Praktyczny kurs Java 11. Poziom I.

Spis treści

[Skróty 2](#_Toc60144772)

[Obiekty i klasy 3](#_Toc60144773)

[Konstruktory, bloki ststyczne i inicjalizacyjne 3](#_Toc60144774)

[Przekazywanie zmiennych do metod: pass-by-value 4](#_Toc60144775)

[Porównanie zmiennych == vs .equals 6](#_Toc60144776)

[Zasięg i widoczność zmeinnych 8](#_Toc60144777)

[Cykl Życia zmiennych. stos i sterta 9](#_Toc60144778)

[Opcjonalność zmiennych – null vs Optional 9](#_Toc60144779)

[metody statyczne i importy statyczne 11](#_Toc60144780)

[Dziedziczenie, abstrakcja i polimorfizm 11](#_Toc60144781)

[dziedziczenie klas 11](#_Toc60144782)

[Dziedziczenie metod instancji i konstruktorów 12](#_Toc60144783)

[Dziedziczenie elementów statycznych 12](#_Toc60144784)

[Nadpisywanie metod super klasy 12](#_Toc60144785)

[Klasy final 12](#_Toc60144786)

[Jak działa framework mockito 12](#_Toc60144787)

[Polimorfizm 12](#_Toc60144788)

[Przeciążanie metod i konstruktorów 14](#_Toc60144789)

[Klasy abstrakcyjne 16](#_Toc60144790)

[Interfejsy 16](#_Toc60144791)

[Metodo domyślne, prywatne i statyczne w interfejsach 17](#_Toc60144792)

[Dziedziczenie interfejsów 18](#_Toc60144793)

[Wyjątki i aseracje 19](#_Toc60144794)

[Obsługa sytuacji wyjątkowych 19](#_Toc60144795)

[Definiowanie niestandardowych wyjątków 21](#_Toc60144796)

[Blok try-catch-finally i multi-catch 22](#_Toc60144797)

[Blok try-with-resources 23](#_Toc60144798)

[Nadpisywanie metod z throws 23](#_Toc60144799)

[Wyjątki w konstruktorach 23](#_Toc60144800)

[Używanie asercji 24](#_Toc60144801)

[Kolekcje i mapy 25](#_Toc60144802)

[ArrayList 27](#_Toc60144803)

[PriorityQueue 30](#_Toc60144804)

[LinkedList 31](#_Toc60144805)

[ArrayDeque 32](#_Toc60144806)

[Hashowanie i kontrakt Equals-Hashcode 32](#_Toc60144807)

[HashSet 34](#_Toc60144808)

[LinkedHashSet 37](#_Toc60144809)

[TreeSet 39](#_Toc60144810)

[HashMap 41](#_Toc60144811)

[LinkedHashMap 41](#_Toc60144812)

[TreeMap 41](#_Toc60144813)

# Skróty

|  |  |
| --- | --- |
| Skrót | Działanie |
| Ctrl + Alt + L | Formatting code |
| Alt + Enter | Create test/Quick fixes |
| Ctrl + P | Show method params |
| psvm | public static void main(String[] args) |
| Suot (-,-m,-p,-v) | System.*out*.println(""); |
| Ctrl + B | Go to declaration |
|  |  |
|  |  |

Import java.lang – pakiet z najpopularniejszymi klasami

CamelCase

Java – silnie typizowana

Typ obiektowy – prymitywny

* Bajt – 1 bajt
* Short – 2 bajty
* Int – 4 bajty
* Long – 8 bajtów

Zakres 2(n-1) – 2(n-1)-1

Zmiennoprzecinkowe

* Float – 4 bajty (6-7 cyfr po przecinku)
* Double – bajtów (15 cyfr po przecinku)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  | 8 bitów | | |  | 32-8-1 bitów | | | |
|  |  |  |  |  |  |  |  |  |  |

Rzutowanie

* Widening: int widening=(int) byteVar
* Narrowing: byte narrowing=(byte) intVar

Hermetyzacja

Static – można stosować bez instancji

Static != instance

Wszystkie klasy dziedziczą po Object

Pole final – musi być zdeklarowane w konstruktorze

JVM – wirtualna maszyna JAVY

Kod źródłowy – w plikach .java → kompilacja → język pośredni .class

JVM odczytyje bytecode

Domyślny modyfikator dostępu: public> protected > default > private

Immutable – np. LocalDate, BigDecimal – wywołanie metody na danym obiekcie zwróci nowy obiekt – nie zmieni wywoływanego obiektu

# Obiekty i klasy

## Konstruktory, bloki ststyczne i inicjalizacyjne

Kolejność wywoływania konstruktorów:

* Bloki statyczne – wg kolejności w pliku źródłowym – wywoływane, kiedy klasa ładowana jest do pamięci
  + Jeśli inicjalizacyjnych jest więcej niż jeden to są wywoływane w takiej kolejności jak są napisane w kodzie
  + Jeśli nie utworzymy żadnej instancji, to bloki inicjalizacyjne statyczne nie zostaną wywołane (jeśli np. chcemy wyświetlić wartość statycznego pola, bez utworzenia instancji, to wywołane zostaną tylko bloki statyczne inicjalizacyjne)
* Bloki inicjalizacyjne - wg kolejności w pliku źródłowym
* Konstruktory

Klasa Person\_Konstruktoryblokiinicjalizacyjne

import java.math.BigDecimal;  
  
public class Person\_KonstruktoryBlokiInicjalizacyjne {  
 //statycznego pola nie można dekklarować  
 private static BigDecimal *RATE*;  
 //wyjątkiem jest inicjalizacja static CONST  
 private static BigDecimal *RATE\_CONST*=BigDecimal.*valueOf*(1.1);  
 //inicjalizacja bezpośrednio przy deklaracji  
 private String name ="John";  
 private Integer age;  
  
 static {  
 System.*out*.println("Static Initializer 2");  
 }  
  
 {  
 System.*out*.println("Instance Initializer 2");  
 }  
  
 //inicjalizacja w bloku inicjalizacyjnym - pola i pola statyczne  
 {  
 name="John";  
 *RATE* = BigDecimal.*valueOf*(10.0);  
 System.*out*.println("Instance Initializer 1");  
 }  
  
 //inicjalizacja w statycznym bloku inicjalizacyjnym - tylko pola statyczne  
 static  
 {  
 *RATE*=BigDecimal.*valueOf*(10.0);  
 System.*out*.println("Static Initializer 1");  
 }  
  
 //konsktuktor niestandardowy  
 public Person\_KonstruktoryBlokiInicjalizacyjne(String name)  
 {  
 this.name=name;  
 System.*out*.println("Konsruktor(name)");  
 }  
  
  
 public Person\_KonstruktoryBlokiInicjalizacyjne(String name, Integer age)  
 {  
 //wywołanie innego konstruktora, zawsze jako pierwsza linijka  
 this(name);  
 this.age=age;  
 System.*out*.println("Konsruktor(name, age)");  
 }  
}

Main

new Person\_KonstruktoryBlokiInicjalizacyjne("Kasia", 23);

konsola

Static Initializer 2

Static Initializer 1

Instance Initializer 2

Instance Initializer 1

Konsruktor(name)

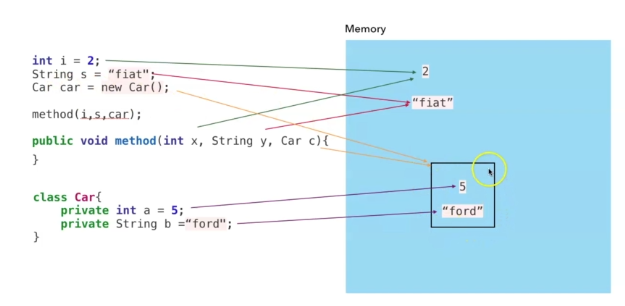
Konsruktor(name, age)

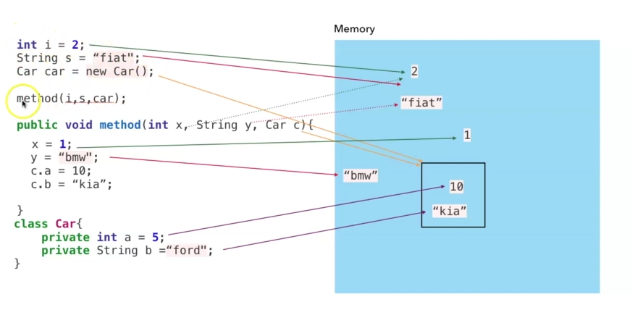
## Przekazywanie zmiennych do metod: pass-by-value

W Javie zmienne przekazywane są do metod za pomocą mechanizmu **pass-by-value**.

Do metody nie jest przekazywana wartość zmiennej, tylko **kopia** referencji do adresu w pamięci, gdzie zapisana jest zmienna.

Jeśli przekazujemy do metody, to w Javie kopiuje się adres referencji





Klasa Przekazywanie zmiennych

public class ZmienneFinal {  
 //inicjalizacja razem z deklaracją  
 private final String name="";  
  
 //LUB  
  
 //inicjalizacja w bloku inicjalizacyjnym  
 {  
 name="";  
 }  
  
 //LUB  
  
 //inicjalizacja w konstruktorze  
 public ZmienneFinal()  
 {  
 name="";  
 }  
  
 //---------------------------------------------  
  
 //static+final > STALA - wielkimi literami  
 //inicjalizacja w zmiennej z deklaracją  
 private final static String *NAME*="";  
  
 //inicjalizacja w statycznym bloku inicjalizacyjnym  
 static {  
 *NAME*="";  
 }  
  
 //---------------------------------------------  
  
 //zmienna lokalna final  
 public void method()  
 {  
 final String name;  
 //może być zadeklarowana potem  
 name="";  
 }  
  
 //---------------------------------------------  
  
 //ale finala raczej się nie pisze i po prostu pilnuje, żeby nie zmienić jej wartości  
 public void method2(final String s) //pass-by-value  
 {  
 s="xyz"; //BLĄD - nie kompiluje się, bo nie można zmienić final  
 }  
}

Main

new PrzekazywanieZmiennych().passingVariablesToMethod();

Konsola

Przed wywołaniem method()

i: 10

s: fiat

car.a: 5

car.b: ford

Po wywołaniu method()

i: 10

s: fiat

car.a: 10

car.b: kia

## Porównanie zmiennych == vs .equals

Main porównywanie

public class MainPorownywanie {  
  
 public static void main(String[] args)  
 {  
 int i1=100;  
 int i2=100;  
 Integer i3=new Integer(100);  
 Integer i4=new Integer(100);  
  
 System.*out*.println("i1==i2: "+(i1==i2));  
 //jeśli ma prymitywny i wraper to wraper sprowadza do prymitywnego  
 System.*out*.println("i1==i3: "+(i1==i3)); //to samo co i3.intValue()  
 System.*out*.println("i3==i4: "+(i3==i4));  
  
 //bo Cash do 128  
 Integer i5=Integer.*valueOf*(127);  
 Integer i6=Integer.*valueOf*(127);  
 Integer i7=Integer.*valueOf*(1000);  
 Integer i8=Integer.*valueOf*(1000);  
  
 System.*out*.println("i5==i6: "+(i5==i6));  
 System.*out*.println("i7==i8: "+(i7==i8));  
  
 String car1=new String("ford");  
 String car2=new String("ford");  
 System.*out*.println("ca1==car2: "+(car1==car2));  
  
 //bo StringPool  
 String car3=new String("ford").intern();  
 String car4=new String("ford").intern();  
 System.*out*.println("ca3==car4: "+(car3==car4));  
  
 String car5=new String("ford").intern();  
 String car6=new String("ford").intern();  
 System.*out*.println("car5==car6 "+(car5==car6));  
  
 }  
}

konsola

i1==i2: true

i1==i3: true

i3==i4: false

i5==i6: true

i7==i8: false

ca1==car2: false

ca3==car4: true

car5==car6 true

MainPorownywanieInstancje

public class MainPorownywanieInstancje {  
 static class Person{  
 private final String name;  
  
 Person(String name) {  
 this.name = name;  
 }  
  
 @Override //nie trzeba w nowych Javach deklarować @Override  
 public boolean equals(Object obj)  
 {  
 //ale jak ktoś przy deklaracji instancji uzyje new Person(new String("Kasia") to bd false  
 //dlatego trzeba też użyć equals na Stringu i wtedy porówna semantycznie  
 return this.name.equals(((Person)obj).name); //rzutowanie Object do Person  
 }  
 }  
  
 public static void main(String[] args) {  
 Person p1=new Person("Kasia");  
 Person p2=new Person("Kasia");  
 System.*out*.println("p1==p2: "+(p1==p2));  
 System.*out*.println("p1.equals(p2): "+p1.equals(p2)); //domyslna implementacja - ona używa ==, przed nadpisaniem @Override - zwróci FALSE  
  
 System.*out*.println("p1.name==p2.name: "+(p1.name==p2.name));  
  
 Person p3=p1;  
 System.*out*.println("p1==p3: "+(p1==p3));  
 }  
}

Konsola

p1==p2: false

p1.equals(p2): true

p1.name==p2.name: true

p1==p3: true

## Zasięg i widoczność zmeinnych

Zasiegiwartosczmiennych

public class ZasiegIWidocznoscZmiennych {  
 private String president="Duda";  
 private String city="Warszawa";  
  
 public void katowice() {  
 String president = "Krupa";  
  
 //shadowing - przysłanianie  
 System.*out*.println("president from void: " + president);  
 System.*out*.println("president from instance of ZasiegIWidocznoscZmiennych: " + this.president);  
  
 //klasa zagnieżdżona w metodzie  
 class MusicClub {  
 private String president = "Kowalski";  
  
 public void liverpool() {  
 String president = "Nowak";  
  
 System.*out*.println("president from loverpool: " + president);  
 System.*out*.println("president from instance of MusicClub: " + this.president);  
  
 //nie mozna wyswietlic president z ZasiegIWidocznoscZmiennych, ale nieprzysłonięte można  
 System.*out*.println("City from instance of ZasiegIWidocznoscZmiennych: " + city);  
  
 System.*out*.println("president from instance of ZasiegIWidocznoscZmiennych: " + ZasiegIWidocznoscZmiennych.this.president);  
  
