
boolean b5 = anotherHippo instanceof Elephant; // DOES NOT COMPILE
```

```
public void feedAnimal(Animal animal) {
   if(animal instanceof Cow) {
      ((Cow)animal).addHay();
   } else if(animal instanceof Bird) {
      ((Bird)animal).addSeed();
   } else if(animal instanceof Lion) {
       ((Lion)animal).addMeat();
   } else {
      throw new RuntimeException("Unsupported animal");
   } }
```

BLOKI INICJALIZACYJNE

Bloki inicjalizacyjne występują w dwu rodzajach: jako **bloki instancyjne** i **bloki statyczne**. Blok inicjalizacyjny to fragment kodu, który wykonuje się – w przypadku **bloków statycznych** – **gdy ładowana jest klasa**, bądź – w przypadku **bloków instancyjnych** – gdy **tworzona jest instancja**.

```
public class Test {
    public Test() {
        System.out.println("constructor");
    }

    static {
        System.out.println("static init block");
    }

    {
        System.out.println("instance init block");
    }

    public static void main(String[] args) {
        System.out.println("start main");

        Test test = new Test();

        System.out.println("end main");
    }
}
```

BLOKI INICJALIZACYJNE

```
public class Counter {
        private static int count;
        static {
                // code in this static block will be executed when JVM loads the class into memory
                count = 1;
        }
        public Counter() {
                count++;
        }
        public static void printCount() {
                System.out.println("Number of instances created so far is: " + count);
        }
        public static void main(String []args) {
                Counter anInstance = new Counter();
                Counter.printCount();
                Counter anotherInstance = new Counter();
                Counter.printCount();
        }
}
```

ZAAWANSOWANE ASPEKTY BUDOWY KLAS W JAVIE

DEFINICJE

Klasa abstrakcyjna jest definicją klasy, której instancji (podobnie jak interfejsu) **nie możemy stworzyć**. Aby zadeklarować taką klasę, pomiędzy modyfikatorem dostępu a nazwą klasy, powinniśmy wykorzystać słowo kluczowe **abstract**. W przeciwieństwie do zwykłych klas, **klasa abstrakcyjna może dodatkowo posiadać metody abstrakcyjne.**

DEFINICJE

Metody abstrakcyjne to takie metody, które:

- **nie mogą posiadać ciała w definicji**
- ciało musi być zaimplementowane **w klasie, która po takiej klasie abstrakcyjnej dziedziczy**
- mogą być deklarowane **tylko i wyłącznie w klasach abstrakcyjnych oraz interfejsach**
- wymagają wykorzystania **słowa kluczowego abstract** przed typem zwracanym w jej sygnaturze

UWAGA: Metody abstrakcyjne nie mogą być deklarowane jako prywatne.

```
public abstract class Button { // klasa zdefiniowana jako abstrakcyjna

    public String getComponentName() { // zwykła, nieabstrakcyjna metoda
        return "Button";
    }

    public abstract void onClick(); // metoda abstrakcyjna, NIE posiada ciała
}
```

TWORZENIE INSTANCJI

Próba stworzenia instancji klasy zdefiniowanej na poprzednim slajdzie, kończy się **błędem kompilacji**:

```
public static void main(String[] args) {
  Button button = new Button(); // BŁĄD
}
```

Aby stworzyć instancję takiej klasy, **musimy najpierw ją rozszerzyć**, np.

```
public class SimpleButton extends Button {
  // dziedziczymy klasę abstrakcyjną tak, jak każdą inną klasę

  // KONIECZNIE musimy zaimplementować WSZYSTKIE metody abstrakcyjne
  @Override
  void onClick() {
    System.out.println("Simple Button was clicked");
  }
}
```

PORÓWNANIE Z INTERFEJSAMI

W odróżnieniu od interfejsów, **w klasach abstrakcyjnych możemy deklarować pola**. Ponadto, zdefiniowane w tej klasie metody, nie posiadają żadnych domyślnych słów kluczowych.

Jeśli chcemy współdzielić część kodu pomiędzy wieloma powiązanymi klasami, wtedy warto zastanowić się nad użyciem klasy abstrakcyjnej. Jeśli chcemy tylko wydzielić pewną abstrakcję, która może zostać wykorzystana w przypadku wielu niepowiązanych ze sobą klas, wtedy powinniśmy skorzystać z interfejsu.

PORÓWNANIE Z INTERFEJSAMI

Klasa abstrakcyjna może implementować interfejsy bez deklarowania ciał ich metod. Nie zmienia to faktu, że tworząc klasę pochodną, należy nadać ciało wszystkim metodom abstrakcyjnym, bez względu na to, czy są one przynależne do interfejsu, czy do klasy abstrakcyjnej.

```java
public interface ComponentClickListener {

    void onClick();
}

public abstract class AbstractButton implements ComponentClickListener {
    public static final String TAG = "Button";

    String componentName;

    String getComponentName() {
        return componentName;
    }

    abstract void click();
}

public class ButtonComponent extends AbstractButton {
  @Override
  void click() {
    // implementacja wymagana -> metoda abstrakcyjna klasy abstrakcyjnej
    System.out.println("I just clicked a button! Amazing");
  }

  @Override
  public void onClick() {
    // implementacja wymagana -> metoda abstrakcyjna interfejsu ComponentClickListener
    System.out.println("I am an onClick handler");
  }
}
```

PORÓWNANIE Z INTERFEJSAMI

Ponadto klasy abstrakcyjne **mogą dziedziczyć inne klasy abstrakcyjne**. Definiując klasę abstrakcyjną, która dziedziczy po klasie abstrakcyjnej, **możemy, ale nie musimy implementować dziedziczone metody abstrakcyjne**. Poniższy przykład pokazuje taką definicję:

```java
public abstract class TestTemplate {
  abstract public void run();
  protected abstract int getNumberOfIterations();
}

public abstract class PerformanceTestTemplate extends TestTemplate {
  // klasa abstrakcyjna dziedzicząca po klasie abstrakcyjnej
  @Override
  public void run() { // opcjonalnie decydujemy się na definicję metody abstrakcyjnej
    System.out.println("I should run test here");
  }
  // w tej klasie NIE implementujemy getNumberOfIterations (ale możemy to zrobić)

  public abstract double getAverageExecutionTime();
}

public class SortListPerformanceTest extends PerformanceTestTemplate {
  // definicja klasy nieabstrakcyjnej

  @Override // wymagane - metoda abstrakcyjna nieposiadająca implementacji w hierarchii dziedziczenia
  public double getAverageExecutionTime() {
    // calculate value based on real scenario
    return 5.0;
  }

  @Override // wymagane - metoda abstrakcyjna nieposiadająca implementacji w hierarchii dziedziczenia
  protected int getNumberOfIterations() {
    return 3;
  }
}
```

INTERFEJS

Interfejs jest, podobnie jak klasy, typem referencyjnym.

Może zawierać tylko:

- **stałe**
- **sygnatury metod**
- **metody typu default**
- **metody statyczne**
- **definicje typów zagnieżdżonych**

Nie można stworzyć instancji interfejsu. Interfejsy mogą być:

- implementowane przez klasy (keyword: **implements**)
- rozszerzane przez inne interfejsy (keyword: **extends**)

DEFINIOWANIE INTERFEJSU

Podczas deklarowania interfejsu definiuje się:

- **modyfikator dostępu**
- słowo kluczowe **interface**
- **nazwę interfejsu**
- opcjonalnie, **listę interfejsów** oddzielonych przecinkami
- **ciało interfejsu**, które może być puste

```
public interface MediaPlayer {
    void stop();
    void play();
}
```

CIAŁO INTERFEJSU

Ciało interfejsu może zawierać tylko metody typu:

- **abstract**
- **default**
- **static**

Pewne słowa kluczowe są domyślne:

- metody niestatyczne, w których nie zdefiniowaliśmy ciała, **są domyślnie abstrakcyjne i publiczne**, tzn. muszą zostać zaimplementowane w implementacjach
- stałe są domyślnie publiczne, finalne i statyczne, tzn. są poprzedzone słowami kluczowymi **public static final**
- metody **statyczne są domyślnie publiczne**, tzn. są poprzedzone słowem kluczowym public

```java
public interface MediaPlayer {

    String TAG = "MediaPlayer";
    // domyślnie zawiera słowa kluczowe - public static final

    void stop();
    // metoda bez zdefiniowanego ciała - domyślnie abstrakcyjna

    default void next() {
        // metoda z domyślnym ciałem, można, ale nie trzeba, nadpisywać w implementacji interfejsu
        throw new NoSuchMechanismException("not supported by default");
    }

    static String getName() {
        // metoda statyczna, domyślnie posiada modyfikator public
        return "MediaPlayer Interface";
    }
}
```

IMPLEMENTOWANIE INTERFEJSU

W celu **stworzenia implementacji interfejsu** należy użyć słowa kluczowego **implements** w deklaracji klasy i zaimplementować wszystkie metody abstrakcyjne danego interfejsu, np.:

```
// deklaracja interfejsu
public interface SomeInterface {
  void someMethod();
}

// deklaracja klasy implementującej interfejs
public class SomeInterfaceImpl implements SomeInterface {
  @Override
  public void someMethod() {
    System.out.println("methodImplementation");
  }
}

// stworzenie instancji klasy implementującej interfejs
SomeInterface someInterface = new SomeInterfaceImpl();
```

IMPLEMENTOWANIE INTERFEJSU

Klasa może **implementować więcej niż jeden interfejs** (oddzielając ich nazwy od siebie przecinkiem), np.:

```java
// deklaracja pierwszego interfejsu
public interface SomeInterface {
  void someMethod();
}

// deklaracje drugiego interfejsu, tym razem bez żadnych metod abstrakcyjnych
public interface SomeOtherInterface {
}

// definicja klasy implementującej oba powyższe interfejsy
public class ClassImplementingInterfaces implements SomeInterface, SomeOtherInterface {
  @Override
  public void someMethod() {
    System.out.println("I am interface method implementation");
  }
}
```

IMPLEMENTOWANIE INTERFEJSU

Pojedyncza klasa **oprócz implementowania wielu interfejsów**, może również rozszerzać **co najwyżej jedną klasę**, np.:

```java
// deklaracja pierwszego interfejsu
public interface SomeInterface {
  void someMethod();
}

// deklaracja drugiego interfejsu, tym razem bez żadnych metod abstrakcyjnych
public interface SomeOtherInterface {
}

// definicja klasy, która rozszerza klasę SomeClass i implementuje oba powyższe interfejsy
public class ClassImplementingInterfacesAndExtendingClass
    extends SomeClass implements SomeInterface, SomeOtherInterface {
  @Override
  public void someMethod() {
    System.out.println("I am interface method implementation");
  }
}
```

MODYFIKOWANIE INTERFEJSU

Wykonanie dowolnej modyfikacji na zaimplementowanym interfejsie, **będzie skutkować błędami na poziomie kompilacji, ze względu na brak implementacji nowego (zmodyfikowanego) interfejsu**.

UWAGA: Zmieniając definicję interfejsu, musimy pamiętać o koniecznych zmianach we wszystkich jego implementacjach.

UWAGA: Podobnych błędów kompilacji można uniknąć, wykorzystując opcję refactor w IntelliJ IDEA używając skrótu **Shift + F6**.

