# OOP Basic concepts:

## Obiecte

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

## Classes

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

All classes have the following syntax:



## Interfaces

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's interface with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

Here is an example of an interface:

public interface ShapeBehaviour {

public double area();

}

**Interfaces have the following properties :**

* An interface cannot be instantiated.
* An interface can extend another interface. Use the extends (and not the implements) keyword for this.
* Interfaces cannot contain instance variables. If you declare a data member in an interface,

it should be initialized, and all such data members are implicitly treated as “public static

final” members.

* An interface does not contain any constructors.
* An interface cannot declare static methods. It can only declare abstract instance methods.
* You cannot declare members as protected or private. Only public access is allowed for

members of an interface.

* All methods declared in an interface are implicitly considered to be abstract. If you want, you can explicitly use the abstract qualifier for the method.
* You can only declare (and not define) methods in an interface.
* An interface can be declared with empty body (i.e., an interface without any members. Such interfaces are known as tagging interfaces (or marker interfaces). Such interfaces are useful for defining a common parent, so that runtime polymorphism can be used. For example, java.util defines the interface EventListner without a body.
* An interface can be declared within another interface or class; such interfaces are known as nested interfaces.
* Unlike top-level interfaces that can have only public or default access, a nested interface can be declared as public, protected, or private.

## Packages

When the size of your application grows, you need an effective mechanism to manage all your source files.

Java supports the concept of package, which is a scoping construct to organize your classes and to provide namespace management. All closely related classes can be put together in a single entity: a package. A package not only reduces the complexity of a big application but also provides access protection.

In essence, here are the advantages of using packages:

* Packages reduce complexity by facilitating categorization of similar classes.
* Packages provide namespace management. For example, two developers can define the same type name without ending up in a name clash by putting the name in different packages.
* Packages offer access protection (recall the discussion of the default access modifier).

The Java SDK is categorized in various packages. For example, java.lang provides basic language functionality and fundamental types, and java.io can be used to carry out file-related operations.

## Constructors

Each time you create an object, a constructor of that class gets called. You can make use of the constructor to initialize the newly created object by setting the initial state of the object, and you can acquire some resources (such as file handles). The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Every class has a constructor. If you do not explicitly write a constructor for a class, the Java compiler provides a default constructor (without any parameter) for that class.

## Abstraction

In computer science, abstraction is the process of separating ideas from specific instances of those ideas at work. Computational structures are defined by their meanings (semantics), while hiding away the details of how they work. Abstraction tries to factor out details from a common pattern so that programmers can work close to the level of human thought, leaving out details which matter in practice, but are immaterial to the problem being solved. For example, a system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer; low-level abstraction layers expose details of the computer hardware where the program runs, while high-level layers deal with the business logic of the program.

Abstraction captures only those details about an object that are relevant to the current perspective. For instance, numbers are concepts in programming languages. Numbers can be represented in many different ways in hardware and software, but, irrespective of how this is done, numerical operations will obey similar rules.

Abstraction can apply to control or to data: Control abstraction is the abstraction of actions while data abstraction is that of data structures.

* + Control abstraction involves the use of subprograms and related concepts control flows
  + Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype.

One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code.

The same abstract definition can be used as a common interface for a family of objects with different implementations and behaviors but which share the same meaning. The inheritance mechanism in object-oriented programming can be used to define an abstract class as the common interface.

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The requirement that a programming language provide suitable abstractions is also called the abstraction principle.

### Abstract classes

Abstraction refers to the ability to make a class abstract in OOP. An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

An example of an abstract class is the following :

public abstract class AbstractShape {

// declare fields

// declare nonabstract methods

public abstract double area();

}

Each nonabstract subclass of AbstractShape, such as Circle and Rectangle or Shape, must provide implementations for the area method:

public class Shape extends AbstractShape

{

…

public double area() {

return 0; // default implementation

}

}

### Abstract methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.

The abstract keyword is also used to declare a method as abstract. An abstract method consists of a method signature, but no method body.

Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces

Declaring a method as abstract has two results:

* The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
* Any child class must either override the abstract method or declare itself abstract.

A child class that inherits an abstract method must override it. If they do not, they must be abstract and any of their children must override it.

Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

## Encapsulation

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Example of Encapsulation :

public class EncapsulationDemo{

private int ssn;

private String empName;

private int empAge;

//Getter and Setter methods

public int getEmpSSN(){

return ssn;

}

public String getEmpName(){

return empName;

}

public int getEmpAge(){

return empAge;

}

public void setEmpAge(int newValue){

empAge = newValue;

}

public void setEmpName(String newValue){

empName = newValue;

}

public void setEmpSSN(int newValue){

ssn = newValue;

}

}

## Inheritance

Inheritance is a reusability mechanism in object-oriented programming in which the common properties of various objects are exploited to form relationships with each other. The abstract and common properties are provided in the superclass, which is available to the more specialized subclasses. For example, a color printer and a black-and-white printer are kinds of a printer (single inheritance); an all-in-one printer is a printer, scanner, and photocopier (multiple inheritance). It should be noted that Java does not support multiple inheritance but does support multiple-interface inheritance.

When we say that a class B is inherited from another class A, then class B is referred to as a derived class (or subclass) and class A is called as a base class (or superclass). By inheritance, the derived class receives the behavior of the base class, such that all the visible member methods and variables of the base class are available in the derived class. Apart from the inherited behavior, the derived class specializes its behavior by adding to or overriding base class behavior.

class Shape {

protected int color;

}

class Circle extends Shape {

private int radius; // private field

public void area() { // public method

// access to private field radius inside the class

System.out.println("area:"+3.14\*radius\*radius);

}

void fillColor() {

System.out.println("color:" + color); //access to protected field, in subclass

}

}

## Polymorphism

The Greek roots of the term polymorphism refer to the “several forms” of an entity. In the real world, every message you communicate has a context. Depending on the context, the meaning of the message may change and so may the response to the message. Similarly in OOP, a message can be interpreted in multiple ways (polymorphism), depending on the object.

For example, in function overloading (one of the polymorphic constructs in Java), you can provide methods with the same name but with different numbers of arguments or types of arguments. The concept is simple, yet it provides a lot of power and flexibility to the programmer. In FunPaint, you can fill the shapes with different colors.

Let’s assume that you have a method named area() in the Shape base class. The area() method returns the area of the drawn shape. Hence, area() is implemented (overridden) in all the derived classes of Shape. A Shape reference can point to any derived class object. When you call the area() method from the Shape reference, it results in calling the area() method of the actual object type (i.e. the dynamic type of the object). This dynamic behavior is known as runtime polymorphism.

Example :

public interface ShapeBehaviour {

public double area();

}

public class Shape implements ShapeBehaviour{

public double area() {

// some calculations

}

}

public class Circle extends Shape {

public Circle(int x){

//some code

}

public double area() {

// some calculations

}

}

public class RunApp {

public static void main(String[] args){

//Runtime polymorfism

Shape shape1 = new Circle(10);

System.out.println(shape1.area());

ShapeBehaviour shape2 = new Circle(10);

System.out.println(shape2.area());

}

}

## Access Modifiers

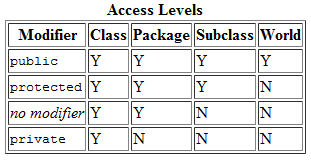
Java supports four types of access modifiers:

• Public

• Private

• Protected

• Default (no access modifier specified)



#### Public Access Modifier:

// Shape.java

package graphicshape;

class Shape {

// class definition elided

}

// Circle.java

package graphicshape;

public class Circle extends Shape {

public void area() { //public method

// code for area method elided

}

}

// Circles.java

package graphicshape;

class Circles {

void getArea() {

Circle circle = new Circle();

circle.area(); //call to public method area(), within package

}

}

// Canvas.java

package appcanvas;

import graphicshape.Circle;

class Canvas {

void getArea() {

Circle circle = new Circle();

circle.area(); //call to public method area(), outside package

}

}

The public method area() is accessible within the same package (in the Circles class), as well as outside of the package (in the Canvas class).

#### Private Access Modifier

// Shape.java

package graphicshape;

class Shape {

//class definition

}

//Circle.java

package graphicshape;

public class Circle extends Shape {

private int radius; //private field

public void area() { //public method

// access to private field radius inside the class

System.out.println("area:"+3.14\*radius\*radius);

}

}

// Circles.java

package graphicshape;

class Circles {

void getArea() {

Circle circle = new Circle();

circle.area(); //call to public method area(), within package

}

}

// Canvas.java

package appcanvas;

import graphicshape.Circle;

class Canvas {

void getArea() {

Circle circle = new Circle();

circle.area(); // call to public method area(), outside package

}

}

In this example, radius is accessible only inside the Circle class and not in any other class, regardless of the enclosing package.

