



Index:

1. Introduction to OPP

2. class

3. object

4. method

5. Constructor

6. Argument passing

7. Access Modifiers

8. Encapsulation

9. Inheritance

10. Polymorphism

11. super keyword

12. this keyword

13. final keyword

14. Abstraction

15. Generics

16. Lambda Expressions

17. Type Casting

18. compareTo() method

19. Anonymous class

20. Exception handling

21. UML

22.-----

23.-----

24.-----

25.-----

26.-----

27.-----

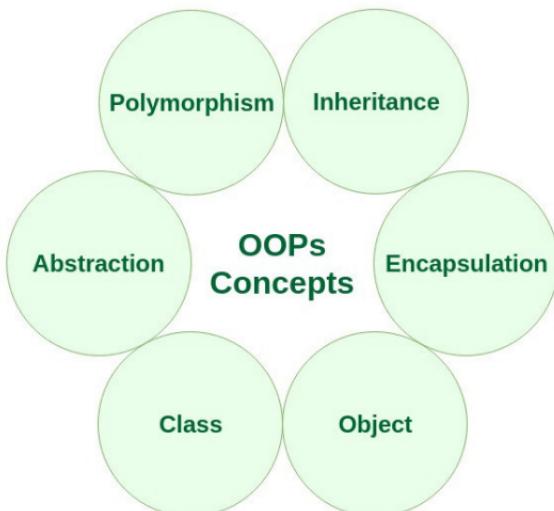
28.-----

1. Introduction to OPP

OOP stands for Object-Oriented Programming.

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use objects in programming, they use objects as a primary source to implement what is to happen in the code. Objects are seen by the viewer or user, performing tasks assigned by you. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an Object Oriented Programming language



OOPS concepts are as follows:

1. Class
2. Object
3. Method and method passing
4. Pillars of OOPs
 - (a) Abstraction
 - (B) Encapsulation
 - (C) Inheritance
 - (D) Polymorphism
 - (i) Compile-time / static Polymorphism
(Method Overloading, Constructor Overloading)
 - (ii) Runtime / dynamic Polymorphism
(Method Overriding)

2. class

A class is a user-defined blueprint or prototype from which objects are created.

1. Class is a set of object which shares common characteristics/ behavior and common properties/ attributes.
2. Class is not a real world entity. It is just a template or blueprint or prototype from which objects are created.
3. Class does not occupy memory.
4. Class is a group of variables of different data types and group of methods.

It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times.

A class in java can contain:

- 1. Modifiers:** A class can be public or have default access (Refer to this for details).
- 2. Class name:** The class name should begin with the initial letter capitalized by convention.
- 3. Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- 4. Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- 5. Body:** The class body is surrounded by braces, { }.

Syntax to declare a class:

```
accessModifier class className  
{  
    data member;  
    method;  
    constructor;  
    nested class;  
    interface;  
}
```

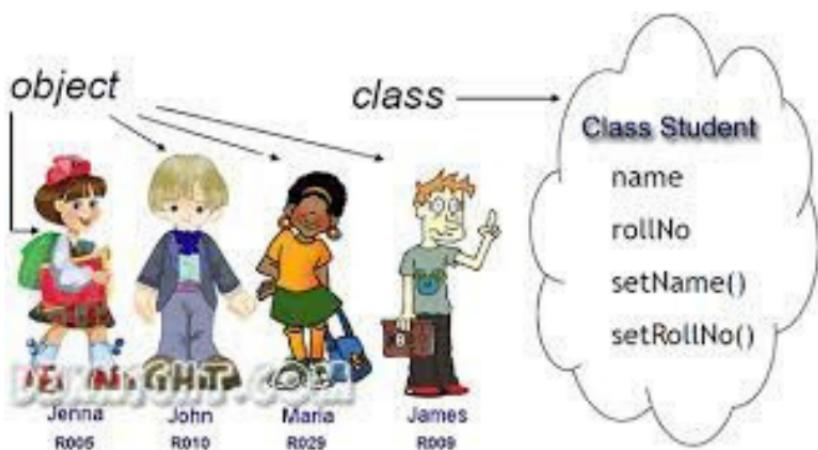
3. object

An object is a basic unit of Object-Oriented Programming that represents real-life entities.

A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
2. **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity:** It is a unique name given to an object that enables it to interact with other objects.
4. **Method:** A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from other languages like C, C++, and Python.

class and object :

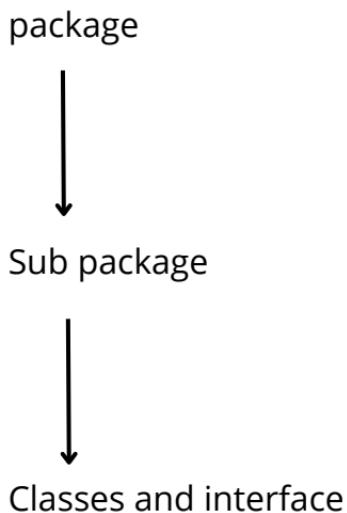


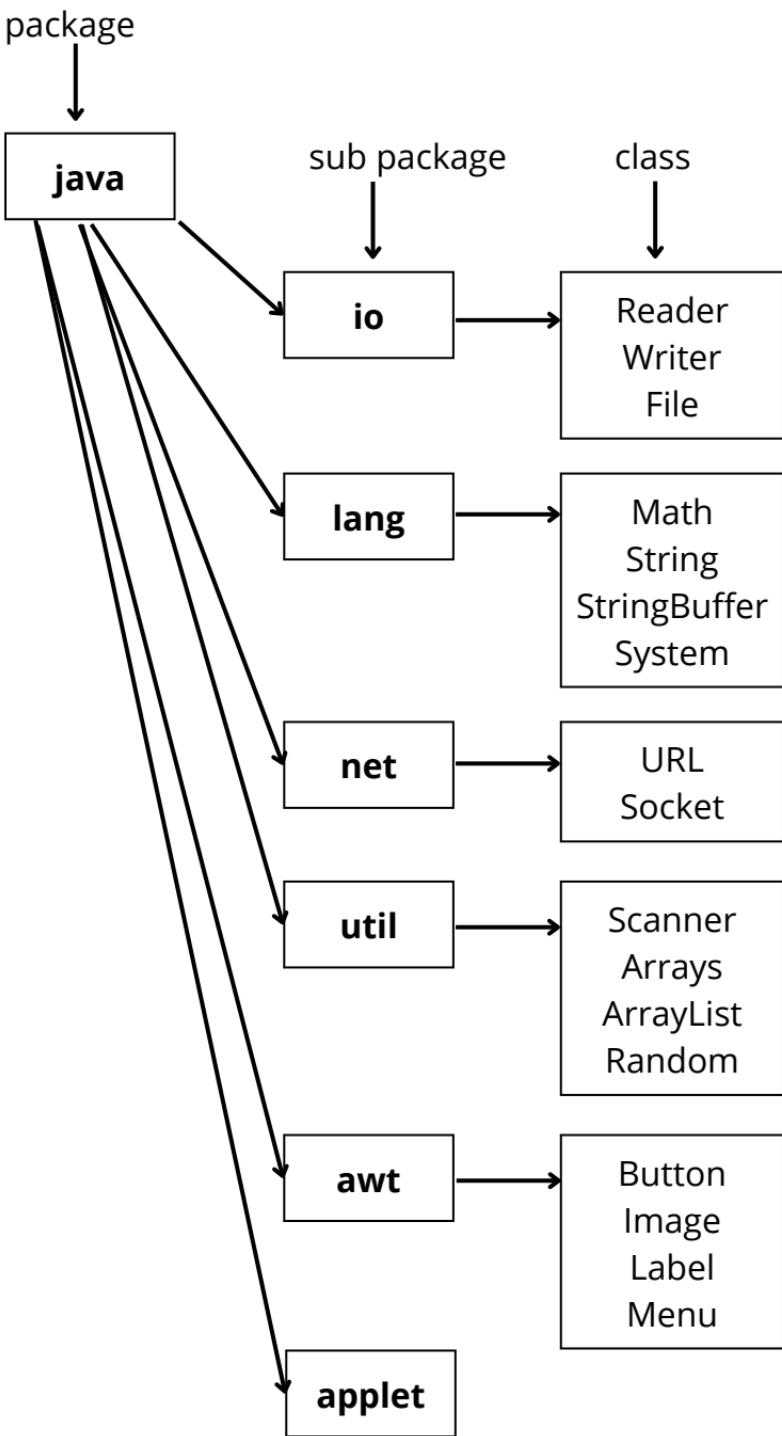
- **Package :**

A package is a group of related classes, interfaces and sub packages.

There are 2 types of package in Java.

1. Built-in package (import java.util.Scanner;)
2. User-defined package





```
import java.util.Scanner;
```

The diagram illustrates the structure of the import statement `import java.util.Scanner;`. Three arrows point from the words "package", "sub package", and "class" to the corresponding parts of the code: "java", "util", and "Scanner".

```
graph TD; package --> java; sub_package --> util; class --> Scanner;
```

- **Make a project OOP then make a package oop under OOP project then make two class Teacher and Test under the oop package. After then create two object teacher1 and teacher2 in the main method which is in the Test class.**

```
package oop;

public class Teacher {

    String name, gender;
    int phone;
}
```

```
package oop;

public class Test {
    public static void main(String[] args){
        Teacher teacher1; // object declare
        teacher1 = new Teacher(); // object create

        teacher1.name = "Hamim Talukdar";
        teacher1.gender = "Male";
        teacher1.phone = 1731767273;

        System.out.println("Name : "+teacher1.name);
        System.out.println("Gender : "+teacher1.gender);
        System.out.println("Phone : "+teacher1.phone);
    }
}
```

```
System.out.println();  
  
Teacher teacher2 = new Teacher(); // object  
declare and create
```

```
teacher2.name = "Ava Talukdar";  
teacher2.gender = "Female";  
teacher2.phone = 1731767274;
```

```
System.out.println("Name : "+teacher2.name);  
System.out.println("Gender : "+teacher2.gender);  
System.out.println("Phone : "+teacher2.phone);  
}  
}
```

or,

```
class Teacher {
```

```
    String name, gender;  
    int phone;  
}
```

```
public class NMP {  
    public static void main(String[] args){  
        Teacher teacher1; // object declare  
        teacher1 = new Teacher(); // object create  
  
        teacher1.name = "Hamim Talukdar";  
        teacher1.gender = "Male";  
        teacher1.phone = 1731767273;
```

```
System.out.println("Name : "+teacher1.name);
System.out.println("Gender : "+teacher1.gender);
System.out.println("Phone : "+teacher1.phone);

System.out.println();

Teacher teacher2 = new Teacher(); // object
declare and create

teacher2.name = "Ava Talukdar";
teacher2.gender = "Female";
teacher2.phone = 1731767274;

System.out.println("Name : "+teacher2.name);
System.out.println("Gender : "+teacher2.gender);
System.out.println("Phone : "+teacher2.phone);

}
```

Output:

Name : Hamim Talukdar
Gender : Male
Phone : 1731767273

Name : Ava Talukdar
Gender : Female
Phone : 1731767274

Here,

The “new” operator dynamically allocates memory for the objects in the heap and returns a reference to it. The reference or dynamically allocated memory address is stored in the object variable.

4. method

A method is a collection of statements that perform some specific task and return the result to the caller. A method can also perform some specific task without returning anything.

Methods allow us to reuse the code without retying it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

1. A method is like a function i.e. used to expose the behavior of an object.
2. It is a set of codes that perform a particular task.

Syntax: Declare a method

```
accessModifier returnType methodName(parameters)
{
    //body
}
```

Note: Methods are time savers and help us to reuse the code without retying the code.

There 2 types of methods:

1. return type method/function;
2. void type method/function;

1. return type method/function :

In Java, a method's return type specifies the type of value that the method will return when it is executed. The return type is declared before the method name and is defined using a data type or a class.

Syntax:

```
AccMo <return_type> methodName(parameters)
{
    // method body
    // return statement
}
```

- **What is the output of the following Java program fragment:**

```
import java.util.Scanner;
```

```
class Test{  
    public int addNumbers(int a, int b) {  
        int sum = a + b;  
        return sum;  
    }
```

```
    public static void main(String[] args) {  
        Test ob = new Test();  
        Scanner scan = new Scanner(System.in);
```

```
            System.out.print("Enter 1st number : "); int a =  
            scan.nextInt();
```

```
            System.out.print("Enter 2nd number : "); int b =  
            scan.nextInt();
```

```
            System.out.println("The sum is  
"+ob.addNumbers(a, b));  
            scan.close();  
        }  
    }
```

Output:

Enter 1st number : 5

Enter 2nd number : 2

The sum is 7

- **What is the output of the following Java program fragment:**

```
import java.util.Scanner;

class Test{
    public String result(int a){
        String str1;
        if(a%2==0 & a>2){
            str1 = "YES";

            return str1;
        }
        else{
            str1 = "NO";

            return str1;
        }
    }

    public static void main(String[] args) {
        Test ob = new Test();
        Scanner scan = new Scanner(System.in);
        int a = scan.nextInt();
        System.out.println(ob.result(a));
        scan.close();
    }
}
```

```
or,
import java.util.Scanner;

class Test{
    public String result(int a){
        if(a%2==0 & a>2){
            return "YES";
        }
        else{
            return "NO";
        }
    }

    public static void main(String[] args) {
        Test ob = new Test();
        Scanner scan = new Scanner(System.in);
        int a = scan.nextInt();
        System.out.println(ob.result(a));
        scan.close();
    }
}
```

Output:

2
NO

- **What is the output of the following Java program fragment:**

```
class returnValue {  
    int square (int value)  
    {  
        return value * value;  
    }  
  
}  
  
public class Test {  
    public static void main(String[] args) {  
        returnValue ob1 = new returnValue(), ob2 =  
        new returnValue();  
        int result = ob1.square(5);  
        System.out.println("square is : "+result);  
        System.out.println(ob2.square(4));  
    }  
}
```

Output:

square is : 25
16

- **What is the output of the following Java program fragment:**

```
class returnValue {  
    int square ()  
    {  
        return 5 * 5;  
    }  
  
}  
  
public class Test {  
    public static void main(String[] args) {  
        returnValue ob1 = new returnValue();  
        int x = ob1.square();  
        System.out.println(x);  
    }  
}
```

Output:

25

2. void type method/function :

In Java, a void method is a method that does not return any value. It is used when you want the method to perform certain actions or operations without producing a result that needs to be returned to the caller.