 }  
 }  
 }  
 public void scope(String name3)  
 {  
 String name="Kasia";  
 int x=0;  
 if(x>0)  
 {  
 // String name="Ala"; - nie można, bo on widzi name z metody  
 String name2="Beata"; //widoczna tylko w if(){ ... }  
 }  
  
 if(x>0)  
 {  
 String name2="Darek"; //widoczna tylko w if(){ ... }  
 }  
 }  
  
}

## Cykl Życia zmiennych. stos i sterta

**Stos – Stack**

* Rośnie i kurczy się wraz z wywoływaniem i wychodzeniem z metod
* Ma stały rozmiar
* Zapisane są tu **zmienne prymitywe lokalne** oraz **referencje** do obiektów
* Zmienne istnieją do czasu działania metody, w której są zdefiniowane
* Każdy wątek ma swój stos
* **Jeżeli brakuje pamięci Java wyrzuci wyjątek java.lang.StackOverFlowError**, np.:

public void method()  
{  
 method();  
}

**Sterta – Heap**

* Jest dynamiczna
* Nie ma ograniczeń w rozmiarze
* Zapisane są tutaj wartości **typów obiektowych** i **zmienne prymitywne** należące do obiektów
* **Sterta** jest wspólna dla wszystkich wątków
* **String pool** jest częścią **sterty**
* Jeśli brakuje pamięci java wyrzuci wyjątek **java.lang.OutOfMemoryError** (np. nieskończona pętla tworząca obiekty i ich lista)

public static void method()  
{  
 List<String> lista=new ArrayList<String>();  
 while (true)  
 {  
 lista.add(UUID.*randomUUID*().toString());  
 }  
}

## Opcjonalność zmiennych – null vs Optional

Szkoła amerykańska – American Dream – zakładamy, że aplikacja poda prawidłowe wartości

Szkoła rosyjska – zakładamy, że użytkownik może podać niepoprawne dane

Calculatoropcjonalnosc

public Double calculate(Integer factor1, Double factor2)  
{  
 //szkoła rosyjska, bo jest zabezpieczenie przed podaniem nulla - bez ifa - szkoła amerykańska  
 if(factor1!=null&&factor2!=null) {  
 return factor1 \* factor2;  
 }return null;  
}

main

new CalculatorOpcjonalnosc().calculate(10,10.0);  
new CalculatorOpcjonalnosc().calculate(10, null); //jeśli nie zabezpieczymy przed nullem poleci NullPointerException

Optional

Calculatoropcjonalnosc

public class CalculatorOpcjonalnosc {  
 public Optional<Double> calculate(Integer factor1, Optional<Double> factor2)  
 {  
 //1 sposób  
 if(factor2.isPresent()) {  
 return Optional.*of*(factor1 \* factor2.get());  
 }  
 return Optional.*empty*();  
  
 //2 sposób  
 return Optional.*of*(factor1\*factor2.orElse(1.0));  
  
 }  
}

MAIN

new CalculatorOpcjonalnosc().calculate(10, Optional.*empty*());  
new CalculatorOpcjonalnosc().calculate(10, Optional.*of*(10.0));  
new CalculatorOpcjonalnosc().calculate(10, Optional.*ofNullable*(null));  
  
final Optional<Double> result=new CalculatorOpcjonalnosc().calculate(10, Optional.*ofNullable*(null));  
if(result.isPresent())  
{  
 result.get().intValue();  
}

Obsługa nulli

Personoptional

public class PersonOptional {  
 String name;  
  
 @Override  
 public boolean equals(Object obj)  
 {  
 if (Optional.*ofNullable*(this.name).isEmpty())  
 {  
 return false;  
 }  
 final String name=((PersonOptional)obj).name;  
 if(Optional.*ofNullable*(name).isEmpty())  
 {  
 return false;  
 }  
  
 return (this.name.equals(name));  
 }  
}

## metody statyczne i importy statyczne

GEnderenum

public enum GenderEnum {  
 *MELE*,  
 *FEMALE*}

Genderclass

public class GenderClass {  
 public static String *MALE*="MELE";  
 public static String *FEMALE*="FEMALE";  
}

IMPORTOPTIONAL

import static java.util.Optional.\*;  
import static Sekcja8\_ObiektyKlasy.GenderClass.*MALE*;  
  
public class ImportOptional {  
 public static void main(String[] args) {  
 //CalculatorOpcjonalnosc.calculate(10, Optional.empty()); //bez importu  
 CalculatorOpcjonalnosc.*calculate*(10, *empty*()); //po imporcie nie trzeba dawać Optional  
  
  
 System.*out*.println(*MALE*); //po imporcie można używać statyczne bez nazwy klasy  
  
 }  
}

# Dziedziczenie, abstrakcja i polimorfizm

## dziedziczenie klas

W Javie – klasa może dziedziczyć TYLKO po jednej klasie.

Do dziedziczenie – słowo kluczowe **extends** w deklaracji.

Wszystkie elementy instancji klasy oznaczone jako **protected** zostaną odziedziczone.

**Klasa nadrzędna / Parent class/ Super class,   
Klasa podrzędna / Child class / Subclass**

Nie można użyć razem w konstruktorze super() i this().

Super() zawsze pierwsze w konstruktorze

## Dziedziczenie metod instancji i konstruktorów

Kolejność wywołania konstruktorów i bloków inicjalizacyjnych przy dziedziczeniu:

* Statyczne bliki inicjalizacyjne – od klas nadrzędnej do podrzędnej
* Bloki inicjalizacyjne + konstruktory – po kolei od wszystkich klas zaczynając od nadrzędnej

## Dziedziczenie elementów statycznych

## Nadpisywanie metod super klasy

Modyfikatory dostępu: public > protected > default > private

Nadpisywana metoda nie może mieć „mniejszego” modyfikatora dostępu, niż zapisywana metoda.

Prywatnych metod nie da się nadpisać. Prywatne metody nie są dziedziczone.

Typ zwracany musi mieć taki sam typ jak metody nadpisywanej lub być jego subklasą.

Nadpisywanie metod statycznych jest zabronione.

Lista parametrów nadpisywanej metody musi być zgodna pod względem typu i kolejności.

Jeżeli parametry nie są takie same, to wówczas jest to **przeładowanie** metody.

Nie można nadpisywać metod, które oznaczone są jako **final**.

Nadpisując metodę można odwołać się do implementacji z klas nadrzędnych przy użyciu **super**.

Opcjonalnie można oznaczać nadpisywanie metody anotacją @Override.

Zasady – wyjątki – o tym bd później.

## Klasy final

Nie powinniśmy używać final w przypadku klas.

Final w przypadku metod i klas blokuje możliwość nadpisywania.

Final w przypadku zmiennych blokuje możliwość przypisania nowej wartości wartości do zmiennej.

## Jak działa framework mockito

Tworzy w locie dynamicznie subklasę i nadpisuje daną metodę.

W przypadku oznaczenia klasy/metod final to Mockito nie da rady nadpisać klasy/metody.

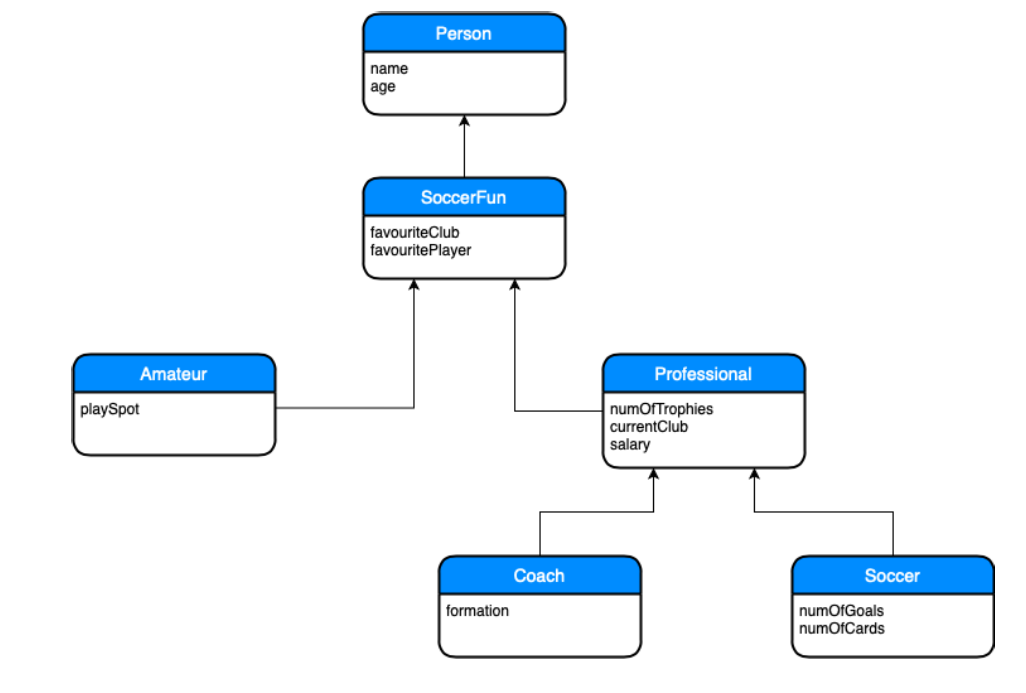
## Polimorfizm

Dziedziczenie – przekazywanie pewnych cech innym klasom

Enkapsulacja – ukrywanie widoczności elementów danej klasy dla innych klas

Polimorfizm (wielopostaciowość) – zdolność obiektu, żeby działać inaczej dla innych typów

schemat



Amateur

public class Amateur extends SoccerFun {  
 protected String playSpot;  
  
 public Amateur(String name, int age) {  
 super(name, age);  
 }  
  
  
 @Override  
 protected void whoAmI()  
 {  
 System.*out*.println("I'm Amateur");  
 }  
  
 public void amateur()  
 {  
 System.*out*.println("amateur void");  
 }  
}

main

public static void main(String[] args) {  
 SoccerFun soccerFun=new Amateur("Jan", 23);  
 Person coach=new Coach("Adam", 45);  
  
 soccerFun.whoAmI();  
 ((Amateur)soccerFun).amateur();  
 coach.whoAmI();  
  
 //((Coach)soccerFun).whoAmI(); //błąd ClassCastException  
 //dlatego  
 if(soccerFun instanceof Coach)  
 {  
 ((Coach)soccerFun).whoAmI();  
 }  
}

konsola

I'm Amateur

amateur void

I'm Coach

## Przeciążanie metod i konstruktorów

Sygnatury metody – unikalny identyfikator metody w klasie. Składa się z:

* Nazwy metody
* Uporządkowanej listy parametrów

Przeciążanie metody – definiowanie metod o tej samej nazwie, ale innych parametrach.

Zasady selekcji metody

* Dokładne dopasowanie (1)
* Najbardziej konkretna metoda (2)  
  >double > float > int > char  
   int > short > byte
* Widening przez autoboxing (3)
* Autoboxing przez varargs (4)

Ad. 2

Main

public static void main(String[] args) {  
  
 //>double > float > int > char  
 // int > short > byte  
  
 char c = 1;  
 *process*(c);  
 }  
  
 public static void process(int var) {  
 System.*out*.println("int");  
 }  
  
 public static void process(float var) {  
 System.*out*.println("float");  
 }  
}

Konsola

int

Ad. 3

Main

public static void main(String[] args) {  
  
 //>double > float > int > char  
 // int > short > byte  
  
 byte b = 1;  
 *process*(b);  
}  
  
public static void process(double var) {  
 System.*out*.println("double");  
}  
  
public static void process(Byte var) {  
 System.*out*.println("Byte");  
}

Konsola

double

Ad. 4

MAIN

public static void main(String[] args) {  
  
 //>double > float > int > char  
 // int > short > byte  
  
 int i = 1;  
 *process*(i);  
}  
  
public static void process(int... var) {  
 System.*out*.println("int...");  
}  
  
public static void process(Integer var) {  
 System.*out*.println("Integer");  
}

KONSOLA

Integer

## Klasy abstrakcyjne

Klasa abstrakcyjna – ma zablokowaną możliwość tworzenia instancji

W klasach abstrakcyjnych oprócz tworzenia elementów zwykłej klasy można deklarować metody abstrakcyjne.

## Interfejsy

Jest to zestaw wymagań dla klas, które chcą być zgodne z interfejsem. Interfejs != klasa.

Do oznaczenia, że klasa implementuje interfejs używa się słowa kluczowego **implements** w deklaracji klasy.

Interfejs vs. Klasa abstrakcyjna

* Klasy mogą implementować wiele interfejsów, ale dziedziczyć tylko po jednej klasie.
* Interfejsy nie mogą mieć implementacji do metod oznaczonych jako public, protected i default.

Deklaracja

* Każdy interfejs domyślnie jest oznaczany jako **abstract**.
* Wszystkie metody interfejsu są domyślnie **public abstract**. (do Javy 8)
* Wszystkie zmienne są domyślnie **public static final**. Deklarowane jedynie razem z inicjalizacją. (nie działa statyczny blok inicjalizacyjny)

Konwencje nazewnicze

* Nazwa interfejsu powinna być przymiotnikiem i wyrażać cechę którą nadaje ten interfejs. Często stosuje się sposób tworzenia przymiotników od rzeczowników: rzeczownik + **-able**.
* Nazwa interfejsu może być też rzeczownikiem kiedy reprezentuje rodzinę klas, np. List Map.

