



Java developer in 12 months

Mentor-supported training program from CodeGym

MODULE 2. JAVA CORE

Lesson #1

OOP: encapsulation,
polymorphism



Lesson plan

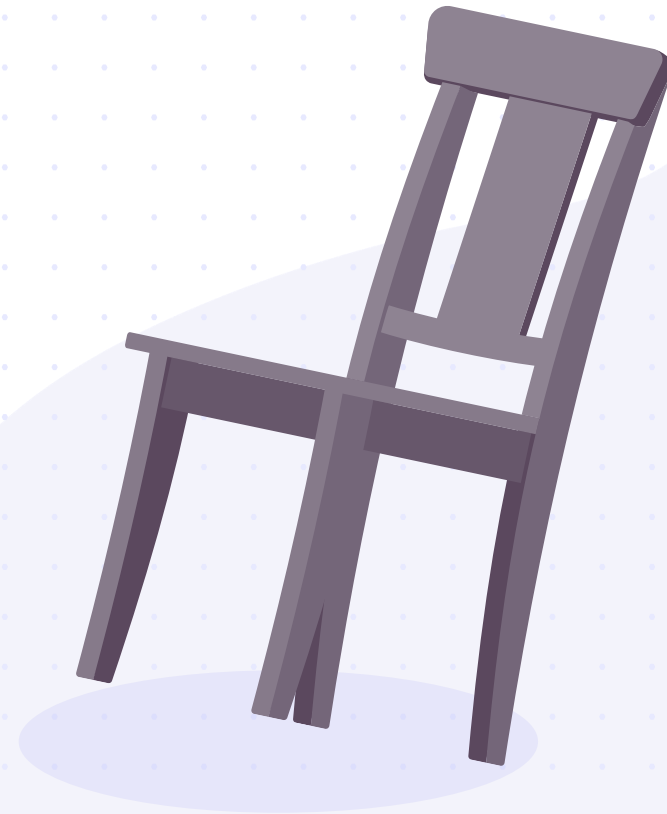
- OOP — basic principles
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Low coupling, clear functions
- Encapsulation and polymorphism
in more detail



OOP — basic principles

OOP represents principles. Internal laws. Each of them limiting us in some way, but in return providing bigger advantages when the program grows to a large size.

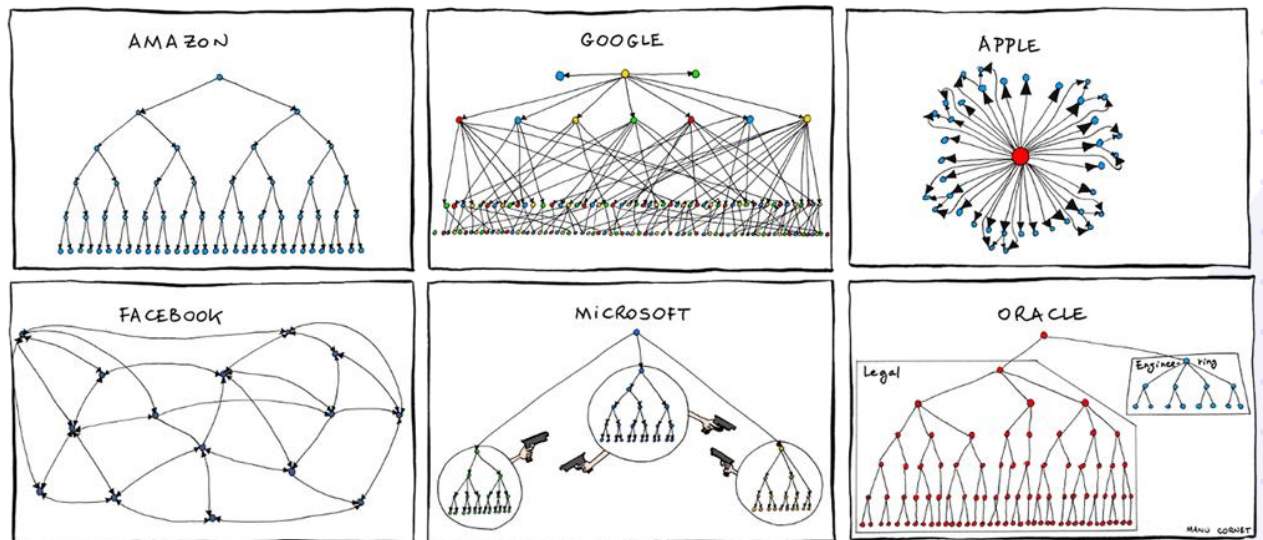
Four principles of OOP — like four legs of a chair. Remove just one of them, and the entire system becomes unstable.