Syntax:

```
AccMod <void> methodName(parameters) {  
    // method body  
}
```

- **What is the output of the following Java program fragment:**

```
import java.util.Scanner;

class Test{
    public void printMessage(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        Test ob = new Test();
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a message : ");
        String a = scan.nextLine();

        ob.printMessage(a);
        scan.close();
    }
}
```

Output:

Enter a message : I am HaMeem
I am HaMeem

- **What is the output of the following Java program fragment:**

```
class Teacher {  
  
    String name, gender;  
    int phone;  
  
    void setInformation(String s1, String s2, int i)  
    {  
        name = s1;  
        gender = s2;  
        phone = i;  
    }  
  
    void displayInformation()  
    {  
        System.out.println("Name : " + name);  
        System.out.println("Gender : " + gender);  
        System.out.println("Phone : " + phone);  
        System.out.println();  
    }  
}  
  
public class hameem {  
    public static void main(String[] args) {  
        Teacher teacher1; // object declare  
        teacher1 = new Teacher(); // object create  
  
        teacher1.setInformation("Hamim  
Talukder", "Male", 1731767273);  
        teacher1.displayInformation();
```

```
Teacher teacher2 = new Teacher(); // object  
declare and create  
  
teacher2.setInformation("Ava  
Talukder","Female",1731767275);  
teacher2.displayInformation();  
}  
}
```

Output:

Name : Hamim Talukder
Gender : Male
Phone : 1731767273

Name : Ava Talukder
Gender : Female
Phone : 1731767275

- **What is the output of the following Java program fragment:**

```
class Overload{  
    void add(int a, int b){  
        System.out.println(a+b);  
    }  
  
    void add(double a, double b){  
        System.out.println(a+b);  
    }  
  
    void add(int a, int b, int c){  
        System.out.println(a+b+c);  
    }  
  
    void add(){  
        System.out.println("Nothing to add.");  
    }  
}  
public class Test{  
    public static void main(String[] args)  
    {  
        Overload ob = new Overload();  
        ob.add();  
        ob.add(5, 10);  
        ob.add(6.5, 5.5);  
        ob.add(5, 10, 20);  
    }  
}
```

Output:

Nothing to add.

15

12.0

35

- **Automatic Type Conversion :**
- **What is the output of the following Java program fragment:**

```
class Overload{  
  
    void add(double a, double b){  
        System.out.println(a+b);  
    }  
  
    void add(int a, int b, int c){  
        System.out.println(a+b+c);  
    }  
  
    void add(){  
        System.out.println("Nothing to add.");  
    }  
  
}  
public class Test{  
    public static void main(String[] args)  
    {  
        Overload ob = new Overload();  
        ob.add();  
        ob.add(5, 10); // Automatic Type Conversion  
        ob.add(6.5, 5.5);  
        ob.add(5, 10, 20);  
    }  
}
```

Output:

Nothing to add.

15.0

12.0

35

- **variable length arguments :**

Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments.

Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.

Syntax of Varargs :

Internally, the Varargs method is implemented by using the single dimensions arrays concept.

Hence, in the Varargs method, we can differentiate arguments by using Index. A variable-length argument is specified by three periods or dots(...).

```
public static void fun(int ... a)
{
    // method body
}
```

- **What is the output of the following Java program fragment:**

```
class AddDemo{  
void add(int ... n){  
int sum = 0;  
  
for(int i=0; i<n.length; i++){  
sum = sum + n[i];  
}  
  
System.out.println(sum);  
}  
}  
  
public class Test{  
public static void main(String[] args)  
{  
AddDemo ob = new AddDemo();  
ob.add(10, 20);  
ob.add(10, 20, 30);  
ob.add(10, 20, 30, 40);  
}  
}
```

Output:

30
60
100

- What is the output of the following Java program fragment:

```
class AddDemo{  
    void add(int ... n){  
        int sum = 0;  
        for(int x : n){  
            sum = sum + x;  
        }  
  
        System.out.println(sum);  
    }  
}  
  
public class Test{  
    public static void main(String[] args)  
    {  
        AddDemo ob = new AddDemo();  
        ob.add(10, 20);  
        ob.add(10, 20, 30);  
        ob.add(10, 20, 30, 40);  
    }  
}
```

Output:

30
60
100

5. Constructor

A constructor in Java is a special method that is used to initialize objects.

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the heap memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

Properties of constructor:

1. Constructor has the same name as that of the class it belongs.
2. Constructor is a special type of method.
3. It has no return type of method.
4. It is called automatically.
5. Default constructor (no parameter), parameterized constructor (parameter).

Note: It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor (constructor with no-arguments) if your class doesn't have any.

Types of Constructors in Java:

primarily there are two types of constructors in java:

1. Parameterized Constructor
2. Default Constructor
- No-argument constructor

How Constructors are Different From Methods in Java?

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or void if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Syntax:

```
class Geek  
{
```

```
// A Constructor  
Geek() {  
    statement;  
}  
  
}
```

```
// We can create an object of the above class  
// using the below statement. This statement  
// calls above constructor.  
Geek obj = new Geek();
```

1. Parameterized Constructor :

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Example:

```
import java.io.*;  
class Geek {  
  
    String name;  
    int id;  
    Geek(String name, int id)  
    {  
        this.name = name;  
        this.id = id;  
    }  
}  
  
class GFG {  
    public static void main(String[] args)  
    {
```

```
Geek geek1 = new Geek("avinash", 68);
System.out.println("GeekName :" + geek1.name
+ " and GeekId :" + geek1.id);
}
}
```

Output:

GeekName :avinash and GeekId :68

Remember: Does constructor return any value?

There are no “return value” statements in the constructor, but the constructor returns the current class instance. We can write ‘return’ inside a constructor.

Now the most important topic that comes into play is the strong incorporation of OOPS with constructors known as constructor overloading. Just like methods, we can overload constructors for creating objects in different ways. The compiler differentiates constructors on the basis of the number of parameters, types of parameters, and order of the parameters.

Example:

```
import java.io.*;  
  
class Geek {  
    // constructor with one argument  
    Geek(String name)  
    {  
        System.out.println("Constructor with one "  
            + "argument - String : " + name);  
    }  
  
    // constructor with two arguments  
    Geek(String name, int age)  
    {  
  
        System.out.println(  
            "Constructor with two arguments : "  
            + " String and Integer : " + name + " " + age);  
    }  
  
    Geek(long id)  
    {  
        System.out.println(  
            "Constructor with one argument : "  
            + "Long : " + id);  
    }  
}  
  
class GFG {  
    public static void main(String[] args)  
    {
```

```
// Invoke the constructor with one argument of
// type 'String'.
Geek geek2 = new Geek("Shikhar");

// Invoke the constructor with two arguments
Geek geek3 = new Geek("Dharmesh", 26);

// Invoke the constructor with one argument of
// type 'Long'.
Geek geek4 = new Geek(325614567);
}

}
```

Output:

Constructor with one argument - String : Shikhar
Constructor with two arguments : String and
Integer : Dharmesh 26
Constructor with one argument : Long :
325614567

2. Default Constructor :

A constructor that has no parameters is known as default the constructor. A default constructor is invisible. And if we write a constructor with arguments or no arguments then the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can't change the default constructor.

- What is the output of the following Java program fragment:**

```
import java.io.*;  
  
class GFG {  
    GFG()  
    {  
        System.out.println("Default constructor");  
    }  
    public static void main(String[] args)  
    {  
  
        GFG hello = new GFG();  
    }  
}
```

Output:

Default constructor

- **What is the output of the following Java program fragment:**

```
class Teacher {  
  
    String name, gender;  
    int phone;  
  
    Teacher(String s1, String s2, int i) // constructor  
    {  
        name = s1;  
        gender = s2;  
        phone = i;  
    }  
  
    Teacher() // Default Constructor  
    {  
        System.out.println("No value");  
    }  
  
    void displayInformation()  
    {  
        System.out.println("Name : " + name);  
        System.out.println("Gender : " + gender);  
        System.out.println("Phone : " + phone);  
        System.out.println();  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Teacher teacher1; // object declare
```

```
teacher1 = new Teacher("Hamim  
Talukder","Male",1731767273); // object create  
teacher1.displayInformation();  
  
Teacher teacher2 = new Teacher("Ava  
Talukder","Female",1731767275); // object  
declare and create  
teacher2.displayInformation();  
  
Teacher teacher3 =new Teacher();  
teacher3.displayInformation();  
}  
}
```

Output:

Name : Hamim Talukder
Gender : Male
Phone : 1731767273

Name : Ava Talukder
Gender : Female
Phone : 1731767275

No value

Name : null
Gender : null
Phone : 0

- **What is the output of the following Java program fragment:**

```
class Teacher {  
  
    String name, gender;  
    int phone;  
  
    void displayInformation()  
    {  
        System.out.println("Name : " + name);  
        System.out.println("Gender : " + gender);  
        System.out.println("Phone : " + phone);  
        System.out.println();  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
  
        Teacher teacher3 =new Teacher();  
        teacher3.displayInformation();  
    }  
}
```

Output:

Name : null
Gender : null
Phone : 0

Here,

If we do not make any constructor in Java it automatically creates a default constructor. In Java language the default value of String is null and for primitive data type here the integer variable is 0.

Thats why here String was printed null and integer variable was printed 0.

- **No-argument constructor :**

A constructor that has no parameter is known as the No-argument or Zero argument constructor. If we don't define a constructor in a class, then the compiler creates a constructor (with no arguments) for the class. And if we write a constructor with arguments or no arguments then the compiler does not create a default constructor.

Note: Default constructor provides the default values to the object like 0, null, etc. depending on the type.

Example:

```
import java.io.*;  
  
class Geek {  
    int num;  
    String name;  
  
    Geek()  
    {  
        System.out.println("Constructor called");  
    }  
}  
  
class GFG {  
    public static void main(String[] args)  
    {
```

```
Geek geek1 = new Geek();
System.out.println(geek1.name);
System.out.println(geek1.num);
}
}
```

Output:

Constructor called
null
0

Difference between Constructor And Method :

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or void if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Syntax:

```
class Geek  
{
```

.....

```
// A Constructor  
Geek() {  
}
```

.....

```
}
```

```
// We can create an object of the above class  
// using the below statement. This statement  
// calls above constructor.  
Geek obj = new Geek();
```

- **What is the output of the following Java program fragment:**

```
class Teacher {  
  
    String name, gender;  
    int phone;  
  
    Teacher(String s1, String s2, int i) // constructor  
    {  
        name = s1;  
        gender = s2;  
        phone = i;  
    }  
  
    void displayInformation()  
    {  
        System.out.println("Name : " + name);  
        System.out.println("Gender : " + gender);  
        System.out.println("Phone : " + phone);  
        System.out.println();  
    }  
}  
  
public class hameem {  
    public static void main(String[] args) {  
        Teacher teacher1; // object declare  
        teacher1 = new Teacher("Hamim  
Talukder", "Male", 1731767273); // object create  
        teacher1.displayInformation();  
    }  
}
```

```
    Teacher teacher2 = new Teacher("Ava  
Talukder","Female",1731767275); // object  
declare and create  
    teacher2.displayInformation();  
}  
}
```

Output:

Name : Hamim Talukder

Gender : Male

Phone : 1731767273

Name : Ava Talukder

Gender : Female

Phone : 1731767275

6. Argument passing

Two ways to pass an argument :

1. Call-by-value / pass-by-value
2. Call-by-reference / pass-by-reference

1. Call-by-value :

- If we call a method by passing-a-value (primitive data) then it is known as call-by-value.
 - The value is copied to a method parameter.
 - Changes to that formal parameter doesn't affect the actual parameter.
 - In call-by-value original value doesn't change.
-
- **What is the output of the following Java program fragment:**

```
class CallByValue{  
    void change(int i){  
        i = 20;  
    }  
}  
public class Test{  
    public static void main(String[] args)  
    {  
        CallByValue ob = new CallByValue();  
        int x =10; //primitive data  
        System.out.println("x before call : "+x);  
  
        ob.change(x);  
    }  
}
```

```
        System.out.println("x after call : "+x);
    }
}
```

Output:

x before call : 10

x after call : 10

2.Call-by-reference :

- If we call a method by passing-a-reference type data(object, String etc) then it is known as call-by-reference.
- changes to that formal parameter does affect the actual parameter.
- In call-by-reference original value gets changed.

- **What is the output of the following Java program fragment:**

```
class CallByReference{  
    String name;  
    void change(CallByReference r2){  
        r2.name = "Jim";  
    }  
}  
public class Test{  
    public static void main(String[] args)  
    {  
        CallByReference r1 = new CallByReference();  
        r1.name = "Hamim";  
        System.out.println("before calling : "+r1.name);  
  
        r1.change(r1);  
        System.out.println("after calling : "+r1.name);  
    }  
}
```

Output:

before calling : Hamim
after calling : Jim

- **What is the output of the following Java program fragment:**

```
class CallByReference{  
    String name;  
    void change(CallByReference r2, String str){  
        r2.name = str;  
    }  
}  
  
public class Test{  
    public static void main(String[] args)  
    {  
        CallByReference r1 = new CallByReference();  
        r1.name = "Hamim";  
        System.out.println("before calling : "+r1.name);  
  
        r1.change(r1, "Jim");  
        System.out.println("after calling : "+r1.name);  
    }  
}
```

Output:

before calling : Hamim
after calling : Jim

- **What is the output of the following Java program fragment:**

```
class CallByReference {  
    int x;  
  
    void change(CallByReference r2, int str) {  
        r2.x = str;  
    }  
  
}  
  
public class Test {  
    public static void main(String[] args) {  
        CallByReference ob = new CallByReference();  
        ob.x = 10;  
        System.out.println("x before call : " + ob.x);  
        ob.change(ob, 20);  
        System.out.println("x after call : " + ob.x);  
    }  
}
```

Output:

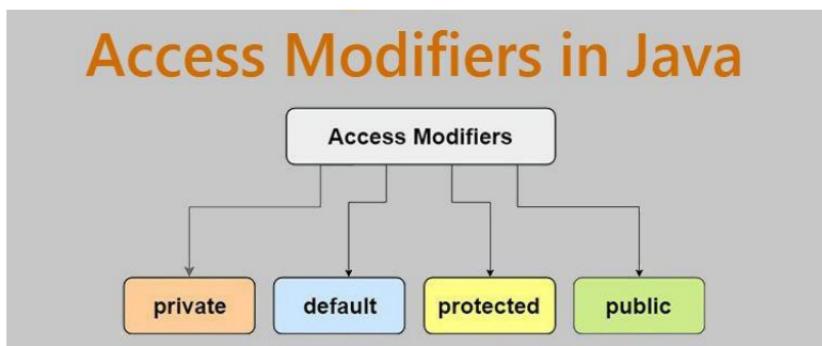
x before call : 10
x after call : 20

7. Access Modifiers

As the name suggests access modifiers in Java help to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

Diagram –



- **Certainly! Here's an example of a Java subclass:**

```
class Animal {  
    void eat() {  
        System.out.println("This animal eats food.");  
    }  
}  
  
class Dog extends Animal {  
    void bark() {  
        System.out.println("The dog barks.");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Dog myDog = new Dog();  
        myDog.eat(); // Inherited from the Animal class  
        myDog.bark(); // Defined in the Dog subclass  
    }  
}
```

Output:

This animal eats food.
The dog barks.