Walkable

public interface Walkable {  
 public abstract void walk();  
}

Person

public abstract class Person implements Walkable, Eatable{  
 protected String name;  
 protected int age;  
  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
}

Coach

public class Coach extends Proffessional{  
 protected String formation;  
  
 public Coach(String name, int age) {  
 super(name, age);  
 }  
  
 @Override  
 public void eat() {  
 System.*out*.println("I'm eating. Coach");  
 }  
  
 @Override  
 public void walk() {  
 System.*out*.println("I'm walking. Coach");  
 }  
}

Cat

public class Cat implements Walkable, Eatable {  
 @Override  
 public void eat() {  
 System.*out*.println("I'm eating. Cat");  
 }  
  
 @Override  
 public void walk() {  
 System.*out*.println("I'm walking. Cat");  
 }  
}

Main

public class MainInterfejsy {  
  
 public static void main(String[] args) {  
  
 Amateur amateur=new Amateur("Jan", 23);  
 Coach coach =new Coach("Adam", 43);  
 Cat cat=new Cat();  
  
 *goToWalk*(amateur);  
 *goToWalk*(coach);  
 *goToWalk*(cat);  
  
 }  
  
 public static void goToWalk(Walkable walkable)  
 {  
 walkable.walk();  
 }  
}

## Metodo domyślne, prywatne i statyczne w interfejsach

Każda zmienna w interfejsie: public static final

walkable

public abstract interface Walkable {  
  
 //wyszarzone - nie trzeba ich pisać  
  
 public static final String *name* = "hello";  
  
 public abstract void walk();  
  
 public default void doIt() //od Java 8, prywatnych domyslnych nie mozna  
 {  
 System.*out*.println("Do it");  
 }  
  
 private void privMethod() //od Java 9  
 {  
 System.*out*.println("privMethod()");  
 }  
  
 public static int staticInt() //protected static nie mozna  
 {  
 return 1;  
 }  
  
 private static int privateStaticInt() {  
 return 1;  
 }  
}

## Dziedziczenie interfejsów

**Interfejsy mogą dziedziczyć interfejsy, klasy mogą implementować interfejsy i dziedziczyć inne klasy.**

Readable

public interface Readable {  
 void read();  
  
 default void doIt()  
 {  
 System.*out*.println("Do it: read");  
 }  
}

Writeable

public interface Writeable {  
 void write();  
}

ReadWriteable

public interface ReadWriteable extends Readable, Writeable{  
  
 void delete();  
  
 default void doIt()  
 {  
 System.*out*.println("Do it: read&write");  
 }  
}

DatabaseObject

public class DatabaseObject implements ReadWriteable {  
 @Override  
 public void read() {  
 }  
  
 @Override  
 public void write() {  
 }  
  
 @Override  
 public void delete() {  
 }  
}

Main

public static void main(String[] args) {  
 DatabaseObject dbObj=new DatabaseObject();  
 dbObj.doIt();  
  
 //doIt - Java wybierze bardziej konkretną czyli tą z ReadWriteable  
 //w przypadku gdyby doIt miała zarówno Readable jak i Writeable należaloby ją nadpisać w pierwszej klasie w której są uzywane i nie jest ona abstract  
}

# Wyjątki i aseracje

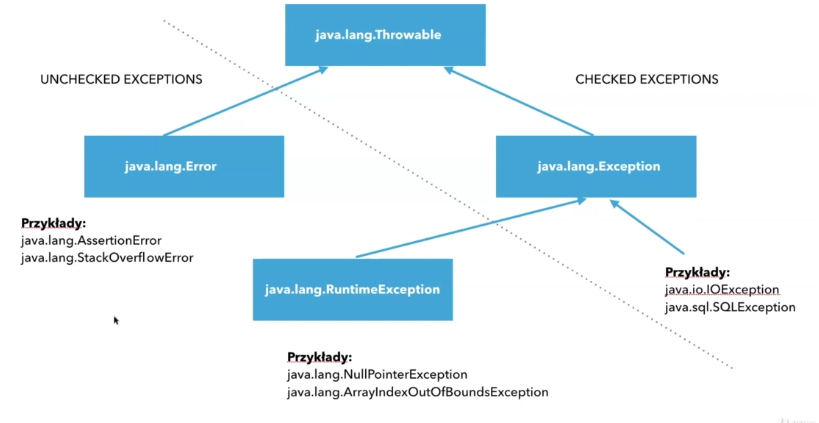
## Obsługa sytuacji wyjątkowych

Wszystkie wyjątki dziedziczą po java.lang.Throwable, ma on 2 subklasy:

* Java.lang.Error (nie powinno się po niej dziedziczyć)
* Java.lang.Exception

Wyjątki

* Chcecked – należy je obsłużyć
* unchecked



Calculator

public class Calculator {  
   
 public void calculate\_unchecked()  
 {  
 //throw + coś co dziedziczy Throwable (nic innego nie może być)  
 throw new NullPointerException();  
 }  
   
 //1 sposób obsługi wyjatkow  
 public void calculate\_checked(String input) {  
 try {  
 throw new IOException();  
 }  
 catch (IOException ioException)  
 {  
 System.*out*.println("Enexpected error, input: "+input);  
 ioException.printStackTrace();  
 }  
 }  
  
 //2 sposob obslugi wyjatkow  
 public void calculate\_checked2() throws IOException{  
  
 throw new IOException();  
 }  
}

Main

public class MainObslugaSytuacjiWyjatkowych {  
 public static void main(String[] args) throws IOException {  
  
 System.*out*.println(">> Checked Exception (1)");  
 new Calculator().calculate\_checked("\*ERROR\*");  
  
 //System.out.println(">> Checked Exception (2)");  
 //new Calculator().calculate\_checked2();  
   
 System.*out*.println(">> Unchecked Exception");  
 new Calculator().calculate\_unchecked();  
  
  
 }

Konsola

>> Checked Exception (1)

Enexpected error, input: \*ERROR\*

>> Unchecked Exception

java.io.IOException

at Sekcja10\_WyjatkiIAseracje.Calculator.calculate\_checked(Calculator.java:16)

at Sekcja10\_WyjatkiIAseracje.MainObslugaSytuacjiWyjatkowych.main(MainObslugaSytuacjiWyjatkowych.java:11)

Exception in thread "main" java.lang.NullPointerException

at Sekcja10\_WyjatkiIAseracje.Calculator.calculate\_unchecked(Calculator.java:10)

at Sekcja10\_WyjatkiIAseracje.MainObslugaSytuacjiWyjatkowych.main(MainObslugaSytuacjiWyjatkowych.java:17)

## Definiowanie niestandardowych wyjątków

ServiceLevel1

public class ServiceLevel1 {  
 private final ServiceLevel2 service=new ServiceLevel2();  
  
 public double calculate(int var)  
 {  
 return service.calculate(var\*2);  
 }  
}

ServiceLevel2

public class ServiceLevel2 {  
 private final ServiceLevel3 service=new ServiceLevel3();  
  
 public double calculate(int var)  
 {  
 return (var+10)\*service.calculate(var);  
 }  
}

ServiceLevel3

public class ServiceLevel3 {  
 private final ServiceLevel4 service=new ServiceLevel4();  
  
 public double calculate(int var)  
 {  
 return 10\*service.calculate(var);  
 }  
}

ServiceLevel4

public class ServiceLevel4 {  
  
 public double calculate(int var)  
 {  
 //jesli ServiceLevel4 wyrzucilby CheckedException, to jesli chcielibysmy obsluzyc go dopiero w mainie. trzeba by go przepychac przez wsyzstkie serwisy 4-3-2-1-main, dlatego lepiej tutaj Unchecked  
 if(var==0)  
 {  
 throw new UncheckedException(var);  
 }  
 return var\*1.5;  
 }  
}

ChceckedException

public class CheckedException extends Exception {  
}

UncheckedException

public class UncheckedException extends RuntimeException {  
  
 public UncheckedException(int var)  
 {  
 super("Variable is invalid (value: "+var+")");  
 }  
}

Main

public class Main {  
 public static void main(String[] args) {  
 try {  
 new ServiceLevel1().calculate(0);  
 }  
 catch (UncheckedException e) //Unchecked bardziej kokretny, więc musi być wyżej w catchu  
 {  
 System.*out*.println("I'm know about this Exception. "+e.getMessage());  
 }  
 catch (Exception e)  
 {  
 System.*out*.println("Unexpected Error"+e.getMessage());  
 e.printStackTrace(); //StackTrace tylko w przypadku wyjkatkow ktorych sie nie spodziewamy  
 }  
 }  
}

Konsola

I'm know about this Exception. Variable is invalid (value: 0)

## Blok try-catch-finally i multi-catch

Finally – np. przy pracy z plikami, jeśli w trakcie poleci wyjątek, to plik należy zamknąć w finally; przy pracy z bazą danych - zamknięcie

Try+catch / try+catch+finally / try+finally – finally zawsze na końcu

Hierarchia

* *RuntimeException*
* **UncheckedRoot** extends *RuntimeException*
  + - **UncheckedLevel1a** extends UncheckedRoot
      * **UncheckedLevel2a** extends UncheckedLevel1a (+pole public static String name)
      * **UncheckedLevel2b** extends UncheckedLevel1a
    - **UncheckedLevel1b** extends UncheckedRoot

Main

public static void main(String[] args) {  
 try{  
 throw new UncheckedLevel2a();  
 }  
 catch (UncheckedLevel2a | UncheckedLevel2b e) //nie można listować razem bardziej konkretne i mniej konkretne z tego samego drzewa dziedziczenia  
 //typ tej zmiennej e bd pierwszy wspolny typ po ktorym dziedziczą  
 {  
  
 //ale można zrobić coś takiego  
 //ale wtedy nie powinno się używac multicatcha, tylko osobno  
 if(e instanceof UncheckedLevel2a)  
 {  
 System.*out*.println(((UncheckedLevel2a) e).name);  
 }  
  
 }  
 finally //po wszystkich catchach  
 {  
 System.*out*.println("The end...");  
 }  
}

## Blok try-with-resources

Dla klas implementujących interfejs AutoCloseable

Java automatycznie zamknie zasób (plik, połączenie bazodanowe, scanner ...)

W takim try więcej niż jeden zadeklarować

DATABASEMENAGER

public class DataBaseMenager implements AutoCloseable{  
  
 public void openSession()  
 {  
 //open session  
 }  
  
 @Override  
 public void close() throws Exception {  
 //close session and clean  
 }  
}

MAIN

public static void main(String[] args) {  
  
 //dla klas implementujących interfejs AutoCloseable  
 //Java automatycznie zamknie zasób (plik, połączenie bazodanowe, scanner ...)  
 //można w takim try więcej niż jeden zadeklarować  
 try (FileWriter fileWriter = new FileWriter("test.txt");  
 Scanner scanner = new Scanner(System.*in*);  
 DataBaseMenager dataBaseMenager=new DataBaseMenager()) {  
 } catch (IOException ioException) {  
 ioException.printStackTrace();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

## Nadpisywanie metod z throws

Wyjątki – dziedziczenie – dotyczy tylko wyjątków **chcecked**

* Typ wyjątku wyrzucany z metody nie może być bardziej ogólny od wyrzucanego z nadpisywanej metody
* Metoda nie może wyrzucać nowych wyjątków
* Metoda (nadpisywana) może w ogóle, nie wyrzucać wyjątków

## Wyjątki w konstruktorach

W blokach inicjalizacyjnych nie można wyrzucać wyjątków – musi być szansa, że blok się wykona bez wyrzucenia wyjątku  
ALE! Jeżeli istnieje możliwość, że blok inicjalizacyjny nie wyrzuci wyjątku, to można go wyrzucać.

**Jeśli w bloku inicjalizacyjnym wyrzucamy wyjątek checked to musimy go zadeklarować we wszystkich konstruktorach.**

W blokach statycznych – tylko bloki unchecked oraz musi istnieć szansa, że blok wykona się bez wyrzucenia błędu.

Calculator

public class Calculator {  
 int x=5;  
  
 {  
 if (x == 4)  
 {  
 //jeśli to bd unchecked to nic nie musimy robic  
 throw new RuntimeException();  
 }  
 if (x==3)  
 {  
 //jeśli to jest checked to trzeba taki wyjątek wyrzucić ze wszystkich konstruktorow  
 throw new IOException();  
 }  
 }  
  
 public Calculator() throws IOException {  
 throw new IOException();  
 }  
}

Main

public static void main(String[] args) {  
 try {  
 new Calculator();  
 } catch (IOException ioException) {  
 ioException.printStackTrace();  
 }  
}

## Używanie asercji

Asercje – testy, które sprawdzają jakieś warunki, które muszą być Boolean

Asercje mają 3 stany:

* Failed – jeżeli warunek nie jest spełniony
* Passed – jeżeli warunek jest spełniony
* Turned off

Domyślnie asercje są wyłączone, żeby je włączyć należy w konfiguracji projektu w VM options podać „-ea”

Asercje – nie powinny być używane na produkcji, używamy **tylko** w fazie developmentu, aby pomóc w debagowaniu i znalezieniu błędu.

W fazie developmentu, używając asercji bd w stanie zobaczyć co nie działa poprawnie, ale w przypadku ifa dany fragment jesteśmy w stanie wykonać tylko jak jest on bezbłędny.

Nie zmieniają kodu, są „przezroczyste”

Calculator

public class Calculator {  
  
 public void calculate(String var) {  
 assert var != null : "Unexpected error (assert)";  
  
 //to samo co  
  
 if (var == null) {  
 throw new AssertionError("Unexpected error (if)");  
 }  
 }  
}

Main

public static void main(String[] args) {  
 new Calculator().calculate(null);  
}

konsola

Exception in thread "main" java.lang.AssertionError: Unexpected error (assert)

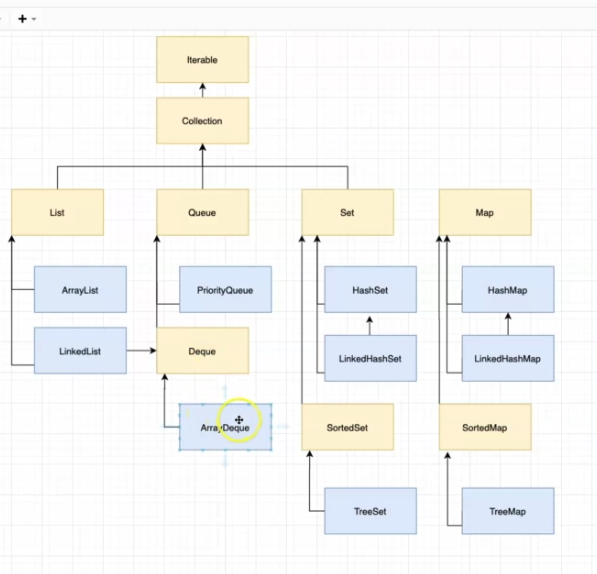
at Sekcja10\_WyjatkiIAseracje.Asercje.Calculator.calculate(Calculator.java:8)

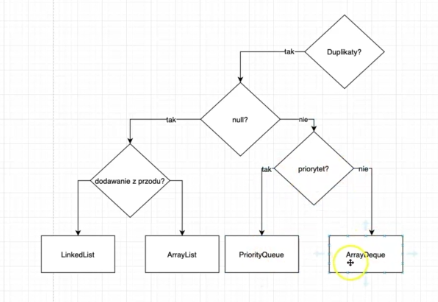
at Sekcja10\_WyjatkiIAseracje.Asercje.MainAsercje.main(MainAsercje.java:6)