```java
public interface NewsletterAPI {

    void subscribe();

    void unsubscribe();
}
```

```java
public interface NewsletterAPI {

    void subscribe(String type);

    void unsubscribe();
}
```

```java
// Class must either be declared abstract or implement abstract method error
public class ShopNewsletterAPI implements NewsletterAPI {
    @Override
    public void subscribe() {
        System.out.println("Subscribe to shop newsletter!");
    }

    @Override
    public void unsubscribe() {
        System.out.println("Unsubscribe from shop newsletter!");
    }
}
```

METODY DOMYŚLNE

Metody domyślne zostały wprowadzone w wersji Javy 1.8. Ich główne zastosowanie to **wprowadzenie modyfikacji w istniejącym już interfejsie**, bez prowokowania błędów kompilacji związanych ze zmianą ciała interfejsu. Domyślne metody **pozwalają dodawać nowe funkcje do interfejsów bibliotek** i zapewniają zgodność binarną z kodem napisanym dla starszych wersji tych interfejsów. W przypadku implementacji interfejsów z metodami domyślnymi, nie ma potrzeby, by nadpisywać te metody, tak jak w poniższym przykładzie.

```
public interface Encoder {

    default String encode(String encodeText) {
        return encodeText;
    }

    default String decode(String decodeText) {
        return decodeText;
    }
}

public class SampleEncoder implements Encoder { }
```

KLASY WEWNĘTRZNE

W Javie **możliwe jest deklarowanie klas wewnątrz innych klas**. Klasy te nazywamy klasami zagnieżdżonymi. Mogą one być zdeklarowane jako:

- **klasy statyczne** (tzw. static nested), z wykorzystaniem słowa kluczowego static
- klasy niestatyczne (tzw. **non-static** lub **inner**)

UWAGA: Istnieją dwa specjalne rodzaje klas wewnętrznych: **klasy lokalne** i **klasy anonimowe**.

```
public class Outer {

    static class NestedStatic {
        // class body
    }

    class Inner {
        // class body
    }
}
```

KLASY WEWNĘTRZNE - WIDOCZNOŚĆ

Niestatyczne klasy zagnieżdżone mają bezpośredni dostęp do klasy nadrzędnej, tzn. z klasy nadrzędnej mogą wykorzystywać:

- **pola** (również statyczne i prywatne)
- **metody** (również statyczne i prywatne)

```java
public class OuterClass {

  private static int outerClassStaticField;
  private int outerClassField;

  void outerClassMethod() {
    System.out.println("I am outer class method");
  }

  public class InnerClass {
    void useOuterClassField() {
      System.out.println(outerClassStaticField); // wykorzystanie statycznego pola
      outerClassMethod();                        // wykorzystanie metody
      System.out.println(outerClassField);       // wykorzystanie prywatnego pola
    }
  }
}
```

KLASY WEWNĘTRZNE - WIDOCZNOŚĆ

UWAGA: W przeciwieństwie do klas lokalnych, klasy wewnętrzne **mogą wykorzystywać modyfikatory dostępu**.

UWAGA: Niestatyczne klasy lokalne **nie mogą definiować pól i metod statycznych**.

Z kolei zagnieżdżone klasy statyczne z klasy nadrzędnej:

- mogą korzystać ze statycznych pól i metod klasy nadrzędnej
- nie mogą korzystać z niestatycznych pól i metod

```java
public class OuterClass {

  private static int outerClassStaticField;
  private int outerClassField;

  void outerClassMethod() {
    System.out.println("I am outer class method");
  }

  protected static void outClassStaticMethod() {
    System.out.println("I am out class static method");
  }

  static class InnerStaticClass {

    void useOuterClassField() {
      System.out.println(outerClassStaticField);
      outClassStaticMethod();
      //outerClassMethod(); -> błąd kompilacji, NIE możemy korzystać z metod niestatycznych
    }
  }
}
```

KLASY WEWNĘTRZNE - KREACJA

Kolejne przykłady oprzemy o następujące klasy:

```
public class OuterClass {

  class InnerClass {
  }

  static class InnerStaticClass {
  }
}
```

Aby stworzyć **instancję wewnętrznej klasy niestatycznej**, musimy **najpierw stworzyć instancję klasy zewnętrznej**, np.:

```
OuterClass outerClass = new OuterClass();
final OuterClass.InnerClass innerClass = outerClass.new InnerClass();
```

```
OuterClass.InnerStaticClass innerStaticClass = new OuterClass.InnerStaticClass();
```

KLASY LOKALNE

Klasy lokalne są **klasami deklarowanymi w bloku kodu**, np. metodzie, pętli for, czy instrukcji if. Deklarując klasę lokalną **pomijamy informacje o modyfikatorze dostępu**. Obiekty klasy lokalnej mają dostęp do pól metod klas nadrzędnych. Kolejny przykład pokazuje wykorzystanie klasy lokalnej bezpośrednio w metodzie main.

```
public static void main(String[] args) {
  final List<String> names = List.of("Kasia", "Magda", "Gosia");
  final List<String> surnames = List.of("Piszczyk", "Olszańska", "Budrzeńska");
  final int someVariable = 3; // zmienna do zaprezentowania dostępu zmiennych z klas lokalnych

  class Name { // stworzenie definicji klasy lokalnej, bez modyfikatora dostępu
    private final String firstName;
    private final String lastName;

    public Name(final String firstName, final String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
    }

    public String getReadableName() {
      System.out.println("Hey I can use outer variable " + someVariable);
      return firstName + " " + lastName;
    }
  }

  for (int idx = 0; idx < names.size(); idx++) {
    final Name name = new Name(names.get(idx), surnames.get(idx)); // wykorzystanie klasy lokalnej
    System.out.println(name.getReadableName());
  }
}
```

KLASY LOKALNE

UWAGA: **W bloku kodu nie można deklarować interfejsów.**

Klasy lokalne nie są jednak tak elastyczne jak "zwykłe" klasy. Otóż:

- nie mogą definiować metod statycznych
- nie mogą zawierać pól statycznych
- ale mogą zawierać statyczne stałe, tzn. takie z modyfikatorami **static final**

```
class InnerClassExample {
  private static final String APP_NAME = "DummyApp"; // OK
  // Błąd kompilacji, brak modyfikatora final
  private static String INCORRECT_FIELD = "IAmMissingFinal";

  public void printAppName() {
    System.out.println(APP_NAME);
  }

  // Błąd kompilacji, metoda statyczna w klasie lokalnej
  public static void shouldNotBeDeclaredHere() {}
}
```

KLASY ANONIMOWE

Klasy anonimowe **działają tak samo, jak klasy lokalne**.

Różnią się tylko tym, że klasy anonimowe:

- nie mają nazwy
- powinny być deklarowane jeśli potrzebujemy ich tylko jeden raz

KLASY ANONIMOWE - SKŁADNIA

Tworzenie obiektu klasy anonimowej wygląda niemal **identycznie jak w przypadku tworzenia zwykłego obiektu**. Różnica polega na tym, że podczas tworzenia obiektu, implementuje się wszystkie wymagane metody. Wyrażenie składa się z:

- **operatora new** lub zastosowania **lambdy** w przypadku interfejsu funkcyjnego
- **nazwy interfejsu**, który implementujemy lub **klasy abstrakcyjnej**, którą dziedziczymy
- **parametrów konstruktora** (w przypadku interfejsu korzystamy z pustego konstruktora)
- **ciała klasy/interfejsu**

KLASY ANONIMOWE - SKŁADNIA

Tworzenie obiektu klasy anonimowej wygląda niemal **identycznie jak w przypadku tworzenia zwykłego obiektu**. Różnica polega na tym, że podczas tworzenia obiektu, implementuje się wszystkie wymagane metody. Wyrażenie składa się z:

- **operatora new** lub zastosowania **lambdy** w przypadku interfejsu funkcyjnego
- **nazwy interfejsu**, który implementujemy lub **klasy abstrakcyjnej**, którą dziedziczymy
- **parametrów konstruktora** (w przypadku interfejsu korzystamy z pustego konstruktora)
- **ciała klasy/interfejsu**

KLASY ANONIMOWE - SKŁADNIA

W ramach klasy anonimowej możemy deklarować:

- pola (w tym pola statyczne)
- metody (w tym metody statyczne)
- Stałe

Natomiast nie można deklarować:

- konstruktorów
- Interfejsów
- bloków statycznej inicjalizacji

```java
public interface ClickListener {
  void onClick();
}

public class UIComponents {
    void showComponents() {
        // implementacja klasy anonimowej z wykorzystaniem słowa kluczowego new
        ClickListener buttonClick = new ClickListener() {
            @Override
            public void onClick() {
                System.out.println("On Button click!");
            }
        };
        // koniec implementacji klasy anonimowej

        buttonClick.onClick();

        // implementacja klasy anonimowej z wykorzystaniem lambdy
        // jest to możliwe, ponieważ ClickListener jest interfejsem funkcyjnym,
        // tzn. ma jedną metodę do zaimplementowania
        ClickListener checkboxClick  = () -> System.out.println("On Checkbox click!");
        checkboxClick.onClick();
    }
}
```

KLASY ANONIMOWE - SKŁADNIA

Kolejny przykład pokazuje **dodatkowe możliwości klasy anonimowej** zaimplementowanej z wykorzystaniem słowa kluczowego new:

```
public interface ClickListener {
  void onClick();
}

void showComponentsV2() {
    // implementacja klasy anonimowej
    ClickListener buttonClick = new ClickListener() {

      private String name; // pole w klasie anonimowej
      // pole statyczne w klasie anonimowej
      private static final String BUTTON_CLICK_MESSAGE ="On Button click!";

      public void sayHello() { // implementacja metody w klasie anonimowej
        System.out.println("I am new method in anonymous class");
      }

      @Override
      public void onClick() {
        sayHello();
        System.out.println(BUTTON_CLICK_MESSAGE);
      }
    };

  }
```

DOSTĘP DO KLAS NADRZĘDNYCH

Klasy anonimowe **mogą odwoływać się do pól klasy nadrzędnej i zmiennych lokalnych**, pod warunkiem, że są **finalne**. Podobnie jak w przypadku klas zagnieżdżonych zmienne, o nazwach takich samych jak pola klasy nadrzędnej, przysłaniają ich właściwości.

ENUMERACJE

Typy wyliczeniowe, zwane częściej **enumami**, to specjalne typy obiektów w języku Java. Ich główną charakterystyką jest z góry znana ilość instancji. W celu stworzenia typu wyliczeniowego powinniśmy postąpić podobnie jak w przypadku tworzenia zwykłej klasy, tzn. stworzyć plik o dowolnej nazwie z rozszerzeniem java, a następnie wewnątrz pliku zdefiniować **typ wyliczeniowy o takiej samej nazwie jak plik.** Zamiast słowa kluczowego class wykorzystujemy **słowo kluczowe enum**.