#### Protected and Default Access Modifier

// Shape.java

package graphicshape;

class Shape {

protected int color;

}

// Circle.java

package graphicshape;

public class Circle extends Shape {

private int radius; // private field

public void area() { // public method

// access to private field radius inside the class

System.out.println("area:"+3.14\*radius\*radius);

}

void fillColor() {

System.out.println("color:" + color); //access to protected field, in subclass

}

}

// Circles.java

package graphicshape;

class Circles {

void getArea() {

Circle circle = new Circle();

circle.area(); // call to public method area() within package

circle.fillColor(); // call to a method with default access modifier

within package

}

}

// Canvas.java

package appcanvas;

import graphicshape.Circle;

class Canvas {

void getArea() {

Circle circle = new Circle();

circle.area(); // call to public method area(), outside package

}}

As the example shows, the protected field color is accessed in the class Circle and the default method fillColor() is called from the class Circles.

## Method Overloading/Overwriting

Remenber the abstract class example ?

public abstract class AbstractShape {

// declare fields

// declare nonabstract methods

public abstract double area();

}

Each nonabstract subclass of AbstractShape, such as Circle and Rectangle or Shape, must provide implementations for the area method:

public class Shape extends AbstractShape

{

…

public double area() {

return 0; // default implementation

}

}

This is one example of method overwriting. Let’s extend some more :

public Circle extends Shape {

…

Public double shape{

//Overwritten behavior

}

}

Overloading methods is when more than one method shares the same name, but has a different parameter list. The parameter list may comprise more or less parameters than the other methods, or different types. Overloaded methods must not all have the same return type.

public Circle extends Shape {

//package access method

void fillColor() {

System.out.println("color:" + color); //access to protected field, in subclass

}

public void fillColor (int color) {

/\* color the circle using RGB color values \*/

super.setColor(color);

System.out.println("The circle color is now " + color);

}

public void fillColor (float saturation) {

/\* color the circle using HSB values \*/

super.setSaturation(saturation);

System.out.println("The circle color saturation is now " + saturation);

}

}

## Keywords

* **break**
* **class**
* **const**
* Although reserved as a keyword in Java, const is not used and has no function. For defining constants in java, see the 'final' reserved word.
* **continue**
* **default**
* The default keyword can optionally be used in a switch statement to label a block of statements to be executed if no case matches the specified value; see switch.[3][4] Alternatively, the default keyword can also be used to declare default values in a Java annotation. From Java 8 onwards, the default keyword is also used to specify that a method in an interface provides the default implementation of an optional method.
* used in loops : **do, for, while**
* primitives : **double, float, int, long, short, char, byte, boolean**
* used in conditional statements : **if, else**
* **enum**

A Java keyword used to declare an enumerated type. Enumerations extend the base class Enum.

* **extends**
* **final**
* **finally**
* **goto**
* **implements**
* i**mport**
* **instanceof**
* **interface**
* **native**

Used in method declarations to specify that the method is not implemented in the same Java source file, but rather in another language

* **new**
* Used to create an instance of a class or array object.
* **package**
* **private protected public**
* **return**
* **static**

Used to declare a field, method, or inner class as a class field. Classes maintain one copy of class fields regardless of how many instances exist of that class. static also is used to define a method as a class method. Class methods are bound to the class instead of to a specific instance, and can only operate on class fields. (Classes and interfaces declared as static members of another class or interface are actually top-level classes and are not inner classes.)

* **super**
* **switch**
* **synchronized**
* **this**
* **throw**
* **throws**
* **transient**

Declares that an instance field is not part of the default serialized form of an object. When an object is serialized, only the values of its non-transient instance fields are included in the default serial representation. When an object is deserialized, transient fields are initialized only to their default value. If the default form is not used, e.g. when a serialPersistentFields table is declared in the class hierarchy, all transient keywords are ignored.

* **try**
* **void**
* **volatile**

Used in field declarations to specify that the variable is modified asynchronously by concurrently running threads. Methods, classes and interfaces thus cannot be declared volatile, nor can local variables or parameters.

* Reserved words for literal values
* **false**
* **null**
* **true**

# DRY

In software engineering, don’t repeat yourself (DRY) is a principle of software development, aimed at reducing repetition of information of all kinds, especially useful in multi-tier architectures. The DRY principle is stated as “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

Examples:

# Abstraction principle (programming)

"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

# Code reuse

Code reuse is the idea that a partial computer program written at one time can be, should be, or is being used in another program written at a later time. The reuse of programming code is a common technique which attempts to save time and energy by reducing redundant work.