Process finished with exit code 1

# Kolekcje i mapy

Niebieski - implementacje  
Żółte - interfejsy





|  |  |
| --- | --- |
| Kolekcja | Kiedy użyć |
| ArrayList | * Duplikaty * Uporządkowane (kolejność wstawiania) * Losowy dostęp do elementów * Szybkie dodawanie elementów na końcu listy |
| PriorityQueue | * Duplikaty * Uporządkowane (Comparable) * Sortowana kolejka * Szybkie dodawanie i usuwanie z przodu (HEAD) |
| ArrayDeque | * Duplikaty * Uporządkowana(wstawianie z przodu i z tyłu) * Szybkie dodawanie i usuwanie z przodu i z tyłu (HEAD, TAIL) * Kolejki FIFO, LIFO |
| LinkedList | * Duplikaty * Uporządkowana (wstawianie z przodu i z tyłu) * Szybkie dodawanie i usuwanie z przodu i z tyłu (HEAD i TAIL) * Szybkie usuwanie losowych elementów |
| HashSet | * Unikalne wartości * Szybkie usuwanie i pobieranie (hashowanie) * Nieuporządkowany |
| LinkedHashSet | * Unikalne wartości * Szybkie usuwanie i pobieranie (hashowanie) * Uporządkowana (kolejność wstawiania) |
| TreeSet | * Unikalne wartości * Szybkie usuwanie i pobieranie (brak hashowania) * Uporządkowana (comparable) |
| HashMap | * Unikalne klucze * Szybkie usuwanie i pobieranie (hashowanie) * Nieuporządkowany |
| LinkedHashMap | * Unikalne klucze * Szybkie usuwanie i pobieranie (hashowanie) * Uporządkowana (kolejność wstawiania) |
| TreeMap | * Unikalne klucze * Szybkie usuwanie i pobieranie (brak hashowania) * Uporządkowana (comparable) |

Cechy kolekcji

* Duplikacja elementów
* Możliwość dodawania null
* Uporządkowanie elementów
* Wstawianie i wyciąganie elementów
* Thread-safety

Iterowanie po listach – mają iterable więc:

for (type var :iterable){  
   
}

## ArrayList

Person

public class Person implements Comparable<Person>{  
 //Comparable - generyczny - jak nie damy typów <> to bd Object  
 private final String name;  
 private final String pesel;  
  
 public Person(String name, String pesel) {  
 this.name = name;  
 this.pesel = pesel;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public String getPesel() {  
 return pesel;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Person person = (Person) o;  
 return Objects.*equals*(name, person.name) &&  
 Objects.*equals*(pesel, person.pesel);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(name, pesel);  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "name='" + name + '\'' +  
 ", pesel='" + pesel + '\'' +  
 '}';  
 }  
  
 @Override  
 public int compareTo(Person o) {  
 return this.pesel.compareTo(o.pesel);  
 }  
}

Main

public static void main(String[] args) {  
  
 String[] array = new String[5];  
  
  
 //operator diament  
 //new - wywołujemy konstruktor, dlatego nawiasy okrągłe, od (chyba) Javy 8 nie trzeba typu w konstruktorze, JAVA sama określi go z deklaracji  
 ArrayList<String> l1 = new ArrayList<String>();  
 //lepiej chować to pod samym interfejsem List  
 //typ nie może być prymitywny  
 List<String> l2 = new ArrayList();  
  
 //można też w konstruktorze dodawać liczbę elementów, bd działać szybciej, ale z zasady listy są DYMANICZNE  
 List<Integer> l3 = new ArrayList(10);  
  
  
 //konwersja z tablicy do listy, np;.:  
 //1) przeiterować po elementach tablicy i dodać je do listy  
 //2) Collections.addAll(kolekcja, tablica)  
 //3) Arrays.asList(array) - ale lista staje się NIEMODYFIKOWALNA  
  
 Collections.*addAll*(l1, array);  
 List<String> l4 = Arrays.*asList*(array); //ale lista NIEMODYFIKOWALNA  
  
 //konwersja listy do tablicy - zwraca tablicę obiektów  
 //aby otrzymać konkretny typ - podać go należy przy wywołaniu metody  
  
 l3.toArray(new Integer[]{});  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 List<String> list = new ArrayList();  
  
 //dodawanie do listy - add - doda na końcu  
 list.add("Aa");  
 list.add("Bb");  
 list.add(0, "Cc");  
 list.add("Dd");  
  
 //usuwanie z listy  
 //remove - usunie TYLKO pierwsze wystąpienie, aby usunąć wszytskie - należy przeiterować po liście  
 list.remove(0);  
 list.remove("Aa"); //jesli nie zostanie znaleziony to nic się nie stanie (NIE poleci wyjatek)  
  
 //podstawianie  
 list.set(1, "Ee");  
  
 //sortowanie - tylko dla typów implementujących Comparable  
 Collections.*sort*(list);  
 System.*out*.println(list);  
 Collections.*reverse*(list); //odwraca kolejnosc  
  
 //czy dany element jest w liście  
 list.contains("Bb");  
  
 //rozmiar  
 list.size();  
  
 //możemy przechowywać DUPLIKATY I NULLe  
 list.add(null);  
 list.add("Bb");  
  
 //sublista  
 list.subList(0, 2);  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 List<Person> people = new ArrayList();  
 people.add(new Person("Kasia", "123"));  
 people.add(new Person("Ala", "454"));  
  
 people.remove(new Person("Kasia", "123")); //ponieważ equels jest przeciążona, nie porównuje domyślnie referencji, to usunie obiekt Kasia 123  
  
 System.*out*.println(people);  
  
 for (Person person : people) {  
 System.*out*.println(person);  
 }  
  
  
}

## PriorityQueue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5,2,6,1; add(5) |  |  |  | 5 |
| 2,6,1; add(2) |  |  | 5 | 2 |
| 6,1; add(6) |  | 6 | 5 | 2 |
| 1; add(1) | 6 | 5 | 2 | 1 |

Ostatnie miejsce – **Head** – możemy tylko ściągać „od prawej”.

Aby przeiterować używamy **Iterator**. Dla kolejek dodatkowe while.

Dodawanie: add, offer:   
Pobieranie: peek, poll  
Ściąganie: element, remowe

Elementy muszą implementować **Comparable**. Można dodawać w kodzie, nie poleci błąd, ale przy odpaleniu programu poleci ClassCastException jeśli nie bd Comparable. Można też dodać elementy bez Comparable, ale wtedy w konstruktorze kolejki musimy dodać komparator.

Implements Comparable – tylko jedna implementacja  
Implements Comparator (nowa klasa) – można wiele komparatorów

Cechy PriorityQueue

* Przechowuje duplikaty
* **Nie** można dodawać null
* Elementy są uporządkowane
* Wstawianie wg priorytetu
* Wyciąganie od głowy (**head**)

PersonComparator

public class PersonComparator implements Comparator<Person> {  
 @Override  
 public int compare(Person o1, Person o2) {  
 return o1.getPesel().compareTo(o2.getPesel());  
 }  
}

Main

public static void main(String[] args) {  
 Queue<String> q1=new PriorityQueue<>();  
  
 q1.add("Blue");  
 q1.add("Red");  
 q1.add("Black");  
  
 q1.offer("White"); //w przypadku kolejek z zadeklarowanym rozmiarem, gdy bd już całkowicie wypełniona z offer poleci false  
  
 //iterator  
 Iterator<String> it=q1.iterator();  
  
 while(it.hasNext()){  
 System.*out*.println(it.next());  
 }  
  
 //dla kolejek można też użyć while  
 while (q1.isEmpty())  
 {  
 //System.out.println(q1.peek()); //jesli bd pusta to wyjdzie null  
 System.*out*.println(q1.element()); //w przypadku braku poleci wyjątek  
 }  
  
 //komparator można od razu odwrócic, żeby sortowanie bylo przeciwne  
 Queue<Person> people=new PriorityQueue<>(new PersonComparator().reversed());  
 people.add(new Person("Adam", "443"));  
 people.add(new Person("John", "123"));  
  
 System.*out*.println(people);  
  
}

## LinkedList

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ArayList | | | | |
| add(4) | 3 | 2 | 1 |  |
|  | 3 | 2 |  | 1 |
|  |  | 3 | 2 | 1 |
|  | 4 | 3 | 2 | 1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LinkedList | | | | |
| add(4) | 3 | 2 | 1 |  |
|  |  | 3 | 2 | 1 |
|  | 4 | 3 | 2 | 1 |

LinkedLista – każdy „wagon” ma połączenie do poprzedniego i następnego

Decque – mamy dostęp od przodu (**head**) i od tyłu (**tail**)

Kiedy co stosować

* ArrayList – losowy dostęp do elementów, w środku tablica
* LinkedList – w środku podwójne wiązanie za pomocą węzłów

Cechy LinkedList

* Przechowuje duplikaty
* **Nie** można dodawać nulli
* Elementy są uporządkowane (kolejność wstawiania, podwójne łączenie)
* Wstawianie i wyciąganie elementów na początku i na końcu jest szybkie
* Usuwanie elementu jest wydajniejsze niż w ArrayList

Main

public static void main(String[] args) {  
 //przy deklaracji można używać: List/Queue/Deque  
 Deque<String> d=new LinkedList<>();  
  
 d.addFirst("Black");  
 d.addFirst("Red");  
 d.addLast("Lila");  
  
 System.*out*.println(d); //[Red, Black, Lila]  
}

## ArrayDeque

Cechy ArrayDeque (w środku taki sam mechanizm jak ArrayList), dodane w Java6, wydajniej niż ArrayDeque wydajniej niż ArrayList

* Przechowuje duplikaty
* **Nie** można dodwać nulli
* elementy są uporządkowane (kolejność wstawiania, z przodu albo z tyłu)
* wstawianie i wyciąganie elementów tylko na początku i końcu
* usuwanie elementu tylko na początku i końcu kolejki

Main

public static void main(String[] args) {  
 Deque<String> fifo=new ArrayDeque<>();  
 Deque<String> lifo=new ArrayDeque<>();  
  
 //FIFO - first in, first out  
 fifo.addFirst("one");  
 fifo.addFirst("two");  
  
 System.*out*.println(fifo.pollLast()); //one  
  
  
 //LIFO - last in, first out - STOS  
 fifo.addFirst("one");  
 fifo.addFirst("two");  
  
 System.*out*.println(fifo.pollFirst()); //two  
}

## Hashowanie i kontrakt Equals-Hashcode

Kontrakt equals

* **zwrotna** = x.equals(x)==true
* **symetryczna** – jeżeli x.equals(y)==true, to y.equals(x)==true
* **przechodnia** – x.equals(y)=true oraz y.equals(z)=true, to x.equals(z)=true
* **spójna** – wielokrotne wywoływanie x.equals(y) musi zawsze dawać taką samą wartość boolean
* zawsze x.equals(null)=false

Kontrakt HashCode

* zawsze wywołanie metody **hashCode** na tym samym obiekcie musi kończyć się zwróceniem tej samej liczby całkowitej
* Jeżeli dwa obiekty są **równe** (wg metody **equals**), to ich hashCode również **musi** być równy
* Jeżeli obiekty są **różne** (wg metody **equals**), to ich **hashCode może** być równy, jednak ze względów wydajnościowych powinno to być unikalne.

Dobre praktyki

* Użycie tych samych pól w equals i hashCode
* Jednoznaczne identyfikatory (np. pesel, NIP, regon, albo para pól, np.: waluta i ilość)
* Nie używać identyfikatorów bazodanowych
* Null check

Main

public static void main(String[] args) {  
 List<String> list =new ArrayList<>();  
  
 list.add("Black");  
 list.add("Pink");  
 list.add("White");  
 list.add("Red");  
 list.add("Blue");  
 list.add("Green");  
 list.add("Orange");  
 list.add("Gray");  
  
 System.*out*.println(list.contains("Gray")); //8 elemetów(iteracji) + 8 sprawdzeń equals  
  
 //hashowanie  
  
 List<String>[] arrayOfLists=new ArrayList[10];  
  
 //ile liter taki indeks  
 arrayOfLists[2]= Arrays.*asList*("Red");  
 arrayOfLists[3]= Arrays.*asList*("Pink", "Blue", "Gray");  
 arrayOfLists[4]= Arrays.*asList*("Black", "White","Green");  
 arrayOfLists[5]= Arrays.*asList*("Orange");  
  
 System.*out*.println(arrayOfLists["Gray".length()].contains("Gray")); //3 iteracje + 3 sprawdzenia equals  
}

HashColor

public class HashColor {  
 private final String value;  
  
 public HashColor(String value) {  
 this.value = value;  
 }  
  
 @Override  
 public int hashCode() {  
 return value.length();  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) //sprawdzenia "zwrotna"  
 {  
 return true;  
 }  
 if (o == null) {  
 return false;  
 }  
 if(!(o instanceof HashColor))  
 {  
 return false;  
 }  
 return this.value.equals(((HashColor) o).value);  
  
  
 //return new Random().nextBoolean(); //ZABRONIONE - musi być "spójne"  
 }  
}

Money

public class Money {  
 protected final Double amount;  
 protected final String currency;  
  
 public Money(Double amount, String currency) {  
 this.amount = amount;  
 this.currency = currency;  
 }  
  
  
 //wygenerowane automatycznie alt+insert  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Money money = (Money) o;  
 return amount.equals(money.amount) &&  
 currency.equals(money.currency);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(amount, currency);  
 }  
}

## HashSet

Cechy HashSet

* Przechowuje unikalne elementy
* Można dodawać nulle
* Elementy **nie są** uporządkowane
* Nie ma metody get, trzeba iterować
* Wydajność wyciągania i usuwania elementów zależy od **hashcode**

Należy zawsze nadpisywać obie metody – equals i hashCode

Main

public static void main(String[] args) {  
 Set<String> set = new HashSet<>();  
  
 set.add("Black");  
 set.add("Pink");  
 set.add("White");  
 set.add("Red");  
 set.add("Blue");  
 set.add("Green");  
 set.add("Orange");  
 set.add("Gray");  
 set.add("Gray"); // w srodku tej metody wywolany zostanie contains > poleci true, wiec nie zostanie dodany  
  
 //dla stringów domyslny hashcode unikalny dla kazdego stringa - każdy ma swój bucket  
  
 System.*out*.println(set.contains("Gray")); //1 operacja (wejscie do odpowiedniego bucketa) + 1 sprawdzenie equals (w tym buckecie)  
   
 //iterowanie po hashSet  
 //for  
 for (String string : set) {  
 System.*out*.println(string);  
 }  
  
 //iterator  
 Iterator<String> it = set.iterator();  
 while (!it.hasNext()) {  
 it.next();  
 }  
  
