# *Chapter Five*

# *OOP Concepts*

## Encapsulation

* Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. Java’s basic unit of encapsulation is the class.
* Refers to keeping all the related members (variables and methods) together in an object.
* Facilitates data abstraction, which is the relationship between a class and its data members. The **has-a** relationship is associated with encapsulation, whereas the **is-a** relationship is associated with inheritance.
* Encapsulation refers to the fact that the data variables (also called the properties) and the methods (also called the behavior) are encapsulated together inside a template called a class.
* Implements information hiding and modularity (abstraction).
* The data members (variable) encapsulated inside a class may be declared public, private, or protected. However, good object-oriented programming practice requires **tight encapsulation**.
* Tight encapsulation means all data members (variable) of the class should be declared private. That is, the code outside of the class in which the data members are declared can access them only through method calls, and not directly. This is called **data abstraction (or data hiding)**, because now the data is hidden from the user, and the user can have access to it only through the methods.
* The benefits of encapsulation are: reliable code, easy maintenance, extensibility, security, integrity and re-usability.
* Sample code for demonstrating encapsulation :

public class Student{

private String name;

private int age ;

private String department;

. . .

public void setname(String sname){

this.name=sname;

}

public void setDepartment(String sDepartment){

this.department=sDepartment;

}

public void setAge(int sAge){

this.age=sAge;

}

public String getname( ){

return this.name;

}

public String getDepartment( ){

return this.department;

}

public int getAge( ){

return this.age;

}

. . .

}

Public class StudentDemo{

public static void main (String [ ] args){

Student st=new Student();

St.setName(“Anwar Mohammed”);

St.setDepartment(“computer science”);

St.setAge(26);

System.out.println(“student name:”+st.getName());

System.out.println(“student Department:”+st.getDepartment());

System.out.println(“student Age:”+st.getAge());

. . . .

}

**Note:** A class's private **fields** can be manipulated only by methods of that class. So a client of an objectthat is, any class that calls the object's methods calls the class's public methods to manipulate the private fields of an object of the class. Classes often provide public methods to allow clients of the class to **set** (i.e., assign values to) or **get** (i.e., obtain the values of) private instance variables.

## Inheritance

* You can derive a class from another class. If you do so, your class, called a subclass, inherits the state (variables) and the behavior (methods) of the class from which it is derived. The class from which you derive your class is called the superclass. The property of inheriting the state and behavior of the superclass is called inheritance.
* Inheritance can also be defined as the process whereby one object acquires characteristics from one or more other objects the same way children acquire characteristics from their parents.
* A subclass is a specialized version of a superclass. It inherits all of the variables and methods defined by the super class and add its own, unique elements.
* Java supports inheritance by allowing one class to incorporate another class into its declaration by using the **extends** keyword. Thus, the subclass adds to (extends) the superclass.

Class <subclass-name> **extends** <superclass-name>{

// body of class

}

* Being a superclass for a subclass does not mean that the superclass cannot be used by itself.
* Unlike C++, Java offers only single inheritance; that is, a class can be directly derived only from one superclass.
* The **is-a** relationship corresponds to inheritance. i.e. sub-class **is-a** superclass.
* The main advantages of inheritance are :
  + - **Code reusability:** Inheritance automates the process of reusing the code of the super classes in the subclasses. With inheritance, an object can inherit its more general properties from its parent object, and that saves the redundancy in programming.
    - **Code maintenance:** Organizing code into hierarchical classes makes its maintenance and management easier.
    - **Implementing OOP:** Inheritance helps to implement the basic OOP philosophy to adapt computing to the problem and not the other way around, because entities (objects) in the real world are often organized into a hierarchy.
* Simple example of inheritance:

|  |  |
| --- | --- |
| public class Person{  private String name;  private String sex;  private int age;  private String dateofBirth;  …  Public void setName(String name){  this.name=name;  }  Public void setSex(String sex){  this.sex=sex;  }  Public void setAge(int age){  this.age=age;  }  Public void setDateOfBirth(String dateofBirth){  this.dateofBirth=dateofBirth;  }  Public String getName( ){  return this.name;  }  Public String getSex( ){  return this.sex;  }  Public int setAge( ){  return this.age;  }  Public String setDateOfBirth( ){  return this.dateofBirth;  }  } | Public class Student extends person {  Private string department;  Private String year;  Private double gpa;  …  Public void setDepartment(String department){  this.department=department;  }  Public void setYear(String year){  this.year=year;  }  Public void setGPA(double gpa){  this.gpa=gpa;  }  Public String getDepartment( ){  return this.department;  }  Public String getyear( ){  return this.year;  }  Public double getGPA( ){  return this.gpa;  }  }  Public class StudentDemo{  Public static void main(String args[ ]{  Student st=new Student ( );  st.setName(“Arsema”);  st.setDepartment(“IT”);  system.out.println(st.getName( ));  system.out.println(st.getDepartment( )) |

* On the above example class student inherits class person, person is a super-class and student is a sub-class. Student inherits the behavior of the super-class person.no need for defining redundant code.
* St object of student can refers to set and get method of person class because student is part of person.(student **is-a** person)
* A subclass inherits only non-private members of the superclass—that is, the members that do not have an access modifier of private.in the above code private fields of person cannot be inherited by student.

### Constructor and inheritance

* It is possible for both super-classes and subclasses to have their own constructors.
* If both the subclass and the super class have no an explicit constructor, the superclass and subclass constructor of the object is constructed automatically using its implicit default constructor.
* When only the subclass defines a constructor, the process is straightforward: simply construct the subclass object. The superclass portion of the object is constructed automatically using its default constructor.
* When both the superclass and the subclass define constructors both the superclass and subclass constructors must be executed. In this case you must use another of Java’s keywords, **super**, which has two general forms. The first one is calls a superclass constructor and the second one is used to access member of the super class.
* If we define a superclass explicit a constructor (with parameters) the compiler will not provide the default constructor.at this time the subclass constructor must call the super class constructor explicitly by using super keyword with parameter .

For example, consider the following code fragment:

class A {

int myNumber;

A(int i) {

myNumber = i;

}

}

class B extends A {

String myName;

B (String name) {

Super(30);// without calling the supper class constructor it generates error

myName = name;

}

}

* When Are Constructors Called? Or When a subclass object is created, whose constructor is executed first?

Answer is that in a class hierarchy, constructors are **called in order of derivation**, from superclass to subclass. Since super( )must be the first statement executed in a subclass’ constructor, this order is the same whether or not super( )is used.

// Demonstrate when constructors are called.

// Create a super class.

class A {

A() {

System.out.println("Constructing A.");

}

}

// Create a subclass by extending class A.

class B extends A {

B() {

System.out.println("Constructing B.");

}

}

// Create another subclass by extending B.

class C extends B {

C() {

System.out.println("Constructing C.");

}

}

// Create another subclass by extending B.

class C extends B {

C() {

System.out.println("Constructing C.");

}

}

The output from this program is shown here:

Constructing A.

Constructing B.

Constructing C.

As you can see, the constructors are called in order of derivation.

Exercise:

In a class hierarchy, in what order are the constructors called?

### Superclass References and Subclass Objects

* A reference variable for one class type cannot normally refer to an object of another class type.
* There is an exception to Java’s strict type enforcement. A reference variable of a subclass can be assigned to a reference of superclass because the subclass is derived from the superclass. But after the assignment you will have access only to those parts of the object defined by the superclass.
* We can’t assign reference of super class to the reference of subclass.

Example that demonstrates a superclass reference can refer to a subclass object:

class X {

int a;

X(int i) { a = i; }

}

class Y extends X {

int b;

Y(int i, int j) {

super(j);

b = i;}

}

class SupSubRef {

public static void main(String args[]) {

X x = new X(10);

X x2;

Y y = new Y(5, 6);

//y=x; error because x object is superclass and y is objet of subclass

x2 = x; // OK, both of same type

System.out.println("x2.a: " + x2.a);

x2 = y; // still Ok because Y is derived from X

System.out.println("x2.a: " + x2.a);

// X references know only about X members

x2.a = 19; // OK

// x2.b = 27; // Error, X doesn't have a b member

}

}

Exercise 1:

A laptop is a specific kind of computer, and it has a network card installed on it. A desktop is also a specific kind of computer. Which of the following statements are true?