Abstraction

Abstraction is a process of breaking up of something large, monolithic, into many smaller components.

Abstraction, let's say, is the **correct division** of a program into objects.



Manu Cornet

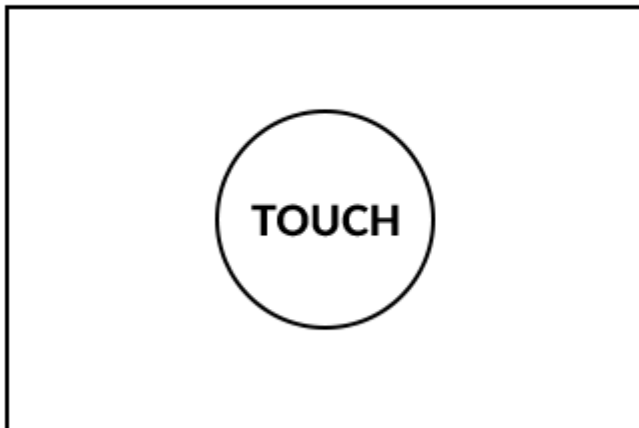
Abstraction allows you to **select the main characteristics** and omit the secondary ones.

Encapsulation

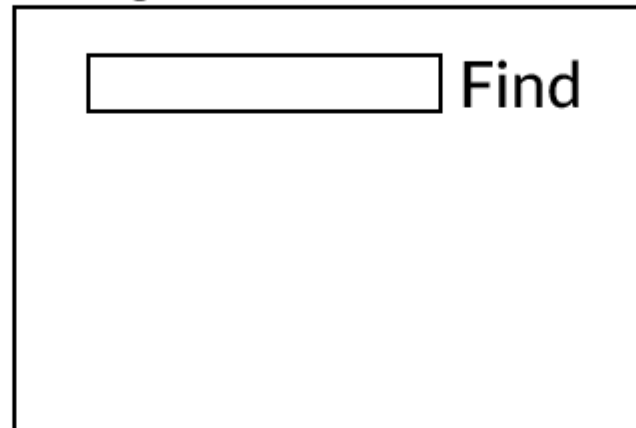
The purpose of encapsulation is to improve the quality of how things interact, by simplifying them.

In terms of programming, encapsulation is “**hiding of implementation**”.

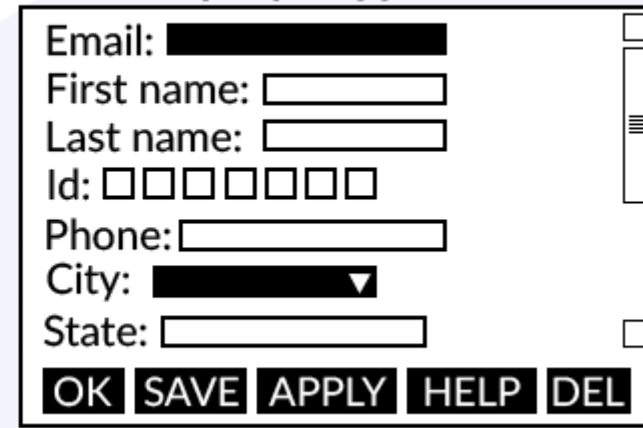
Typical Apple Product...



A Google Product...



Your Company's App...

A rectangular box representing a complex user interface form. It contains several input fields: "Email:" followed by a long blacked-out field; "First name:" followed by a short input field; "Last name:" followed by a short input field; "Id:" followed by six small square input fields; "Phone:" followed by a long input field; "City:" followed by a dropdown menu with a downward arrow; and "State:" followed by a long input field. At the bottom, there is a row of five buttons: "OK", "SAVE", "APPLY", "HELP", and "DEL". On the right side of the form, there are three small UI elements: a small square at the top, a vertical scrollbar in the middle, and another small square at the bottom.