Here,

Dog class is a subclass (or derived class) of the Animal class (the superclass). In this example, we have a base class Animal with a method eat(),

and a subclass Dog that extends the Animal class. The Dog class has an additional method bark().

When you create an instance of the Dog class and call its methods, it can both access the methods inherited from the Animal class and the methods defined in the Dog subclass. This is an example of inheritance in Java, where the Dog class is a subclass (or derived class) of the Animal class (the superclass).

Access	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

1.Default:

When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default.

- The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifier are accessible only within the same package.

- **What is the output of the following Java program fragment:**

```
package p1;

class Geek
{
    void display()
    {
        System.out.println("Hello World!");
    }
}
```

```
package p2;
import p1.*;

class GeekNew
{
    public static void main(String args[])
    {

        Geek obj = new Geek();

        obj.display();
    }
}
```

Output:

Compile time error

2.Private:

The private access modifier is specified using the keyword private.

- The methods or data members declared as private are accessible only within the class in which they are declared.
- Any other class of the same package will not be able to access these members.
- Top-level classes or interfaces can not be declared as private because
 1. private means “only visible within the enclosing class”.
 2. protected means “only visible within the enclosing class and any subclasses”
- **What is the output of the following Java program fragment:**

```
package p1;
```

```
class A
{
    private void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
```

```
class B
{
```

```
public static void main(String args[])
{
    A obj = new A();
    // Trying to access private method
    // of another class
    obj.display();
}
```

Output:

```
error: display() has private access in A
    obj.display();
```

- **Accessing Private Members in Java :**

If we want to access Private Field and method using Reflection we just need to call setAccessible(true) on the field or method object which you want to access.

Class.getDeclaredField(String fieldName) or Class.getDeclaredFields() can be used to get private fields. Whereas

Class.getDeclaredMethod(String methodName, Class<?>... parameterTypes) or Class.getDeclaredMethods() can be used to get private methods.

Below program may not work on online IDEs like, compile and run the below program on offline IDEs only.

1. Accessing private Field :

Example :

```
import java.lang.reflect.Field;

class Student {

    private String name;
    private int age;

    public Student(String name, int age)
    {
        this.name = name;
```

```
this.age = age;
}
public String getName() {
    return name; }

public void setName(String name) {
    this.name = name; }

private int getAge() { return age; }

public void setAge(int age) { this.age = age; }

@Override
public String toString()
{
    return "Employee [name=" + name + ", age=" + age
        + "]";
}

}

class GFG {

    public static void main(String[] args)
    {
        try {

            Student e = new Student("Kapil", 23);

            Field privateField
                = Student.class.getDeclaredField("name");

            privateField.setAccessible(true);
```

```
String name = (String)privateField.get(e);

        System.out.println("Name of Student:" + name);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Output:

Name of Student:Kapil

2. Accessing private Method :

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

class Student {

    private String name;
    private int age;

    public Student(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
```

```
public void setName(String name) { this.name = name;
}

private int getAge() { return age; }

public void setAge(int age) { this.age = age; }

@Override public String toString()
{
    return "Employee [name=" + name + ", age=" + age
           + "]";
}

}

class GFG {

    public static void main(String[] args)
    {
        try {
            Student e = new Student("Kapil", 23);

            Method privateMethod
                = Student.class.getDeclaredMethod("getAge");

            privateMethod.setAccessible(true);

            int age = (int)privateMethod.invoke(e);
            System.out.println("Age of Student: " + age);
        }
    }
}
```

```
        catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

Age of Student: 23

3.protected:

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as **protected** are accessible within the same package or subclasses in different packages.

- In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.**

```
package p1;
```

```
public class A
{
    protected void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
```

```
package p2;
```

```
import p1.*; // importing all classes in package p1

class B extends A
{
public static void main(String args[])
{
B obj = new B();
obj.display();
}

}
```

Output:

GeeksforGeeks

- **Accessing Protected Members in Java :**

1. Accessing in the same class
2. Accessing in other classes of the same package
3. Accessing protected members of a class in its subclass in the same package
4. Accessing another class in a different package
5. Accessing in sub-class in a different package

Case 1: Accessing protected members in the same class

We can access protected members of a class anywhere in it.

Example:

```
class Sample {  
  
    protected int year = 2021;  
    protected void printYear()  
    {  
        System.out.println("Its " + year + " !!");  
    }  
  
    public static void main(String[] args)  
    {  
        Sample sample = new Sample();  
        System.out.println(sample.year);  
        sample.printYear();  
    }  
}
```

Output:

2021

Its 2021 !!

Case 2: Accessing protected members in other classes of the same package

We can access protected members of a class in another class that is present in the same package.

Example:

```
// Class 1
class Sample {
    protected int year = 2021;
    protected void printYear() {
        System.out.println("Its "+year+" !!");
    }
}
```

```
// Class 2
public class Test {
```

```
// Main driver method
public static void main(String[] args) {
    Sample sample = new Sample();
    System.out.println(sample.year);
    sample.printYear();
}
```

Output

2021

Its 2021 !!

Case 3: Accessing protected members of a class in its subclass in the same package

We can access protected members of a class in its subclass if both are present in the same package.

Example :

```
// Class 1
class Sample {
    static protected String title = "geekforgeeks";
    protected int year = 2021;
    protected void printYear() {
        System.out.println("Its "+year+" !!");
    }
}
```

```
// Class 2
public class Test extends Sample {
    public static void main(String[] args) {
        Sample sample = new Sample();
        System.out.println(sample.year);
        sample.printYear();
        System.out.println(Sample.title);
    }
}
```

Case 4: Accessing protected members in another class in a different package

We cannot access the protected members of a class in a class (non-subclass) that is present in a different package.

Example 1: Package 1

```
package package1;

public class Sample {

    static protected String title = "geeksforgeeks";
    protected int year = 2021;

    protected void printYear() {
        System.out.println("Its "+year+" !!");
    }
}
```

Example 2: Package 2

```
package package2;

import package1.Sample;
```

```
// Main class
public class Test {

// Main driver method
public static void main(String[] args {

    Sample sample = new Sample();
    System.out.println(sample.year);
    sample.printYear();
    System.out.println(Sample.title);
}
}
```

Output:

```
error: year has protected access in Sample
        System.out.println(sample.year);
                           ^
```

```
error: printYear() has protected access in Sample
        sample.printYear();
                           ^
```

```
error: title has protected access in Sample
        System.out.println(Sample.title);
```

Here,

It will give a compile-time error. In the following example, we will create two classes. Sample class in package1 and Test class in package2 and try to access protected members of Sample class in Test class. It is justified in the above two examples.

Case 5: Accessing protected members in sub-class in a different package

We can access protected members of a class in its subclass present in a different package. In the following example, we will create two classes. Sample class in package1 and Child class in package2. Child class extends Sample class.

Example 1.1

```
package package1;

// Class
public class Sample {

    // Protected attributes
    static protected String title = "geeksforgeeks";
    protected int year = 2021;
    protected void printYear()
    {
        System.out.println("Its " + year + " !!");
    }
}
```

Example 1.2

```
package package2;
// Importing class from above package
import package1.Sample;

// Main class
public class Child extends Sample {
```

```
void helper()
{
    System.out.println(year);
    printYear();
    System.out.println(Sample.title);
}

public static void main(String[] args)
{
    Child child = new Child();
    child.helper();
}
```

Output
2021
Its 2021 !!
geeksforgeeks

Note: From the above output it can be perceived we have successfully accessed the protected members directly as these are inherited by the Child class and can be accessed without using any reference. The protected members are inherited by the child classes and can access them as its own members. But we can't access these members using the reference of the parent class. We can access protected members only by using child class reference.

Example 2

```
package package2;
import package1.Sample;

public class Child extends Sample {

    void helper()
    {
        Child myself = new Child();

        System.out.println(Sample.title);

        System.out.println(year);
        System.out.println(myself.year);

        printYear();
        myself.printYear();

        Sample sample = new Sample();

        Sample child = new Child();

        System.out.println(sample.year);
        sample.printYear();
        child.printYear();
    }

    public static void main(String[] args)
    {
        Child child = new Child();
```

```
    child.helper();
}
}
```

Output :

```
error: year has protected access in Sample
    System.out.println(sample.year);
          ^
```

```
error: printYear() has protected access in Sample
    sample.printYear();
          ^
```

```
error: printYear() has protected access in Sample
    child.printYear();
          ^
```

Here,

So the main difference between default access modifiers and the protected modifier is that default members are accessible only in the current package. While protected members can be accessed anywhere in the same package and outside package only in its child class and using the child class's reference variable only, not on the reference variable of the parent class. We can't access protected members using the parent class's reference.

4. public:

The public access modifier is specified using the keyword public.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.
- **What is the output of the following Java program fragment:**

```
// public modifier
package p1;
public class A
{
    public void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
```

```
package p2;
import p1.*;
class B {
    public static void main(String args[])
    {
        A obj = new A();
        obj.display();
    }
}
```

Output:
GeeksforGeeks

Important Points:

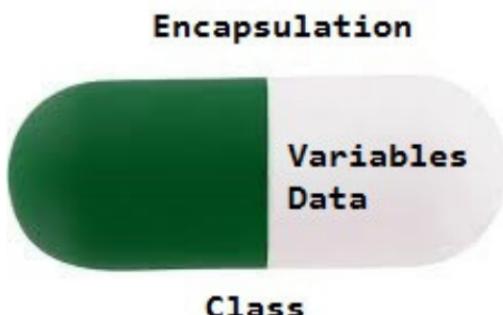
- If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants.

8. Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.

Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a combination of data-hiding and abstraction.
- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.



Advantages of Encapsulation:

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
- **Reusability:** Encapsulation also improves the code for unit testing.

re-usability and is easy to change with new requirements.

Testing code is easy: Encapsulated code is easy to test

Process of encapsulation :

- Declare the variables as private.
- Provide public setter and getter method to modify and get the variables value.
- **What is the output of the following Java program fragment:**

```
class Person{  
    private String name;  
    private int age;  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public void setAge(int age){  
        this.age = age;  
    }  
}
```

```
public int getAge(){
    return age;
}

}

public class Test{
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.setName("Hamim");
        p1.setAge(23);

        System.out.println(p1.getName());
        System.out.println(p1.getAge());

    }
}

or,
class Person{
    private String name;
    private int age;

    public void setter(String s, int a){
        name = s;
        age = a;
    }

    public void getter(){
        System.out.println("Name : "+name);
        System.out.println("Age : "+age);
    }
}
```

```
public class Test{
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.setter("Hamim",23);
        p1.getter();
    }
}
```

or,

```
class Person{
    private String name;
    private int age;

    public void setter(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void getter(){
        System.out.println("Name : "+name);
        System.out.println("Age : "+age);
    }
}
```

```
public class Test{
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.setter("Hamim",23);
        p1.getter();
    }
}
```

Output:

Name : Hamim

Age : 23

9. Inheritance

It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Syntax :

```
class derived-class extends base-class  
{  
    //methods and fields  
}
```

Advantage of inheritance in Java :

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

Important terminologies used in Inheritance:

- **Class:** Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class/Child Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

- In the below example of inheritance, class **Bicycle** is a base class, class **MountainBike** is a derived class that extends the **Bicycle** class and class **Test** is a driver class to run the program.