}

Book

public class Book {  
 private final String title;  
 private final String isbn;  
  
 public Book(String title, String isbn) {  
 this.title = title;  
 this.isbn = isbn;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Book book = (Book) o;  
 return Objects.*equals*(isbn, book.isbn);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(isbn);  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public String getIsbn() {  
 return isbn;  
 }  
}

Library

public class Library {  
  
 private final Set<Book> avaliable = new HashSet<>();  
 private final Set<Book> borrowed = new HashSet<>();  
  
 public Library(List<Book> books) {  
 avaliable.addAll(books);  
 }  
  
 public void borrowABook(String isbn) {  
 Optional<Book> bookOpt = findBook(avaliable, isbn);  
 if (bookOpt.isPresent()) {  
 borrowed.add(bookOpt.get());  
 avaliable.remove(bookOpt.get());  
 System.*out*.println("Książka " + bookOpt.get().getTitle() + " została wypożyczona");  
 }  
  
 }  
  
 public void checkABook(String isbn) {  
 Optional<Book> bookOpt = findBook(avaliable, isbn);  
 Optional<Book> borrowedOpt = findBook(borrowed, isbn);  
  
 if (bookOpt.isEmpty()) {  
 if (borrowedOpt.isEmpty()) {  
 System.*out*.println("Brak ksiazki o ISBN: " + isbn);  
 } else {  
 System.*out*.println("Książka " + borrowedOpt.get().getTitle() + " niedostepna");  
 }  
 } else {  
 System.*out*.println("Książka " + bookOpt.get().getTitle() + " dostepna");  
 }  
  
 }  
  
 public void returnABook(String isbn) {  
 Optional<Book> bookOpt = findBook(borrowed, isbn);  
 if (bookOpt.isPresent()) {  
 borrowed.remove(bookOpt.get());  
 avaliable.add(bookOpt.get());  
 System.*out*.println("Książka " + bookOpt.get().getTitle() + " zostala zwrocona");  
 }  
 }  
  
 private Optional<Book> findBook(Set<Book> set, String isbn) {  
 for (Book book : set) {  
 if (book.getIsbn().equals(isbn)) {  
 return Optional.*of*(book);  
 }  
 }  
 return Optional.*empty*();  
 }  
}

MainLibrary

public static void main(String[] args) {  
 */\*\*  
 \* Napisz implementację bilbioteki. Wypożyczjący mogą sprawdzić czy książka o danym isbn istnieje.  
 \* Mogą ją wypożyczyć i zwrócić. Biblioteka powinna pamiętać również wypożyczone książki.  
 \*/* List<Book> books = Arrays.*asList*(  
 new Book("Potop", "1WR"),  
 new Book("Chłopki", "2ASD"),  
 new Book("Zbrodnia i Kara", "3DSA"),  
 new Book("Lalka", "4ASF")  
 );  
  
 //System.out.println((new Book("x", "qwe").equals(new Book("x", "qwe"))));  
 Library library = new Library(books);  
 library.checkABook("5ASD");  
 library.checkABook("3DSA");  
 library.borrowABook("3DSA");  
 library.checkABook("3DSA");  
 library.returnABook("3DSA");  
  
// Brak książki o isbn 5ASD.  
// Książka Zbrodnia i Kara dostępna.  
// Książka Zbrodnia i Kara została wypożyczona.  
// Książka niedostępna.  
// Książka Zbrodnia i Kara została zwrócona.  
 }

## LinkedHashSet

Cechy LinkedHashSet

* Przechowuje unikalne elementy
* Można dodawać nulle (tylko jeden)
* Elementy **są** uporządkowane (kolejność dodawania)
* Nie ma metody get, trzeba iterować
* Wydajność usuwania elementów zależy od **hascode**

Main

public static void main(String[] args) {  
 Set<String> set = new LinkedHashSet<>();  
  
 //pod spodem - zachowana kolejnosc bucketow  
 set.add("Black");  
 set.add("Red");  
  
 for (String color : set) {  
 System.*out*.println(color);  
 }  
}

WebinarManager

public class WebinarManager {  
  
 private final Set<String> list = new LinkedHashSet<>();  
  
 public void signUp(String emailAddress) {  
 list.add(emailAddress);  
 System.*out*.println(emailAddress + "zapisany");  
 }  
  
 public void resign(String emailAddress) {  
 if (list.contains(emailAddress)) {  
 list.remove(emailAddress);  
 System.*out*.println(emailAddress + "wypisany");  
 }  
 }  
  
 public void printFirstFive() {  
 Set<String> top5 = new LinkedHashSet<>();  
 int i = 0;  
 for (String email : list) {  
 i++;  
 top5.add(email);  
 if (i >= 5) {  
 break;  
 }  
 }  
 System.*out*.println("Top5: " + top5);  
 }  
  
}

MainWebinar

public static void main(String[] args) {  
 */\*\*  
 \* Napisz implementację managera webinara z ograniczoną liczbą miejsc.  
 \* Decyduje kolejność zgłoszeń. Dwie osoby (adresy email) nie mogą zapisać się na ten sam webinar.  
 \* Na koniec wydrukuj na ekran 5 pierwszych osób, które zapisały się na webinar.  
 \*/* WebinarManager manager = new WebinarManager();  
 manager.signUp("john@");  
 manager.signUp("jane@");  
 manager.signUp("cindy@");  
 manager.signUp("steve@");  
 manager.resign("jane@");  
 manager.signUp("chris@");  
 manager.signUp("paul@");  
 manager.signUp("james@");  
  
 manager.printFirstFive();  
  
// john@ zapisany.  
// jane@ zapisany.  
// cindy@ zapisany.  
// steve@ zapisany.  
// jane@ wypisany.  
// chris@ zapisany.  
// paul@ zapisany.  
// james@ zapisany.  
// Top 5: [john@,cindy@,steve@,chris@,paul@]  
  
 }

## TreeSet

Cechy TreeSet

* Przechowuje unikalne elementy
* Nie używa hashowania
* **Nie** można dodawać nulli (przy dodawaniu compareTo (żeby nie było duplikatów))
* Elementy **są** uporządkowane (kolejność wg Comparable albo Comparatora)
* Nie ma metody get, trzeba iterować
* Wydajność usuwania elementów **nie** zależy od **hashcode**

Main

public static void main(String[] args) {  
  
 //implementacja treeSet NIE UŻYWA HASHOWANIA  
 //obiekty muszą implementowąc Comparable lub musi być Comparator  
  
 Set<String> set = new TreeSet<>();  
  
 set.add("Pink");  
 set.add("Black");  
 set.add("Green");  
  
 System.*out*.println(set); //[Black, Green, Pink]  
 //dla zwyklego Seta nie ma first, last  
 //System.out.println(set.first());  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
 //TreeSet implementuje: NavigableSet i SortedSet - mają ciekawe metody  
  
 NavigableSet<String> colors = new TreeSet<>();  
  
 colors.add("Pink");  
 colors.add("Black");  
 colors.add("Green");  
 colors.add("Violet");  
  
 System.*out*.println(colors.higher("B")); //pierwszy "większy" od "B" - Black  
  
 System.*out*.println(colors.higher("Black")); //większy od - Green  
 System.*out*.println(colors.ceiling("Black")); //większy lub równy - Black  
  
 System.*out*.println(colors.lower("H")); //Green  
 System.*out*.println(colors.subSet("Black", "Pink")); //[Black, Green]  
 System.*out*.println(colors.subSet("B", "P")); //[Black, Green]  
 System.*out*.println(colors.headSet("C")); //wszystkie przed C, [Black]  
 System.*out*.println(colors.tailSet("G")); //[Green, Pink, Violet]  
  
 //nie ma metody get, ale są first i last  
 System.*out*.println(colors.first());  
 System.*out*.println(colors.last());  
  
}

MainZadanie

public class TreeSetZadanie {  
 static class Application implements Comparable<Application> {  
 final String name;  
 final Integer points;  
  
 Application(String name, Integer points) {  
 this.name = name;  
 this.points = points;  
 }  
  
 @Override  
 public String toString() {  
 return "\n" + name + " " + points;  
 }  
  
 @Override  
 public int compareTo(Application o) {  
 if (o.points.compareTo(this.points) != 0) { //zeby punkty byly malejaco  
 return o.points.compareTo(this.points);  
 }  
 return this.name.compareTo(o.name);  
 }  
 }  
  
 static class NameComparator implements Comparator<Application> {  
  
 @Override  
 public int compare(Application o1, Application o2) {  
 return o1.name.compareTo(o2.name);  
 }  
 }  
  
 static class ApplicationList extends TreeSet<Application> {  
  
 void printList() {  
 SortedSet<Application> sorted = this.headSet(new Application("", 200));  
 NavigableSet<Application> subSetK = new TreeSet<>(new NameComparator());  
 subSetK.addAll(sorted);  
 System.*out*.println(subSetK.subSet(new Application("K", 0), new Application("L", 0)));  
 }  
 }  
  
 @SuppressWarnings("SpellCheckingInspection")  
 public static void main(String[] args) {  
 */\*\*  
 \* Napisz implementację listy zgłoszeń na studia dzienne.  
 \* O kolejnosci decyduje liczba punktów.  
 \* Wyświetl listę wszystkich aplikantów zaczynających się na literę K, spośród mających conajmniej 200 pkt.  
 \* Stwórz klasę ApplicationList, która dziedziczy po odpowiedniej kolekcji.  
 \*/* ApplicationList applicationList = new ApplicationList();  
 applicationList.add(new Application("Kowalski", 100));  
 applicationList.add(new Application("Zieliński", 299));  
 applicationList.add(new Application("Kania", 501));  
 applicationList.add(new Application("Orzeł", 500));  
 applicationList.add(new Application("Kwiatkowiski", 500));  
 applicationList.printList();  
  
 // Kania  
 // Kwiatkowiski  
  
 }  
  
}

## HashMap

Cechy HashMap

* Przechowują pary klucz-wartość
* Klucze są unikalne
* Używa hashowania do kluczy
* Można dodać jeden null jako klucz
* Elementy **nie** są uporządkowane
* Jest metoda get, pobieranie po kluczu
* Wydajność pobierania i usuwania po kluczu zależy od **hashcode**

IntegerKey

public class IntegerKey {  
  
 private final Integer value;  
  
 public IntegerKey(Integer value) {  
 this.value = value;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 IntegerKey that = (IntegerKey) o;  
 return Objects.*equals*(value, that.value);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(value);  
 }  
  
 @Override  
 public String toString() {  
 return value.toString();  
 }  
}

Person

public class Person {  
 private final String name;  
 private final Gender gender;  
  
 public Person(String name, Gender gender) {  
 this.name = name;  
 this.gender = gender;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public Gender getGender() {  
 return gender;  
 }  
  
 @Override  
 public String toString() {  
 return name;  
 }  
}

HashMapMain

public static void main(String[] args) {  
 Map<String, List<String>> map = new HashMap<>();  
  
 map.put("B", Arrays.*asList*("Black", "Brown"));  
 map.put("P", Arrays.*asList*("Pink"));  
 map.put(null, new ArrayList<>());  
 map.put("B", Arrays.*asList*("Black", "Brown")); //wartość mapy dla klucza "B" zostanie nappisana nową listą  
  
 System.*out*.println(map);  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 Map<IntegerKey, String> map2 = new HashMap<>();  
 IntegerKey key1 = new IntegerKey(1);  
 IntegerKey key2 = new IntegerKey(1); //tu ma byc 1  
 IntegerKey key3 = new IntegerKey(3);  
  
 map2.put(key1, "Black");  
 map2.put(key2, "Blue");  
 map2.put(key3, "Lila");  
 //ponieważ nie zostały nadpisane hashcode to domyślnie referencja do obiektów kluczy i dodane zostaną obie pary  
 //jeśli nadpiszemy hashcode i uzależnimy od value, to zostanie dodany key1 i potem zostanie napisany, czyli bd jedna pozycja, ktora jako wartosc bd miala "Blue"  
  
 System.*out*.println(map2);  
  
 //nie można iterowac po mapie, więc  
 final Set<Map.Entry<IntegerKey, String>> entries = map2.entrySet();  
  
 for (Map.Entry<IntegerKey, String> entry : entries) {  
 entry.getKey();  
 }  
  
 System.*out*.println(map2.get(key1));  
 System.*out*.println("Is map contains key1: " + map2.containsKey(key1)); //true  
 System.*out*.println("Is map contains key2: " + map2.containsKey(key2)); //true  
 System.*out*.println("Is map contains 'Blue' value: " + map2.containsValue("Blue")); //true  
  
 System.*out*.println(map2.remove(key1)); //Blue  
 System.*out*.println(map2.remove(key3, "Violet")); //false  
 System.*out*.println(map2.remove(key3, "Lila")); //true  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 Set<Person> people = new HashSet<>();  
 people.add(new Person("Nowak", Gender.*MALE*));  
 people.add(new Person("Zielinska", Gender.*FEMALE*));  
 people.add(new Person("Kowalski", Gender.*MALE*));  
  
 Map<Gender, List<Person>> peopleMap = new HashMap<>();  
  
 for (Person person : people) {  
 if (peopleMap.containsKey(person.getGender())) {  
 peopleMap.get(person.getGender()).add(person);  
 } else {  
 List<Person> peoplePerGender = new ArrayList<>();  
 peoplePerGender.add(person);  
 peopleMap.put(person.getGender(), peoplePerGender);  
 }  
 }  
  
 System.*out*.println(peopleMap); //{FEMALE=[Zielinska], MALE=[Nowak, Kowalski]}  
  
 //iterowanie przez same klucze  
 for (Gender gender : peopleMap.keySet()) {  
 System.*out*.println(gender); // 1 iteracja "FEMALE", 2 iteracja "MALE"  
 }  
  
 //w Javie jest prostszy sposob na wygenerowanie takiej mapy - za pomocą streamów  
   
}

RegisterZadanie

public class RegisterZadanie {  
 static class RegistrationManager {  
 private final Map<String, Set<String>> map = new HashMap<>();  
  
 public void register(String className, String studentName) {  
 if (map.containsKey(className)) {  
 map.get(className).add(studentName);  
 } else {  
 Set<String> names = new TreeSet<>();  
 names.add(studentName);  
 map.put(className, names);  
 }  
 }  
  
 public void printStudents(String className) {  
 System.*out*.println(map.get(className));  
 }  
 }  
  
