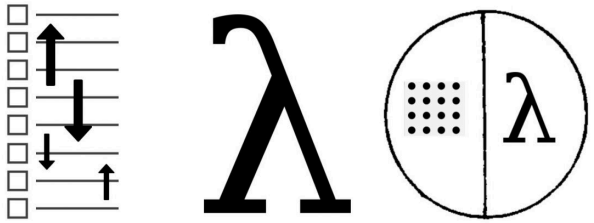
Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”



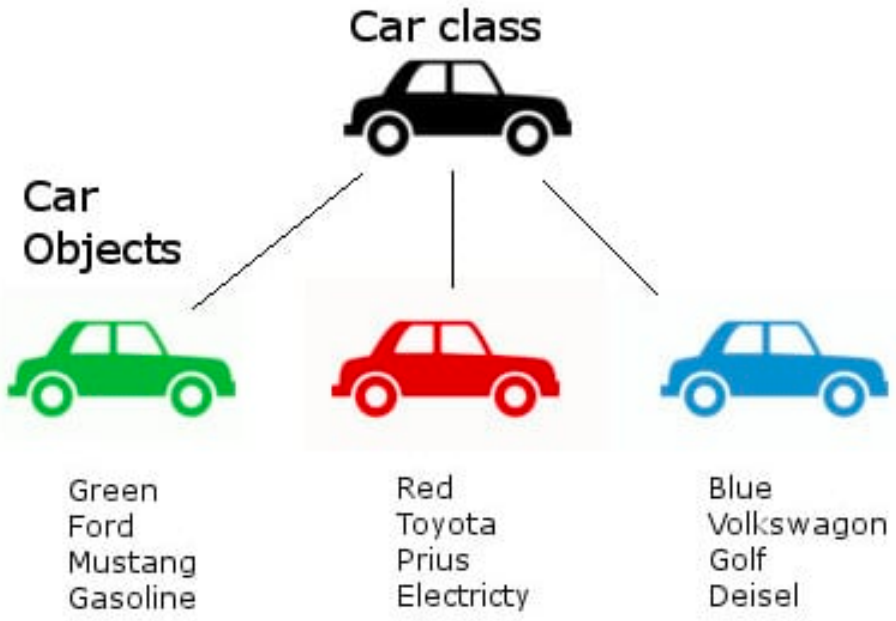
Programming paradigms

L09: Object Oriented Programming

by Michał Szczepanik

OOP (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects.



OOP (Object-Oriented Programming System)

It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Class vs Object

The difference is simple and conceptual:

A class is a template for objects.

A class defines object properties including a valid range of values, and a default value. A class also describes object behavior. An object is a member or an "instance" of a class.
An object has a state in which all of its properties have values that you either explicitly define or that are defined by default settings.

Pokemon
Name: Pikachu
Type: Electric
Health: 70
attack()
dodge()
evolve()

Fields

Methods

Classes in Java

A class is a blueprint from which individual objects are created.

```
public class Dog {
    String breed;
    int age;
    String color;
    void barking() { … }
    void hungry() { … }
    void sleeping() { … }
}
```

Classes in Java

Local variables − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables − Class variables are variables declared within a class, outside any method, with the static keyword.

Constructor

Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

```
public class Puppy {
        public Puppy() { }

        public Puppy(String name) {
                // This constructor has one parameter, name.
        }
}
```

Creating an Object

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Accessing Instance Variables and Methods

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Derived classes and inheritance

Sometimes it is convenient to develop a class that shares properties with another class but yet is distinct from the original.

The new class derives properties from an existing class but also extends or adds its own properties. This new class is called a "derived class" and is said to "inherit" its properties and functionality from the original class.

Polymorphism

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages.

- Ad hoc polymorphism: defines a common interface for an arbitrary set of individually specified types.
- Parametric polymorphism: when one or more types are not specified by name but by abstract symbols that can represent any type.
- Subtyping (also called subtype polymorphism or inclusion polymorphism): when a name denotes instances of many different classes related by some common superclass

Abstract Methods and Classes

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

Abstract Classes

```
public abstract class GraphicObject {
      abstract methods abstract void draw();
}
```

Interface

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Abstract classes vs Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.

Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

Encapsulation in Java:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

SOLID

Single responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Open–closed principle

"Software entities ... should be open for extension, but closed for modification."

Liskov substitution principle

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

Interface segregation principle

"Many client-specific interfaces are better than one general-purpose interface."

Dependency inversion principle

One should "depend upon abstractions, [not] concretions."

Other

KISS - "keep it simple, stupid" or "keep it stupid simple”

DRY - Don’t Repeat Yourself

Boy scout rule - Always leave the code cleaner than you found it.

Demo time

Please check attached source code

Thank you for your attention

michal.szczepanik@pwr.edu.pl