```
class Bicycle {  
    public int gear;  
    public int speed;  
  
    public Bicycle(int gear, int speed)  
    {  
        this.gear = gear;  
        this.speed = speed;  
    }  
  
    public void applyBrake(int decrement)  
    {  
        speed -= decrement;  
    }  
  
    public void speedUp(int increment)  
    {  
        speed += increment;  
    }  
  
    public String toString()  
    {  
        return ("No of gears are " + gear + "\n"  
            + "speed of bicycle is " + speed);  
    }  
}
```

```
class MountainBike extends Bicycle {  
  
    public int seatHeight;  
  
    public MountainBike(int gear, int speed,  
                        int startHeight)  
    {  
        super(gear, speed);  
        seatHeight = startHeight;  
    }  
  
    public void setHeight(int newValue)  
    {  
        seatHeight = newValue;  
    }  
  
    @Override public String toString()  
    {  
        return (super.toString() + "\nseat height is "  
                + seatHeight);  
    }  
}  
  
public class Test {  
    public static void main(String args[])  
    {  
  
        MountainBike mb = new MountainBike(3, 100, 25);  
        System.out.println(mb.toString());  
    }  
}
```

Output:

No of gears are 3
speed of bicycle is 100
seat height is 25

- In the below example of inheritance, class Employee is a base class, class Engineer is a derived class that extends the Employee class and class Test is a driver class to run the program.**

```
import java.io.*;
```

```
class Employee {  
    int salary = 60000;  
}
```

```
class Engineer extends Employee {  
    int benefits = 10000;  
}
```

```
class Test {  
    public static void main(String args[])  
    {  
        Engineer E1 = new Engineer();  
        System.out.println("Salary : " + E1.salary  
            + "\nBenefits : " + E1.benefits);  
    }  
}
```

Output:

Salary : 60000
Benefits : 10000

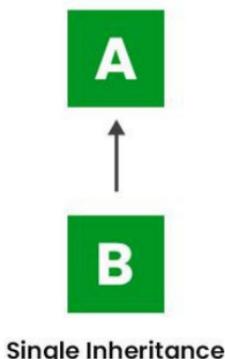
In practice, inheritance and polymorphism are used together in java to achieve fast performance and readability of code.

Types of Inheritance in Java :

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance :

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



- **What is the output of the following Java program fragment:**

```
import java.io.*;
import java.lang.*;
import java.util.*;

class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one {
    public void print_for()
    {
        System.out.println("for");
    }
}

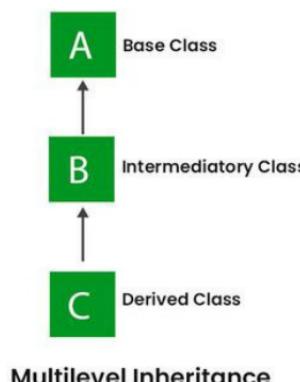
public class main {
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

Output:

Geeks
for
Geeks

2. Multilevel Inheritance :

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



- **What is the output of the following Java program fragment:**

```
import java.io.*;
import java.lang.*;
import java.util.*;

class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}
```

```
class two extends one {  
    public void print_for() { System.out.println("for"); }  
}
```

```
class three extends two {  
    public void print_geek()  
    {  
        System.out.println("Geeks");  
    }  
}
```

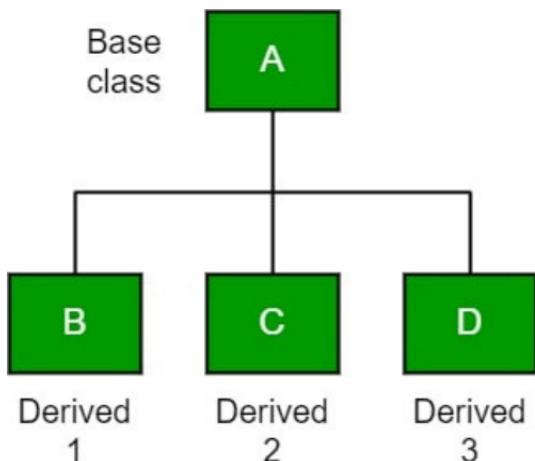
```
public class Test {  
    public static void main(String[] args)  
    {  
        three g = new three();  
        g.print_geek();  
        g.print_for();  
        g.print_geek();  
    }  
}
```

Output:

Geeks
for
Geeks

3. Hierarchical Inheritance :

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



- What is the output of the following Java program fragment:**

```
class A {  
    public void print_A()  
    {  
        System.out.println("Class A");  
    }  
}
```

```
class B extends A {  
    public void print_B()  
    {  
        System.out.println("Class B");  
    }  
}
```

```
class C extends A {  
    public void print_C()  
    {  
        System.out.println("Class C");  
    }  
}  
  
class D extends A {  
    public void print_D()  
    {  
        System.out.println("Class D");  
    }  
}  
  
public class Test {  
    public static void main(String[] args)  
    {  
        B obj_B = new B();  
        obj_B.print_A();  
        obj_B.print_B();  
  
        C obj_C = new C();  
        obj_C.print_A();  
        obj_C.print_C();  
  
        D obj_D = new D();  
        obj_D.print_A();  
        obj_D.print_D();  
    }  
}
```

Output:

Class A

Class B

Class A

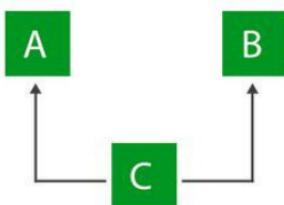
Class C

Class A

Class D

4. Multiple Inheritance (Through Interfaces) :

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

- **What is the output of the following Java program fragment:**

```
import java.io.*;
import java.lang.*;
import java.util.*;

interface one {
    public void print_geek();
}

interface two {
    public void print_for();
}

interface three extends one, two {
    public void print_geek();
}

class child implements three {
    @Override public void print_geek()
    {
        System.out.println("Geeks");
    }

    public void print_for()
    {
        System.out.println("for");
    }
}

public class Test {
    public static void main(String[] args){
```

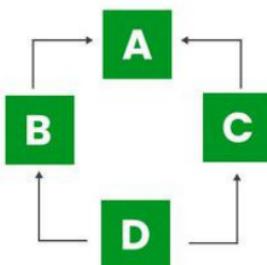
```
child c = new child();
c.print_geek();
c.print_for();
c.print_geek();
}
}
```

Output:

Geeks
for
Geeks

5. Hybrid Inheritance(Through Interfaces) :

It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

10. Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Polymorphism

= poly + morphs (2 Greek words)
= many + forms
= many forms

Types of polymorphism :

In Java polymorphism is mainly divided into two types:

1. Compile-time/static Polymorphism
(Method Overloading, Constructor Overloading)
2. Runtime/dynamic Polymorphism
(Method Overriding)

1. Overloading:

There are two types of overloading :

1. Method Overloading
2. Constructor Overloading

1. Method Overloading :

Method overloading in Java is the concept of defining multiple methods in a class with the same name but with different parameters. When a method is called, the compiler decides which version of the method to execute based on the arguments passed to it.

Different Ways of Method Overloading in Java

1. Changing the Number of Parameters.
2. Changing Data Types of the Arguments.
3. Changing the Order of the Parameters of Methods
4. Changing return Types .

1. Changing the Number of Parameters:

Method overloading can be achieved by changing the number of parameters while passing to different methods.

- What is the output of the following Java program fragment:**

```
class Sum {  
  
    public int sum(int x, int y)  
    {  
        return (x + y);  
    }  
  
    public int sum(int x, int y, int z)  
    {  
        return (x + y + z);  
    }  
  
    public double sum(double x, double y)  
    {  
        return (x + y);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Sum sum1 = new Sum();  
        System.out.println(sum1.sum(10,20));  
        System.out.println(sum1.sum(10,20,30));  
        System.out.println(sum1.sum(10.5,20.5));  
    }  
}
```

Output:

30
60
31.0

- **What is the output of the following Java program fragment:**

```
import java.io.*;
```

```
class Product {  
  
    public int multiply(int a, int b)  
    {  
        int prod = a * b;  
        return prod;  
    }  
  
    public int multiply(int a, int b, int c)  
    {  
        int prod = a * b * c;  
        return prod;  
    }  
}
```

```
}

}

class GFG {

    public static void main(String[] args)
    {
        Product ob = new Product();

        int prod1 = ob.multiply(1, 2);

        System.out.println(
            "Product of the two integer value :" + prod1);

        int prod2 = ob.multiply(1, 2, 3);

        System.out.println(
            "Product of the three integer value :" + prod2);
    }
}
```

Output:

Product of the two integer value :2
Product of the three integer value :6

- **Varargs (variable-length argument lists) :**

Varargs, short for "variable-length argument lists," is a feature in Java that allows a method to accept a variable number of arguments. This feature simplifies the process of writing methods that operate on a variable number of inputs.

- **Method Overloading with Varargs:**

```
class OverloadingExample {  
    // Overloaded method with varargs  
    void printValues(int... values) {  
        System.out.println("Printing int values:");  
        for (int value : values) {  
            System.out.println(value);  
        }  
    }  
  
    // Overloaded method with varargs of a different  
    type  
    void printValues(String... values) {  
        System.out.println("Printing String values:");  
        for (String value : values) {  
            System.out.println(value);  
        }  
    }  
  
    public class Main {  
        public static void main(String[] args) {
```

```
OverloadingExample example = new
OverloadingExample();

// Call the overloaded methods with different
argument types
example.printValues(1, 2, 3);
example.printValues("Hello", "World");
}

}
```

Output:

Printing int values:

1
2
3

Printing String values:

Hello
World

2. Changing Data Types of the Arguments :

In many cases, methods can be considered Overloaded if they have the same name but have different parameter types, methods are considered to be overloaded.

- What is the output of the following Java program fragment:**

```
import java.io.*;  
  
class Product {  
  
    public int Prod(int a, int b, int c)  
    {  
  
        int prod1 = a * b * c;  
        return prod1;  
    }  
  
    public double Prod(double a, double b, double c)  
    {  
  
        double prod2 = a * b * c;  
        return prod2;  
    }  
}  
  
class GFG {  
    public static void main(String[] args)  
    {
```

```
Product obj = new Product();

int prod1 = obj.Prod(1, 2, 3);
System.out.println("Product of the three integer
value :" + prod1);
double prod2 = obj.Prod(1.0, 2.0, 3.0);
System.out.println("Product of the three double
value :" + prod2);
}
}
```

Output:

Product of the three integer value :6
Product of the three double value :6.0

3. Changing the Order of the Parameters of Methods :

Method overloading can also be implemented by rearranging the parameters of two or more overloaded methods. For example, if the parameters of method 1 are (String name, int roll_no) and the other method is (int roll_no, String name) but both have the same name, then these 2 methods are considered to be overloaded with different sequences of parameters.

- What is the output of the following Java program fragment:**

```
import java.io.*;  
  
class Student {  
  
    public void StudentId(String name, int roll_no)  
    {  
        System.out.println("Name :" + name + " " +  
"Roll-No :" + roll_no);  
    }  
    public void StudentId(int roll_no, String name)  
    {  
        System.out.println("Roll-No :" + roll_no + " " +  
"Name :" + name);  
    }  
}
```

```
class GFG {  
  
    public static void main(String[] args)  
    {  
        Student obj = new Student();  
  
        obj.StudentId("Jim", 1);  
        obj.StudentId(2, "Kamlesh");  
    }  
}
```

Output:

Name :Jim Roll-No :1
Roll-No :2 Name :Kamlesh

2.Constructor Overloading :

Constructor overloading in Java is the concept of defining multiple constructors in a class with different parameter lists. Like method overloading, constructor overloading allows you to create multiple constructors with the same name, but with different arguments. When you create an object of the class, the appropriate constructor is called based on the arguments you provide.

- What is the output of the following Java program fragment:**

```
class Teacher {
```

```
    String name, gender;  
    int phone;
```

```
    Teacher() // Default Constructor
```

```
    {  
        System.out.println("No information");  
    }
```

```
    Teacher(String n, String g) // constructor
```

```
    {  
        name = n;  
        gender = g;  
    }
```

```
Teacher(String n, String g, int p) // constructor
{
    name = n;
    gender = g;
    phone = p;
}

void displayInformation() {
    System.out.println("Name : " + name);
    System.out.println("Gender : " + gender);
    System.out.println("Phone : " + phone);
    System.out.println();
}
}

public class Test {
    public static void main(String[] args) {
        Teacher teacher1 = new Teacher();

        Teacher teacher2; // object declare
        teacher2 = new Teacher("Hamim Talukder",
        "Male"); // object create
        teacher2.displayInformation();

        Teacher teacher3 = new Teacher("Ava
        Talukder", "Female", 1731767275); // object
        declare and create
        teacher3.displayInformation();

    }
}
```

Output:

No information

Name : Hamim Talukder

Gender : Male

Phone : 0

Name : Ava Talukder

Gender : Female

Phone : 1731767275

- **Constructor Overloading with Varargs:**

```
public class VarargsConstructorExample {  
  
    private String[] values;  
  
    // Constructor with varargs  
    public VarargsConstructorExample(String... args) {  
        values = args;  
    }  
  
    // Method to display values  
    public void displayValues() {  
        System.out.println("Values:");  
        for (String value : values) {  
            System.out.println(value);  
        }  
    }  
  
    public static void main(String[] args) {  
        // Create objects using different constructors  
        VarargsConstructorExample obj1 = new  
        VarargsConstructorExample("Java", "is", "powerful");  
        VarargsConstructorExample obj2 = new  
        VarargsConstructorExample("Varargs", "make", "life",  
        "easier");  
  
        // Display values  
        obj1.displayValues();  
        System.out.println(); // Adding a newline for  
        clarity
```

```
    obj2.displayValues();
}
}
```

Output:

Values:
Java
is
powerful

Values:
Varargs
make
life
easier

- Create a class called Box that include three pieces of information as instance variables- height, width and depth(type double) of two boxes. Your class should have a constructor and initializes the tree instance variables. Provide a method displayVol that display the volume of two boxes, Suppose, the values of instance variables for the first box's are (10,10,10) and second box's are (20,30,10). Write a test application named BoxVolume that demonstrate class Box's capablities.

```
class Box{  
    double height, width, depth;  
  
    Box(double h,double w, double d){  
        height = h;  
        width = w;  
        depth = d;  
    }  
  
    void displayVol(){  
        double vol = height * width * depth;  
        System.out.println("Volume is : "+vol);  
    }  
}  
  
public class Test{  
    public static void main(String[] args){  
        Box box1 = new Box(10,10,10);  
        Box box2 = new Box(20,30,10);  
    }  
}
```

```
    box1. displayVol();
    box2. displayVol();

}
```

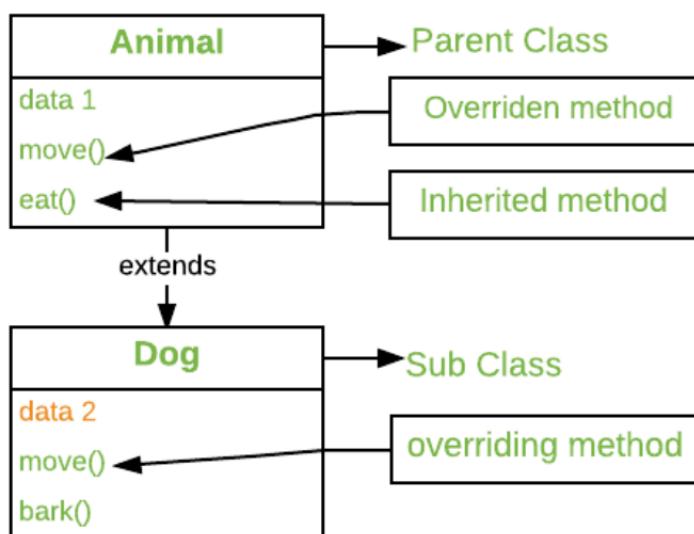
Output:

Volume is : 1000.0

Volume is : 6000.0

2. Overriding:

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.