```
public enum ExampleEnum {
 // zawartość
}
```

DEFINIOWANIE TYPÓW WYLICZENIOWYCH

Typ wyliczeniowy może posiadać **dowolną ilość predefiniowanych wartości** (instancji). Definiujemy je na początku ciała enuma. Wartości te najczęściej piszemy wielkimi literami i oddzielamy od siebie przecinkami, np.:

```
public enum JsonSerializationStrategy {
  SNAKE_CASE,
  CAMEL_CASE,
  KEBAB_CASE
}
```

ENUMERACJE - POBIERANIE INSTANCJI

W celu odwołania się do konkretnej wartości wykorzystujemy notację z kropką, np.:

```
JsonSerializationStrategy strategy = JsonSerializationStrategy.SNAKE_CASE;
```

Nie jest z kolei możliwe stworzenie nowej instancji takiego obiektu **przy pomocy słowa kluczowej new**:

```
// niemożliwe dla typu wyliczeniowego
JsonSerializationStrategy jsonSerializationStrategy = new JsonSerializationStrategy();
```

ENUMERACJE - PORÓWNYWANIE

Instancje enumów są **tworzone na starcie aplikacji**, każda z nich tworzona jest dokładnie raz. Stąd też chcąc porównywać instancje enumów możemy to zrobić przy pomocy metody **equals** ale również **==**, np.:

```
JsonSerializationStrategy strategyA = JsonSerializationStrategy.CAMEL_CASE;
JsonSerializationStrategy strategyB = JsonSerializationStrategy.CAMEL_CASE;
System.out.println(strategyA == strategyB); // true
System.out.println(strategyA.equals(strategyB)); // true
```

ENUMERACJE - PODOBIEŃSTWA

Typ wyliczeniowy pod względem mechanizmów jest bardzo podobny do klasy. Oprócz listy możliwych wartości enum może:

- zdefiniować konstruktory
- zdefiniować pola
- zdefiniować metody
- implementować interfejsy
- wykorzystywać modyfikatory dostępu

Enum ma jednak pewne ograniczenia, w przeciwieństwie do klasy **nie może**:

- **dziedziczyć po klasie**
- **nie może być wykorzystany jako klasa**

ENUMERACJE - KONSTRUKTORY, POLA

Jeżeli typ wyliczeniowy, oprócz predefiniowanych wartości, definiuje jakiekolwiek dodatkowe elementy (metody czy konstruktory) to **po liście wartości zawsze stawiamy średnik**.

Każdy enum może definiować wiele konstruktorów i wszystkie z nich muszą być **prywatne** (a najlepiej nie wykorzystywać żadnego modyfikatora dostępu).
W przypadku gdy konstruktor posiada parametry, chcąc go wykorzystać, musimy przekazać wartości argumentów bezpośrednio przy predefiniowanej wartości typu wyliczeniowego.

```
public enum JsonSerializationStrategy {
  SNAKE_CASE("snake case"),
  CAMEL_CASE("camel case"),
  KEBAB_CASE("kebab case");

  private final String readableName;

  JsonSerializationStrategy(final String readableName) {
    this.readableName = readableName;
  }
}
```

ENUMERACJE - METODY, INTERFEJSY

Każdy typ wyliczeniowy może również **tak jak klasa**, **implementować interfejsy i definiować metody**:

```
JsonSerializationStrategy strategy = JsonSerializationStrategy.CAMEL_CASE;
System.out.println(strategy.getId() + " " + strategy.getReadableName());
```

```
public interface IdProvider {
  String getId();
}

public enum JsonSerializationStrategy implements IdProvider {
  SNAKE_CASE("snake case"),
  CAMEL_CASE("camel case", "1"),
  KEBAB_CASE("kebab case", "2");

  private final String readableName;
  private final String id;

  JsonSerializationStrategy(final String readableName) {
    this.readableName = readableName;
    this.id = "0";
  }

  JsonSerializationStrategy(final String readableName, final String id) {
    this.readableName = readableName;
    this.id = id;
  }

  public String getReadableName() {
    return readableName;
  }

  @Override
  public String getId() {
    return id;
  }
}
```

ENUMERACJE - DZIEDZICZENIE

Typy wyliczeniowe nie mogą dziedziczyć, tzn. poniższy kod nie skompiluje się poprawnie:

```
public class SomeBaseClass {
}

public enum EnumExample extends SomeBaseClass {
}
```

Również inne klasy nie mogą dziedziczyć po typie wyliczeniowym. Ponownie, **poniższy kod nie skompiluje się**:

```
public enum EnumExample {
}

public class SomeBaseClass extends EnumExample {
}
```

ENUMERACJE
- METODY WBUDOWANE

Każdy typ wyliczeniowy ma dostęp do metod statycznych:

- **values** - zwraca tablicę dostępnych wartości
- **valueOf** - zwraca predefiniowaną wartość na podstawie nazwy

Ponadto każda instancja enuma posiada metodę name(), która zwraca nazwę instancji jako String

```
public enum Difficulty {
  EASY,
  MEDIUM,
  HARD
}

Stream.of(Difficulty.values()).forEach(System.out::println); // EASY MEDIUM HARD
System.out.println(Difficulty.valueOf("EASY") == Difficulty.EASY); // true
System.out.println(Difficulty.MEDIUM.name()); // MEDIUM
```

WZORCE PROJEKTOWE

Wzorce projektowe są zestawem gotowych szkieletów rozwiązań, które powinniśmy wykorzystywać w naszych aplikacjach. Wyróżniamy wzorce:

- konstrukcyjne
- strukturalne
- czynnościowe (behawioralne)

WZORCE STRUKTURALNE

Wzorce konstrukcyjne są podgrupą, które opisują sposoby tworzenia obiektów. **Obiekty możemy tworzyć za pomocą konstruktorów**, ale ograniczając się tylko do tej metody, w przypadku bardziej skomplikowanych obiektów, nasz kod **może stać się bardzo nieczytelny**.

Wyróżniamy m.in.:

- **Singleton** - opisuje, w jaki sposób stworzyć pojedynczą instancję obiektu w aplikacji.
- **Builder** - wykorzystywany do tworzenia skomplikowanych obiektów.

SINGLETON

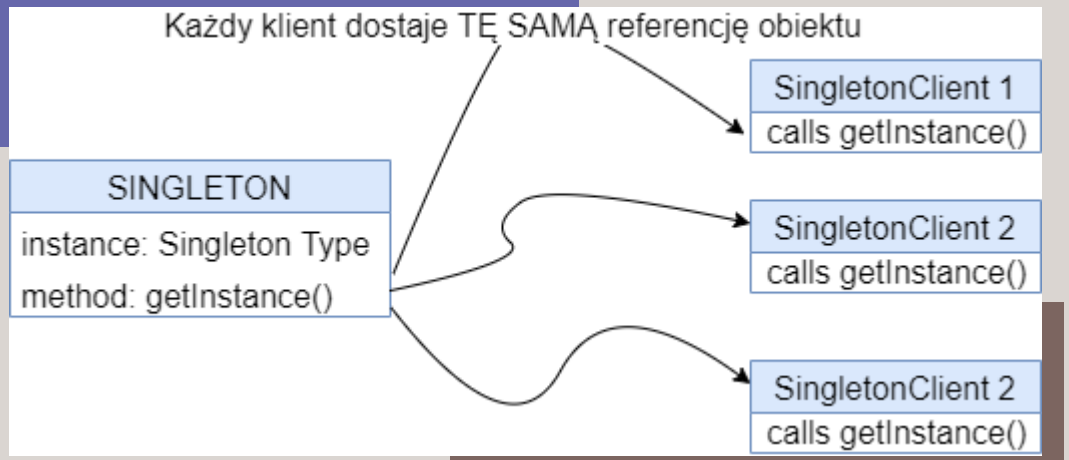
Singleton jest wzorcem projektowym, który opisuje w jaki sposób stworzyć obiekt, który w danej aplikacji **może być skonstruowany co najwyżej raz**.

Singletony możemy podzielić na dwie główne grupy:

- **lazy**
- **Eager**

Z kolei singleton typu lazy również możemy stworzyć na dwa sposoby:

- **lazy, który nie nadaje się do wykorzystania w aplikacjach wielowątkowych**
- **lazy double checked** , który można wykorzystać w aplikacjach wielowątkowych.



SINGLETON - EAGER

```java
public class SimpleCounter {

  // pole statyczne, w którym przetrzymujemy referencję singletonu
  // jest to singleton typu eager więc instancję tworzymy od razu,
  // przypisując ją do pola
  private static final SimpleCounter INSTANCE = new SimpleCounter();

  // getter dla referencji singletonu
  public static SimpleCounter getInstance() {
    return INSTANCE;
  }

  // ukryty konstruktor
  private SimpleCounter() {}

  private int currentCount = 0;

  public int getCurrentCount() {
    return currentCount;
  }

  public void increment() {
    currentCount++;
  }
}
```

SINGLETON - LAZY

```java
import java.util.ArrayList;
import java.util.List;

public class CommonStorage {
  private static CommonStorage instance;

  public static CommonStorage getInstance() {
    if (instance == null) { // (1)
      instance = new CommonStorage(); // (2)
    }
    return instance;
  }

  private List<Integer> values = new ArrayList<>();

  private CommonStorage() {
  }

  public void addValue(final int value) {
    values.add(value);
  }

  public List<Integer> getValues() {
    return values;
  }
}
```

SINGLETON
- DOUBLE CHECKED

```java
import java.util.ArrayList;
import java.util.List;

public class CommonStorage {
  private static CommonStorage instance;

  public static CommonStorage getInstance() {
    if (instance == null) { // (1)
      synchronized (CommonStorage.class) {
        if (instance == null) { // (2)
          instance = new CommonStorage();
        }
      }
    }
    return instance;
  }

  private List<Integer> values = new ArrayList<>();

  private CommonStorage() {
  }

  public void addValue(final int value) {
    values.add(value);
  }

  public List<Integer> getValues() {
    return values;
  }
}
```

IMMUTABLE PATTERN

Obiekt niezmienny (**immutable**) to taki, który po utworzeniu i inicjalizacji **pozostaje niezmienny** i co ważne, **nie ma możliwości jego zmiany** (oczywiście w konwencjonalny sposób). Czyli taki obiekt nie udostępnia metod, które pozwalają **na zmianę jego stanu**. A jego wszystkie pola są prywatne.

IMMUTABLE PATTERN

Wymagania:

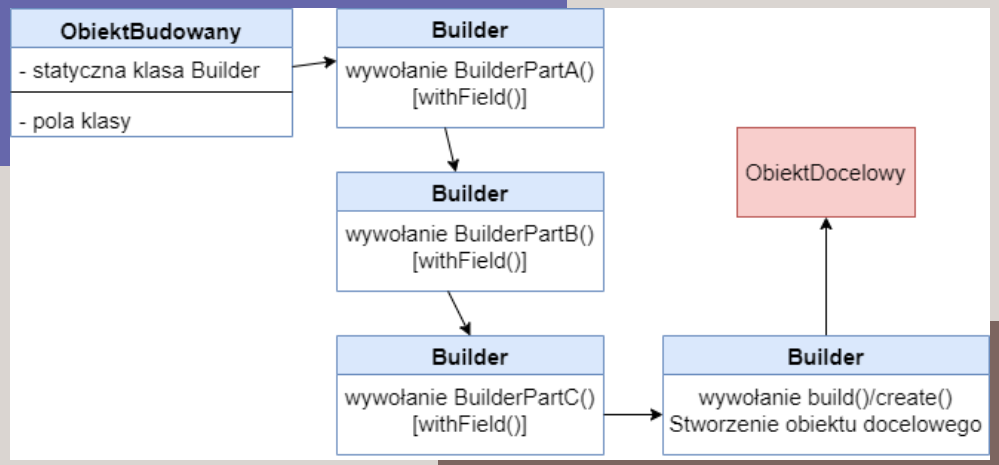
- Użycie konstruktora ustawiającego wszystkie wartości parametrów danej klasy (ewentualnie wykorzystanie wzorców konstrukcyjnych)
- Oznaczenie wszystkich pól klasy jako **private final**
- Brak metod typu seter
- Brak bezpośredniego dostępu do obiektów „mutowalnych”
- Zabezpieczenie przed możliwością nadpisania metody

