



# Object Oriented Programming

# HELLO!

## Krzysztof Łopucki

Full Stack Developer

Transition Technologies – Managed Services

# Object Oriented Programming

## Programowanie zorientowane na obiekty

- Wykorzystywane w wielu językach takich jak Java, C#, C++, Ruby.
- Podstawowa składowa każdej aplikacji napisanej w języku java
- Przestajemy operować na zmiennych w zamian za obiekty.
- Koncentrujemy swoją uwagę na mechanizmach rządzących światem.

# Object Oriented Programming

## Klasa

- Struktura danych.
- Może posiadać pola i metody.
- Na jej podstawie tworzymy obiekty.

# Object Oriented Programming

## Klasa

```
public class Document { }
```

# Object Oriented Programming

## Metoda main

- Nic nie zwraca
- Jest statyczna
- Jest publiczna
- Przyjmuje argumenty

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

# Object Oriented Programming

## Obiekt

- Egzemplarz/Instancja/Naturalna reprezentacja klasy.
- Zmienna typu danej klasy, tworzona za pomocą słowa kluczowego *new*.
- Pozwala ustawić wartości oraz wywoływać opisane w klasie metody.

# Object Oriented Programming

## Tworzenie obiektów

- Operator new rezerwuje przestrzeń w pamięci oraz wywołuje właściwy konstruktor.
- Zwraca referencję do nowego obiektu.



# Object Oriented Programming

## Obiekt

Operator **new** tworzy nowy obiekt – instancje klasy.

```
public static void main(String[] args) {  
    Document document = new Document();  
    document.setDetail("invoice document");  
    document.setId(11);  
  
    System.out.println(  
        document.getId() + " " + document.getDetail());  
}
```

# Object Oriented Programming

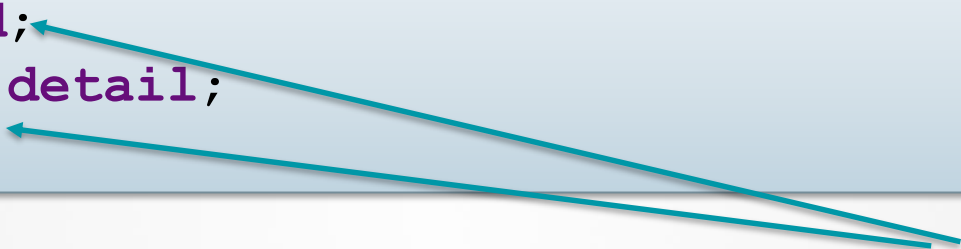
## Pole klasy

- Pojedyncza informacja, właściwość obiektu danej klasy.
- Zmienna należąca do klasy.
- Składa się z typu przechowywanych danych oraz nazwy.

# Object Oriented Programming

## Pole klasy

```
public class Document {  
    Long id;  
    String detail;  
}
```



Pola klasy Document

# Object Oriented Programming

## Metoda

- Działanie, czynność którą można wykonać na obiekcie.
- Funkcja należąca do klasy.
- Składa się z deklaracji (typ zwracanych danych oraz nazwa) oraz definicji (kod opisujący czynność, w klamrach, po deklaracji).

# Object Oriented Programming

## Metoda

```
String concatStrings(String a, String b) {  
    return a + b;  
}
```

# Object Oriented Programming

## Konstruktor

- Wywoływany przy tworzeniu każdego obiektu.
- Może posiadać argumenty.
- Domyślnie konstruktor bezargumentowy.

# Object Oriented Programming

## Konstruktor

```
String variable;  
  
Document() {  
    this.variable = "Simple String";  
}  
  
Document(String variable) {  
    this.variable = variable;  
}
```

# Object Oriented Programming

## POJO

- Plain Old Java Object – prosty obiekt Java.
- Zawiera pola.
- Zawiera akcesory – gettery, settery.
- Może posiadać konstruktor.
- Służy do przechowywania danych zbiorowych.



# Object Oriented Programming

## Ćwiczenie pierwsze

Napisz klasę POJO do przechowywania informacji o fakturach.