A. A laptop is-a computer.

B. A computer is-a laptop.

C. A laptop has-a network card.

D. A computer is-a desktop

## Method Overriding

* In a class hierarchy, when a method in a subclass has the same return type and signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
* Overriding deals with two methods, one in the parent class and the other one in the child class and has the same name and signatures.
* The same signature in this case means the same name and the same number of parameters and their types appearing in the same order.
* In other words, method overriding is a feature of Java that lets the programmer declare and implement a method in a subclass that has the same signature as a method in the superclass.

Example of Method overriding:

|  |  |
| --- | --- |
| class A {  int i, j;  A(int a, int b) {  i = a;  j = b;  }  // display i and j  void show() {  System.out.println("i and j: " + i + " " + j);  }  } | class B extends A {  int k;  B(int a, int b, int c) {  super(a, b);  k = c;  }  // display k – this overrides show() in A  void show() {// This show( )in B overrides the one defined by A  System.out.println("k: " + k);  }  } |

class OverrideDemo {

public static void main(String args[]) {

B subOb = new B(1, 2, 3);

subOb.show(); // this calls show() in B

}

}

The output produced by this program is shown here:

k: 3

* If you want to access the superclass version of an overridden method, you can do so by using super. For example, in this version of B, the superclass version of show( )is invoked within the subclass’ version. This allows all instance variables to be displayed.

class B extends A {

int k;

B(int a, int b, int c) {

super(a, b);

k = c;

}

void show() {

super.show(); // this calls A's show()

System.out.println("k: " + k);

}

}

If you substitute this version of show( )into the previous program, you will see the following output:

i and j: 1 2

k: 3

* The following are the rules for overriding a method:
* You cannot override a method that has the final modifier.
* You cannot override a static method to make it non-static and vice versa.
* The overriding method and the overridden method must have the same return type
* The number of parameters and their types in the overriding method must be same as in the overridden method and the types must appear in the same order. However, the names of the parameters may be different.
* You cannot override a method to make it less accessible. For example, overriding a public method and declaring it protected will generate a compiler error.

Exercise 1:

Consider the following classes defined in separate source files:

class SuperClass {

SuperClass() {

System.out.print(" I was in Super Class." );

}

public void aMethod (int i) {

System.out.print (" The value of i is " + i );

}

}

class SubClass extends SuperClass {

public void aMethod(int j) {

System.out.print (" The value of j is " + j );

}

}

class Test {

public static void main(String args[]) {

SubClass sub = new SubClass();

sub.aMethod(5);

}

}

What output is generated when the class Test is run?

Exercise 2:

**protected int aMethod(String st, int i,double number){….}**

Which one is Valid and Invalid Overriding of the above Method? And write the reason for your answer

1. protected int aMethod(String st,int i, double number){….}
2. protected int aMethod(String st,int j, double num){….}
3. protected double aMethod(String st,int i, double number){….}
4. protected int aMethod(String st,int i, double number,int j){….}
5. int aMethod(String st,int i, double number){….}
6. private int aMethod(String st,int i, double number){….}

## Method Overloading

* Method overloading is helpful when the same task is to be performed in slightly different ways under different conditions.
* Overloading lets you define the same operation in different ways for **different data** but Overriding lets you define the same operation in different ways for **different object types**.
* Method overloading is a feature of Java that facilitates defining multiple methods in a class with identical names.
* In contrast to overriding methods, no two overloaded methods could have the same parameter types in the same order.
* The return types in overloaded methods may be the same or different, while the return type of an overriding method must match that of the overridden method.
* For example, assume that we want to write a class for calculating the area of different shapes such as a triangle, a rectangle, and a square. All of these tasks are the same to the extent that they all relate to calculating the area, but they are performed differently because the formulae to calculate the area in all of these cases would differ and would require different input.

