Object Oriented Concepts

Table of Contents

[Objects and Classes 2](#_Toc134711383)

[Methods 4](#_Toc134711384)

[Access 5](#_Toc134711385)

[Components of OOP 5](#_Toc134711386)

[Inheritance 5](#_Toc134711387)

[Polymorphism 6](#_Toc134711388)

[Encapsulation 7](#_Toc134711389)

[Abstraction 8](#_Toc134711390)

**Object-Oriented Principles** (OOP) is a principle of design and development of programs that uses a **modular approach**. It organizes software design around data or **objects** rather than functions and logic. This approach is well suited for programs that are large, complex and actively maintained.

## Objects and Classes

A **class** is a blueprint for objects. It defines a set of attributes and behaviours that are common to every object of that class. An **object** is a specific implementation of a class. For example, we can have an Employee class that represents the information common to all employees, such as the salary. The specific value of the salary variable though, will be defined by an object of the Employee class.

class Employee {  
 // Fields  
 private String name; // Can get but not change  
 private double salary; // Cannot get or set  
 // Constructor  
 Employee(String n, double s) {  
 name = n; salary = s;  
 }  
 // Methods  
 void pay () {  
 System.*out*.println("Pay to the order of " + name + " $" + salary);  
 }  
 public String getName() { return name; } // getter  
}

JAVA

When creating objects for a class, there is an important distinction compared to primitives. With a primitive, all you need to do is **declare** the variable.

int n;

JAVA

This also allocates the space in memory required to hold the variable. For objects of a class however, doing this only allocates memory for a **reference** to the object.

Employee employee;

JAVA

To actually allocate space for the object, we need to **initialize** the object.

employee = new Employee();

JAVA

We can of course combine the two steps.

Employee employee = new Employee();

JAVA

This still does not set values for the variables, since we use the **default constructor**. We can set these using the **dot notation** (assuming the variables are public).

employee.name = "Adele";

JAVA

The dot notation can also be used to access **methods** of the objects.

employee.birthday();

JAVA

The dot notation is only required when working outside of the class. Inside, we can directly access the variables, since it is not necessary to explicitly specify which object we are talking about. If we do need to do this however, perhaps due to ambiguity caused by duplicate variables names, we can use the this keyword.

this.name = "Adele";

JAVA

## Methods

There are four types of methods that a class can have:

* **Modifiers** – Also called mutators, these methods change the value of some attribute of the object.
* **Accessors** – These return the value of some attribute of the object.
* **Constructors** – These are called only once per object, when initializing the object. They can be used to set default values for attributes.
* **Destructors** – Also called once per object, these are used when the object is being destroyed, for clean-up.

## Access

Most classes provide three types of access to their attributes and methods:

* **Public** – A public attribute or method is accessible from anywhere.
* **Protected** – A protected attribute or method is only accessible by the class itself or classes that inherit from the current one.
* **Private** – Private attributes and methods are only accessible from the class itself.

## Components of OOP

There are four major components of OOP, Inheritance, Polymorphism, Abstraction and Encapsulation.

### Inheritance

**Inheritance** involves one class inheriting the methods and attributes of another one. This allows us to specify behaviour that all the child classes must have in one places, **avoids code duplication** by reusing code and provides the ability to **generalize** in the sense that a subclass’s objects can be used in places where a superclass is expected.

class Dog { ... }  
class Poodle extends Dog { ... }  
Dog myDog;  
Dog rover = new Dog ();  
Poodle yourPoodle;  
Poodle fifi = new Poodle ();  
myDog = rover; // ok  
yourPoodle = fifi; // ok  
myDog = fifi; //ok  
yourPoodle = rover; // illegal  
yourPoodle = (Poodle) rover; //runtime check

JAVA

### Polymorphism

**Polymorphism** is the ability of a method to take on multiple forms. It is exhibited as either **method overriding**, where a child class changes the behaviour of a method defined in the parent class, or **overloading**, where multiple methods have the same name but different signatures.

// Method overriding  
class Bird extends Animal {  
 void fly (String destination) {  
 location = destination;  
 }  
}

class Penguin extends Bird {  
 void fly (String whatever) { }  
}

JAVA

An overridden method can also access the method in the parent class using the super keyword.

class FamilyMember extends Person {  
 void birthday () { // override birthday() in Person  
 super.birthday (); // call overridden method  
 givePresent (); // and add your new stuff  
 }  
}

JAVA

// Method overloading

int add(int x, int y) {  
 return (x + y);  
}  
  
int add(int x, int y, int z) {  
 return (x + y + z);  
}  
  
int add(int w, int x, int y, int z) {  
 return (w + x + y + z);  
}

JAVA

An alternative to method overloading is **operator overloading**, where an operator is overloaded to accept different data types.

### Encapsulation

**Encapsulation** allows an object to maintain control over its data. The attributes of the object are not publicly exposed but are only accessible through setters and getters defined by the class.

class Employee extends Person {  
 private double salary;  
 private boolean male;  
 public void setSalary (double newSalary) {  
 salary = newSalary;  
 }  
 public double getSalary () { return salary; }  
 public boolean isMale() { return male; }  
}

JAVA

### Abstraction

**Abstraction** is the concept of hiding the implementation details and only showing the functionality. This allows us to focus on what the object does instead of how it does it. Abstraction is achievable using either abstract classes or interfaces.

An **abstract** class can have both non-abstract and abstract methods. It is not possible to instantiate objects of such a class. To be able to create objects that can use the methods, we need to create new classes that **inherit** from the abstract class and implement the abstract methods.

abstract class Bike {  
 Bike() {  
 System.*out*.println("Bike is created.");  
 }  
 abstract void run(); // abstract method  
 void changeGear() { // non-abstract method  
 System.*out*.println("Gear changed.");  
 }  
}

class Honda extends Bike {  
 void run() { // child class must implement abstract method  
 System.*out*.println("Running.");  
 }  
}

class Main {  
 public static void main(String[] args) {  
 // child class can be used everywhere parent class is expected  
 Bike bike = new Honda();  
 bike.changeGear();  
 }  
}

JAVA

An **interface** on the other hand can only have abstract methods, not non-abstract ones. To create objects that use the methods, we need to create new classes that **implement** the interface.

interface *Drawable* {  
 void draw();  
}  
  
class Rectangle implements *Drawable* {  
 public void draw() {  
 System.*out*.println("Drawing a rectangle.");  
 }  
}  
  
class Circle implements *Drawable* {  
 public void draw() {  
 System.*out*.println("Drawing a circle.");  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
 *Drawable* drawable = new Circle();  
 drawable.draw();  
 }  
}

JAVA

Although abstract classes and interfaces seem similar, they have completely different use cases:

* To avoid the [diamond problem](https://www.javatpoint.com/what-is-diamond-problem-in-java), many programming languages do not allow a class to **inherit** from **multiple classes**. However, since interfaces do not have any data to cause conflicts, implementing **multiple interfaces** is acceptable.
* **Abstract classes** are useful in the same cases where **inheritance** would be useful: when several classes have common methods and attributes. The need for an abstract class instead of just a common parent class comes in specific cases where objects of the parent would not make sense, such as a generic Animal class being the parent when the system only makes sense for specific types of animals and not the general entity.
* **Interfaces** on the other hand are used when there is a need for **generalization**. We may have cases where we need to be sure that an object implements certain methods (because we are calling the methods), but we have no idea what the methods do internally. As such, it is not possible to provide implementation details. Technically we could also have achieved this using an abstract class with non-abstract methods, but that would remove our ability to work with multiple interfaces and also creates the possibility of non-abstract methods being unintentionally introduced in the future.