# Object Oriented Programming

## Porównywanie obiektów

- Obiekty w javie można porównać za pomocą metody *equals*.
- Pochodzi ona z klasy `Object` dzięki czemu można ją wywołać na każdej zmiennej typu obiektowego.
- Porównanie obiektów za pomocą operatora `==` jest błędne. Ten operator porównuje adresy obiektów (obiekty utworzone za pomocą operatora *new* mają zawsze różne adresy)

# Object Oriented Programming

## Hash code

- Metoda hash code zwraca zmienną typu int na podstawie danych obiektu.
- Pochodzi z klasy Object.
- Jeśli metoda equals porównująca dwa obiekty zwróci true to obiekty mają takie same wartości skrótu.
- Jeśli equals zwróci false to wartości skrótu mogą być różne lub takie same.

# Object Oriented Programming

## Kontrakt pomiędzy equals a hashCode

- Nadpisujemy albo obie metody albo żadnej.
- Każde wywołanie metody *hashCode* na tym samym obiekcie musi kończyć się zwróceniem tej samej liczby całkowitej.
- Jeżeli dwa obiekty są sobie **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 unikane.
- Relacja wyznaczona metodą *equals* musi być **zwrotna**, czyli dla każdej zmiennej *x* różnej od null wyrażenie *x.equals(x)* musi zwracać wartość = true.

# Object Oriented Programming

## Kontrakt pomiędzy equals a hashCode

- Relacja wyznaczona metodą equals musi być **symetryczna**, czyli dla każdej pary zmiennych  $x$  i  $y$ , wyrażenie  $x.equals(y)$  ma wartość `true` i wtedy i tylko wtedy gdy  $y.equals(x) = true$ .
- Relacja wyznaczona metodą equals musi być **przechodnia**, czyli dla dowolnych zmiennych  $z$ ,  $y$  i  $x$ , jeżeli  $x.equals(y) = true$  oraz  $y.equals(z) = true$ , to  $x.equals(z)$  musi również  $= true$ .
- Relacja wyznaczona metodą equals musi być **spójna**, czyli każdorazowe wywołanie  $x.equals(y)$  (przy założeniu, że między wywołaniami obiekty nie były modyfikowane) zawsze musi zwracać tę samą wartość – zawsze `true` albo zawsze `false`.
- Każdy obiekt jest różny od `null`, czyli wywołanie  $x.equals(null)$  dla obiektu  $x$  różnego od `null`, zawsze musi zwrócić `false`.

# Object Oriented Programming

## Kontrakt pomiędzy equals a hashCode

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Document document = (Document) o;

    if (!getId().equals(document.getId())) return false;
    if (!getDetail().equals(document.getDetail())) return false;
    return variable.equals(document.variable);
}

@Override
public int hashCode() {
    int result = getId().hashCode();
    result = 31 * result + getDetail().hashCode();
    result = 31 * result + variable.hashCode();
    return result;
}
```

# Object Oriented Programming

## Ćwiczenie drugie

- Stwórz dwa obiekty tej samej klasy
- Porównaj je wykorzystując metodę `.equals` oraz operator `==`
- Sprawdź otrzymane wyniki
- Spróbuj sam zaimplementować metody w taki sposób aby sprawdzały konkretne wartości

# Object Oriented Programming

## static

- Słowo kluczowe static pozwala na utworzenie statycznych metod i zmiennych.
- Statyczne pola i metody mają taką samą wartość dla wszystkich instancji danej klasy.
- Nie trzeba tworzyć instancji nowego obiektu w celu wywołania metody lub pobrania wartości zmiennej.



# Object Oriented Programming

## static