1. class TestAreaCalculator {

2. public static void main(String[] args) {

3. AreaCalculator ac = new AreaCalculator();

4. System.out.println("Area of a rectangle with length 2.0, and width 3.0:" + ac.calculateArea(2.0f, 3.0f));

5. System.out.println("Area of a triangle with sides 2.0, 3.0, and 4.0: "

+ ac.calculateArea(2.0, 3.0, 4.0));

6. System.out.println("Area of a circle with radius 2.0: " +

ac.calculateArea(2.0));

7. }

8. }

9. class AreaCalculator {

10. float calculateArea(float length, float width) {//calculate area of rectangle

11. return length\*width;

12. }

13. double calculateArea(double radius) {//overloaded to calculate for circle

14. return ((Math.PI)\*radius\*radius);

15. }

16. double calculateArea(double a, double b, double c) {//overloaded to calculate for triangle

17. double s = (a+b+c)/2.0;

18. return Math.sqrt(s\*(s-a)\*(s-b)\*(s-c));

19. }

20. }

The output from this code follows:

Area of a rectangle with length 2.0, and width 3.0: 6.0

Area of a triangle with sides 2.0, 3.0, and 4.0: 2.9047375096555625

Area of a circle with radius 2.0: 12.566370614359172

* Overloaded versions of a method can have different modifiers.
* A method in a subclass inherited from its superclass can be both overridden and overloaded.

Example :

class VolumeCalculator extends AreaCalculator {

int calculateArea (int i, int j) {//overload the method in the super class. In the above code

}

double calculateVolume (double x, int y, double z);

}

}

**Comparison of Overriding and Overloading**

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Overriding** | **Overloading** |
| Name | Must have the same method name. | Must have the same method name |
| Access modifier | The overriding method cannot be less public than the overridden method. | Different versions can have different access modifiers. |
| Parameter types | Identical list of parameter types. | The set of parameter types of any two versions must be different or should appear in a different order. |
| Return types | Must be the same. | Can be the same or different |

Exercise :

Consider the following class:

1. class MyClass {

2. public int myMethod (double a, int i) {

3. }

4.

5. }

Which of the following methods, if added at line 4 independently, would be valid?

A.public int myMethod(int i, double a) { }

B.public double myMethod(double b, int j){ }

C.public int myMethod(double a, double b, int i){ }

D.public int yourMethod(double a, int i) { }

## Polymorphism

* (from the Greek, meaning “many forms”) is the quality that allows one interface to access a general class of actions.
* means the ability of a single variable of a given type to be used to reference objects of different types, and automatically call the method that is specific to the type of object the variable references.
* Refers to a feature of Java that allows an object of a superclass to refer to an object of any subclass.
* Polymorphism helps to prevent the code from becoming more complex and cumbersome, by allowing you to make different implementations of related functionalities (behaviors) under one name: object or method.
* The concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities.
* Way we can implement polymorphism :
* Method overloading b/c: want to implement essentially the same method for different types of data.
* Method overriding b/c: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
* Inheritance: by enabling different objects respond to the same message in different ways.
* Abstract class and interface b/c: different class implements the abstract method or interface in different ways.

## Abstract class

* A class in which one or more methods are not implemented is defined as an **abstract class**. A class in which all methods are implemented is a **concrete class**.
* You design abstract classes to implement common sets of behavior, which are then shared by the concrete (instantiable) classes you derive from them.
* Java lets us define a method without implementing it by declaring the method with the abstract modifier. An abstract method has no body; it simply has a signature definition followed by a semicolon.
* Here are the rules about abstract methods and the abstract classes that contain them:
* Any class with an abstract method is automatically abstract itself and must be declared as such.
* An abstract class cannot be instantiated.
* A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides an implementation (i.e., a method body) for all of them. Such a class is often called a concrete subclass, to emphasize the fact that it is not abstract.
* If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract.
* A class can be declared abstract even if it does not actually have any abstract methods.
* **Example of an Abstract Class and Concrete Subclasses**

public abstract class Shape {

public abstract double area(); // Abstract methods: note

public abstract double circumference(); // semicolon instead of body.

}

class Circle extends Shape {

public static final double PI = 3.14159265358979323846;

protected double r; // Instance data

public Circle(double r) { this.r = r; } // Constructor

public double getRadius() { return r; } // Accessor

public double area() { return PI\*r\*r; } // Implementations of

public double circumference() { return 2\*PI\*r; } // abstract methods.