```
public final class Animal {
   private final String species;
   private final int age;
   private final List<String> favoriteFoods;

   public Animal(String species, int age, List<String> favoriteFoods) {
      this.species = species;
      this.age = age;
      if(favoriteFoods == null) {
         throw new RuntimeException("favoriteFoods is required");
      }
      this.favoriteFoods = new ArrayList<String>(favoriteFoods);
   }

   public String getSpecies() {
      return species;
   }

   public int getAge() {
      return age;
   }

   public int getFavoriteFoodsCount() {
      return favoriteFoods.size();
   }

   public String getFavoriteFood(int index) {
      return favoriteFoods.get(index);
   }
}
```

BUILDER

Wzorzec **Builder** jest kolejnym sposobem tworzenia obiektów. **Wykorzystywany jest najczęściej do tworzenia obiektów, które składają się z wielu pól**. Obiekty, które składają się z wielu pól (np. 4 lub więcej) W celu stworzenia klasy będącej builderem dla pewnej klasy, musimy stworzyć:

- klasę statyczną w klasie
- lub osobną klasę



BUILDER - PRZYKŁAD

```java
import java.util.ArrayList;
import java.util.List;

public class Weapon {

  private String type;
  private String name;
  private Integer damage;
  private Long durability;
  private List<String> perks;

  private Weapon(final String type, final String name, final Integer damage,
      final Long durability, final List<String> perks) {
    this.type = type;
    this.name = name;
    this.damage = damage;
    this.durability = durability;
    this.perks = perks;
  }

  // getters + setters
}
```

```java
  public static class Builder {
    private String type;
    private String name;
    private Integer damage;
    private Long durability;
    private List<String> perks = new ArrayList<>();

    // metody budownicze
    public Builder withType(final String type) {
      this.type = type;
      return this;
    }

    public Builder withName(final String name) {
      this.name = name;
      return this;
    }

    public Builder withDamage(final Integer damage) {
      this.damage = damage;
      return this;
    }

    // ...

    // metoda budująca obiekt docelowy
    public Weapon build() {
      return new Weapon(type, name, damage, durability, perks);
    }
  }
}
```

KOLEKCJE I TYPY GENERYCZNE

TYPY GENERYCZNE

Typy generyczne są nazywane inaczej **szablonami**. Umożliwiają one parametryzowanie **klasy/metody/interfejsu**, które realizowane jest poprzez przekazanie typów argumentów w **momencie rzeczywistego wykorzystania ich w kodzie**. Wykorzystując taki mechanizm, możemy wielokrotnie wykorzystywać te same fragmenty kodu, ponieważ nie są na sztywno powiązane z żadną konkretną implementacją. **Dzięki typom generycznym dodatkowo jesteśmy w stanie pozbyć się zbędnego rzutowania**.

TYPY GENERYCZNE - PRZYKŁAD

W przypadku przedstawionego fragmentu kodu jesteśmy w stanie przekazać dowolny obiekt do obiektu klasy Box (ze względu na korzystanie z typu Object). Niemniej jednak, **nie jesteśmy w żaden sposób chronieni na poziomie kompilacji** przed omyłkowym przekazywaniem różnego typu obiektów do tego samego obiektu klasy Box, co **może prowadzić do błędów w trakcie działania aplikacji**. Dzięki możliwości wykorzystania typów generycznych, kompilator jest w stanie sprawdzić poprawność przekazywanych typów.

```
public class Box {

    private Object item;

    public Object getItem() {
        return item;
    }

    public void setItem(Object item) {
        this.item = item;
    }
}
```

```
public class Box<T> {
    private T item;

    public T getItem() {
        return item;
    }

    public void setItem(T item) {
        this.item = item;
    }
}
```

TYPY GENERYCZNE KONWENCJA NAZEWNICZA

Zgodnie z konwencją, **nazwy typów generycznych są najczęściej zapisywane za pomocą pojedynczej wielkiej litery**. Sugeruje się wykorzystywać poniższe formy nazewnicze:

- **E** - Element (używany m.in. dla Java Collection API)
- **K** – Klucz
- **N** – Liczba
- **T** – Typ
- **V** - Wartość

KLASY GENERYCZNE - KREACJA

W celu stworzenia obiektu, do którego musimy przekazać wartość typu generycznego, **musimy podczas jego tworzenia podać konkretny typ, który zastępuje parametr, np. T**:

```
new Box<Integer>();
```

Tak stworzony obiekt możemy przypisać do odpowiedniej referencji, która również powinna mieć informacje, jaki typ generyczny został wyspecyfikowany, tzn.:

```
Box<Integer> numberBox = new Box<Integer>(); // T zastąpione przez Integer
```

KLASY GENERYCZNE - KREACJA

W Javie, jeżeli referencja, do której przypisujemy tworzony obiekt generyczny, zawiera informacje o typie, **typu tego nie musimy powtarzać tworząc obiekt, ale wykorzystanie <> jest ciągle obowiązkowe**:

```
Box<Integer> numberBox = new Box<>(); // <> wymagane
```

Z kolei typ ten musimy wyspecyfikować podczas tworzenia obiektu, jeżeli korzystamy ze słowa kluczowego var, np.:

```
var intList = new ArrayList<Integer>();
```

ILOŚĆ TYPÓW GENERYCZNYCH

W ramach jednej klasy można zadeklarować wiele typów generycznych, które będą częścią klasy. Każdy parametr typu powinien być deklarowany jako unikalny znak zgodnie z konwencją.

Kolejny przykład definiuje parę dwóch generycznych obiektów. Zarówno pole **key** jak i pole **value** może **mieć dowolny typ**.

Tworzenie instancji w przypadku powyższej klasy będzie wyglądać następująco:

```
Pair<String, Float> pair = new Pair<String, Float>();
```

```
public class Pair<K, V> {
    private K key;
    private V value;

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }
}
```

ROZSZERZANIE KLASY GENERYCZNEJ

Klasę generyczną możemy bez problemu rozszerzyć. **Klasa dziedzicząca klasę generyczną musi wyspecyfikować typ generyczny lub dalej pozostać generyczna**. Poniższe definicje klas pokazują te możliwości:

```
public class BaseClass<T, V> {
}

// wyspecyfikowano typy generyczne
public class NoLongerGenericClass extends BaseClass<String, Integer> {
}

// wyspecyfikowano jeden typ generyczny - V.
// Klasa jest ciągle generyczna i wymaga wyspecyfikowania parametru T.
public class StillGenericClass<T> extends BaseClass<T, Integer> {
}
```

METODY GENERYCZNE

Nie tylko klasy mogą być generyczne. Również metody umożliwiają **deklarowanie własnych typów parametrycznych**. Widoczność tych typów jest ograniczona do konkretnej metody (sygnatury i ciała). Dozwolone są zarówno **statyczne, jak i niestatyczne metody generyczne**. Składnia metod generycznych rozszerza deklarację metod o typy parametryczne, umieszczone przed typem zwracanym. Podczas korzystania z takich metod, nie musimy, ale **możemy wyspecyfikować typy generyczne**. Robimy to za pomocą nawiasów <>, w których podajemy wartości typów generycznych.

```
public class PairGenerator {
  public static <K, V> Pair<K, V> generatePair(K key, V value) {
    Pair<K, V> pair = new Pair<K, V>();
    pair.setKey(key);
    pair.setValue(value);
    return pair;
  }

  public static void main(String[] args) {
    final Pair<Integer, String> firstPair = PairGenerator.generatePair(1, "value1");
    final Pair<Long, String> secondPair = PairGenerator
        .<Long, String>generatePair(2L, "value2");
  }
}
```

OGRANICZANIE TYPÓW PARAMETRÓW

Zazwyczaj chcemy, aby tworzone przez nas klasy generyczne, mogły korzystać jedynie z określonych wartości typów generycznych, np. takich, które dziedziczą po pewnej klasie lub implementują konkretny interfejs. W tym celu korzystamy z tzw**. bounded types parameters**. Deklaracja polega na określeniu typu parametrycznego, a następnie wykorzystaniu słowa kluczowego **extends** i zdefiniowaniu ograniczenia.

```
public class NumberBox<T extends Number> {

  private T value;

  public T getValue() {
    return value;
  }

  public void setValue(T value) {
    this.value = value;
  }

  public static void main(String[] args) {
    NumberBox<Double> doubleBox = new NumberBox<>();
    doubleBox.setValue(3.3);
    NumberBox<Integer> intBox = new NumberBox<>();
    intBox.setValue(10);
    System.out.println(intBox.getValue() + " " + doubleBox.getValue());
  }
}
```

OGRANICZANIE TYPÓW PARAMETRÓW

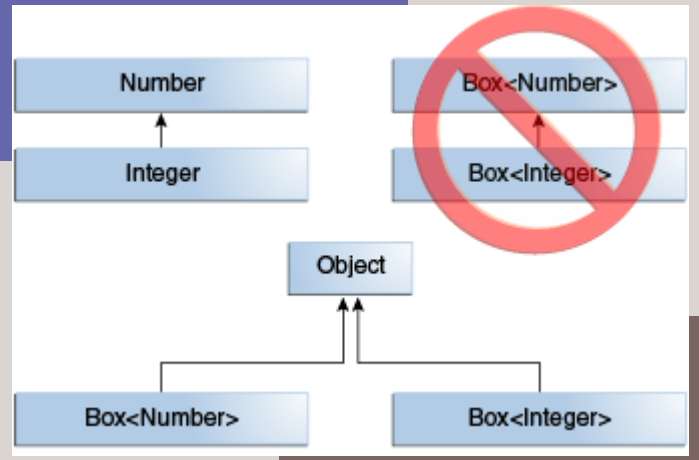
W przypadku **bounded types parameters** słowo kluczowe **extends** wykorzystujemy zarówno do klas i interfejsów. Ponadto możliwe jest wymuszenie, aby typ generyczny z którego chcemy korzystać implementował wiele interfejsów, np.:

```
public class NumberBox<T extends Number & Cloneable & Comparable<T>>
```

PODTYPY

W przypadku typów generycznych, **błędem jest traktowanie parametrów typów w ten sam sposób, co klasy generyczne**, np. jeśli **Integer jest podtypem Number**, to **nie znaczy to, że typ Box<Integer> jest podtypem Box<Number>**.

W celu uzyskania oczekiwanej relacji powinno się korzystać z tzw. Wildcards.



WILDCARDS

W przypadku typów generycznych, znak **? reprezentuje typ nieznany**. Wildcard może reprezentować:

- typ zmiennej
- typ pola klasy
- opcjonalnie typ zwracany

Nie może natomiast reprezentować:

- argumenntu metody generycznej
- argumentu klasy generycznej

OGRANICZENIE GÓRNE

Załóżmy, że chcemy stworzyć metodę, która będzie działać **dla list zawierających dowolne typy numeryczne** (tzn. obiekty **dziedziczące po klasie Number**). W tym celu możemy użyć tzw. **upper-bounded wildcard**, który reprezentowany jest przez znak **?** oraz słówko kluczowe **extends**, np.:

```java
public class UpperBoundedWildcards{

  // metoda akceptuje tylko typy rozszerzające klasę Number
  public static double sum(final List<? extends Number> numbers) {
    double sum = 0;
    for (Number number : numbers) {
      sum += number.doubleValue();
    }
    return sum;
  }

  public static void main(String[] args) {
    List<Integer> values = List.of(1, 2, 3);
    System.out.println(sum(values));
  }
}
```

OGRANICZENIE DOLNE

Załóżmy, że musimy napisać metodę, która jako argument przyjmie listę obiektów **typu Integer** lub tych, po których ta klasa dziedziczy (np. **Number lub Object**). W tym celu możemy użyć tzw. **lower-bounded wildcard**, który reprezentowany jest przez znak **?** oraz słówko kluczowe **super**, które uwzględnia wskazaną klasę oraz wszystkie klasy nadrzędne.