```
public static String DOCUMENT_KEY = "invoice";
```

```
System.out.println(Document.DOCUMENT_KEY);
```

Nie potrzebujemy tworzyć nowego obiektu. Odwołujemy się bezpośrednio do klasy.

# Object Oriented Programming

## Dziedziczenie

- Rozszerzamy właściwości klasy
- Przechowujemy właściwości wspólne wielu klas
- Można wyróżnić nadklasę i podklasę, gdzie podklasa dziedziczy po nadklasie.
- Podklasa rozszerza funkcjonalność nadklasy oraz może korzystać z jej zmiennych i metod.
- Dziedziczenie pozwala ograniczyć powielanie kodu.
- W Javie dana klasa może dziedziczyć tylko po jednej klasie (w javie nie ma wielodziedziczenia).
- Dziedziczenie realizujemy za pomocą słowa kluczowego **extends**.

# Object Oriented Programming

## Dziedziczenie

```
public class InvoiceDocument extends Document {  
    private String NIP;  
    private String REGON;  
    // GETTERS and SETTERS ...  
}
```

# Object Oriented Programming

## Ćwiczenie trzecie

- Wykorzystaj mechanizm dziedziczenia i utwórz 4 klasy posiadające konkretne pola. Opis klas :
  1. Klasa pierwsza będzie posiadała id pojazdu (typu Long), markę, model i rocznik.
  2. Druga klasa rozszerzy pierwszą klasę dodając ilość osób którą pojazd może przewozić.
  3. Trzecia klasa rozszerzy drugą klasę dodając informacje specyficzne dla autokarów jak np. ilość miejsc siedzących i stojących.
  4. Ostatnia klasa rozszerzy pierwszą klasę o informacje specyficzne dla koparki.

# Object Oriented Programming

## Ćwiczenie trzecie

- Zaimplementuj w każdej klasie metodę info wyświetlającą napis zbudowany z wszystkich dostępnych pól informujący o stanie pojazdu
- Stwórz każdy z tych obiektów
- Wypełnij wszystkie pola
- Dla każdego obiektu wywołaj metodę wyświetlającą informacje o pojeździe

# Object Oriented Programming

## Modyfikatory dostępu

- Określają widoczność danego elementu w kodzie.
- Można ich używać w stosunku do zmiennych, metod, klas lub interfejsów podając odpowiednie słowo kluczowe przed typem danego elementu.
- Wyróżniamy cztery modyfikatory: public, private, protected i package.

# Object Oriented Programming

## Modyfikatory dostępu

- public - element jest dostępny z wszystkich miejsc w kodzie.
- private - element jest dostępny tylko dla klasy w której się znajduje (nie stosuje się przed klasami).
- protected - element jest dostępny dla danej klasy oraz jej podklas (nie stosuje się przed klasami).
- package - element jest dostępny w pakiecie w którym się znajduje.

# Object Oriented Programming

## Enkapsulacja

- Inaczej hermetyzacja.
- Ograniczenie dostępu do pewnych informacji dla elementów z zewnątrz.
- Realizowana za pomocą modyfikatorów dostępu.
- Przykładem są prywatne pola klasy lub metody, które nie są dostępne bezpośrednio dla innych klas.



# Object Oriented Programming

## Ćwiczenie czwarte

- Zgodnie z zasadami enkapsulacji napisz klasę POJO przechowującą informacje o pracownikach.

# Object Oriented Programming

## Przesłanianie - Overriding

- Możliwe w klasie która dziedziczy po klasie bazowej
- Nadpisanie implementacji metody w klasie, która dziedziczy
- Pozwala na dostosowaniu działania metody w zależności od potrzeb

# Object Oriented Programming

## Ćwiczenie piąte

Mechanizm o pojazdach który już zaimplementowałeś zmień w taki sposób, aby zdefiniować w pierwszej klasie metodę info wyświetlającą odpowiedni napis.

Napisuj tą metodę w klasach kolejnych.

# Object Oriented Programming

Obiektem bazowym dla wszystkich klas jest klasa Object.

