Object-Oriented Programming Basics

Java: OOP in simple terms

Classes and objects, encapsulation, inheritance, polymorphism, abstraction and composition

1. Class and object

A **class** describes what data and actions objects of a certain type can have.
An **object** is a particular instance of a class.
For example, we can create a `Cat` class:

```
public class Cat {
    String name;

    public void meow() {
        System.out.println("Meow!");
    }
}
```

Then we can create an object:

```
Cat cat = new Cat();

cat.name = "Murka";
cat.meow();
```

Here `Cat` is the class and `cat` is an object of that class.

Main idea

An object usually combines:

- data — its **state** ;
- methods — its **behaviour** .

2. Encapsulation

Encapsulation means protecting the internal state of an object from incorrect changes.
Consider a player who has health:

```
public class Player {

    private int health = 100;

    public void takeDamage(int damage) {
        if (damage > 0) {
            health = Math.max(0, health - damage);
        }
    }

    public int getHealth() {
        return health;
    }
}
```

The field `health` is declared `private`. This means that other code cannot directly assign any value to it.
The state is changed through a clear method:

```
player.takeDamage(20);
```

But this code is not allowed:

```
player.health = -500; // Error
```

Encapsulation helps an object keep its state valid.

Important

Encapsulation is not just about using `private`.
The main idea is to allow the state to change only through safe and meaningful operations.

3. Inheritance

Inheritance allows us to create a new class based on an existing class.
It usually represents an:

"is a" relationship.

For example, dogs and cats are animals.

```
public class Animal {

    public void sound() {
        System.out.println("Some sound");
    }
}
```

The `Dog` class inherits from `Animal`:

```
public class Dog extends Animal {

    @Override
    public void sound() {
        System.out.println("Woof!");
    }
}
```

The `Cat` class can provide its own version of the same method:

```
public class Cat extends Animal {

    @Override
    public void sound() {
        System.out.println("Meow!");
    }
}
```

The keyword `extends` means that one class inherits from another.

4. Polymorphism

Polymorphism allows us to work with different objects through a common type.
For example:

```
Animal animal = new Dog();

animal.sound();
```

The variable has the type `Animal`, but the actual object is a `Dog`.
Therefore, the method from the `Dog` class is called:

```
Woof!
```

We can also work with several animals:

```
Animal[] animals = {
    new Dog(),
    new Cat()
};

for (Animal animal : animals) {
    animal.sound();
}
```

The result is:

```
Woof!
Meow!
```

The same method call

```
animal.sound();
```

can perform different actions depending on the actual object stored in the variable.

5. Abstraction

Abstraction helps us focus on **what** an object can do without needing to know all the details of **how** it does it.
In Java, interfaces are often used for this.
For example:

```
public interface Switchable {

    void turnOn();

    void turnOff();
}
```

The interface tells us that:

- the object can be turned on;
- the object can be turned off.

It does not tell us exactly how those actions are performed.
For example, a lamp can implement this interface:

```
public class Lamp implements Switchable {

    public void turnOn() {
        System.out.println("Lamp is on");
    }

    public void turnOff() {
        System.out.println("Lamp is off");
    }
}
```

The keyword `implements` means that a class implements an interface.

6. Composition

Composition means that one object contains or uses another object.
It represents a:

"has a" relationship.

For example, a car has an engine.

```
public class Engine {

    public void start() {
        System.out.println("Engine started");
    }
}
```

Now we can use an engine inside a car:

```
public class Car {

    private final Engine engine =
        new Engine();

    public void start() {
        engine.start();
    }
}
```

A `Car` is not an engine.
It **has** an engine and uses it.

7. Inheritance or composition?

A useful way to decide is to ask a simple question.
Is one object a type of another object?
Then inheritance may be appropriate.
For example:

Dog is an Animal.

A dog is an animal.
Does one object have another object?
Then composition is usually appropriate.
For example:

Car has an Engine.

A car has an engine.

Important

Do not use inheritance only to reuse some existing code.
First check whether the classes really have an "is a" relationship.

8. Method overriding

Overriding means that a subclass provides its own implementation of a method that already exists in the parent class.
For example, the `Animal` class contains:

```
public void sound() {
    System.out.println("Some sound");
}
```

The `Dog` class can override it:

```
@Override
public void sound() {
    System.out.println("Woof!");
}
```

The annotation

```
@Override
```

helps the compiler check that we are actually overriding an existing method.
Overriding is closely connected with polymorphism.

9. Method overloading

Overloading means having several methods with the same name but different parameters.
For example:

```
public void print(String text) {
    System.out.println(text);
}

public void print(int number) {
    System.out.println(number);
}
```

Now we can write:

```
print("Hello");
print(25);
```

In the first case, the method with a `String` parameter is called.
In the second case, the method with an `int` parameter is called.

Do not confuse them

Overriding — a new implementation of an inherited method.
Overloading — the same method name with different parameters.

10. What does a good class look like?

A good class usually:

- has one clear purpose;
- stores only the data it needs;
- does not allow its state to be accidentally corrupted;
- uses clear names for fields and methods;
- provides operations that match the meaning of the object.

For example, for a bank account this method is clearer:

```
account.withdraw(100);
```

than changing the balance directly:

```
account.balance = -100;
```

The first version describes the action “withdraw money” and allows the class to check whether the operation is valid.

11. Four main principles of OOP

Encapsulation
Protects the internal state of an object and provides safe ways to work with it.
Inheritance
Allows us to create new types based on existing ones when there is a real “is a” relationship.
Polymorphism
Allows different objects to be used through a common type.
Abstraction
Focuses on the important features of an object and hides unnecessary implementation details.

The main ideas of OOP

A class describes objects.
An object stores state and performs actions.
Encapsulation protects the state.
Inheritance creates relationships between related types.
Polymorphism allows one common type to work with different implementations.
Abstraction shows the important capabilities of an object and hides details.
Composition allows complex objects to be built from simpler objects.

This reference sheet is intended as a first introduction to object-oriented programming. The examples are intentionally simple so that the main ideas can be understood without unnecessary details.