 @SuppressWarnings("SpellCheckingInspection")  
 public static void main(String[] args) {  
 */\*\*  
 \* Napisz implementację systemu zapisów na wykłady.  
 \* System ma błąd i pozwala, żeby student mógł zapisać się wielokrotnie na ten sam wykład.  
 \* Zapisy muszą działać w taki sposób, żeby dostęp do listy wszystkich studentów dla danego wykładu był szybki,  
 \* a lista zwracanych studentów unikalna i posortowana alfabetycznie.  
 \*/* RegistrationManager manager = new RegistrationManager();  
 manager.register("Analiza matematyczna", "Kowalski");  
 manager.register("Analiza matematyczna", "Zieliński");  
 manager.register("Algebra", "Kowalski");  
 manager.register("Analiza matematyczna", "Kowalski");  
 manager.register("Analiza matematyczna", "Cebula");  
  
 manager.printStudents("Analiza matematyczna");  
  
 // [Cebula, Kowalski, Zieliński]  
 }  
  
}

## LinkedHashMap

Cechy LinkedHashMap

* Przechowuje kolekcję par klucz-wartość
* Klucze są unikalne
* Używa hashowania do kluczy
* Można dodać tylko jeden null jako klucz
* Elementy są uporządkowane (wg kolejności, albo wg najrzadziej używanego)
* Wydajność pobierania i usuwania po kluczu zależy od hashcode

Cache

public class Cache extends LinkedHashMap<String, Object> {  
  
 private final int maxElements;  
  
 public Cache(int maxElements) {  
 this.maxElements = maxElements;  
 }  
  
 //wskazuje najrzadziej używane, wywoływana po każdym put  
 @Override  
 protected boolean removeEldestEntry(Map.Entry<String, Object> eldest) {  
 return size() > maxElements;  
 }  
}

LinkedHashMapMain

public static void main(String[] args) {  
  
 Map<String, String> map = new java.util.LinkedHashMap<>(10, 0.75f, true); //taki konstruktor uporządkuje mapę od najczęściej do najrzadziej używanych  
  
 map.put("Blue", null);  
 map.put("Red", null);  
 map.put("Black", null);  
  
 System.*out*.println(map); //{Blue=null, Red=null, Black=null}  
  
 map.get("Red");  
 System.*out*.println(map); //{Blue=null, Black=null, Red=null} - używaliśmy "Red", więc idzie na koniec  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 Cache cache = new Cache(2);  
  
 cache.put("Blue", null);  
 cache.put("Red", null);  
 cache.put("Black", null);  
  
 System.*out*.println(cache); //{Red=null, Black=null}, bo zadeklarowano pojemność =2, czyli najstarszy - "Blue" został usunięty  
  
}

LinkedHashMapZadanie

static class BrowserHistory {  
 private final Cache cache = new Cache(3);  
  
 public void browse(String site) {  
 cache.put(site, null);  
 }  
  
 public void print() {  
 ArrayDeque<String> deque = new ArrayDeque<>();  
 for (String key : cache.keySet()) {  
 deque.addFirst(key);  
 }  
 System.*out*.println(deque);  
 }  
}  
  
@SuppressWarnings("SpellCheckingInspection")  
public static void main(String[] args) {  
 */\*\*  
 \* Napisz implementację historii przeglądarki, pamiętającą 3 ostatnich przeglądanych stron internetowych.  
 \* Wyświetl historię przeglądarki, ale w kolejnosci od ostatnio przegladanej strony.  
 \*/* BrowserHistory manager = new BrowserHistory();  
 manager.browse("onet.pl");  
 manager.browse("interia.pl");  
 manager.browse("gazeta.pl");  
 manager.browse("onet.pl");  
 manager.browse("pudelek.pl");  
 manager.browse("plotka.pl");  
  
 manager.print();  
  
 // [plotka.pl, pudelek.pl, onet.pl]  
}

## TreeMap

Cechy TreeMap

* Przechowuje kolekcję klucz-wartość
* Klucze są unikalne
* Nie używa hashowania dla kluczy
* Nie można dodać nulla jako klucz, ale jako wartość tak
* Elementy są uporządkowane (domyślnie rosnąco, wg naturalnej kolejności)
* Pod spodem TreeMap używa algorytmu „drzewo czerwono-czarne”

Person

public class Person implements Comparable<Person> {  
 private final String name;  
 private final String pesel;  
  
 public Person(String name, String pesel) {  
 this.name = name;  
 this.pesel = pesel;  
 }  
  
 @Override  
 public String toString() {  
 return name + "; " + pesel;  
 }  
  
 @Override  
 public int compareTo(Person o) {  
 return this.pesel.compareTo(o.pesel);  
 }  
}

TreeMapMain

public static void main(String[] args) {  
 //sortowanie po kluczach  
  
 NavigableMap<String, String> map = new java.util.TreeMap<>();  
  
 map.put("Red", null);  
 map.put("Blue", null);  
 map.put("Pink", null);  
  
 System.*out*.println(map); //{Blue=null, Pink=null, Red=null}  
  
 //nie ma hashowania  
  
 //-------------------------------------------------------  
  
 NavigableMap<Person, String> people=new java.util.TreeMap<>();  
  
 people.put(new Person("Adam", "123"),null);  
 people.put(new Person("Barbara", "123"),null);  
 people.put(new Person("Jan", "456"),null);  
  
 System.*out*.println(people); //{Adam; 123=null, Jan; 456=null}, bo w person compare to zalezy od peselu - element nie został dodany, bo już taki element istnieje  
  
}

TreeMapZadanie

public class TreeMapZadanie {  
 static class Dictionary {  
  
 NavigableMap<String, NavigableSet<String>> dict = new TreeMap<>();  
  
 public Dictionary(Set<String> words) {  
  
 for (String word : words) {  
 String letter = word.substring(0, 1).toLowerCase();  
 if (dict.containsKey(letter)) {  
 dict.get(letter).add(word.toLowerCase());  
 } else {  
 NavigableSet<String> set = new TreeSet<>(Comparator.*reverseOrder*());  
 set.add(word.toLowerCase());  
 dict.put(letter, set);  
 }  
 }  
 }  
  
 public void printWordsByLetter(String letter) {  
 System.*out*.println(dict.get(letter.toLowerCase()));  
 }  
 }  
  
 @SuppressWarnings("SpellCheckingInspection")  
 public static void main(String[] args) {  
 */\*\*  
 \* Napisz implementację prostego słownika.  
 \* Słownik powinien być posortowany alfabetycznie.  
 \* Zależy nam, na szybkim dostępie do wszystkich słów dla danej litery.  
 \* Napisz metodę drukującą wszystkie słowa,  
 \* zaczynające się od danej litery, w odwrotnej kolejności.  
 \* Uwaga na małe i duże litery!  
 \*/* Set<String> words = Set.*of*(  
 "kopytko",  
 "Kapusta",  
 "ananas",  
 "pomarańcza",  
 "Kiwi",  
 "kalafior"  
 );  
 Dictionary manager = new Dictionary(words);  
 manager.printWordsByLetter("K");  
  
 // [kopytko, kiwi, kapusta, kalafior]  
 }  
  
}

# Anotacje i refrelekcja

## Tworzenie anotacji

Anotacje

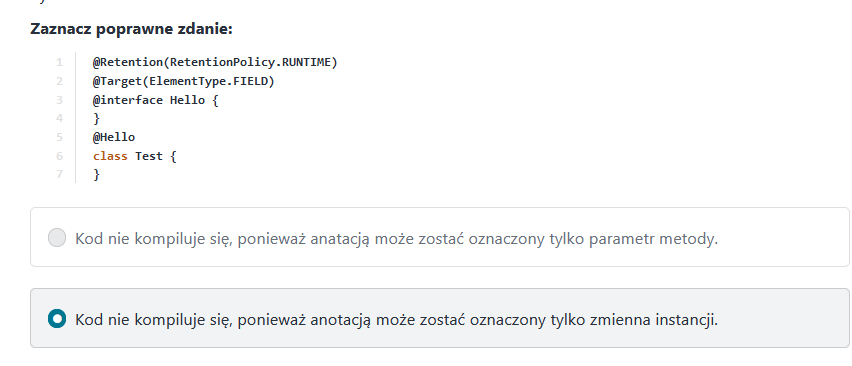
* Mechanizm na dodawanie metadanych do kodu źródłowego
* Oznaczania nie mają wpływu na działanie programu

Metody w anotacjach

* Muszą być bezparametrow
* Nie mogą wyrzucać błędów (wyjątkow)
* Zwracany typ to: **typ prymitywny, String, Class, enum** albo **tablice** tych typów
* Wartość domyślna **nie** może być **null**

@Retention: RetentionPolicy – enum: SOURCE, CLASS, RUNTIME

@Target: ElementType – enum: TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE, ANNOTATION\_TYPE, PACKAGE, TYPE\_PARAMETERM TYPE\_USE, MODULE



Calculator

public class Calculator {  
  
 @Author(name = "Kasia Baluch")  
 public Calculator(){}  
  
 @Author  
 @LogInutParams  
 public double calculate(double income, double debt)  
 {  
 return income-debt;  
 }  
}

Person

@ExactlyOneNotNull({"nip", "pesel"})  
public class Person {  
 @Regex(value = "[a-zA-Z]\*", message = "Field name has invalid regex") //jak jest jedna wlasciwosc nie trzeba jej nazwy pisać, jak więcej trzeba nazwę i wartośc  
 @NotNull(message = "Field name must be not null")  
 private String name;  
 private String nip;  
 private String pesel;  
  
 public Person(String name, String nip, String pesel) {  
 this.name = name;  
 this.nip = nip;  
 this.pesel = pesel;  
 }  
}

Author

@Retention( RetentionPolicy.*RUNTIME*)  
@Target(value = {ElementType.*CONSTRUCTOR*, ElementType.*METHOD*})  
public @interface Author {  
 String *DEFAULT\_NAME*="Unknown2"; //tylko stałe, muszą byc zainicjalizowane, stałą można użyć tylko w ciele anitacji  
  
 String name() default "Unknown";  
  
}

Regex

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*FIELD*)  
public @interface Regex {  
 String value();  
 String message();  
}

NotNull

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*FIELD*)  
public @interface NotNull {  
 String message();  
}

ExactlyOneNotNull

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*TYPE*)  
public @interface ExactlyOneNotNull {  
 String[] value();  
}

LogInputParams

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface LogInutParams {  
}

## Procesowanie anotacji przy użyciu reflekcji

ObjectValidator

public class ObjectValidator {  
 public static void validate(Object object) throws IllegalAccessException {  
 final Class oClass = object.getClass();  
  
 //anotacje na klasie  
 *processExactlyOneNotNull*(object, oClass);  
  
  
 //anotacje na polach  
 //getDeclaredFields pobierze też pola prywatne, samo getFields tylko publiczne  
  
 for (Field field : oClass.getDeclaredFields()) {  
 field.setAccessible(true); //dostep do pol prywatnych  
  
 *processNotNull*(object, field);  
 *processRegex*(object, field);  
 }  
 }  
  
 private static void processExactlyOneNotNull(Object object, Class oClass) throws IllegalAccessException {  
 if (oClass.isAnnotationPresent(ExactlyOneNotNull.class)) {  
 ExactlyOneNotNull annotation = (ExactlyOneNotNull) oClass.getAnnotation(ExactlyOneNotNull.class);  
 List<Field> fields = new ArrayList<>();  
  
 for (String fieldName : annotation.value()) {  
 for (Field field : oClass.getDeclaredFields()) {  
 field.setAccessible(true);  
  
 if (field.getName().equals(fieldName)) {  
 fields.add(field);  
 }  
 }  
 }  
  
 int numOfNotNullFields = 0;  
 for (Field field : fields) {  
  
 if (field.get(object) != null) {  
 numOfNotNullFields++;  
 }  
 }  
 if (numOfNotNullFields != 1) {  
 throw new IllegalAccessException("Exactly one field must be not null");  
 }  
 }  
 }  
  
 private static void processRegex(Object object, Field field) throws IllegalAccessException {  
 if (field.isAnnotationPresent(Regex.class)) {  
 Regex annotation = field.getAnnotation(Regex.class);  
 String fieldValue = (String) field.get(object);  
  
 if (!fieldValue.matches(annotation.value())) {  
 throw new IllegalArgumentException(annotation.message());  
 }  
 }  
 }  
  
  
 private static void processNotNull(Object object, Field field) throws IllegalAccessException {  
 if (field.isAnnotationPresent(NotNull.class)) {  
 NotNull annotation = field.getAnnotation(NotNull.class);  
 String fieldValue = (String) field.get(object);  
  
 if (fieldValue == null) {  
 throw new IllegalArgumentException(annotation.message());  
 }  
 }  
 }  
}

Main

public static void main(String[] args) throws IllegalAccessException {  
  
 Person person1 = new Person("Kasia", "123", null);  
 ObjectValidator.*validate*(person1); //nic nie wyrzuci, bo regex pasuje, dodkladnie jeden nie jest nullem i name nie jest nullem  
  
 Person person2 = new Person(null, null, null);  
 ObjectValidator.*validate*(person2); //poleci pierwszy możliwy wyjątek, ale: regex nie pasuje, więcej niż jedno pole null i name jest null  
  
}

## Dynamiczne Proxy

ICalculaor

public interface ICalculator {  
  
 double calculate(double income, double debt);  
 double calculate2(double income, double debt);  
}

LoggingInvocationHandler

public class LoggingInvocationHandler implements InvocationHandler {  
  
 private final Object targetObject;  
  
 private final Map<String, Method> methods = new HashMap<>();  
  
 public LoggingInvocationHandler(Calculator targetObject) {  
 this.targetObject = targetObject;  
 for (Method method : targetObject.getClass().getMethods()) {  
 if (method.isAnnotationPresent(LogInutParams.class)) {  
 methods.put(method.getName(), method);  
 }  
 }  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 if (methods.containsKey(method.getName())) {  
 for (int i = 0; i < method.getParameters().length; i++) {  
 System.*out*.println(String.*format*("Param %s = %s", method.getParameters()[i].getName(), args[i]));  
 }  
 }  
  
 return method.invoke(targetObject, args);  
 }  
}

ProxyCreator

public class ProxyCreator {  
  
 public static ICalculator createProxy(Calculator calculator) {  
 InvocationHandler invocationHandler = new LoggingInvocationHandler(calculator);  
   
 return (ICalculator) Proxy.*newProxyInstance*(ProxyCreator.class.getClassLoader(), new Class[]{ICalculator.class}, invocationHandler);  
 }  
}