```
Object x = new Object();  
x.  
m equals (Object obj) boolean  
m hashCode () int  
m toString () String  
m getClass () Class<? extends Object>  
m notify () void  
m notifyAll () void  
m wait () void  
m wait (long timeout) void  
m wait (long timeout, int nanos) void  
inst expr instanceof Type ? ((Type) expr). : null
```

# Object Oriented Programming

## Przeciążanie metod - Overloading

- Polega na tworzeniu metod o tych samych nazwach ale różnych parametrach.
- Metody muszą się różnić ilością lub typem argumentów.
- Nieprawidłowa jest różnica tylko zwracanego typu.
- Metody mogą być przeciążane w tej samej klasie lub podklasie.

# Object Oriented Programming

## Ćwiczenie szóste

Zamień metodę wyświetlającą dane z poprzedniego ćwiczenia w taki sposób, żeby przeciążały metodę toString zamiast implementować własną.

# Object Oriented Programming

## Klasy i metody abstrakcyjne

- Tworzone za pomocą słowa kluczowego *abstract*.
- Nie można tworzyć obiektów klas abstrakcyjnych, można po nich dziedziczyć.
- Klasy abstrakcyjne mogą dziedziczyć po innych klasach oraz implementować interfejsy.
- Metoda abstrakcyjna nie posiada ciała, należy ją zaimplementować w klasie która dziedziczy po klasie abstrakcyjnej.

# Object Oriented Programming

## Klasy i metody abstrakcyjne

```
public abstract class DocumentsService {  
    protected abstract String getType();  
}
```

- W klasach dziedziczących po klasie abstrakcyjnej z metodami abstrakcyjnymi pojawi się błąd.
- Po najechaniu kursorem na ten błąd, dostaniemy podpowiedź że musimy zaimplementować metodę.



# Object Oriented Programming

## Ćwiczenie siódme

Napisz klasę abstrakcyjną przechowującą wartość liczbową x;

Napisz metodę abstrakcyjną calculate.

Zaimplementuj dwie implementacje tej klasy. Pierwsza będzie inkrementowała zmienną x, druga będzie ją dekrementowała.

# Object Oriented Programming

## Interfejsy

- API
- Posiada definicję metod bez ich implementacji
- Powiada pola ale koniecznie muszą być zainicjowane
- Interfejsy się implementuje (**implements**)

# Object Oriented Programming

## Interfejsy

```
public interface WordsPrinter {  
    String value = "ISSUE";  
    String printSomething();  
}
```

# Object Oriented Programming

## Klasa anonimowa

- Klasa wewnętrzna (tworzona wewnątrz innej klasy, nieanonimowej).
- Nie posiada nazwy, definicja klasy jest tworzona za pomocą operatora *new*.
- Można utworzyć tylko jeden obiekt takiej klasy (jej definicja jest jej jedyną instancją).

# Object Oriented Programming

## Klasa anonimowa

```
public boolean adult(PersonalDocument personalDocument, IsOldChecker callback) {  
    return callback.check(personalDocument.getAge());  
}
```

```
public interface IsOldChecker {  
    boolean check(Long id);  
}
```

```
public interface WordsPrinter {  
    String printSomething();  
}
```

# Object Oriented Programming

```
System.out.println(new WordsPrinter() {  
    @Override  
    public String printSomething() {  
        return "JUPI!!";  
    }  
});
```