Inheritance

In programming, it is possible to create one class based on another. The new class becomes a descendant (**heir**) of an existing one.

This is very beneficial when there is a class that contains 80%-90% of the data and methods we need.

We simply declare the appropriate class as the parent of our new class, then all the data and methods of the parent class automatically appear in the new class.

Polymorphism

A person can drive a car regardless of what kind of car it is, because they all have the same control interface: **steering wheel, pedals and gear lever**.

The internal structure of cars is different, but they all have the same interface.

In programming, polymorphism allows you to refer to objects of different classes in a uniform way.

Low coupling, clear functions

The internal structure of the product must be maintained in such a state that it will be possible to make significant (and not so) changes with minimal rework.

The program is divided into several parts, often layers, the logic of which is strongly tied to their internal structure and very weakly to other layers / parts.

Usually the interaction of layers is very regulated. One layer can access the second one and use only a small part of its classes. This is called the "**principle of low coupling**".

Encapsulation

Advantages:

1. Valid internal state

The object must keep track of changes to its internal data, or better yet, make them itself.

Proper use of encapsulation ensures that no class can directly access our class's internal data and therefore change it without our control.

2. Parameter control

We must check all incoming data for their compliance with the logic of the program and the logic of our class.

3. Minimizing errors when changing class code

When we change methods that are declared **private**, we know that there is no class anywhere that calls those methods. We can remake them, change the number of parameters and their types, and the dependent code will continue to work. Well, it will at least compile.

4. We set the way our object interacts with third-party objects

Due to encapsulation, we can make sure that an object can only be created once, for example. Even if its creation occurs in several places of the project at the same time.

Encapsulation allows you to add additional restrictions that can be turned into additional benefits.

Encapsulation principles:

The original meaning of the word “**encapsulation**” in programming is the combination of data and methods for working with this data in one package (“capsule”).

In Java, the encapsulation role is represented by a class. The class contains both data (class fields) and methods for working with this data.

It doesn't matter what's going on inside the car. The main thing is that when you press the right pedal, the car goes forward, and when you press the left pedal, it slows down.

This is the essence of concealment (**hiding of the implementation**). All the "insides" of the program are hidden from the user. For him, this information is superfluous, unnecessary. The user needs the end result, not the internal process.

Java also has **concealment of data**.
With this we are helped by:

- Access modifiers (**private**, **protected**, **package default**);
- Getters and setters.



Polymorphism

1. Method overriding

If we have inherited a method that doesn't exactly do what we need in our new class, we can replace that method with another one. The main thing is not in which class the method is written, but the type (class) of the object from which this method is called.

Only **non-static methods** can be inherited and overridden. Static methods are not inherited and therefore not overridden.

2. Casting

When inheriting, a class receives all the methods and data of the parent class, so an object of this class is allowed to be stored (assigned) to the class variables of the parent (and the parent's parent, etc., up to Object).

3. Calling a method of an object (dynamic method dispatch)

The set of methods which can be called on a variable is determined by the type of the variable. And which method / which implementation will be called is determined by the type / class of the object, the reference to which is stored by the variable.

4. Widening and narrowing of types

Classic type widening:

The **inheritor** has been generalized (widened) to the **ancestor**, but only the methods described in the **Ancestor** class can be called on an object of the **Inheritor** type.

Classic type narrowing with a check:

An **ancestor** variable of the **Ancestor** type stores a reference to an object of the **Inheritor** class. We check that this is the case, and then perform the type conversion (narrowing) operation.

Referential type narrowing can be done without checking the type of the object. But at the same time, if an object of a class other than the **Inheritor** was stored in the **ancestor** variable, then an **InvalidClassCastException** exception will be generated.

5. Calling the original method

When overriding a method, if you only need to slightly supplement it without replacing it, you can execute your code in the new method and call the same method, but of the **base class**.

In Java this is done like so:

```
super.method()
```

Homework

Module 2

Level 1 OOP: encapsulation,
polymorphism



Questions and Answers