Calculator

public class Calculator implements ICalculator {  
  
 @Author(name = "Kasia Baluch")  
 public Calculator() {  
 }  
   
 @Override  
 @Author  
 @LogInutParams  
 public double calculate(double income, double debt) {  
  
 return income - debt;  
 }  
  
 @Override  
 public double calculate2(double income, double debt) {  
  
 return income - debt;  
 }  
}

Main

public static void main(String[] args) {  
 ICalculator iCalculator = ProxyCreator.*createProxy*(new Calculator());  
  
 iCalculator.calculate(10.0, 10.0);  
 iCalculator.calculate2(20.0, 20.0);  
}

Konsola

Param arg0 = 10.0

Param arg1 = 10.0

## Dependency Injection

ServiceA

public class ServiceA {  
  
 @Inject  
 private ServiceD serviceD;  
 @Inject  
 private ServiceB serviceB;  
  
 public int getInt() {  
 return serviceB.getInt() + serviceD.getInt();  
 }  
}

ServiceB

public class ServiceB {  
  
 @Inject  
 private ServiceD serviceD;  
 @Inject  
 private ServiceC serviceC;  
   
 public int getInt() {  
 return serviceC.getInt() + serviceD.getInt();  
 }  
}

ServiceC

public class ServiceC {  
  
 @Inject  
 private ServiceE serviceE;  
  
 public int getInt() {  
 return serviceE.getInt() + 1;  
 }  
}

ServiceD

public class ServiceD {  
 public int getInt() {  
 return 2;  
 }  
}

ServiceE

public class ServiceE {  
  
 public int getInt() {  
 return 1;  
 }  
}

Inject

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*FIELD*)  
public @interface Inject {  
}

ClassInnitializer

public class ClassInitializer {  
  
 private Map<String, Object> instances = new HashMap<>();  
  
 public Object createInstance(Class classToInstantiate) throws Exception {  
 return getInstance(classToInstantiate);  
 }  
  
 private Object getInstance(Class classToInstantiate) throws Exception {  
  
 System.*out*.println("Creating instance of class: " + classToInstantiate.getSimpleName());  
  
 if (instances.containsKey(classToInstantiate.getCanonicalName())) {  
 System.*out*.println("Instance of class " + classToInstantiate.getSimpleName() + " returned from cache.");  
 return instances.get(classToInstantiate.getCanonicalName());  
 }  
 Object instance = classToInstantiate.getConstructor().newInstance();  
  
 for (Field field : classToInstantiate.getDeclaredFields()) {  
 field.setAccessible(true);  
 if (field.isAnnotationPresent(Inject.class)) {  
 System.*out*.println("Dependency " + field.getName() + " found in classs " + classToInstantiate.getSimpleName());  
  
 Object dependencyInstance = getInstance(field.getType());  
 Field dependencyField = instance.getClass().getDeclaredField(field.getName());  
 dependencyField.setAccessible(true);  
 dependencyField.set(instance, dependencyInstance);  
 System.*out*.println("Dependency " + field.getName() + " injected into class " + classToInstantiate.getSimpleName());  
 }  
 }  
  
 System.*out*.println("Instance of class: " + classToInstantiate.getSimpleName() + " created");  
 instances.put(classToInstantiate.getCanonicalName(), instance);  
 return instance;  
 }  
}

Main

public static void main(String[] args) throws Exception {  
  
 ClassInitializer classInitializer = new ClassInitializer();  
 ServiceA serviceA = (ServiceA) classInitializer.createInstance(ServiceA.class);  
}

Konsola

Creating instance of class: ServiceA

Dependency serviceD found in classs ServiceA

Creating instance of class: ServiceD

Instance of class: ServiceD created

Dependency serviceD injected into class ServiceA

Dependency serviceB found in classs ServiceA

Creating instance of class: ServiceB

Dependency serviceD found in classs ServiceB

Creating instance of class: ServiceD

Instance of class ServiceD returned from cache.

Dependency serviceD injected into class ServiceB

Dependency serviceC found in classs ServiceB

Creating instance of class: ServiceC

Dependency serviceE found in classs ServiceC

Creating instance of class: ServiceE

Instance of class: ServiceE created

Dependency serviceE injected into class ServiceC

Instance of class: ServiceC created

Dependency serviceC injected into class ServiceB

Instance of class: ServiceB created

Dependency serviceB injected into class ServiceA

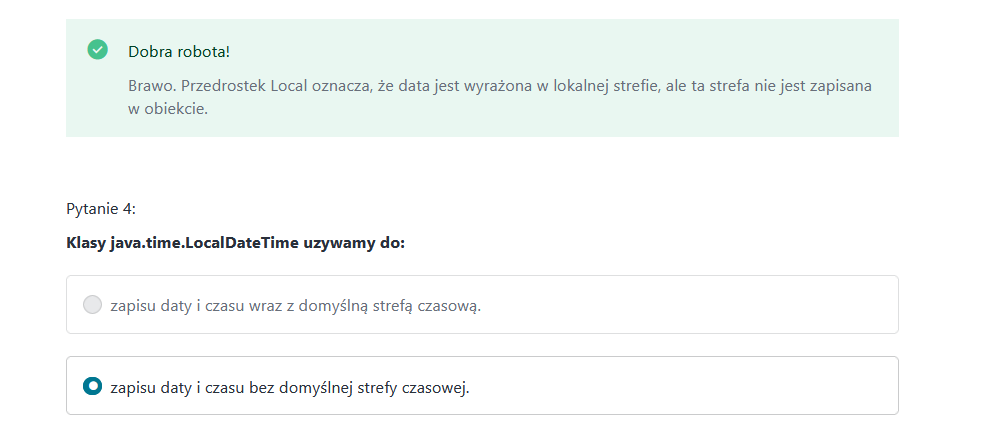
Instance of class: ServiceA created

Process finished with exit code 0

# Internacjonalizacja, lokalizacja i czas

## Time API

Czas „zero” to północ, 1 stycznia 1970 w strefie UTC



Main

public static void main(String[] args) {  
 String date1 = "2020-03-11 10:00";  
 System.*out*.println(date1); //2020-03-11 10:00  
  
 long now = System.*currentTimeMillis*();  
 System.*out*.println(now); //1609324899396  
  
 Date date2 = new Date(); //stare API  
 System.*out*.println(date2); //Wed Dec 30 11:41:39 CET 2020  
  
 Date date3 = new Date(Long.*MAX\_VALUE*); //na jak długo wystarczy longa  
 System.*out*.println(date3); //Sun Aug 17 08:12:55 CET 292278994  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 //Time API  
 Month month = Month.*DECEMBER*;  
 DayOfWeek dayOfWeek = DayOfWeek.*FRIDAY*;  
  
 LocalDate localDate = LocalDate.*of*(2020, 03, 11);  
 TemporalAdjuster adj = TemporalAdjusters.*next*(DayOfWeek.*FRIDAY*);  
 LocalDate nextFridayFromLocalDate = localDate.with(adj);  
 System.*out*.println(nextFridayFromLocalDate); //2020-03-13  
  
 LocalDate localDateMinus5Months = localDate.minus(5, ChronoUnit.*MONTHS*);  
 System.*out*.println(localDateMinus5Months); //2019-10-11  
  
 YearMonth yearMonth = YearMonth.*of*(2010, Month.*FEBRUARY*);  
 System.*out*.println(yearMonth.lengthOfMonth()); //28 (ile dni miał luty)  
  
 LocalDate date4 = LocalDate.*of*(2010, 2, 5);  
 LocalTime time = LocalTime.*of*(12, 20);  
 System.*out*.println(time); //12:20  
  
 LocalDateTime localDateTime1 = LocalDateTime.*of*(date4, time);  
 System.*out*.println(localDateTime1); //2010-02-05T12:20  
  
 LocalDateTime localDateTime2 = LocalDateTime.*of*(2020, 12, 31, 12, 34);  
 System.*out*.println(localDateTime2); //2020-12-31T12:34  
  
 //System.out.println(ZoneId.getAvailableZoneIds()); //[Asia/Aden, America/Cuiaba, Etc/GMT+9 ... Europe/Warsaw, America/Chicago, ...]  
 ZoneId warsawZone = ZoneId.*of*("Europe/Warsaw");  
 ZonedDateTime zonedDateTime = ZonedDateTime.*of*(localDateTime1, warsawZone);  
 System.*out*.println(zonedDateTime); //2010-02-05T12:20+01:00[Europe/Warsaw]  
  
 ZoneOffset zoneOffset = ZoneOffset.*of*("+01:00"); //musi być poprawny!, nie można wpisać np. "+111:00"  
 OffsetDateTime offsetDateTime = OffsetDateTime.*of*(localDateTime1, zoneOffset);  
 System.*out*.println(offsetDateTime); //2010-02-05T12:20+01:00  
  
 Instant instant = Instant.*now*();  
 LocalDate localDateOfInstant = LocalDate.*ofInstant*(instant, ZoneId.*systemDefault*());  
 System.*out*.println(localDateOfInstant); //2020-12-30  
  
 Instant instant2 = localDate.atStartOfDay().toInstant(ZoneOffset.*UTC*);  
 System.*out*.println(instant2); //2020-03-11T00:00:00Z  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 String dateStr = "2020/12-10";  
 DateTimeFormatter dateTimeFormatter = DateTimeFormatter.*ofPattern*("yyyy/MM-dd");  
 LocalDate localDateFromStr = LocalDate.*parse*(dateStr, dateTimeFormatter);  
 System.*out*.println(localDateFromStr); //2020-12-10  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 long duration = Duration.*between*(localDateFromStr.atStartOfDay().toInstant(ZoneOffset.*UTC*), Instant.*now*()).toDays();  
 System.*out*.println(duration); //20  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 Period period = Period.*ofMonths*(2);  
 LocalDate dateFromStrMinusPeriod = localDateFromStr.minus(period);  
 System.*out*.println(dateFromStrMinusPeriod); //2020-10-10  
  
 Period periodBetween = Period.*between*(localDateFromStr, LocalDate.*now*());  
 System.*out*.println(periodBetween.getDays()); //20  
  
}

Zadanie1

public static void main(String[] args) {  
 Date now = new Date();  
 final LocalDateTime startDate = LocalDateTime.*of*(2020, 2, 1, 20, 0);  
 final LocalDateTime endDate = LocalDateTime.*of*(2025, 10, 30, 12, 0);  
  
 System.*out*.println(*isBetween*(startDate, endDate, now));  
}  
  
private static boolean isBetween(LocalDateTime startDate, LocalDateTime endDate, Date dateToCompare) {  
  
 final LocalDateTime localDateTime = LocalDateTime.*ofInstant*(dateToCompare.toInstant(), ZoneId.*systemDefault*());  
  
 return (startDate.isBefore(localDateTime) || startDate.isEqual(localDateTime))  
 && (endDate.isAfter(localDateTime) || endDate.isEqual(localDateTime));  
  
}

Zadnie2

public static void main(String[] args) {  
  
 Instant start = Instant.*now*();  
 for (int i = Integer.*MIN\_VALUE*; i < Integer.*MAX\_VALUE*; i++) {  
 int operation = 45 / 5;  
 }  
 Instant stop = Instant.*now*();  
 long ms = Duration.*between*(start, stop).toMillis();  
  
 System.*out*.println("Program took " + ms + " ms");  
}

## Internacjonalizacja i lokalizacja

Internacjonalizacja – przygotowanie aplikacjo do pracy w wielu językach, regionach, kulturach



Internacjonalizacja

public static void main(String[] args) {  
  
 double dbl = 3100.0;  
 String us = "3,1000.0";  
 String de = "3.100,0";  
  
 LocalDate date = LocalDate.*of*(2020, 02, 10);  
 String usDate = "2020 February 10";  
 String plDate = "10 luty 2020";  
  
 double price = 1000;  
 String usPrice = "$1000.0";  
 String plPrice = "1000 zł";  
  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 System.*out*.println(Locale.*getDefault*()); //pl\_PL jaki ustawiony domyslnie  
  
 Locale locale1 = new Locale.Builder()  
 .setLanguage("pl")  
 .setRegion("PL")  
 .build();  
 Locale locale2 = Locale.*US*;  
 Locale locale3 = Locale.*forLanguageTag*("en-PL");  
  
 Locale.*setDefault*(locale2);  
 System.*out*.println(Locale.*getDefault*()); //en\_US  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 NumberFormat defaultNumberFormat = NumberFormat.*getInstance*();  
 NumberFormat deNumberFormat = NumberFormat.*getNumberInstance*(Locale.*GERMANY*);  
  
 double d = 1000.01;  
  
 System.*out*.println(defaultNumberFormat.format(d)); //1,000.01  
 System.*out*.println(deNumberFormat.format(d)); //1.000,01  
  
 NumberFormat defaultCurrencyFormat = NumberFormat.*getCurrencyInstance*();  
 NumberFormat deCurrencyFormat = NumberFormat.*getCurrencyInstance*(Locale.*GERMANY*);  
  
 System.*out*.println(defaultCurrencyFormat.format(d)); //$1,000.01  
 System.*out*.println(deCurrencyFormat.format(d)); //1.000,01 €  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 System.*out*.println(Month.*MARCH*.getDisplayName(TextStyle.*SHORT*, Locale.*GERMANY*)); //März  
 System.*out*.println(DayOfWeek.*WEDNESDAY*.getDisplayName(TextStyle.*FULL*, Locale.*GERMANY*)); //Mittwoch  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 LocalDateTime localDateTime = LocalDateTime.*now*();  
 String pattern = "dd-MMMM-yyyy; HH:mm::ss.SSS";  
 DateTimeFormatter myDateTimeFormatter = DateTimeFormatter.*ofPattern*(pattern);  
 DateTimeFormatter deDateTimeFormatter = DateTimeFormatter.*ofPattern*(pattern, Locale.*GERMANY*);  
  
 System.*out*.println(myDateTimeFormatter.format(localDateTime)); //30-December-2020; 14:10::41.695  
 System.*out*.println(deDateTimeFormatter.format(localDateTime)); //30-Dezember-2020; 14:10::41.695  
  
}

translations\_en\_US.properties

hello=What's up?  
hi=Hi %s!

translations\_pl\_PL.properties

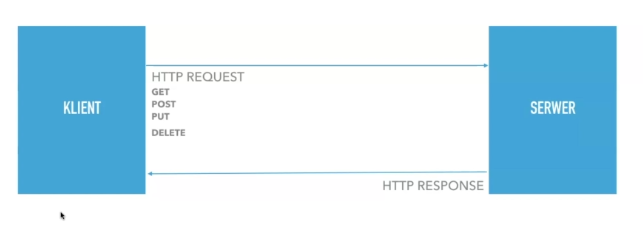
hello=Co słychać?  
hi=Cześć %s!