```
System.out.println(((PersonDocumentService)  
personalService).adult(personalDocument, new IsOldChecker() {  
    @Override  
    public boolean check(Long id) {  
        return id > 18;  
    }  
}));
```

# Object Oriented Programming

## Ćwiczenie ósme

Napisz interfejs i zdefiniuj dowolną metodę.

Napisz metodę w już istniejącej klasie, która przyjmie wartość jak argument, napisany przed chwilą interfejs.

Stwórz obiekt klasy i wywołaj jej metodę przekazując klasę anonimową stworzoną na podstawie interfejsu.

# Object Oriented Programming

## Interfejsy C.D.

- Interfejsy oddzielają implementację od definicji.
- Interfejsów używamy wszędzie gdzie to jest tylko możliwe.



# Object Oriented Programming

## Polimorfizm

- Wielopostaciowość; zdolność obiektu do różnych zachowań.
- Występuje również, kiedy w klasie pochodnej nadpisujemy metodę z klasy bazowej.
- Zapobiega przeoczeniu implementacji pewnych metod.

# Object Oriented Programming

## Polimorfizm

- Występuje w dziedziczeniu np.:
- `Object o = new DocumentService();`
- `DocumentService ds = new PersonalDocumentService();`
- `Object o = new PersonalDocumentService();`
- Dostęp do metod będzie możliwy po rzutowaniu lub zadeklarowaniu konkretnej klasy.

# Object Oriented Programming

## Ćwiczenie dziewiąte

Napisz interfejs Pet i zadeklaruj w nim metodę voice typu void.

Napisz 5 różnych implementacji tej metody dla 5 różnych zwierząt.

utwórz obiekty stworzonych klas i wywołaj odpowiednie metody.

# Object Oriented Programming

## final

- Słowo kluczowe którego można użyć w stosunku do zmiennej, metody i klasy
- Zmienna final oznacza, że tylko raz można przypisać do niej wartość. Zmienne najczęściej ustawiamy w konstruktorze lub bezpośrednio przy deklaracji
- Metoda final oznacza, że można ją zaimplementować tylko w klasie do której należy (nie można jej nadpisać w innej klasie)
- Klasa final oznacza, że nie można po niej dziedziczyć
- Zmienne final inicjujemy podczas deklaracji lub w konstruktorze

# Object Oriented Programming

## Typy generyczne

- „szablony” dla klas, metod i interfejsów.
- Parametryzujemy typy danych.
- Dzięki temu klasy, metody i interfejsy nie są związane z konkretną implementacją.
- Generyki pozwalają uniknąć rzutowania.

# Object Oriented Programming

## Typy generyczne

- Możemy ustawić typ który dziedziczy po konkretnej implementacji : `<? extends IPerson>`
- Deklarujemy przy definicji klasy i interfejsów.

# Object Oriented Programming

```
public class GenericDocument<T extends Document> {  
    private Map<Long, T> list = new HashMap<>();  
    private T document;  
    public T storeDocument(Long id, T document) {  
        list.put(id, document);  
        return document;  
    }  
    public T findById(Long id) {  
        return list.get(id);  
    }  
}
```

```
GenericDocument<PersonalDocument> genericPersonalDocument  
    = new GenericDocument<>();  
genericPersonalDocument.storeDocument(11, pd);  
pd = genericPersonalDocument.findById(11);
```

# Object Oriented Programming

## Ćwiczenie dziesiąte

Repozytoria to klasy które integrują aplikację oraz miejsce do przechowywania pliku jak np.: baza danych, dysk.

Zaimplementuj swóje własne repozytorium generyczne w taki sposób żeby operowało na plikach oraz posiadało metody do zapisu oraz pobierania danych z plików.

Bądź kreatywny i wymyśl mechanizm oraz tematykę.

Nie obawiaj się zadawania pytań.



# Object Oriented Programming

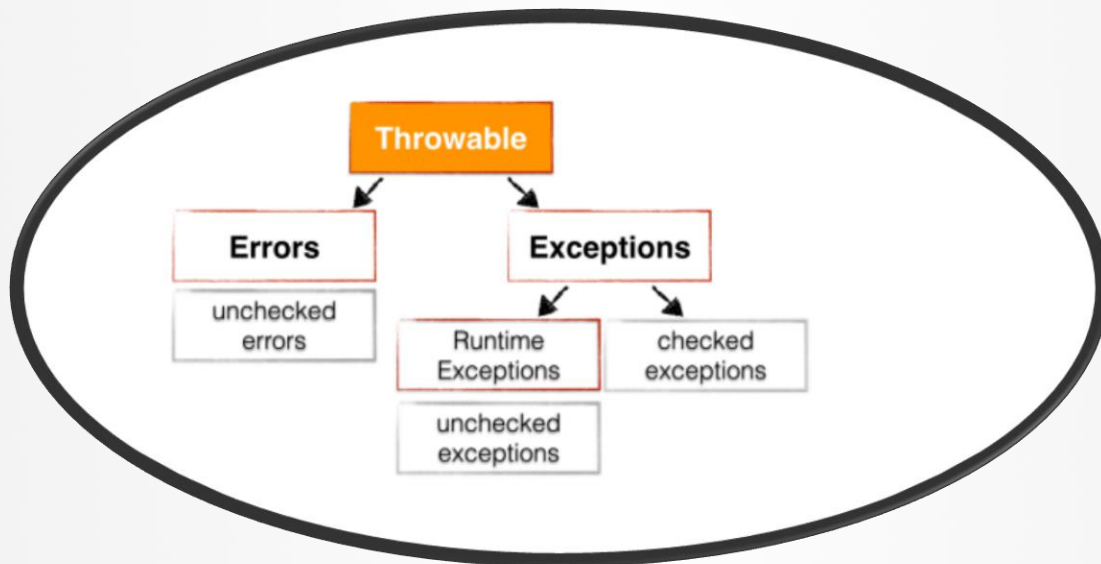
## Wyjątki

- Obiekty, które są odpowiedzią programu na nieoczekiwane zdarzenia, np. błąd spowodowany próbą odczytu pliku, który nie istnieje.
- Wyjątki przechwytujemy jeśli trzeba lub nie jeśli nie trzeba.
- Przechwycone wyjątki obsługujemy.

# Object Oriented Programming

## Wyjątki

- Już spotkałeś się z wyjątkami. **IOException**.



# Object Oriented Programming

## Wyjątki

- Pierwszym z wyjątków jest Throwable, wyłapuje każdą anomalię programu.
- Error powinien zatrzymać aplikację. Najczęściej pojawia się podczas nieoczekiwanych sytuacji jak np.: brak pamięci lub inny błąd maszyny wirtualnej. Ten wyjątek nie powinien nigdy się pojawić.
- Exception ( checked exceptions ) wyjątki które musimy złapać i obsłużyć samodzielnie.
- Runtime Exceptions (unchecked) wyjątki których nie musimy łapać.

# Object Oriented Programming

Implementacja logiki biznesowej

Wykona się tylko wtedy kiedy  
spotka wyjątek ValidationException