Method Overriding rules:

- Name, signature type, parameter must be same.
- If a method can't be inherited, then it can't be overridden.
- A method declared as final or static can't be overridden.
- Constructor can't be overridden.

Method overriding is one of the way by which java achieve Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

- **What is the output of the following Java program fragment:**

```
class Parent {  
    void show()  
    {  
        System.out.println("Parent's show()");  
    }  
}
```

```
class Child extends Parent {  
    @Override  
    void show()  
    {  
        System.out.println("Child's show()");  
    }  
}
```

```
class Test {  
    public static void main(String[] args)  
    {  
        Parent obj1 = new Parent();  
        obj1.show();  
  
        Parent obj2 = new Child();  
        obj2.show();  
    }  
}
```

Output:

Parent's show()
Child's show()

- **Rules for method overriding:**

1. Overriding and Access-Modifiers :

The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

- **What is the output of the following Java program fragment:**

```
class Parent {  
    private void m1()  
    {  
        System.out.println("From parent m1()");  
    }  
  
    protected void m2()  
    {  
        System.out.println("From parent m2()");  
    }  
}  
  
class Child extends Parent {  
    private void m1()  
    {  
        System.out.println("From child m1()");  
    }  
  
    @Override  
    public void m2()  
    {  
        System.out.println("From child m2()");  
    }  
}  
  
class Test {  
    public static void main(String[] args)  
    {
```

```
Parent obj1 = new Parent();
obj1.m2();
Parent obj2 = new Child();
obj2.m2();
}
}
```

Output:

From parent m2()
From child m2()

2.Final methods can not be overridden :

If we don't want a method to be overridden, we declare it as final. Please see Using final with Inheritance .

- What is the output of the following Java program fragment:**

```
class Parent {
    final void show() {}
}

class Child extends Parent {
    void show() {}
}
```

```
class Test {
    public static void main(String[] args)
{
```

```
Parent obj1 = new Parent();
obj1.show();
Parent obj2 = new Child();
obj2.show();
}
}
```

Output:

error: show() in Child cannot override show() in Parent

```
void show() {}
^
```

overridden method is final

1 error

3.Static methods can not be overridden(Method Overriding vs Method Hiding) :

When you define a static method with same signature as a static method in base class, it is known as method hiding.

The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

- **What is the output of the following Java program fragment:**

```
class Parent {  
    static void m1()  
    {  
        System.out.println("From parent "  
            + "static m1()");  
    }  
  
    void m2()  
    {  
        System.out.println("From parent "  
            + "non-static(instance) m2()");  
    }  
}  
  
class Child extends Parent {  
    static void m1()  
    {  
        System.out.println("From child static m1()");  
    }  
  
    @Override  
    public void m2()  
    {  
        System.out.println("From child "  
            + "non-static(instance) m2()");  
    }  
}
```

```
class Test {  
    public static void main(String[] args)  
    {  
        Parent obj1 = new Child();  
  
        obj1.m1();  
  
        obj1.m2();  
    }  
}
```

Output:

From parent static m1()

From child non-static(instance) m2()

4. Private methods can not be overridden :

Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.(See this for details).

5. The overriding method must have same return type (or subtype) :

From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be subtype of parent's return type. This phenomena is known as covariant return type.

6.Invoking overridden method from sub-class:

:We can call parent class method in overriding method using super keyword.

- What is the output of the following Java program fragment:**

```
class Parent {  
    void show()  
    {  
        System.out.println("Parent's show()");  
    }  
}
```

```
class Child extends Parent {  
    @Override  
    void show()  
    {  
        super.show();  
        System.out.println("Child's show()");  
    }  
}
```

```
class Test {  
    public static void main(String[] args)  
    {  
        Parent obj = new Child();  
        obj.show();  
    }  
}
```

Output:

Parent's show()

Child's show()

7.Overriding and constructor :

We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).

8.Overriding and Exception-Handling :

Below are two rules to note when overriding methods related to exception-handling.

- **Rule#1 :** If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.
- **What is the output of the following Java program fragment:**

```
class Parent {  
    void m1()  
    {  
        System.out.println("From parent m1()");  
    }  
  
    void m2()  
    {  
        System.out.println("From parent m2()");  
    }  
}  
  
class Child extends Parent {
```

```
@Override  
void m1() throws ArithmeticException  
{  
    System.out.println("From child m1()");  
}  
  
@Override  
void m2() throws Exception  
{  
    System.out.println("From child m2");  
}  
}
```

Output:

```
error: m2() in Child cannot override m2() in Parent  
    void m2() throws Exception{  
        System.out.println("From child m2");}  
        ^  
overridden method does not throw Exception
```

- **Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

- **What is the output of the following Java program fragment:**

```
class Parent {  
    void m1() throws RuntimeException  
    {  
        System.out.println("From parent m1()");  
    }  
}  
  
class Child1 extends Parent {  
    @Override  
    void m1() throws RuntimeException  
    {  
        System.out.println("From child1 m1()");  
    }  
}  
  
class Child2 extends Parent {  
    @Override  
    void m1() throws ArithmeticException  
    {  
        System.out.println("From child2 m1()");  
    }  
}  
  
class Child3 extends Parent {  
    @Override  
    void m1()  
    {  
        System.out.println("From child3 m1()");  
    }  
}
```

```
class Child4 extends Parent {  
    @Override  
    void m1() throws Exception  
    {  
        System.out.println("From child4 m1()");  
    }  
}
```

Output:

error: m1() in Child4 cannot override m1() in Parent
void m1() throws Exception
 ^
overridden method does not throw Exception

9. Overriding and abstract method: Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

10. Overriding and synchronized/strictfp method :

The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa

Note :

1. In C++, we need virtual keyword to achieve overriding or Run Time Polymorphism. In Java, methods are virtual by default.
2. We can have multilevel method-overriding.

- **What is the output of the following Java program fragment:**

```
class Parent {  
    void show()  
    {  
        System.out.println("Parent's show()");  
    }  
}  
  
class Child extends Parent {  
    void show() { System.out.println("Child's show()"); }  
}  
  
class GrandChild extends Child {  
    void show()  
    {  
        System.out.println("GrandChild's show()");  
    }  
}  
  
class Test {  
    public static void main(String[] args)  
    {
```

```
Parent obj1 = new GrandChild();
obj1.show();
}
}
```

Output:

GrandChild's show()

- **Method Overriding with Varargs:**

```
class Base {  
    void printValues(String... values) {  
        System.out.println("Base class printing values:");  
        for (String value : values) {  
            System.out.println(value);  
        }  
    }  
}  
  
class Derived extends Base {  
  
    @Override  
    void printValues(String... values) {  
        System.out.println("Derived class printing  
values:");  
        for (String value : values) {  
            System.out.println(value);  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Derived derivedObj = new Derived();  
  
        derivedObj.printValues("Java", "is", "fun");  
    }  
}
```

Output:

Derived class printing values:

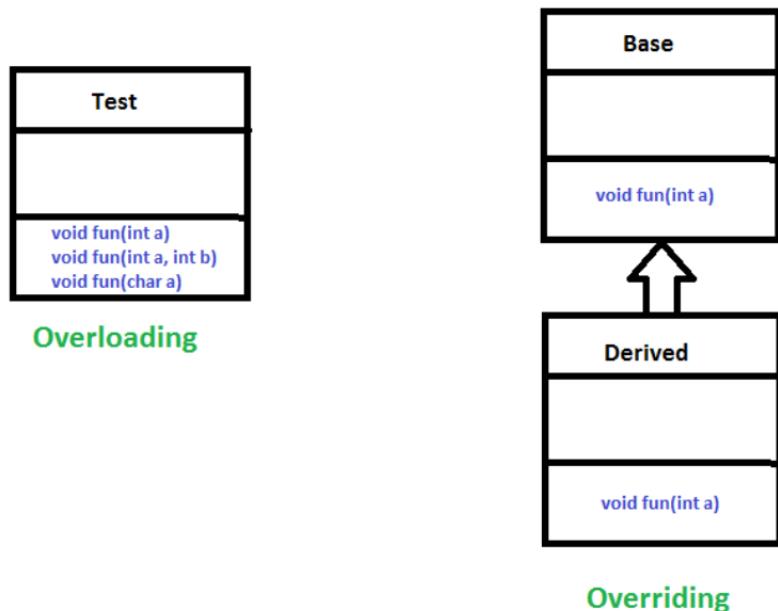
Java

is

fun

- **Overriding vs Overloading :**

1. Overloading is about same method have different signatures. Overriding is about same method, same signature but different classes connected through inheritance.



2. Overloading is an example of compiler-time polymorphism and overriding is an example of run time polymorphism.

Method Overloading	Method Overriding
1.Parameter must be different.	1.Parameter can't be different.
2.It occurs within the same class.	2.It occurs between two classes-sub class and a super class.
3.Inheritance is not involved.	3.Inharitance is involved.
4.Return type may or may not be same.	4.Return type must be same.
5.One method does not hide another.	5.child method hides parent another.

- **What is the output of the following Java program fragment:**

```
class Employee {  
    public static int base = 10000;  
    int salary()  
    {  
        return base;  
    }  
}
```

```
class Manager extends Employee {  
    int salary()  
    {  
        return base + 20000;  
    }  
}
```

```
class Clerk extends Employee {  
    int salary()  
    {  
        return base + 10000;  
    }  
}
```

```
class Test {  
    static void printSalary(Employee e)  
    {  
        System.out.println(e.salary());  
    }  
}
```

```
public static void main(String[] args)
{
    Employee obj1 = new Manager();

    System.out.print("Manager's salary : ");
    printSalary(obj1);

    Employee obj2 = new Clerk();
    System.out.print("Clerk's salary : ");
    printSalary(obj2);
}
```

Output:

Manager's salary : 30000
Clerk's salary : 20000

- **What is the output of the following Java program fragment:**

```
class Person{  
    String name;  
    int age;  
  
    void displayInformation(){  
        System.out.println("Name : "+name);  
        System.out.println("Age : "+age);  
    }  
}  
  
class Teacher extends Person{  
    String qualification;  
  
    void displayInformation(){  
        System.out.println("Name : "+name);  
        System.out.println("Age : "+age);  
        System.out.println("Qualification :  
"+qualification);  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person();  
        p1.displayInformation();  
  
        Person t = new Teacher();  
        t.displayInformation();  
    }  
}
```

```
or,  
class Person{  
    String name;  
    int age;  
  
    void displayInformation(){  
        System.out.println("Name : "+name);  
        System.out.println("Age : "+age);  
    }  
}  
  
class Teacher extends Person{  
    String qualification;  
  
    @Override  
    void displayInformation(){  
        System.out.println("Name : "+name);  
        System.out.println("Age : "+age);  
        System.out.println("Qualification :  
"+qualification);  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person();  
        p1.displayInformation();  
  
        p1 = new Teacher();  
        p1.displayInformation();  
    }  
}
```

Output:

```
Name : null  
Age : 0  
Name : null  
Age : 0  
Qualification : null
```

- **What is the output of the following Java program fragment:**

```
class Person{  
    String name;  
    int age;  
  
    void display(){  
        System.out.println("I am a person");  
    }  
}  
  
class Teacher extends Person{  
    String qualification;  
  
    @Override  
    void display(){  
        System.out.println("I am a Teacher");  
    }  
}  
  
class Student extends Person{  
    String qualification;  
  
    @Override
```

```
void display(){
    System.out.println("I am a Student");
}
}

public class Test{
    public static void main(String[] args) {
        Person p = new Person();
        p.display();
        p = new Teacher();
        p.display();
        p = new Student();
        p.display();
    }
}
```

Output:

I am a person
I am a Teacher
I am a Student

- **What is the output of the following Java program fragment:**

```
class Shape{  
    double area(){  
        System.out.println("Shape area : ");  
        return 0;  
    }  
}  
  
class Rectangle extends Shape{  
    double length, width;  
    Rectangle(double length, double width){  
        this.length = length;  
        this.width = width;  
    }  
    @Override  
    double area(){  
        System.out.println("Area for Rectangle : ");  
        return length * width;  
    }  
}  
  
class Triangle extends Shape{  
    double base, height;  
    Triangle(double base, double height){  
        this.base = base;  
        this.height = height;  
    }  
}
```

```
@Override
double area(){
    System.out.println("Area for Triangle : ");
    return 0.5 * base * height;
}
}

public class Test{
    public static void main(String[] args) {
        Shape s =new Shape();
        Rectangle r =new Rectangle(10,20);
        Triangle t = new Triangle(10,20);
        System.out.println(s.area());
        System.out.println(r.area());
        System.out.println(t.area());
    }
}
```

Output:

Shape area :
0.0
Area for Rectangle :
200.0
Area for Triangle :
100.0

- **What is the output of the following Java program fragment:**

```
class Shape{  
    double area(){  
        System.out.println("Shape area : ");  
        return 0;  
    }  
}  
  
class Rectangle extends Shape{  
    double length, width;  
    Rectangle(double length, double width){  
        this.length = length;  
        this.width = width;  
    }  
  
    @Override  
    double area(){  
        System.out.println("Area for Rectangle : ");  
        return length * width;  
    }  
}  
  
class Triangle extends Shape{  
    double base, height;  
  
    Triangle(double base, double height){  
        this.base = base;  
        this.height = height;  
    }  
}
```

```
@Override
double area(){
    System.out.println("Area for Triangle : ");
    return 0.5 * base * height;
}
}

public class Test{
    public static void main(String[] args) {
        Shape s1 =new Shape();
        Shape s2 =new Rectangle(10,20);
        Shape s3 = new Triangle(10,20);

        System.out.println(s1.area());
        System.out.println(s2.area());
        System.out.println(s3.area());
    }
}
```

or,

```
class Shape{
    double area(){
        System.out.println("Shape area : ");
        return 0;
    }
}
```

```
class Rectangle extends Shape{
    double length, width;
    Rectangle(double length, double width){
        this.length = length;
```

```
        this.width = width;
    }

@Override
double area(){
    System.out.println("Area for Rectangle : ");
    return length * width;
}
}

class Triangle extends Shape{
    double base, height;

    Triangle(double base, double height){
        this.base = base;
        this.height = height;
    }

    @Override
    double area(){
        System.out.println("Area for Triangle : ");
        return 0.5 * base * height;
    }
}

public class Test{
    public static void main(String[] args) {
        Shape[] s = new Shape[3];
        s[0] =new Shape();
        s[1] =new Rectangle(10,20);
        s[2] = new Triangle(10,20);
    }
}
```

```
for(int i = 0; i<3; i++){  
    System.out.println(s[i].area());  
}  
}  
}
```

Output:

Shape area :

0.0

Area for Rectangle :

200.0

Area for Triangle :

100.0

11. super keyword

"super" keyword is used to refer immediate super class object.