```
public class LowerBoundedWildcards {

  public static void main(String[] args) {
    addNumbers(List.of(1, 2, 3));
    addNumbers(List.of(new Object(), new Object(), new Object()));
  }

  public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
      list.add(i);
    }
  }
}
```

KOLEKCJE

Kolekcje, często nazywane również **kontenerami**, są obiektami, które agregują elementy. Kolekcje wykorzystywane są do **przechowywania obiektów**, uzyskiwania przechowywanych danych lub manipulowania danymi. W Javie wbudowane w język kolekcje opierają się o poniższe mechanizmy:

- **Interfejs**: abstrakcyjny typ danych reprezentujący kolekcje.
- **Implementacja**: konkretna realizacja interfejsów kolekcji.
- **Algorytm**: przydatne operacje na strukturach danych

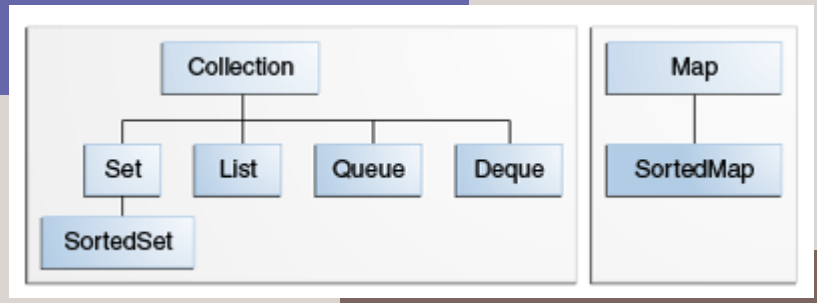
KOLEKCJE

Kolekcje w Javie dają programistom następujące korzyści:

- zwalniają programistę z **obowiązku implementacji struktur danych od zera**, co redukuje czas potrzebny na realizację konkretnej funkcjonalności,
- zaimplementowane struktury danych wykorzystują **najbardziej efektywne mechanizmy, a ich implementacja jest optymalna**,
- w przypadku projektowania własnych struktur danych, nie ma potrzeby „wymyślania koła na nowo”. **Można ponownie użyć już istniejące**.

INTERFEJSY

Interfejsy kolekcji reprezentują kontenery różnego typu. Interfejsy te umożliwiają manipulowanie kolekcjami bez wnikania w szczegóły implementacyjne. Rdzeniem wszystkich kolekcji jest interfejs generyczny: **public interface Collection<E>**



INTERFEJSY

Poniżej znajdują się wszystkie bazowe interfejsy **Collection API**:

- **Set** - kolekcja, która nie może przechowywać duplikatów.
- **List** - kolekcja uporządkowana, może zawierać duplikaty.
- **Queue** - kolekcja realizująca mechanizm FIFO (first in, first out) lub LIFO (last in, first out).
- **Deque** - rodzaj kolejki, gdzie można dodawać i usuwać elementy, zarówno z początku, jak i końca.
- **Map** - kolekcja służąca do przechowywania par klucz-wartość.

SET

java.util.Set jest generyczną strukturą danych odpowiedzialną za przechowywanie elementów unikalnych. Set korzysta tylko i wyłącznie z metod interfejsu Collection. Dodatkowo wprowadza silne połączenie pomiędzy metodami: **equals** i **hashCode**. Istnieją trzy podstawowe implementacje interfejsu Set: **HashSet**, **TreeSet**, **LinkedHashSet**.

SET

- **HashSet**

  - kolejność elementów nie jest zachowana

  - przechowuje informacje w tablicy hashującej

- **TreeSet**

  - kolejność elementów jest zachowana według tzw. kolejności naturalnej lub wg pewnego Comparatora

  - przechowuje dane w drzewie czerwono-czarnym

- **LinkedHashSet**

  - zachowuje informacje o kolejności dodawania poszczególnych elementów

  - implementacja jest oparta o tablicę hashującą wraz z obsługą linked list

```java
final Set<Integer> numbersSet = new HashSet<>(); // stworzenie instancji HashSet
System.out.println(numbersSet.isEmpty()); // true, Set nie zawiera elementów
numbersSet.add(1);
numbersSet.add(17);
numbersSet.add(3);
numbersSet.add(2);
numbersSet.add(1); // Dodanie elementu o wartości,
// który już istnieje - element NIE jest ponownie dodany
numbersSet.forEach(System.out::println);

/* przykładowa kolejność, w jakiej elementy mogą być wypisane:
 1 17 2 3
*/
```

```java
final Set<Integer> numbersSet = new TreeSet<>();
numbersSet.add(1);
numbersSet.add(3);
numbersSet.add(2);
numbersSet.add(1); // Dodanie elementu o wartości,
// który już istnieje - element NIE jest ponownie dodany
numbersSet.forEach(System.out::println);
/* Kolejność elementów ZAWSZE będzie taka sama
   (posortowana wg naturalnej kolejności):
 1 2 3
*/
```

```java
final Set<Integer> numbersSet = new LinkedHashSet<>();
numbersSet.add(1);
numbersSet.add(3);
numbersSet.add(2);
numbersSet.add(1); // Dodanie elementu o wartości,
// który już istnieje - element NIE jest ponownie dodany
numbersSet.forEach(System.out::println);
/* Kolejność elementów ZAWSZE będzie taka sama
   (wg kolejności dodawania elementów)
 1 3 2
*/
```

LIST

java.util.List jest interfejsem, który reprezentuje kolekcje uporządkowane. Charakteryzuje się tym, że:

- może zawierać **zduplikowane elementy** (tzn. o takiej samej wartości)
- element możemy **pobrać na podstawie jego pozycji** w liście (na podstawie indeksu)
- **element możemy wyszukać**.

Najczęściej wykorzystywanymi implementacjami interfejsu List są:

- **ArrayList**, która oparta jest o strukturę tablicową
- **LinkedList**, która zaimplementowana jest na zasadzie węzłów.

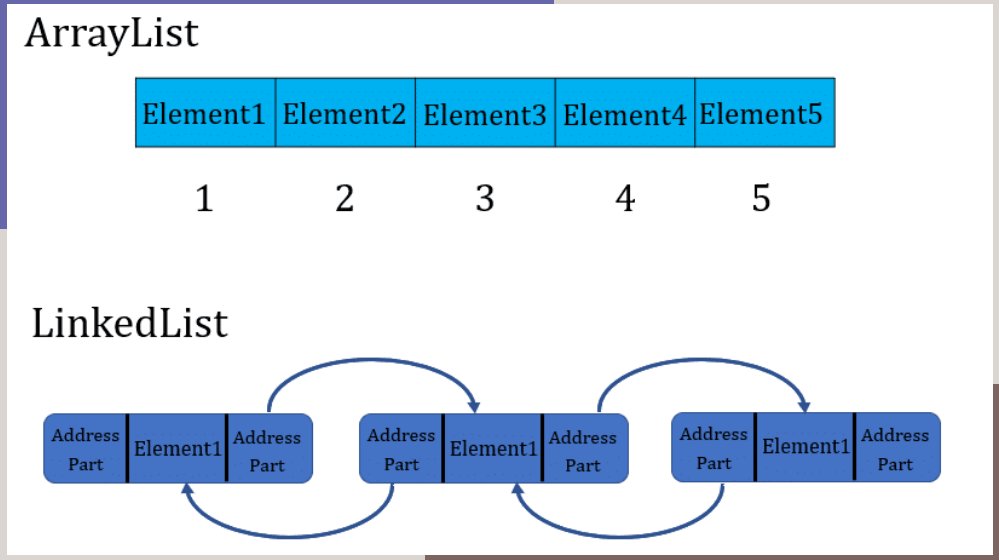
```
final List<String> names = new ArrayList<>();
names.add("Andrzej");  // dodanie elementu na koniec listy
names.add("Grzegorz"); // dodanie elementu na koniec listy
for (final String name: names) {
  System.out.println(name);
  // na ekran zostanie wypisane Andrzej, Grzegorz, z zachowaniem kolejności
}
```