# Single Source of Truth

In Information Systems design and theory Single Source Of Truth (SSOT) refers to the practice of structuring information models and associated schemata such that every data element is stored exactly once (e.g., in no more than a single row of a single table). Any possible linkages to this data element (possibly in other areas of the relational schema or even in distant federated databases) are by reference only. Thus, when any such data element is updated, this update propagates to the enterprise at large, without the possibility of a duplicate value somewhere in the distant enterprise not being updated (because there would be no duplicate values that needed updating)

# Rule of three (computer programming)

Rule of three is a code refactoring rule of thumb to decide when a replicated piece of code should be replaced by a new procedure. It states that the code can be copied once, but that when the same code is used three times, it should be extracted into a new procedure.

# Immutable Objects

Immutable objects are simply objects whose state (the object's data) cannot change after construction. Examples of immutable objects from the JDK include String and Integer.

Immutable objects greatly simplify your program, since they:

* are simple to construct, test, and use
* are automatically thread-safe and have no synchronization issues
* don't need a copy constructor
* don't need an implementation of clone
* allow hashCode to use lazy initialization, and to cache its return value
* don't need to be copied defensively when used as a field
* make good Map keys and Set elements (these objects must not change state while in the collection)
* have their class invariant established once upon construction, and it never needs to be checked again
* always have "failure atomicity" (a term used by Joshua Bloch): if an immutable object throws an exception, it's never left in an undesirable or indeterminate state

Immutable objects have a very compelling list of positive qualities. Without question, they are among the simplest and most robust kinds of classes you can possibly build. When you create immutable classes, entire categories of problems simply disappear.

Make a class immutable by following these guidelines:

* ensure the class cannot be overridden - make the class final, or use static factories and keep constructors private
* make fields private and final
* force callers to construct an object completely in a single step, instead of using a no-argument constructor combined with subsequent calls to setXXX methods (that is, avoid the Java Beans convention)
* do not provide any methods which can change the state of the object in any way - not just setXXX methods, but any method which can change state
* if the class has any mutable object fields, then they must be defensively copied when they pass between the class and its caller

/\*\*

\* Planet is an immutable class, since there is no way to change

\* its state after construction.

\*/

public final class Planet {

public Planet (double aMass, String aName, Date aDateOfDiscovery) {

fMass = aMass;

fName = aName;

//make a private copy of aDateOfDiscovery

//this is the only way to keep the fDateOfDiscovery

//field private, and shields this class from any changes that

//the caller may make to the original aDateOfDiscovery object

fDateOfDiscovery = new Date(aDateOfDiscovery.getTime());

}

/\*\*

\* Returns a primitive value.

\*

\* The caller can do whatever they want with the return value, without

\* affecting the internals of this class. Why? Because this is a primitive

\* value. The caller sees its "own" double that simply has the

\* same value as fMass.

\*/

public double getMass() {

return fMass;

}

/\*\*

\* Returns an immutable object.

\*

\* The caller gets a direct reference to the internal field. But this is not

\* dangerous, since String is immutable and cannot be changed.

\*/

public String getName() {

return fName;

}

/\*\*

\* Returns a mutable object - good style.

\*

\* Returns a defensive copy of the field.

\* The caller of this method can do anything they want with the

\* returned Date object, without affecting the internals of this

\* class in any way. Why? Because they do not have a reference to

\* fDate. Rather, they are playing with a second Date that initially has the

\* same data as fDate.

\*/

public Date getDateOfDiscovery() {

return new Date(fDateOfDiscovery.getTime());

}

// PRIVATE

/\*\*

\* Final primitive data is always immutable.

\*/

private final double fMass;

/\*\*

\* An immutable object field. (String objects never change state.)

\*/

private final String fName;

/\*\*

\* A mutable object field. In this case, the state of this mutable field

\* is to be changed only by this class. (In other cases, it makes perfect

\* sense to allow the state of a field to be changed outside the native

\* class; this is the case when a field acts as a "pointer" to an object

\* created elsewhere.)

\*/

private final Date fDateOfDiscovery;

}

Other immutable classes from the java standard API are :

* java.lang.String
* java.math.BigInteger and java.math.BigDecimal
* The wrapper classes for the primitive types: java.lang.Integer, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Boolean, java.lang.Long, java.lang.Double, java.lang.Float
* java.util.Locale - representing a specific geographical, political, or cultural region.
* etc