1. It is used to call super class instance variable.
2. It is used to call super class method.
(overridden method)
3. It is used to call super class constructor.

- **What is the output of the following Java program fragment:**

```
class A{  
    int x = 10;  
}
```

```
class B extends A{  
    int x = 5;  
  
    void display(){  
        System.out.println(x);  
    }  
}
```

```
public class Test{  
    public static void main(String[] args) {  
        B ob = new B();  
        ob.display();  
    }  
}
```

Output:

5

- **What is the output of the following Java program fragment:**

```
class A{  
    int x = 10;  
}  
  
class B extends A{  
    int x = 5;  
  
    void display(){  
        System.out.println(super.x);  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        B ob = new B();  
        ob.display();  
    }  
}
```

Output:

10

- **What is the output of the following Java program fragment:**

```
class A{  
    void display(){  
        System.out.println("Inside A class");  
    }  
}  
  
class B extends A{  
  
    @Override  
    void display(){  
        System.out.println("Inside B class");  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        B ob = new B();  
        ob.display();  
    }  
}
```

Output:

Inside B class

- **What is the output of the following Java program fragment:**

```
class A{  
    void display(){  
        System.out.println("Inside A class");  
    }  
}  
  
class B extends A{  
  
    @Override  
    void display(){  
        super.display();  
        System.out.println("Inside B class");  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        B ob = new B();  
        ob.display();  
    }  
}
```

Output:

Inside A class
Inside B class

- **What is the output of the following Java program fragment:**

```
class A{  
    A(){  
        System.out.println("A's constructor");  
    }  
}  
  
class B extends A{  
  
    B(){  
        System.out.println("B's constructor");  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        B ob = new B();  
    }  
}
```

Output:

A's constructor
B's constructor

- **What is the output of the following Java program fragment:**

```
class A{  
    A(){  
        System.out.println("A's constructor");  
    }  
}  
  
class B extends A{  
  
    B(){  
        super();  
        System.out.println("B's constructor");  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        B ob = new B();  
    }  
}
```

Output:

A's constructor
B's constructor

- **What is the output of the following Java program fragment:**

```
class Vehical{  
    String color;  
    double weight;  
  
    Vehical(String c, double w){  
        color = c;  
        weight = w;  
    }  
  
    void attribute(){  
        System.out.println("Color : "+color);  
        System.out.println("Weight : "+weight);  
    }  
}  
  
class Car extends Vehical{  
    //color, weight, attribute()  
    int gear;  
  
    Car(String c, double w, int g){  
        super(c,w);  
        gear = g;  
    }  
  
    @Override  
    void attribute(){  
        super.attribute();  
        System.out.println("Gear : "+gear);  
    }  
}
```

```
public class Test{  
    public static void main(String[] args) {  
        Car ob = new Car("Lamborgini", 12000.47,6);  
        ob.attribute();  
    }  
}
```

Output:

Color : Lamborgini
Weight : 12000.47
Gear : 6

12. this keyword

"this" keyword used to refer current class object.

Usages of "this" keyword :

1. Refer current class instance variable.
2. It can be used to call current class constructor.
3. It can be used to call current class method.
4. It can be passed as an argument in the method.
(event handling)

- **What is the output of the following Java program fragment:**

```
class Person{  
    String name;  
    int age;  
  
    Person(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
  
    void display (){  
        System.out.println("Name : "+name);  
        System.out.println("Age : "+age);  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person("Anis",27);  
    }  
}
```

```
    p1.display();
}
}
```

Output:

Name : Anis

Age : 27

- What is the output of the following Java program fragment:**

```
class Person{
    String name;
    int age;
    String hairColor;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    Person(String name, int age, String hairColor){
        this(name,age);
        this.hairColor = hairColor;
    }

    void display (){
        System.out.println("Name : "+name);
        System.out.println("Age : "+age);
        System.out.println("HairColor : "+hairColor);
    }
}
```

```
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person("Anis",47,"Whight");  
        p1.display();  
  
        Person p2 = new Person("Anis",27,"Black");  
        p2.display();  
    }  
}
```

Output:

Name : Anis
Age : 47
HairColor : Whight
Name : Anis
Age : 27
HairColor : Black

- **What is the output of the following Java program fragment:**

```
class Person{  
    void message(){  
        System.out.println("I am message method");  
    }  
  
    void display(){  
        this.message();  
        System.out.println("I am display method");  
    }  
}
```

```
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person();  
        p1.display();  
  
    }  
}
```

or,

```
class Person{  
    void message(){  
        System.out.println("I am message method");  
    }  
  
    void display(){  
        message();  
        System.out.println("I am display method");  
    }  
}
```

```
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person();  
        p1.display();  
  
    }  
}
```

Output:

I am message method
I am display method

13. final keyword

Using final keyword will restrict the user.

- 1) final variable
- 2) final method
- 3) final class

1)final variable :

- 1.final variable
- 2.blank final variable
- 3.static blank final variable

- **What is the output of the following Java program fragment:**

```
class University{  
    final String UNIVERSITY_NAME = "DIU";  
    final int fees; //blank final variable  
  
    University(){  
        fees = 37000; //blank final variable initialized  
    }  
  
    void display(){  
        System.out.println(UNIVERSITY_NAME);  
        System.out.println(fees);  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {
```

```
        University p1 = new University();
        p1.display();

    }

}
```

Output:

DIU
37000

- **What is the output of the following Java program fragment:**

```
class University{
    final String UNIVERSITY_NAME = "DIU";
    static final int fees; //static blank final variable

    static{
        fees = 37000; //static blank final variable initialized
    }

    void display(){
        System.out.println(UNIVERSITY_NAME);
        System.out.println(fees);
    }
}

public class Test{
    public static void main(String[] args) {
        University p1 = new University();
        p1.display();
    }
}
```

```
    }  
}
```

Output:

DIU
37000

2) final method :

- **What is the output of the following Java program fragment:**

```
class University{
```

```
    final void display(){  
        System.out.println("University info");  
    }
```

```
}
```

```
class Student extends University{
```

```
    //display()
```

```
    void display2(){  
        System.out.println("Students info");  
    }
```

```
}
```

```
public class Test{  
    public static void main(String[] args) {  
        Student p1 = new Student();  
        p1.display();  
        p1.display2();  
  
    }  
}
```

Output:

University info
Students info

- **What is the output of the following Java program fragment:**

```
import java.util.Arrays;

class A {
    final int num = 10;
    String name;

    public A(String name){
        this.name = name;
    }
}

public class Solution {
    public static void main(String[] args) {
        final A hamim = new A("Hamim Talukder");

        // when a non primitive is final, you cannot
        // reassign it.
        hamim = new A("Jim Talukder");

        System.out.println(hamim.name);
    }
}
```

Output:

Solution.java:15: error: cannot assign a value to
final variable hamim

```
hamim = new A("Jim Talukder");
^
```

1 error

14. Abstraction

Abstraction is the process of hiding the implementation details and showing only the functionality to the user.

There are two ways to achieve abstraction in Java :

1. Abstract class (0 to 100)
2. Interface (Achieve 100% abstraction)

Points to remember about abstract method :

- Abstract method has no body
- It must be ends with a semicolon
- It must be in the abstract class
- It must be overridden
- It can never be final and static
- Abstract class have abstract and non abstract method
- Non abstract class can't have abstract method
- We can't make object of abstract class but we can make reference variable of it
- If we extend an abstract class we have to override its abstract methods or we have to declare the class as abstract itself.

Algorithm to implement abstraction in java:

1. Determine the classes or interfaces that will be part of the abstraction.
2. Create an abstract class or interface that defines the common behaviors and properties of these classes.
3. Define abstract methods within the abstract class or interface that do not have any implementation details.
4. Implement concrete classes that extend the abstract class or implement the interface.
5. Override the abstract methods in the concrete classes to provide their specific implementations.
6. Use the concrete classes to implement the program logic.

1. Abstract class (0 to 100) :

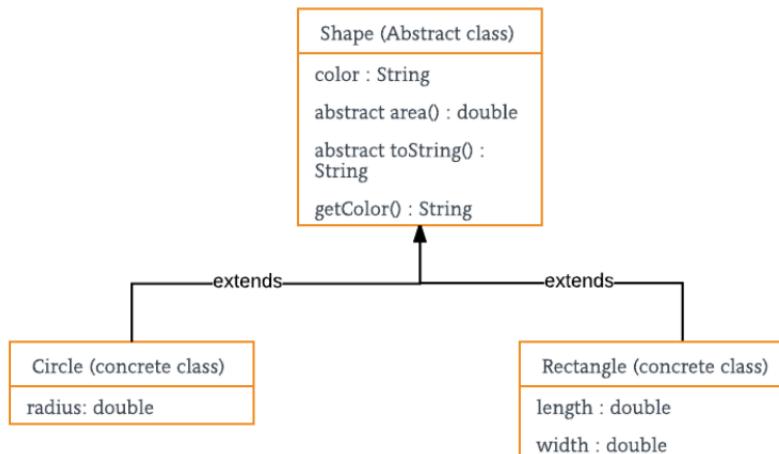
Syntax :

```
abstract class MobileUser{  
    abstract void message();  
  
    void call(){  
        System.out.println("Hello");  
    }  
}
```

- **When to use abstract classes and abstract methods with an example :**

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size, and so on. From this, specific types of shapes are derived(inherited)- circle, square, triangle, and so on — each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



- **What is the output of the following Java program fragment:**

```
abstract class Shape {  
    String color;  
  
    abstract double area();  
    public abstract String toString();  
  
    public Shape(String color)  
    {  
        System.out.println("Shape constructor called");  
        this.color = color;  
    }  
  
    public String getColor() { return color; }  
}  
class Circle extends Shape {  
    double radius;  
  
    public Circle(String color, double radius)  
    {  
  
        super(color);  
        System.out.println("Circle constructor called");  
        this.radius = radius;  
    }  
  
    @Override double area()  
    {  
        return Math.PI * Math.pow(radius, 2);  
    }
```

```
@Override public String toString()
{
    return "Circle color is " + super.getColor()
        + "and area is : " + area();
}
}

class Rectangle extends Shape {

    double length;
    double width;

    public Rectangle(String color, double length,
                    double width)
    {
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override double area() { return length * width; }

    @Override public String toString()
    {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}

public class Test {
    public static void main(String[] args)
    {
```

```
Shape s1 = new Circle("Red", 2.2);
Shape s2 = new Rectangle("Yellow", 2, 4);

System.out.println(s1.toString());
System.out.println(s2.toString());
}

}
```

Output:

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

- What is the output of the following Java program fragment:



```
abstract class MobileUser {
    abstract void sendMessage();
}

class Rahim extends MobileUser{
    @Override
    void sendMessage(){
        System.out.println("I am Rahim");
    }
}

class Karim extends MobileUser{
    @Override
    void sendMessage(){
        System.out.println("I am Karim");
    }
}

public class Test {
    public static void main(String[] args)
    {
```

```
MobileUser mu; //reference variable  
  
    mu = new Rahim();  
    mu.sendMessage();  
  
    mu = new Karim();  
    mu.sendMessage();  
}  
}
```

Output:

I am Rahim
I am Karim

- **What is the output of the following Java program fragment:**

```
abstract class MobileUser {  
    abstract void sendMessage();
```

```
    void call(){  
        System.out.println("I am non-abstract method");  
    }  
}
```

```
class Rahim extends MobileUser{  
    // call(), sendMessage()  
    @Override  
    void sendMessage(){  
        System.out.println("I am Rahim");  
    }  
}
```

```
class Karim extends MobileUser{
    // call(), sendMessage()
    @Override
    void sendMessage(){
        System.out.println("I am Karim");
    }
}
public class Test {
    public static void main(String[] args)
    {
        MobileUser mu; //reference variable

        mu = new Rahim();
        mu.call();
        mu.sendMessage();

        mu = new Karim();
        mu.sendMessage();
    }
}
```

Output:

I am non-abstract method
I am Rahim
I am Karim

2. Interface (Achieve 100% abstraction):

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java **Interface** also represents the **IS-A relationship**.

When we decide a type of entity by its behaviour and not via attribute we should define it as an interface.

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the behaviour.
- Interface do not have constructor.
- Represent behaviour as interface unless every sub-type of the class is guaranteed to have that behaviour.

- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax:

```
interface {  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

Why do we use an Interface?

It is used to achieve total abstraction.

Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.

Any class can extend only 1 class but can any class implement infinite number of interface.

It is also used to achieve loose coupling.

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

```
// A simple interface
```

```
interface Player
```

```
{
```

```
    final int id = 10;
```

```
    int move();
```

```
}
```

Difference Between Class and Interface :

The major differences between a class and an interface are:

S. No.	Class	Interface
1.	In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
2.	Class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
3.	The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Implementation:

To implement an interface we use the keyword **implements**.

- **What is the output of the following Java program fragment:**

```
package FirstPackage;
```

```
public interface A {  
    void play();  
}
```

```
package FirstPackage;
```

```
public interface B {  
    void play();  
}
```

```
package FirstPackage;
```

```
public class C implements A, B {
```

```
    public void play(){  
        System.out.println("Hello I am From C.");  
    }
```

```
}
```

```
package FirstPackage;
```

```
public class Test {
```

```
public static void main(String[] args){  
    C ob = new C();  
    ob.play();  
}  
}
```

Output:

Hello I am From C.

- **What is the output of the following Java program fragment:**

```
import java.io.*;  
  
interface In1 {  
  
    final int a = 10;  
  
    void display();  
}  
  
class Test implements In1 {  
  
    public void display(){  
        System.out.println("Geek");  
    }  
  
    public static void main(String[] args)  
    {  
        Test t = new Test();  
        t.display();  
        System.out.println(a);  
    }  
}
```

Output:

Geek

10

- **Real-World Example:** Let's consider the example of vehicles like bicycle, car, bike....., they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, caretc implement all these functionalities in their own class in their own way.