```
final List<String> names = new LinkedList<>();
names.add(0, "Andrzej");  // dodanie elementu na początek listy
names.add(0, "Grzegorz"); // dodanie elementu na początek listy
for (final String name: names) {
  System.out.println(name);
  // na ekran zostanie wypisane Grzegorz, Andrzej, z zachowaniem kolejności
}
```

ARRAYLIST VS LINKEDLIST

Kiedy powinniśmy korzystać z **ArrayList**, a kiedy z **LinkedList**?

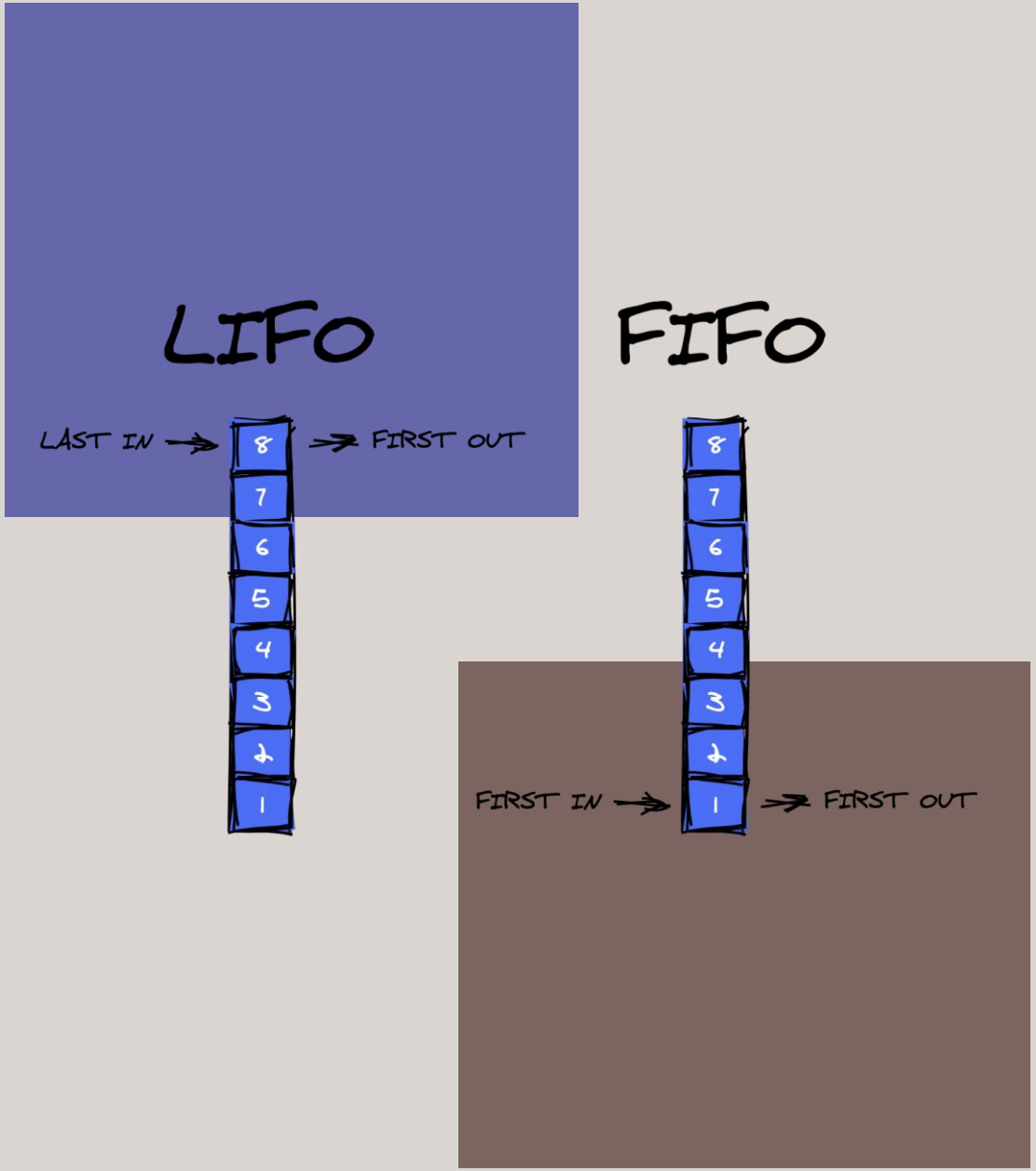
- pobieranie elementu na podstawie jego indeksu z **ArrayList** jest szybsze (**O(1)**) niż z **LinkedList** (**O(n)**)
- dodawanie elementu za pomocą metody **add(E someELement)** ma taką samą złożoność obliczeniową dla obu implementacji, ale w przypadku przepełnienia wewnętrznej tablicy w ArrayList, ta operacja jest wolniejsza (**O(n)**)
- dodawanie elementu na konkretny indeks, tzn. za pomocą metody **add(int index, E someElement)**, jest szybsze w przypadku LinkedList.



QUEUE

java.util.Queue jest generycznym interfejsem reprezentującym strukturę danych, umożliwiającym implementację kolejek **FIFO** i **LIFO**. Kolejki te działają analogicznie jak kolejki do kas w sklepie, tzn.:

- elementy dodawane trafiają na koniec kolejki
- możemy najpierw "obsłużyć" osobę z:
  - **początku kolejki (FIFO)**
  - **końca kolejki (LIFO)**



QUEUE

Kolejka, oprócz podstawowych operacji z interfejsu Collection, realizuje dodatkowe operacje manipulacji danymi:

- **element()** – wraca element z „czoła” kolejki (ale **nie usuwa go ze struktury**) lub wyrzuca wyjątek **NoSuchElementException** w przypadku pustej kolekcji
- **peek()** – działa identycznie, jak metoda element, ale **nie wyrzuca wyjątku w przypadku pustej kolejki**
- **offer()** – dodaje element do kolejki i zwraca informację, czy operacja się udała

QUEUE

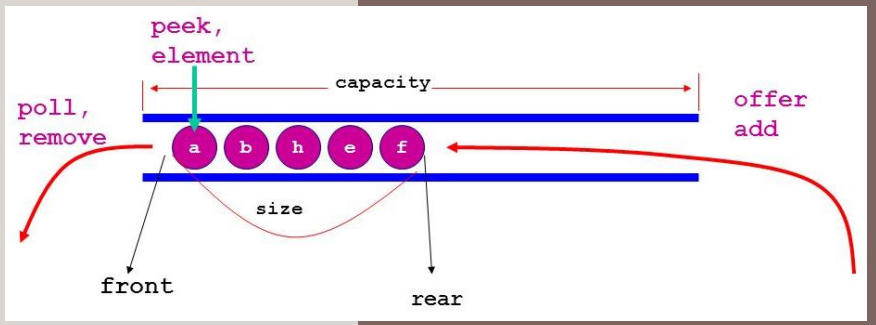
- **remove()** – **usuwa element z „czoła” kolejki** i zwraca jego wartość lub wyrzuca wyjątek **NoSuchElementException** w przypadku pustej kolekcji
- **poll()** - działa identycznie jak metoda **remove**, ale **nie wyrzuca wyjątku w przypadku pustej kolejki**

Podstawową **implementacją kolejki jest LinkedList**.

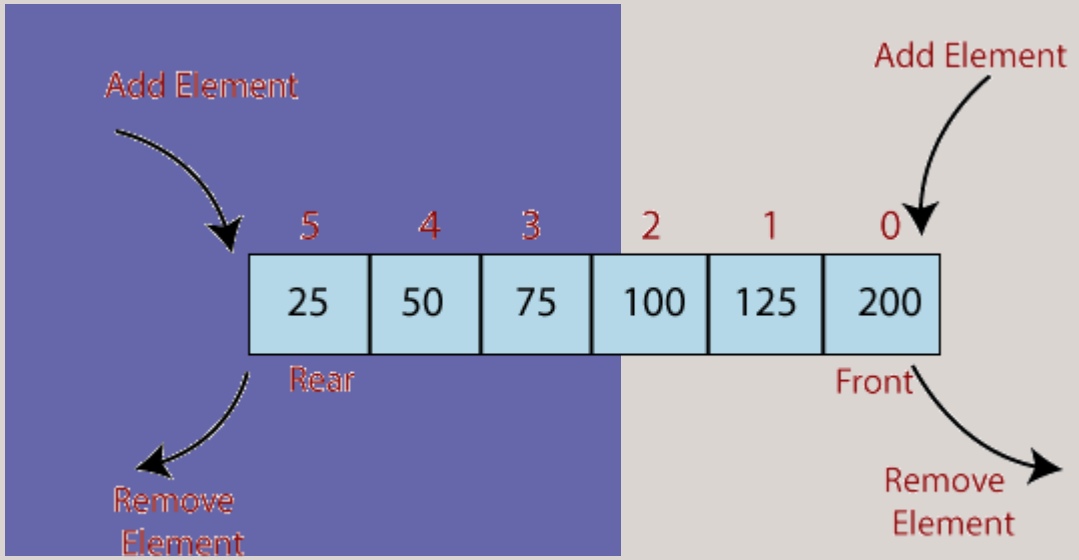
```java
final Queue<String> events = new LinkedList<>();
events.offer("ButtonClicked");
events.offer("MouseMoved");
// podejrzenie pierwszego elementu
System.out.println(events.peek());
// usunięcie pierwszego elementu z kolejki i zwrócenie wartości
System.out.println(events.poll());
// ponowne usunięcie pierwszego elementu z kolejki i zwrócenie wartości
System.out.println(events.poll());
// w tym momencie kolejka jest już pusta
System.out.println(events.isEmpty());
```

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

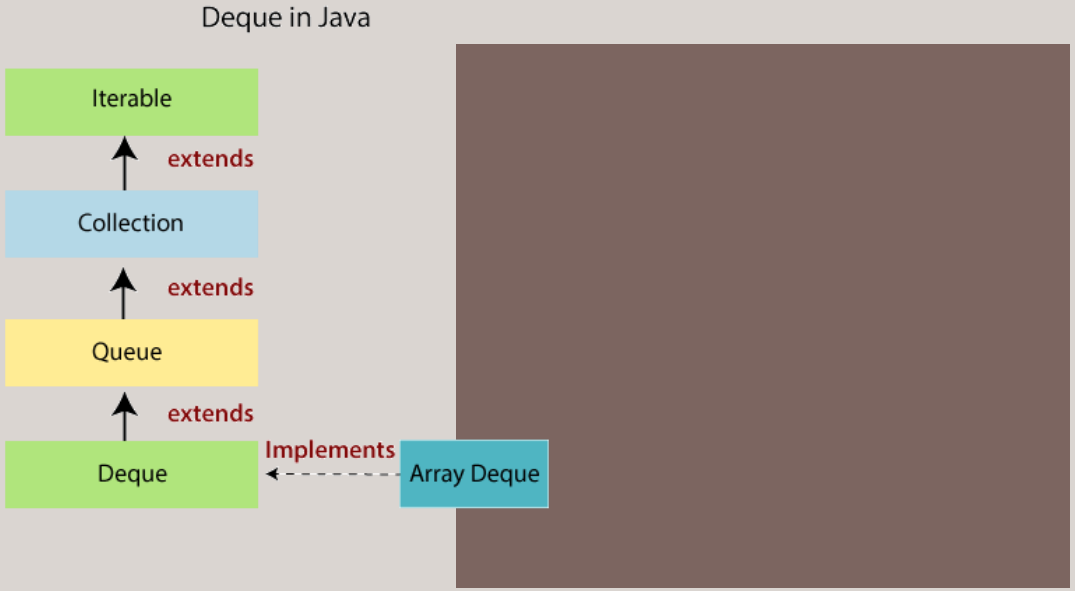


DEQUEUE



Deque, czyli **double-ended queue**, to interfejs opisujący strukturę danych będący rodzajem kolejki, która umożliwia dodawanie i usuwanie elementów, **zarówno z początku, jak i końca**.

Interfejs **java.util.Deque** rozszerza wcześniej omówiony interfejs java.util.Queue. Najczęściej wykorzystywanymi implementacjami Deque są **ArrayDeque** i **LinkedList**.



DEQUEUE

Deque dodatkowo daje nam dostęp do metod:

- **addFirst(e)** i **offerFirst(e)**, które dodają element na początek kolejki
- **addLast(e)** i **offerLast(e)**, które dodają element na koniec kolejki
- **removeFirst()** i **pollFirst()**, które usuwają element z początu kolejki
- **removeLast()** i **pollLast()**, które usuwają element z końca kolejki
- **getFirst()** i **peekFirst()**, które pobierają element z początku kolejki
- **getLast()** i **peekLast()**, które pobierają element z końca kolejki.

DEQUEUE

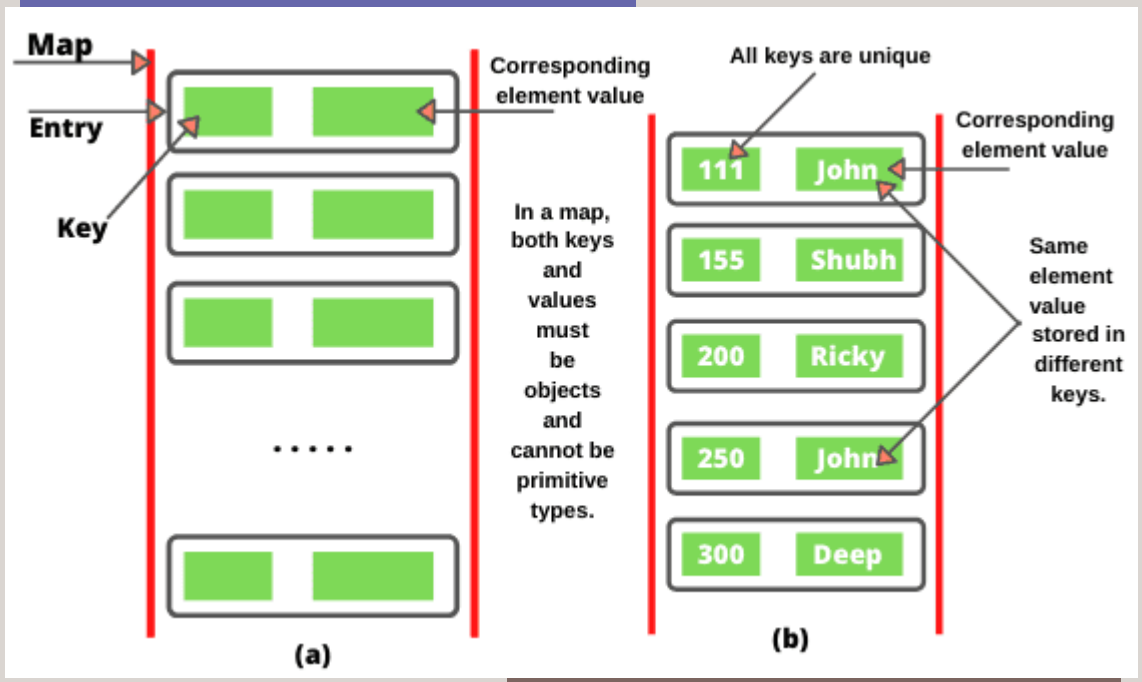
Należy przy tym pamiętać, że powyższe metody z przedrostkiem:

- **add**, **remove** i **get** wyrzucą **wyjątek** w przypadku niepowodzenia operacji
- **offer**, **poll**, **peek** zwrócą **specjalną wartość** w przypadku niepowodzenia operacji (**null dla obiektów i false dla typu boolean**).