}

class Rectangle extends Shape {

protected double w, h; // Instance data

public Rectangle(double w, double h) { // Constructor

this.w = w; this.h = h;

}

public double getWidth() { return w; } // Accessor method

public double getHeight() { return h; } // Another accessor

public double area() { return w\*h; } // Implementations of

public double circumference() { return 2\*(w + h); } // abstract methods.

}

Each abstract method in Shape has a semicolon right after its parentheses. There are no curly braces, and no method body is defined. Using the classes defined above, we can now write code like this:

Circle crc = new Circle(2.0);

Rectangle rct = new Rectangle(1.0, 3.0);

## An interface

* An interface defines a set of methods, without actually defining their implementation.
* Interfaces are syntactically similar to abstract classes. However, in an interface, no method can include a body. It specifies what must be done, but not how. You could think of it as a “pure” abstract class.
* Once an interface is defined, any number of classes can implement it. Also, one class can implement any number of interfaces.
* We can’t create instance variable inside interface.
* An interface is a template that contains only static variables (but must be initialized) and method declarations. The variables of an interface are inherently public, final, and static, while the methods in an interface are inherently public and abstract. Because the methods in an interface are inherently abstract, any class that implements the interface must provide implementation for all the interface methods, if the class itself is not abstract.
* You define an interface by using the keyword interface, as shown in the following:

interface <InterfaceName> {

<dataType1> <var1>;

<dataType2> <var2>;

<ReturnType1> <methodName1> ( );

<ReturnType2> <methodName2>(<parameters>);

} // interface definition ends here.

* The methods declared in an interface are implicitly public and abstract. Therefore, when you implement them in a class, you must declare them public.
* There are a few more features of interfaces related to inheritance and implementation:
* A class can extend (inherit from) another class (only one) by using the keyword extends, but it can inherit from one or more interfaces by using the keyword implements.
* Just like a class, an interface can also extend one or more interfaces by using the keyword extends.
* An interface cannot implement any interface or class.
* Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

<access modifier> class <class name> extends superclass implements <interface name>{

// class-body

}

To implement more than one interface, the interfaces are separated with a comma.

* The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.
* The concrete class that implements the interface must implement all the method inside the interface.
* An abstract class can omit implementing some or all of the methods required by an interface.
* An interface allows multiple inheritances of interfaces.

Code example:

public interface Shape {

double pi=3.14;

double getArea();

double getPerimeter();

}

public class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

public double getRadius() {

return radius;

}

public double getArea() {

return pi \* Math.pow(this.radius, 2);

}

public double getPerimeter() {

return pi \* this.radius \* 2;

}

}

public class Rectangle implements Shape {

private double width;

private double height;

public Rectangle(double width, double height) {

this.width = width;

this.height = height;

}

public double getWidth() {

return width;

}

public double getHeight() {

return height;

}

public double getArea() {

return this.width \* this.height;

}

public double getPerimeter() {

return 2 \* (this.width + this.height);

}

}

public class Square extends Rectangle {

public Square(double side) {

super(side, side);

}

}

public class ShapeDemo {

public static void main(String[] args) {

Circle c = new Circle(5.0);

Rectangle r = new Rectangle(3, 4);

Square s = new Square(6);

System.out.println(“area of rectangle:”+ r.getArae());

System.out.println(“perimeter of rectangle:”+c.getPerimeter());

System.out.println(“area of circle:”+ c.getArae());

System.out.println(“perimeter of circle:”+c.getPerimeter());

System.out.println(“area of squere:”+ s.getArae());

System.out.println(“perimeter of squere:”+s.getPerimeter());

}

}

* One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

// One interface can extend another.

interface A {

void meth1();

void meth2();

}

// B now includes meth1() and meth2() – it adds meth3().

interface B extends A {

void meth3();

}

// This class must implement all of A and B

class MyClass implements B {

public void meth1() {

System.out.println("Implement meth1().");

}

public void meth2() {

System.out.println("Implement meth2().");

}

public void meth3() {

System.out.println("Implement meth3().");

}

}

class IFExtend {

public static void main(String arg[]) {

MyClass ob = new MyClass();

ob.meth1();

ob.meth2();

ob.meth3();

}

}