```
import java.io.*;
```

```
interface Vehicle {
```

```
    void changeGear(int a);
```

```
    void speedUp(int a);
```

```
    void applyBrakes(int a);
```

```
}
```

```
class Bicycle implements Vehicle{
```

```
    int speed;
```

```
    int gear;
```

```
    @Override
```

```
    public void changeGear(int newGear){
```

```
        gear = newGear;
```

```
}
```

```
    @Override
```

```
    public void speedUp(int increment){
```

```
    speed = speed + increment;
}

@Override
public void applyBrakes(int decrement){

    speed = speed - decrement;
}

public void printStates() {
    System.out.println("speed: " + speed
        + " gear: " + gear);
}
}

class Bike implements Vehicle {

    int speed;
    int gear;

    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }
}
```

```
@Override
public void applyBrakes(int decrement){

    speed = speed - decrement;
}

public void printStates() {
    System.out.println("speed: " + speed
        + " gear: " + gear);
}

class Test {

    public static void main (String[] args) {

        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

Output:

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

Advantages of Interfaces in Java :

The advantages of using interfaces in Java are as follows:

1. Without bothering about the implementation part, we can achieve the security of the implementation.
2. In Java, multiple inheritances is not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

- **What is the output of the following Java program fragment:**

```
class Persion{
    String name;
    int age;

    void displayInformation1(){
        System.out.println("Name : "+name);
        System.out.println("Age : "+age);
    }
}

class Teacher extends Persion{
    String qualification;

    void displayInformation2(){
        System.out.println("Name : "+name);
        System.out.println("Age : "+age);
        System.out.println("Qualification : "+qualification);
    }
}

public class Test {
    public static void main(String[] args) {
        Teacher t1 = new Teacher();

        t1.name = "Anisul Islam";
        t1.age = 25;
        t1.qualification = "Online teacher";
        t1.displayInformation2();
    }
}
```

```
Teacher t2 = new Teacher();

t2.name = "Hamim Talukder";
t2.age = 25;
t2.qualification = "BSC in CSE";
t2.displayInformation2();

}

}
```

or,

```
class Persion{
    String name;
    int age;
```

```
void displayInformation1(){
    System.out.println("Name : "+name);
    System.out.println("Age : "+age);
}
```

```
}
```

```
class Teacher extends Persion{
    String qualification;

    void displayInformation2(){
        displayInformation1();
        System.out.println("Qualification : "+qualification);
    }
}
```

```
public class Test {
```

```
public static void main(String[] args) {  
    Teacher t1 = new Teacher();  
  
    t1.name = "Anisul Islam";  
    t1.age = 25;  
    t1.qualification = "Online teacher";  
    t1.displayInformation2();  
  
    Teacher t2 = new Teacher();  
  
    t2.name = "Hamim Talukder";  
    t2.age = 25;  
    t2.qualification = "BSC in CSE";  
    t2.displayInformation2();  
}  
}
```

Output:

Name : Anisul Islam
Age : 25
Qualification : Online teacher
Name : Hamim Talukder
Age : 25
Qualification : BSC in CSE

- **What is the output of the following Java program fragment:**

```
class Persion{  
    private String name;  
    private int age;  
  
    public String getName(){  
        return name;  
    }  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public int getAge(){  
        return age;  
    }  
  
    public void setAge(int age){  
        this.age = age;  
    }  
}  
  
class Teacher extends Persion{  
    private String qualification;  
  
    public void setQualification(String qualification){  
        this.qualification = qualification;  
    }  
}
```

```
public String getQualification(){
    return qualification;
}
}

public class Test {
    public static void main(String[] args) {
        Teacher t1 = new Teacher();

        String name1 = "Anisul Islam";
        int age1 = 25;
        t1.setName(name1);
        t1.setAge(age1);
        t1.setQualification("Online teacher");
        System.out.println("Name : "+t1.getName());
        System.out.println("Age : "+t1.getAge());
        System.out.println("Qualification :
"+t1.getQualification());

        System.out.println();
    }

    Teacher t2 = new Teacher();

    String name2 = "Hamim Talukder";
    int age2 = 23;
    t2.setName(name2);
    t2.setAge(age2);
    t2.setQualification("BSC in CSE");
    System.out.println("Name : "+t2.getName());
    System.out.println("Age : "+t2.getAge());
```

```
        System.out.println("Qualification :  
" + t2.getQualification());  
    }  
}  
  
or,  
class Persion{  
    private String name;  
    private int age;  
  
    public String getName(){  
        return name;  
    }  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public int getAge(){  
        return age;  
    }  
  
    public void setAge(int age){  
        this.age = age;  
    }  
}  
  
class Teacher extends Persion{  
    private String qualification;
```

```
public void setQualification(String qualification){  
    this.qualification = qualification;  
}  
  
public String getQualification(){  
    return qualification;  
}  
  
void displayInfo()  
{  
    System.out.println("Name : "+getName());  
    System.out.println("Age : "+getAge());  
    System.out.println("Qualification :  
"+getQualification());  
}  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Teacher t1 = new Teacher();  
  
        String name1 = "Anisul Islam";  
        int age1 = 25;  
        t1.setName(name1);  
        t1.setAge(age1);  
        t1.setQualification("Online teacher");  
        t1.displayInfo();  
  
        System.out.println();  
        Teacher t2 = new Teacher();
```

```
String name2 = "Hamim Talukder";
int age2 = 23;
t2.setName(name2);
t2.setAge(age2);
t2.setQualification("BSC in CSE");
t2.displayInfo();
}
}
```

Output:

Name : Anisul Islam

Age : 25

Qualification : Online teacher

Name : Hamim Talukder

Age : 23

Qualification : BSC in CSE

- **instanceof Operator :**

instanceof is a keyword that is used for checking if a reference variable is containing a given type of object reference or not. Following is a Java program to show different behaviors of instanceof. Henceforth it is known as a comparison operator where the instance is getting compared to type returning boolean true or false as in java we do not have 0 and 1 boolean return types.

- **What is the output of the following Java program fragment:**

```
class Alian
{
}

class Person extends Alian
{
}

class Teacher extends Person{

}

public class Test{
    public static void main(String[] args) {
        Alian a = new Alian();
```

```
Person p = new Person();
Teacher t =new Teacher();

System.out.println(a instanceof Alian);
System.out.println(p instanceof Alian);
System.out.println(t instanceof Person);
System.out.println(t instanceof Alian);
System.out.println(p instanceof Teacher);

}

}
```

Output:

```
true
true
true
true
false
```

- **What is the output of the following Java program fragment:**

```
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        GFG object = new GFG();
        System.out.println(object instanceof GFG);
    }
}
```

Output:

true

- **What is the output of the following Java program fragment:**

```
class Parent {}
```

```
class Child extends Parent {}
```

```
class GFG {
```

```
    public static void main(String[] args)
    {
```

```
        Child cobj = new Child();
```

```
        if (cobj instanceof Child)
            System.out.println("cobj is instance of Child");
        else
            System.out.println(
                "cobj is NOT instance of Child");
```

```
        if (cobj instanceof Parent)
            System.out.println(
                "cobj is instance of Parent");
        else
            System.out.println(
                "cobj is NOT instance of Parent");
```

```
        if (cobj instanceof Object)
```

```
System.out.println(  
    "cobj is instance of Object");  
else  
    System.out.println(  
        "cobj is NOT instance of Object");  
}  
}
```

Output:

cobj is instance of Child
cobj is instance of Parent
cobj is instance of Object

- **What is the output of the following Java program fragment:**

```
class Test {}  
  
class Main  
{  
    public static void main(String[] args)  
    {  
        Test tobj = null;  
  
        if (tobj instanceof Test)  
            System.out.println("tobj is instance of Test");  
        else  
            System.out.println("tobj is NOT instance of  
Test");  
    }  
}
```

Ourput:

tobj is NOT instance of Test

- **What is the output of the following Java program fragment:**

```
class Parent { }
class Child extends Parent { }

class Test
{
    public static void main(String[] args)
    {
        Parent pobj = new Parent();
        if (pobj instanceof Child)
            System.out.println("pobj is instance of Child");
        else
            System.out.println("pobj is NOT instance of Child");
    }
}
```

Output:

pobj is NOT instance of Child

- **What is the output of the following Java program fragment:**

```
class Parent { }
class Child extends Parent { }

class Test
{
    public static void main(String[] args)
    {
        Parent cobj = new Child();
        if (cobj instanceof Child)
            System.out.println("cobj is instance of Child");
        else
            System.out.println("cobj is NOT instance of Child");
    }
}
```

Output:

cobj is instance of Child

Here,

We have seen here that a parent class data member is accessed when a reference of parent type refers to a child object. We can access child data members using typecasting.

Syntax:

((child_class_name)
Parent_Reference_variable).func.name()

- **What is the output of the following Java program fragment:**

```
class Parent  
{  
    int value = 1000;  
}
```

```
class Child extends Parent  
{  
    int value = 10;  
}
```

```
class Test  
{  
    public static void main(String[] args)  
    {  
        Parent cobj = new Child();  
        Parent par = cobj;  
  
        if (par instanceof Child)  
        {  
            System.out.println("Value accessed through " +  
                "parent reference with typecasting is " +  
                ((Child)par).value);  
        }  
    }  
}
```

Output:

Value accessed through parent reference with typecasting is 10

15. Generics

Generics in Java is a feature that allows you to write classes and methods that can work with different types while providing compile-time type safety. Generics enable you to create classes, interfaces, and methods that operate on parameters of any data type, yet maintain the type information at compile time.

The main benefits of generics in Java include:

- **Type Safety:** With generics, you can specify the type of elements that a class or method can work with. This ensures type safety at compile time, reducing the risk of runtime errors related to type mismatches.
- **Code Reusability:** Generics promote code reusability by allowing you to write classes and methods that are independent of specific data types. This makes your code more flexible and adaptable.
- **Elimination of Type Casting:** Generics eliminate the need for explicit type casting when retrieving elements from collections, making the code cleaner and less error-prone.

- **Enhanced Readability:** Generics make code more readable by clearly expressing the intended type of data being used in a class or method.

Several examples of generics in Java, covering various use cases:

1. Generic Class: Box

```
class Box<T> {  
    private T value;  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
  
    public T getValue() {  
        return value;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Box<Integer> intBox = new Box<>();  
        intBox.setValue(42);  
  
        Box<String> strBox = new Box<>();  
        strBox.setValue("Hello, Generics!");  
  
        int intValue = intBox.getValue();  
        String stringValue = strBox.getValue();  
  
        System.out.println("Integer Box Value: " +  
                           intValue);  
    }  
}
```

```
        System.out.println("String Box Value: " +  
stringValue);  
    }  
}
```

Output:

Integer Box Value: 42
String Box Value: Hello, Generics!

2. Generic Method: Pair

```
class Pair<K, V> {  
    private K key;  
    private V value;  
  
    public Pair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() {  
        return key;  
    }  
  
    public V getValue() {  
        return value;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Pair<Integer, String> pair = new Pair<>(1, "One");  
  
        int key = pair.getKey();  
        String value = pair.getValue();  
  
        System.out.println("Key: " + key);  
        System.out.println("Value: " + value);  
    }  
}
```

Output:

Key: 1
Value: One

3. Generic Interface: List

```
interface List<T> {  
    void add(T element);  
    T get(int index);  
}  
  
class ArrayList<T> implements List<T> {  
    private java.util.ArrayList<T> elements = new  
    java.util.ArrayList<>();  
  
    @Override  
    public void add(T element) {  
        elements.add(element);  
    }  
  
    @Override  
    public T get(int index) {  
        return elements.get(index);  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        List<String> stringList = new ArrayList<>();
```

```
        stringList.add("Java");
        stringList.add("C++");

        System.out.println("Element at index 0: " +
stringList.get(0));
        System.out.println("Element at index 1: " +
stringList.get(1));
    }
}
```

Output:

```
Element at index 0: Java
Element at index 1: C++
```

4. Generic Inheritance: T extends Animal

```
class Animal{
    private String eat;

    public void setEat(String eat){
        this.eat=eat;
    }

    public String getEat(){
        return eat;
    }
}

class Cat extends Animal{
    private String name;
```

```
public void setName(String name){  
    this.name = name;  
}  
  
public String getName(){  
    return name;  
}  
}  
  
class Printer<T extends Animal>{  
    T objectName;  
  
    public Printer(T objectName){  
        this.objectName = objectName;  
    }  
  
    public void print(){  
        System.out.println(objectName.getEat());  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        Cat cat = new Cat();  
        cat.setName("Billu");  
        cat.setEat("Begun");  
  
        Printer<Cat> ob= new Printer<>(cat);  
        ob.print();  
    }  
}
```

Output:

Begun

5. Generic Method : any types of variable

```
public class Test{  
    private static <T> void show (T thingToShow){  
        System.out.println(thingToShow);  
    }  
  
    public static void main(String[] args) {  
        show("Jim");  
        show(57);  
        show(57+"!!!");  
        show(12.5);  
    }  
}
```

Output:

Jim

57

57!!!