```
// tworzenie obiektu Deque
final Deque<Integer> deque = new ArrayDeque<>();
// dodawanie elementów do deque
deque.offerLast(2);
deque.offerFirst(1);
// usuwanie elementów z deque wraz z usunięciem ze struktury -> 2
System.out.println(deque.pollLast());
// usuwanie elementów z deque bez usuwania ich ze struktury -> 1
System.out.println(deque.peekLast());
```

MAP

Interfejs **java.util.Map** jest strukturą danych, umożliwiającą operowanie na danych w postaci **klucz-wartość**. Każdy **klucz** w takim obiekcie **musi być unikalny**, tzn. jeden klucz może zawierać dokładnie jedną wartość.



MAP

Metody mapy, które służą do wykonywania podstawowych operacji, to m.in.:

- **put** - służy do dodania odpowiedniej pary do kolekcji lub zastąpienia starej wartości nową dla konkretnego klucza
- **get** - służy do pobierania wartości na podstawie klucza
- **remove** - usuwa element na podstawie klucza (lub dodatkowo wartości)
- **containsKey** - zwraca informację, czy istnieje wartość w mapie dla danego klucza
- **containsValue** - zwraca informację, czy istnieje klucz w mapie dla danej wartości

MAP

- **size** - zwraca ilość par (tzw. **entry**) znajdujących się w kolekcji
- **isEmpty** - zwraca informację, czy mapa jest pusta

UWAGA: **Wartość null może być kluczem w mapie.**

Metoda zwracające elementy w formie innej kolekcji:

- **keySet** – zwraca zbiór kluczy jako Set
- **values** – zwraca wszystkie elementy jako Collection
- **entrySet** – zwraca Set obiektów klucz-wartość

MAP - IMPLEMENTACJE

- **HashMap**

  - kolejność par nie jest zachowana

  - przechowuje informacje w tablicy hashującej

- **TreeMap**

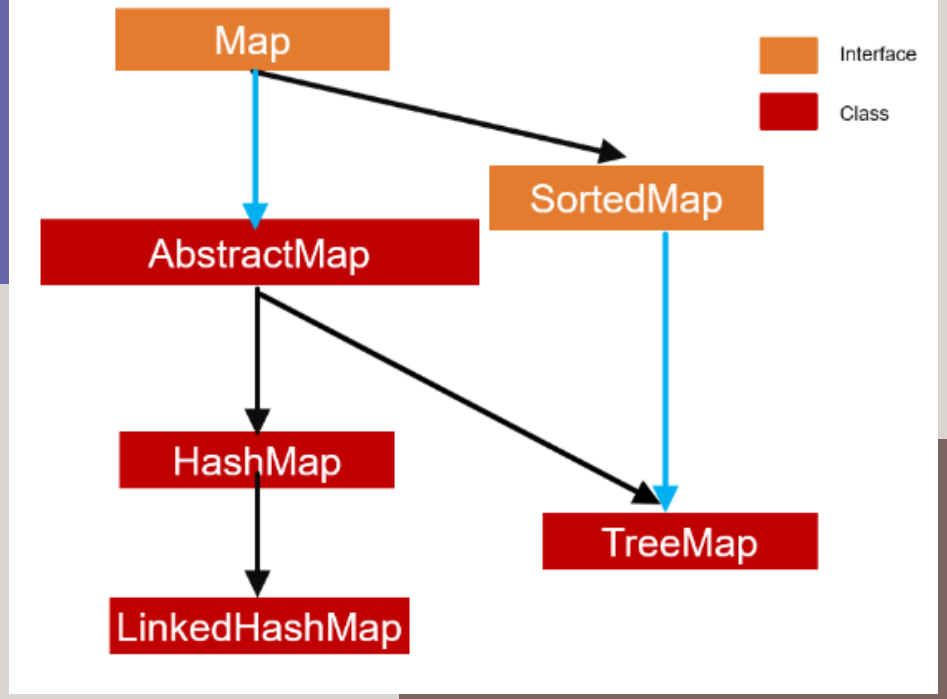
  - kolejność par jest zachowana według tzw. kolejności naturalnej kluczy lub wg pewnego Comparatora kluczy

  - przechowuje dane w drzewie czerwono-czarnym

- **LinkedHashMap**

  - zachowuje informację o kolejności dodawania poszczególnych par

  - implementacja jest oparta o tablicę hashującą, wraz z obsługą linked list



MAP – IMPLEMENTACJE - PRZYKŁADY

```java
final Map<Integer, String> ageToNames = new HashMap<>(); // tworzenie HashMapy
ageToNames.put(11, "Andrzej"); // dodawanie elementów
ageToNames.put(22, "Michal");  // dodawanie kolejnej pary
ageToNames.put(33, "Janusz");  // dodanie trzeciej pary do mapy
ageToNames.remove(22);     // usunięcie elementu na podstawie klucza
// wyświetlenie wartości na podstawie klucza 11 -> Andrzej
System.out.println(ageToNames.get(11));
```

```java
final Map<Integer, Integer> numberToDigitsSum = new TreeMap<>();
numberToDigitsSum.put(33, 6);
numberToDigitsSum.put(19, 10);
numberToDigitsSum.put(24, 6);
numberToDigitsSum.forEach((key, value) -> System.out.println(key + " " + value));

/* Elementy zawsze zostaną wypisane w tej samej kolejności:
  19 10
  24 6
  33 6
*/
```

```java
// stworzenie LinkedHashMap
final Map<Integer, String> ageToNames = new LinkedHashMap<>();
ageToNames.put(20, "Gosia");
ageToNames.put(40, "Kasia");
ageToNames.put(30, "Ania");

// iteracja po kluczach z wykorzystaniem keySet()
for (final Integer key : ageToNames.keySet()) {
  // kolejność kluczy zawsze taka sama -> 20, 40, 30
  System.out.println("Key is map: " + key);
}

// iteracja po wartościach z wykorzystaniem values()
for (final String value : ageToNames.values()) {
  // kolejność wartości zawsze taka sama -> Gosia, Ania, Kasia
  System.out.println("Value in map is: " + value);
}

// iteracja po parach za pomocą entrySet()
for (final Map.Entry<Integer, String> ageToName : ageToNames.entrySet()) {
  System.out.println("Value for key  " + ageToName.getKey()
    + " is " + ageToName.getValue());
  /* wynikiem zawsze będą poniższe 3 linijki, w tej dokładnie kolejności
     (wynika z użycia LinkedHashMap)
     Value for key  20 is Gosia
     Value for key  40 is Kasia
     Value for key  30 is Ania
   */
}
```

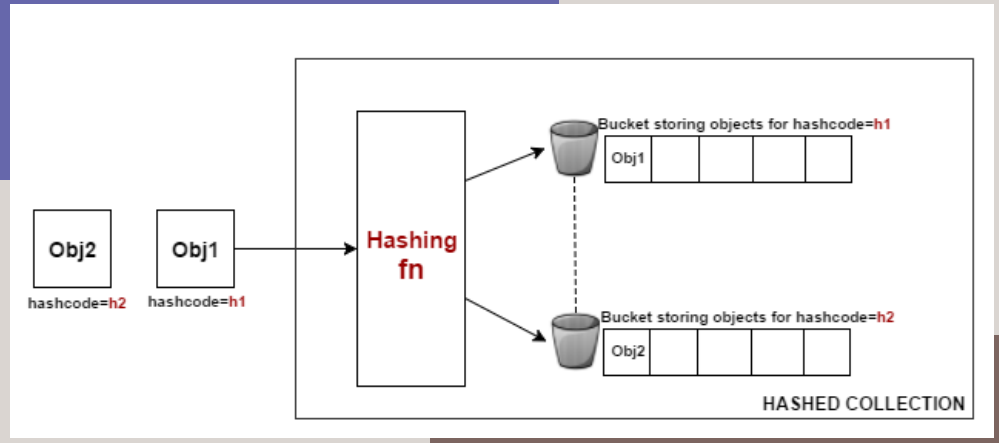
KONTRAKT W JAVIE

Dodając elementy do takich kolekcji, jak np. **Set**, na jakiejś podstawie musimy stwierdzić, **czy taki obiekt w takiej kolekcji się już znajduje**. Kontrakt w Javie jest ściśle powiązany z metodami **equals** i **hashCode**, które **deklarowane są w klasie Object**. Implementacje obydwu metod są często wykorzystywane łącznie.

KONTRAKT W JAVIE

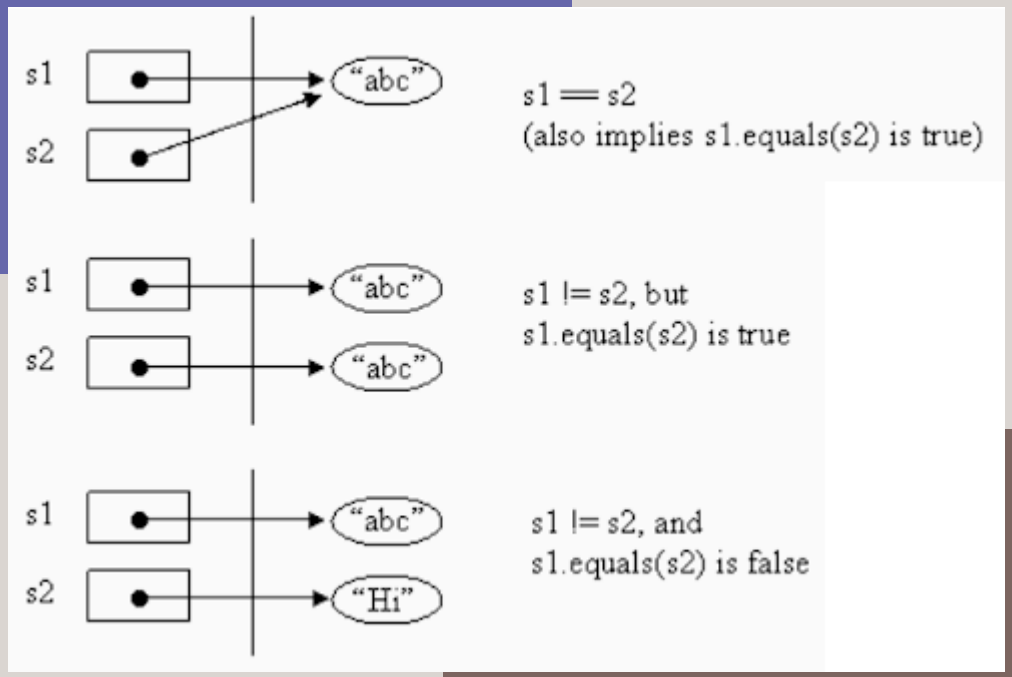
Kontrakt oparty jest o następujące zasady:

- **jeśli x.equals(y) == true**, to wówczas wymagane jest, aby **x.hashCode() == y.hashCode()**
- jeśli **x.hashCode() == y.hashCode()**, to nie jest wymagane, aby **x.equals(y) == true**
- **wielokrotne wywołanie metody hashCode** na tym samym obiekcie zawsze powinno dawać **taki sam wynik**.



EQUALS

Typy prymitywne w Javie porównywane są za pomocą operatora ==, natomiast ten sam operator logiczny, w przypadku porównywania obiektów, **porówna referencję do obiektów** (tzn. czy obie referencje wskazują ten sam adres), nie ich wartości. W celu porównania obiektów ze względu na wskazane wartości, wykorzystuje się nadpisaną metodę **equals**.



EQUALS