```
try {  
    validateDocument(document);  
    store.put(document.getId(), document);  
} catch (ValidationException e) {  
    System.out.println("Cannot save document type " + getType());  
} finally {  
    System.out.println("Run always");  
}
```

Wykona się zawsze

# Object Oriented Programming

## Wyjątki

- Jeśli nie chcemy łapać wyjątku od razu, możemy go propagować dalej. Do definicji metody dodajemy **throws ValidationException**.
- Jeśli chcemy w kodzie wyrzucić konkretny wyjątek wyrzucamy go **throw new ValidationException**.

# Object Oriented Programming

```
private void validateDocument(Document document)
                                throws ValidationException {
    if (document == null) {
        throw new ValidationException("noway");
    }
}
```

# Object Oriented Programming

## Kilka dodatkowych informacji

- Interfejsy mogą po sobie dziedziczyć
- Wśród interfejsów występuje wielodziedziczenie
- Można implementować wiele interfejsów
- Klasa abstrakcyjna może implementować interfejs ale nie musi zapewniać implementacji jego metod

# Object Oriented Programming

## Garbadge Collector

- Odśmiecacz naszej aplikacji.
- JVM przechowuje zmienne, funkcje itd... w pamięci przechowując do nich referencje.
- Garbadge collector co pewien czas uruchamia się i sprawdza które zmienne nie są już używane i je usuwa.
- Dzięki temu nie martwimy się o obiekty zombiee.



# Object Oriented Programming

## Ćwiczenie jedenaste

- Korzystając poznanych mechanizmów stwórz aplikację, która wygeneruje raporty na podstawie podanych wartości z pliku.
- Raporty powinny zawierać :
  - nazwę pojazdu oraz jego id i przebieg ale tylko pojazdów z przebiegiem powyżej 350 tysięcy
  - ile jest ciężarówek konkretnej marki tj. nazwa marki i ilość wystąpień oraz która ciężarówka jest najbardziej eksploatowana
  - id ciężarówce o mocy powyżej 200 KM
- Plik csv oddziela każdą kolumnę przecinkiem, kolejne wiersze są przechowywane w kolejnych liniach.



# Contact

**Krzysztof Łopucki**

Krzysiek.lopucki@gmail.com