Tlumaczenia

public static void main(String[] args) {  
  
 String name = "Kasia";  
  
 Locale.*setDefault*(new Locale("pl", "PL"));  
 ResourceBundle plResourceBundle = ResourceBundle.*getBundle*("translations");  
 System.*out*.println(plResourceBundle.getString("hello")); //Co słychać?  
 System.*out*.println(String.*format*(plResourceBundle.getString("hi"), name)); //Cześć Kasia!  
  
 ResourceBundle usResourceBundle = ResourceBundle.*getBundle*("translations", new Locale("en", "US"));  
 System.*out*.println(usResourceBundle.getString("hello")); //What's up?  
 System.*out*.println(String.*format*(usResourceBundle.getString("hi"), name)); //Hi Kasia!  
   
}

# Komunikacja z serwisami zewnętrznymi

## Wstęp do http i REST API



JSON

* Obiekt: {obiekt}
* Tablica obiektów: [{obiekt1},{obiekt2}]

Metody – CRUD (w sumie metod jest więcej):

* GET – pobranie z serwera, odpowiedź zawsze taka sama, nie zmienia stanu serwera
* POST – może mieć ciało, używa się aby zapisać coś na serwerze, serwer ustala w jakim formacie należy podać dane
* UPDATE
* DELETE

Statusy HTTP

* 1×× Informational
* 2×× Success
* 3×× Redirection
* 4×× Client Error
* 5×× Server Error

## Integracja z zewnętrznym API

User

public class User {  
  
 @JsonProperty("name") //jeśli pole ma inną nazwę, a my chcemy w klasie inny opis tego pola  
 private String n;  
 private int id;  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public void setN(String n) {  
 this.n = n;  
 }  
  
 public String getN() {  
 return n;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 @Override  
 public String toString() {  
 return "User(" + id + "): " + n;  
 }  
}

Main

public static void main(String[] args) throws IOException, InterruptedException {  
 HttpClient client = HttpClient.*newBuilder*().build();  
  
 //GET  
 HttpResponse<String> httpResponse = client.send(HttpRequest.*newBuilder*()  
 .uri(URI.*create*("https://test-api.javaskills.pl/udemy/users"))  
 .GET()  
 .build(),  
 HttpResponse.BodyHandlers.*ofString*());  
  
 System.*out*.println(httpResponse); //(GET https://test-api.javaskills.pl/udemy/users) 200  
 System.*out*.println(httpResponse.body()); //[{"id":1,"name":"John"},{"id":2,"name":"Jane"}]  
  
 //przy użyciu: jackson-databind  
 //aby to zadziałało w tej klasie której obiekty chcemy stworzyć muszą być settery  
 ObjectMapper objectMapper = new ObjectMapper();  
  
 CollectionType type = objectMapper.getTypeFactory().constructCollectionType(List.class, User.class);  
  
 objectMapper.readValue(httpResponse.body(), type);  
  
 List<User> users = objectMapper.readValue(httpResponse.body(), type);  
 System.*out*.println(users); //[User(1): John, User(2): Jane]  
  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
 //CREATE  
 User newUser=new User();  
 newUser.setN("Kasia");  
  
 httpResponse=client.send(HttpRequest.*newBuilder*()  
 .uri(URI.*create*("https://test-api.javaskills.pl/udemy/users"))  
 .POST(HttpRequest.BodyPublishers.*ofString*(objectMapper.writeValueAsString(newUser)))  
 .build(),  
 HttpResponse.BodyHandlers.*ofString*());  
  
 System.*out*.println(httpResponse);  
   
   
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 //UPDATE  
 newUser.setN("Kasia2");  
  
 httpResponse=client.send(HttpRequest.*newBuilder*()  
 .uri(URI.*create*("https://test-api.javaskills.pl/udemy/users/1")) // /1 id uzytkownika do updatu  
 .PUT(HttpRequest.BodyPublishers.*ofString*(objectMapper.writeValueAsString(newUser)))  
 .build(),  
 HttpResponse.BodyHandlers.*ofString*());  
  
 System.*out*.println(httpResponse); //(PUT https://test-api.javaskills.pl/udemy/users/1) 200  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 //DELETE  
 httpResponse=client.send(HttpRequest.*newBuilder*()  
 .uri(URI.*create*("https://test-api.javaskills.pl/udemy/users/1")) // /1 id uzytkownika do updatu  
 .DELETE()  
 .build(),  
 HttpResponse.BodyHandlers.*ofString*());  
  
 System.*out*.println(httpResponse); //(DELETE https://test-api.javaskills.pl/udemy/users/1) 200  
  
  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 System.*out*.println(httpResponse.statusCode());  
  
}

# Strumienie wejście/wyjście, odczyt/zapis z pliku

## Strumienie binanrne, InputStream/OutputStream

Main

public class StrumienieBinarneMain {  
 public static void main(String[] args) {  
 File file = new File("C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA"); //to moze byc tez sciezka do folderu, niekoniecznie plik  
  
 System.*out*.println(file.exists()); //true  
 *filesList*(file);  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 File fileToDelete = new File("C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\to\_delete.txt");  
 if (fileToDelete.exists()) {  
 fileToDelete.delete();  
 }  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 File beeFile = new File("C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src\\bee.jpeg");  
 InputStream in = null;  
 OutputStream out = null;  
 try {  
  
 //InputStream i OutputStream to abstrakcyjne, dlatego potrzebyjemy implementacji  
 //w konstruktorze implementacji przekazujemy File lub path (pod spodem wtedyutworzy File)  
 in = new FileInputStream(beeFile);  
 out = new FileOutputStream("C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src\\bee-copy.jpeg");  
  
 int chunk;  
 while ((chunk = in.read()) != -1) {  
 out.write(chunk);  
 }  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (in != null) in.close();  
 if (out != null) out.close();  
 } catch (IOException ioException) {  
 ioException.printStackTrace();  
 }  
  
 }  
  
 //--------------------------------------  
  
 // żeby nie robićzagnieżdżonego try catch, to można try with resouurces  
  
 //jesli plik istnieje to FileOutputStream nadpisze go  
 try (  
 InputStream in2 = new FileInputStream("C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src\\input.txt");  
 OutputStream out2 = new FileOutputStream("C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src\\output.txt");  
 ) {  
 int chunk2;  
 while ((chunk2 = in2.read()) != -1) {  
 out2.write(chunk2);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 private static void filesList(File file) {  
 for (String filename : file.list()) {  
  
 String pathFir = file.getPath() + "\\" + filename;  
 System.*out*.println(pathFir);  
 File fileDir = new File(pathFir);  
 if (fileDir.isDirectory()) {  
 *filesList*(fileDir);  
 }  
  
 }  
 }  
}

## Strumień znaków, FileReader/FileWriter

Main

public static void main(String[] args) {  
 String projectPath = "C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src";  
  
 System.*out*.println(Charset.*defaultCharset*()); //UTF-8 - domyslne kodowanie  
  
 //UWAGA! nie da się odczytać znaków do wrirer "\*.jpeg" - tylko jako plik tekstowy  
 //w konstruktorze można podać też kodowanie  
  
 try (Reader in = new FileReader(projectPath + "\\bee.jpeg");  
 Writer out = new FileWriter(projectPath + "\\bee.txt", Charset.*forName*("UTF-8"))) {  
 int chunk;  
 while ((chunk = in.read()) != -1) {  
 out.write(chunk);  
 System.*out*.println(chunk);  
 }  
 } catch (IOException e) {  
  
 }  
  
}

## Strumienie buforowane

Main

public static void main(String[] args) {  
  
 String projectPath = "C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src";  
 int size = 8 \* 1024; //i bd 8 razy mniej "przejsc" - odczytu i zapisu  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 try (  
 BufferedInputStream in = new BufferedInputStream(new FileInputStream(projectPath + "\\input.txt"), size);  
 BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(projectPath + "\\output.txt"), size);  
 ) {  
 int chunk;  
 while ((chunk = in.read()) != -1) {  
 out.write(chunk);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 try (BufferedReader in = new BufferedReader(new FileReader(projectPath + "\\bee.jpeg"), size);  
 BufferedWriter out = new BufferedWriter(new FileWriter(projectPath + "\\bee.txt"), size)) {  
 {  
 //tu mnożna czytać po całej linijce  
 String chunk;  
  
 //tu warunek wyjścia null, a nie -1  
 while ((chunk = in.readLine()) != null) {  
 out.write(chunk);  
 System.*out*.println(chunk);  
 }  
 }  
 } catch (IOException e) {  
  
 }  
}

## Serializacja i deserializacja obiektów

Person

public class Person implements Serializable {  
  
 private static final long *serialVersionID* = 1;  
  
 private final String name;  
 private final int age;  
 private final transient String lastName;  
  
 public Person(String name, int age, String lastName) {  
 this.name = name;  
 this.age = age;  
 this.lastName = lastName;  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" + lastName + " " + name + ", " + age + '}';  
 }  
}

Main

public static void main(String[] args) {  
 String projectPath = "C:\\Users\\User\\OneDrive - Politechnika Śląska\\Dokumenty\\JAVA\\Projekt\_JAVA\\src";  
  
  
 //aby móc serialozować, to obiekt musi implementować Seralizable  
  
 //serializacja  
 try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(projectPath + "\\person.dat"))  
 ) {  
 Person person = new Person("John", 23, "Smith");  
 out.writeObject(person);  
 System.*out*.println(person); //Person{Smith John, 23}  
 } catch (IOException ioException) {  
 ioException.printStackTrace();  
 }  
  
 //jesli pomiedzy zmienimy strukture klasy obiektu (np dodanie toString) to poleci wyjątek, bo serialVersionID bd inny, ID jest losowy, ale można ją samemu zdefiniować, aby samemu wersjonować obiekty. Wtedy można oczytać nawet po zmianach w klasie. Można też oznaczyć pole, które będzie ignorowane przy deserializacji transient  
  
  
 //deserializacja  
 try (ObjectInputStream out = new ObjectInputStream(new FileInputStream(projectPath + "\\person.dat"))  
 ) {  
 Object readedPerson = (Person) out.readObject();  
 System.*out*.println(readedPerson); //Person{null John, 23} bo lastName jest transient  
 } catch (IOException | ClassNotFoundException ioException) {  
 ioException.printStackTrace();  
 }  
}

## NIO2 API

Main

public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  
 //NIO2 API - Non-Blockieng Input Output  
  
  
  
 String path=System.*getProperty*("user.dir"); //skrót do pobrania ścieżki głównej projektu, w którym jesteśmy  
 System.*out*.println(path);  
 File file=new File(path);  
  
 //nowe NIO2 Api  
 Path homePath= Paths.*get*(path);  
 System.*out*.println(homePath);  
  
 boolean isExist=Files.*exists*(homePath);  
 System.*out*.println(isExist); //true  
  
 //tworzenie nowego foldeu  
 Path dataDir=homePath.resolve("data"); //dodanie nowego elementu do path  
 if(Files.*notExists*(dataDir)) {  
 Files.*createDirectory*(dataDir); //tworzenie nowego folderu w katalogu  
 }  
  
 //tworzenie nowego pliku  
 Path newFilePath=homePath.resolve("xyz.txt");  
 if(Files.*notExists*(newFilePath))  
 {  
 Files.*createFile*(newFilePath);  
 }  
  
 //zapis do pliku  
 Files.*writeString*(newFilePath, "Ala ma kota", Charset.*defaultCharset*());  
  
 //kopiowanie pliku  
 Path newFilePath2=homePath.resolve("xyz-copy.txt");  
 Files.*copy*(newFilePath, newFilePath2, StandardCopyOption.*REPLACE\_EXISTING*);  
  
 //kopiowanie pliku do OutputStream  
 Path newFilePath3=homePath.resolve("xyz-copy2.txt");  
 try(OutputStream outputStream=Files.*newOutputStream*(newFilePath3)) { //strumień, więc try/try with resources  
 Files.*copy*(newFilePath, outputStream);  
 outputStream.write(" plus dopisane coś".getBytes());  
 }  
  
 //kopiowanie ...  
 Path newFile4=homePath.resolve("xyz-copy3.txt");  
 try(final BufferedWriter bufferedWriter=Files.*newBufferedWriter*(newFilePath3)) {  
 bufferedWriter.write("qwertyuiopsdfghjk");  
 }  
  
 //obiekt - plik  
 Path personFile1=dataDir.resolve("person1.dat");  
  
 //zapis  
 try(ByteArrayOutputStream bos=new ByteArrayOutputStream();  
 ObjectOutputStream oos=new ObjectOutputStream(bos))  
 {  
 oos.writeObject(new Person("John", 23,"Shmith"));  
 byte[] byteObject = bos.toByteArray();  
 Files.*write*(personFile1, byteObject);  
 }  
  
  
 //odczyt  
 try(ByteArrayInputStream bis=new ByteArrayInputStream(Files.*readAllBytes*(personFile1));  
 ObjectInputStream oos=new ObjectInputStream(bis))  
 {  
 Person person=(Person) oos.readObject();  
 System.*out*.println(person);  
 }  
  
  
 //--------------------------------------  
 //watch service - nasłuchiwanie  
 WatchService service=FileSystems.*getDefault*().newWatchService();  
 dataDir.register(service, StandardWatchEventKinds.*ENTRY\_CREATE*, StandardWatchEventKinds.*ENTRY\_DELETE*); //o czym chcemy byc informowani  
  
 WatchKey watchKey;  
 while ((watchKey=service.take())!=null)  
 {  
 //jesli utworzymy plik, to od razu zostanie usuniety  
 for(WatchEvent watchEvent: watchKey.pollEvents())  
 {  
 System.*out*.println(String.*format*("%s - %s", watchEvent.kind(), watchEvent.context()));  
 Path pathToFile = dataDir.resolve(watchEvent.context().toString());  
 System.*out*.println(pathToFile);  
 Files.*deleteIfExists*(pathToFile);  
 }  
 watchKey.reset(); //"właczenie" dalszego nasluchowania  
 }  
  
  
}

Konsola

true

Person{null John, 23}

ENTRY\_CREATE - test.txt

C:\Users\User\OneDrive - Politechnika Śląska\Dokumenty\JAVA\Projekt\_JAVA\data\test.txt

ENTRY\_DELETE - test.txt

C:\Users\User\OneDrive - Politechnika Śląska\Dokumenty\JAVA\Projekt\_JAVA\data\test.txt