```java
public class Car {
    private String name;
    private String type;

    public Car(String name, String type) {
        this.name = name;
        this.type = type;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Car car = (Car) o;
        return Objects.equals(name, car.name) &&
                Objects.equals(type, car.type);
    }

    @Override
    public String toString() {
        return "Car{" +
                "name='" + name + '\'' +
                ", type='" + type + '\'' +
                '}';
    }
}
```

```java
int x = 10;
int y = 10;
boolean primitiveComparison = x == y;

Car car = new Car("BMW", "Sport");
Car car1 = new Car("BMW", "Sport");
// false, referencje są różne
boolean objectComparision = car == car1;
// true, jeżeli metoda equals jest zaimplementowana w intuicyjny sposób
boolean objectComparisionUsingEquals = car.equals(car1);
```

EQUALS

- jest dostępna w klasie Object
- można ją wywołać na każdym obiekcie
- metoda ta powinna być **zwrotna**, tzn. someObject.equals(someObject) powinna zwracać true
- powinna być metodą **symetryczną**, tzn. jeżeli x.equals(y) == true, to y.equals(x) == true
- metoda powinna być **przechodnia**, tzn. jeżeli x.equals(y) == true i y.equals(z) == true, to x.equals(z) == true
- metoda ta powinna byc´**spójna**, tzn. kilkukrotne wywołanie tej samej metody powinno dać ten sam wynik
- przy porównaniu z null metoda powinna zwracać false.

EQUALS

Metoda equals jest **wykorzystywana** podczas wywoływania, np. metody **contains z Collection API**, np.:

```
List<Car> cars = Arrays.asList(new Car("BMW", "Sport"), new Car("Volvo", "suv"));
boolean isVolvoInList = cars.contains(new Car("Volvo", "suv")); // true
```

HASHCODE

Metoda hashCode **zwraca typ prymitywny int** i docelowo ma oznaczać **pewnego rodzaju reprezentację liczbową obiektu**. Ponieważ typ ten ma ograniczony zakres, nie jest to unikalna wartość dla każdego obiektu, a co za tym idzie, **dwa różne obiekty mogą zwracać tę samą wartość hashCode**. To właśnie tę właściwość wykorzystuje wiele API Javy i **to ona jest podstawą kontraktu**. Metoda **hashCode** jest wykorzystywana do implementacji wielu struktur danych i algorytmów, np. **sortowania**.

HASHCODE

Metoda ta charakteryzuje się m.in.:

- metoda dostępna w klasie Object
- **implementacja metody hashCode powinna być powiązana z metodą equals**
- metoda powstaje najczęściej w oparciu o **hashe poszczególnych pól klasy**. Hashe atrybutów mnożymy najczęściej **przez liczby pierwsze** i sumujemy ze sobą.

UWAGA: W praktyce **rzadko ręcznie piszemy implementacje metod equals i hashcode**. Najczęściej je generujemy lub wykorzystujemy w tym celu zewnętrzne biblioteki.

```java
@Override
public int hashCode() {
  int result = name.hashCode();
  result = 31 * result + type.hashCode();
  return result;
}

Set<Car> cars = new HashSet<>();
cars.add(new Car("BMW", "Sport"));
cars.add(new Car("Volvo", "suv"));
boolean isVolvoInSet = cars.contains(new Car("Volvo", "suv"));
// true w przypadku, gdy metoda 'hashcode'
// jest zaimplementowana w klasie Car, false w przeciwnym wypadku
System.out.println(isVolvoInSet);
```

TOSTRING

W Javie **każda klasa dziedziczy po klasie Object**, w której znajduje się kilka publicznych metod, które również są automatycznie dziedziczone. Obok metody **equals()** i **hashCode()** do takich metod należy także metoda **toString()**, która zwraca opis obiektu w postaci Stringa.

```
public String toString()


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

Metodę toString() odziedziczoną z klasy Object warto **nadpisać własną implementacją**, aby wydruk był bardziej przyjazny i użyteczny

```
public static void main(String[] args) {
   System.out.println(new ArrayList());      // []
   System.out.println(new String[0]);        // [Ljava.lang.String;@65cc892e
}
```

```
public class Hippo {
   private String name;
   private double weight;

   public Hippo(String name, double weight) {
      this.name = name;
      this.weight = weight;
   }
   @Override
   public String toString() {
      return name;
   }
   public static void main(String[] args) {
      Hippo h1 = new Hippo("Harry", 3100);
      System.out.println(h1);     // Harry
   } }
```

SORTOWANIE

Porównywanie elementów kolekcji w Javie, realizowane jest za pomocą generycznych interfejsów:

- **Comparator<T>**, który pozwala na porównanie dwóch obiektów tego samego typu za pomocą metody **compare**
- **Comparable<T>**, który porównuje aktualny obiekt z obiektem pewnego typu za pomocą metody **compareTo**.

SORTOWANIE - COMPARATOR

Metoda **compare** powinna zwracać wartość:

- mniejszą niż 0, jeżeli pierwszy argument powinien znaleźć się przed drugim
- równą 0, jeżeli oba obiekty są sobie równe
- większą niż 0, jeżeli pierwszy argument powinien znaleźć się po drugim

```
Comparator<Integer> intComparator = Comparator.naturalOrder();
System.out.println(intComparator.compare(1, 2)); // -1
System.out.println(intComparator.compare(2, 2)); // 0
System.out.println(intComparator.compare(2, 1)); // 1
```

SORTOWANIE - COMPARATOR

Aby umożliwić precyzyjną kontrolę kolejności sortowania, możemy wykorzystywać instancje komparatorów i przekazać je m.in. do metod sortowania (takich jak **Collections.sort**, **Arrays.sort** lub metody sorted w Stream API). Komparatory mogą być również używane do **kontrolowania kolejności obiektów w takich strukturach danych, jak np. TreeMap lub TreeSet.**

```java
public class User {
  private final String name;
  private final int age;

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

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }
}
```

```java
List<User> users = Arrays.asList(
  new User("Piotr", 20),
  new User("Jan", 23));

Collections.sort(users, new Comparator<User>() {
  @Override
  public int compare(User o1, User o2) {
    return  (int)o1.getName().charAt(0) - (int)o2.getName().charAt(0);
  }
});

for (final User user : users) {
  System.out.println(user.getName());
}
```

SORTOWANIE - COMPARABLE

Za pomocą implementacji interfejsu **Comparable** określamy **domyślny porządek klasy**, według którego obiekty mogą być, np. układane w kolekcjach.

Metoda compareTo powinna zwracać:

- wartość większa od 0 oznacza, że obiekt porównywany powinien się znaleźć przed obiektem bazowym
- wartość równa 0 oznacza, że obiekty mają tę samą wartość
- wartość mniejsza od 0 oznacza, że obiekt bazowy powinien się znaleźć przed obiektem, z którym porównujemy.

```
Integer x = 3;
System.out.println(x.compareTo(2)); // 1
System.out.println(x.compareTo(3)); // 0
System.out.println(x.compareTo(4)); // -1
```

SORTOWANIE - COMPARABLE

```java
public class Sorting {

    public static void main(String[] args) {
        User[] users = new User[]{
                new User("Piotr", 40),
                new User("Jan", 23)
        };
        // Metoda ta wykorzystuje implementację interfejsu
        // Comparable do posortowania tablicy
        Arrays.sort(users);
        // output: [User{name='Jan', age=23}, User{name='Piotr', age=40}]
        System.out.println(Arrays.toString(users));
    }
}
```

```java
class User implements Comparable<User>{
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    @Override
    public int compareTo(User o) {
        return  this.getAge() - o.age;
    }
}
```

KLASA ARRAYS

Klasa **Arrays** jest klasą pomocniczą, która pozwala w łatwy sposób wykonywać pewne operacje na tablicach.

- metoda **binarySearch** pozwala na znalezieniu indeksu, na którym znajduje się dany element w posortowanej tablicy

```
int result = Arrays.binarySearch(new int[]{1, 2, 4, 5 ,6}, 5);
System.out.println(result); // 3
```

- metoda **asList** pozwala z dowolnej ilości elementów lub tablicy stworzyć listę

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> ints2 = Arrays.asList(5, 2, 7, 4);
```

- w celu porównania dwóch tablic możemy wykorzystać metodę **compare**.

```
int result1 = Arrays.compare(new int[]{1, 2, 3}, new int[]{1, 2, 3});
System.out.println(result1); // 0

int result2 = Arrays.compare(new int[]{1, 2}, new int[]{1, 2, 3});
System.out.println(result2); // -1

int result3 = Arrays.compare(new int[]{3, 1}, new int[]{1, 3});
System.out.println(result3); // 1
```

KLASA ARRAYS

- aby posortować tablicę (lub jej część) wykorzystujemy metodę **sort**, np.:

```
int[] ints = {3, 1, 5, 4, 2};
Arrays.sort(ints);
System.out.println(Arrays.toString(ints)); //[1, 2, 3, 4, 5]
```

- tablicę (lub jej część) możemy skopiować wykorzystując metodę **copyOf**

```
int[] original = new int[]{1, 2, 3, 4};
int[] copiedResult = Arrays.copyOf(original, 3);
System.out.println(Arrays.toString(copiedResult)); // [1, 2, 3]
```

- metoda **equals** porównuje zawartość tablicy

```
boolean result = Arrays.equals(new int[]{3, 1}, new int[]{1, 3});
System.out.println(result); // false
```

KLASA COLLECTIONS

Klasa **Collections**, podobnie jak klasa Arrays, **daje dostęp zestawu metod statycznych, które możemy wykorzystywać do wykonywania operacji na kolekcjach**.

- w celu uzyskania pustych (**niemutowalnych**) kolekcji wykorzystujemy metody, których nazwy zaczynają się od słowa **empty**, np.:

```
List<String> list = Collections.emptyList();
Map<String, Integer> map = Collections.emptyMap();
Set<Object> set = Collections.emptySet();

list.add("2"); // UnsupportedOperationException
```

- metoda **replaceAll** pozwala zastąpić wszystkie wystąpienia danego elementu w kolekcji wartością zastępczą.

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
ints.add(2);
ints.add(3);
Collections.replaceAll(ints, 2, 4);
ints.forEach(System.out::println); // 1 4 4 3
```

KLASA COLLECTIONS

- operację sortowania na kolekcji (**mutowalnej**) możemy wykonać przy pomocy metody **sort**, opcjonalnie wskazując odpowiedni **Comparator**, np.:

```
List<String> words = new LinkedList<>();
words.add("hi");
words.add("welcome");
words.add("hello!");
Collections.sort(words, Collections.reverseOrder());

System.out.println(words); // [welcome, hi, hello!]
```

DZIĘKUJĘ ZA UWAGĘ