12.5

6. Generic Method : any types of variables pair

```
public class Test{  
    private static <T, V> void show (T thingToShow,  
V otherThingToShow){  
    System.out.print(thingToShow);  
    System.out.println(" "+otherThingToShow);  
}  
  
public static void main(String[] args) {  
    show("Jim", 12);  
    show(57, 13);  
    show(57+"!!!", 14);  
    show(12.5, 15);  
}  
}
```

Output:

Jim 12
57 13
57!!! 14
12.5 15

7. Generic Method : any type of variables paired with return type

```
public class Test{  
    private static <T, V> T show (T thingToShow, V  
otherThingToShow){  
    System.out.print(thingToShow);  
    System.out.println(" "+otherThingToShow);  
  
    return thingToShow;  
}  
  
public static void main(String[] args) {  
    System.out.println(show("Jim", 12));  
    System.out.println(show(57, 13));  
    System.out.println(show(57+"!!!", 14));  
    System.out.println(show(12.5, 15));  
}  
}
```

Output:

```
Jim 12  
Jim  
57 13  
57  
57!!! 14  
57!!!  
12.5 15  
12.5
```

8. Generic List : Wild card (?)

```
import java.util.ArrayList;
import java.util.List;

public class Test{
    private static void printList (List<?> myList){
        System.out.println(myList);
    }
    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<>();
        intList.add(2);
        intList.add(3);
        printList(intList);
    }
}
```

Output:

[2, 3]

9. Generic method : Object type

```
class Printer<T>{
    T data;
    public Printer(T a){
        this.data = a;
    }

    public void print(){
        System.out.println(data);
    }
}

public class Test{
    public static void main(String[] args) {
        Printer<Integer> intPrinter = new Printer<>(23);
        intPrinter.print();

        Printer<Double> doublePrinter = new Printer<>(23.56);
        doublePrinter.print();

        Printer<Object> intPrinter2 = new Printer<>(23);
        intPrinter2.print();

        int a = (int) intPrinter2.data;
        System.out.println(a);

        Printer<Object> doublePrinter2 = new Printer<>(23.56);
        doublePrinter2.print();
```

```
    double b = (double) doublePrinter2.data;
    System.out.println(b);
}
}
```

Output:

```
23
23.56
23
23
23.56
23.56
```

Here,

We used Object type to use any types of variable or value. And if we use Object type variable, we must typecast the value during code implementation.

10. Generic method : class type

```
class Printer<T>{
    T data;
    public Printer(T a){
        this.data = a;
    }
}

class Cat {
    String name;

    public void setName(String name){
        this.name = name;
    }

    public String getname(){
        return name;
    }
}

public class Jim{
    public static void main(String[] args) {
        Cat ob = new Cat();
        ob.setName("Billu");

        Printer<Cat> catPrinter = new Printer<>(ob);
        System.out.println(catPrinter.data.name);

        Cat a = catPrinter.data;
        System.out.println(a.name);
    }
}
```

Output:

Billu
Billu

11. Generic method : Object type (class)

```
class Printer<T>{
    T data;
    public Printer(T a){
        this.data = a;
    }
}

class Cat {
    String name;

    public void setName(String name){
        this.name = name;
    }

    public String getname(){
        return name;
    }
}

public class Jim{
    public static void main(String[] args) {
        Cat ob = new Cat();
        ob.setName("Billu");
```

```
Printer<Object> catPrinter = new Printer<>(ob);
System.out.println(((Cat)catPrinter.data).name);

Cat a = (Cat) catPrinter.data;
System.out.println(a.name);
}

}
```

Output:

Billu

Billu

Here,

We used Object type to use any types of variable or value. And if we use Object type variable, we must typecast the value during code implementation.

16. Lambda Expressions

In Java, "lambda" refers to a feature introduced in Java 8 as part of the Java programming language's evolution. Lambda expressions provide a concise way to express instances of single-method interfaces (functional interfaces). They are a way to enable functional programming style in Java, allowing you to treat functionality as a method argument and create more expressive and readable code.

The basic syntax of a lambda expression looks like this:

(parameters) -> expression

Example: 1

```
interface Printable{
    void print();
}

class Cat implements Printable{
    public String name;
    public int age;

    public Cat(){}
    
    public void print(){
        System.out.println("Meow");
    }
}
```

```
}

public class Test{

    static void printThing(Printable thing){
        thing.print();
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        printThing(myCat);

        // Lambda Expressions ()->
        printThing(
            () -> {
                System.out.println("Meow");
            }
        );

        // or,
        printThing(
            () -> {
                System.out.println("Hamim");
            }
        );

        // or,
        printThing(() -> System.out.println("Meow"));

        // or,
    }
}
```

```
    Printable lambdaPrintable = ()->
        System.out.println("Meow");
        printThing(lambdaPrintable);

    }

}
```

Output:

Meow
Meow
Hamim
Meow
Meow

Example: 2

```
interface Printable{  
    void print(String prifix, String suffix);  
}  
  
public class Test{  
  
    static void printThing(Printable thing){  
        thing.print("-> , , !");  
    }  
  
    public static void main(String[] args) {  
  
        Printable lambdaPrintable = (p, s)->  
        System.out.println(p + "Meow" + s);  
        printThing(lambdaPrintable);  
  
    }  
}
```

Output:

-> Meow !

Example: 3

```
interface Printable{
    String print(String suffix);
}

public class Test{

    static String printThing(Printable thing){
        return thing.print(" !");
    }

    public static void main(String[] args) {

        Printable lambdaPrintable = (s)-> {
            System.out.println("Meow" + s);
            return "Hamim";
        };

        System.out.println(printThing(lambdaPrintable));
    }
}
```

Output:

Meow !
Hamim

Example: 4

```
interface Printable{
    String print(String suffix);
}

public class Test{

    static String printThing(Printable thing){
        return thing.print(" !");
    }

    public static void main(String[] args) {

        Printable lambdaPrintable = (s)-> {
            return "Hamim" + s;
        };

        System.out.println(printThing(lambdaPrintable));

    }
}
```

Output:

Hamim !

Example: 5

```
interface Printable{
    String print(String suffix);
}

public class Test{

    static String printThing(Printable thing){
        return thing.print(" !");
    }

    public static void main(String[] args) {
        Printable lambdaPrintable = (s)-> "Hamim" + s;
        System.out.println(printThing(lambdaPrintable));
    }
}
```

Output:

Hamim !

17. Type Casting

Converting one data type to another is called type casting.

2 types of type casting of primitive and non-primitive data types:

(1) primitive data type casting :

1. Implicit type casting (primitive data type)
2. Explicit type casting (non-primitive data type)

1. Implicit type casting :

byte < short < int < long < float < double

→
widening

Example:

```
package FirstPackage;
```

```
public class Test {  
    public static void main(String[] args){  
        int x = 10;  
        double y = x;  
        System.out.println(y);  
    }  
}
```

2. Explicit type casting :

double > float > long > int > short > byte

Narrowing

Example:

```
package FirstPackage;

public class Test {
    public static void main(String[] args){
        double x = 10.5;
        int y = (int) x;
        System.out.println(y);
    }
}
```

(2) non-primitive data type/ object type casting :

1. Upcasting
2. Downcasting

1. Upcasting :

Example :

```
package FirstPackage;
```

```
public class Person {  
    void display(){  
        System.out.println("Person class");  
    }  
}
```

```
package FirstPackage;
```

```
class Teacher extends Person{  
    void display(){  
        System.out.println("Teacher class");  
    }  
}
```

```
package FirstPackage;
```

```
public class Test {  
    public static void main(String[] args){
```

```
Person p = new Teacher();
p.display();
}
}
```

Output:

Teacher class

2. Downcasting :

Example :

```
package FirstPackage;

public class Person {
    void display(){
        System.out.println("Person class");
    }
}
```

```
package FirstPackage;

class Teacher extends Person{
    void display(){
        System.out.println("Teacher class");
    }
}
```

```
package FirstPackage;

public class Test {
    public static void main(String[] args){
        Person p = new Teacher();
        p.display();

        Teacher t = (Teacher) new Person();
        t.display();
    }
}
```

Output:

Teacher class

```
Exception in thread "main"
java.lang.ClassCastException: class
FirstPackage.Person cannot be cast to class
FirstPackage.Teacher (FirstPackage.Person and
FirstPackage.Teacher are in unnamed module of
loader 'app')
at FirstPackage.Test.main(Test.java:8)
```

18. compareTo() method

Here, Comparable is a built-in interface with generics<T> of Java and compareTo() is an abstract function of it.

Example: 1

```
/* Here, Comparable is a builtin interface of Java and
   compareTo() is an abstract function of it.
*/
class Student implements Comparable<Student>{
    int rollNo;
    float marks;

    public Student(int rollNo, float marks){
        this.rollNo = rollNo;
        this.marks = marks;
    }

    @Override
    public int compareTo(Student o) {
        int diff = (int)(this.marks - o.marks);
        // if diff == 0; means both are equal
        // if diff < 0; means o is bigger
        return diff;
    }
}

public class Test {
```

```
public static void main(String[] args) {  
    Student Hamim = new Student(12, 89.76f);  
    Student Rahul = new Student(13, 99.76f);  
  
    if(Hamim.compareTo(Rahul) < 0){  
        System.out.println("Rahul has more marks");  
    }else  
        System.out.println("Hamim has more marks");  
}  
}
```

Output:

Rahul has more marks

Example: 2

```
import java.util.Arrays;

class Student implements Comparable<Student>
{
    int rollNo;
    float marks;

    public Student(int rollNo, float marks){
        this.rollNo = rollNo;
        this.marks = marks;
    }

    @Override
    public String toString(){
        return marks + "";
    }

    @Override
    public int compareTo(Student o) {
        int diff = (int)(this.marks - o.marks);
        // if diff == 0; means both are equal
        // if diff < 0; means o is bigger
        return diff;
    }
}

public class Test {

    public static void main(String[] args) {
```

```
Student Hamim = new Student(12, 89.76f);
Student Rahul = new Student(13, 99.74f);
Student Mollah = new Student(14,75.45f);
Student Rakib = new Student(15, 37.25f);
Student Abdullah = new Student(16, 68.59f);

Student[] list = {Hamim, Rahul, Mollah, Rakib,
Abdullah};

System.out.println(Arrays.toString(list));
Arrays.sort(list);
System.out.println(Arrays.toString(list));
}
}
```

Output:

```
[89.76, 99.74, 75.45, 37.25, 68.59]
[37.25, 68.59, 75.45, 89.76, 99.74]
```

Example: 3 (using Lamda function in ascending order)

```
import java.util.Arrays;

class Student {
    int rollNo;
    float marks;

    public Student(int rollNo, float marks){
        this.rollNo = rollNo;
        this.marks = marks;
    }

    @Override
    public String toString(){
        return marks + "";
    }
}

public class Test {

    public static void main(String[] args) {
        Student Hamim = new Student(12, 89.76f);
        Student Rahul = new Student(13, 99.74f);
        Student Mollah = new Student(14, 75.45f);
        Student Rakib = new Student(15, 37.25f);
        Student Abdullah = new Student(16, 68.59f);

        Student[] list = {Hamim, Rahul, Mollah, Rakib,
        Abdullah};
    }
}
```

```
System.out.println(Arrays.toString(list));

// sort according to marks using Lambda in
ascending order

Arrays.sort(list, new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2){
        return (int)(o1.marks - o2.marks);
    }

});

System.out.println(Arrays.toString(list));
}
}
```

Output:

```
[89.76, 99.74, 75.45, 37.25, 68.59]
[37.25, 68.59, 75.45, 89.76, 99.74]
```

Example: 4 (using Lamda function and Lambda expression in decending order)

```
import java.util.Arrays;

class Student {
    int rollNo;
    float marks;

    public Student(int rollNo, float marks){
        this.rollNo = rollNo;
        this.marks = marks;
    }

    @Override
    public String toString(){
        return marks + "";
    }
}

public class Test {

    public static void main(String[] args) {
        Student Hamim = new Student(12, 89.76f);
        Student Rahul = new Student(13, 99.74f);
        Student Mollah = new Student(14, 75.45f);
        Student Rakib = new Student(15, 37.25f);
        Student Abdullah = new Student(16, 68.59f);

        Student[] list = {Hamim, Rahul, Mollah, Rakib,
        Abdullah};
    }
}
```

```
System.out.println(Arrays.toString(list));

// sort according to marks using Lambda in
deccending order

Arrays.sort(list, new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2){
        return -(int)(o1.marks - o2.marks);
    }

});

System.out.println(Arrays.toString(list));
}
```

Output:

```
[89.76, 99.74, 75.45, 37.25, 68.59]
[99.74, 89.76, 75.45, 68.59, 37.25]
```

```
or,  
import java.util.Arrays;  
  
class Student {  
    int rollNo;  
    float marks;  
  
    public Student(int rollNo, float marks){  
        this.rollNo = rollNo;  
        this.marks = marks;  
    }  
  
    @Override  
    public String toString(){  
        return marks + "";  
    }  
  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
        Student Hamim = new Student(12, 89.76f);  
        Student Rahul = new Student(13, 99.74f);  
        Student Mollah = new Student(14,75.45f);  
        Student Rakib = new Student(15, 37.25f);  
        Student Abdullah = new Student(16, 68.59f);  
  
        Student[] list = {Hamim, Rahul, Mollah, Rakib,  
        Abdullah};
```

```
System.out.println(Arrays.toString(list));  
  
        // sort according to marks using Lambda in  
        // ascending order  
        Arrays.sort(list, (o1, o2) -> -(int)(o1.marks -  
o2.marks));  
        System.out.println(Arrays.toString(list));  
    }  
}
```

Output:

```
[89.76, 99.74, 75.45, 37.25, 68.59]  
[99.74, 89.76, 75.45, 68.59, 37.25]
```

19. Anonymous class

Example :

```
package FirstPackage;
```

```
public class Person {  
    void display(){  
        System.out.println("Person class");  
    }  
}
```

```
package FirstPackage;
```

```
public class Test {  
    public static void main(String[] args){  
        Person p = new Person() {  
            void display(){  
                System.out.println("Test class");  
            }  
        };  
        p.display();  
    }  
}
```

Output :

Test class

20. Exception handling

There are mainly two types of exceptions:
checked and unchecked
where error is considered as unchecked
exception.
The sun microsystem says there are three types
of exceptions:

1. Error
2. Checked Exception
3. Unchecked Exception

1. Error:

Error is irrecoverable e.g. OutOfMemoryError,
VirtualMachineError, AssertionError
etc.

- **Difference between Error and Exception**

Errors:

- Indicate serious problems and abnormal conditions that most applications should not try to handle.
- Error defines problems that are not expected to be caught under normal circumstances by our program.
- For example:
memory error, hardware error, JVM error etc.
- Errors are typically ignored in code because you can rarely do anything about an error.
- Example: if stack overflow occurs, an error will arise. This type of error is not possible handle in code.

Exceptions:

- Exception is an abnormal condition.
- An exception (or exceptional event) is a problem that arises during the execution of a program.
- In java, exception is an event that disrupts the normal flow of the program.
- It is an object which is thrown at runtime.

2. Checked Exceptions :

- All exceptions other than Runtime Exceptions are known as Checked
- exceptions as the compiler checks them during compilation
- A checked exception is an exception that occurs at the compile time,
- these are also called as compile time exceptions.
- These exceptions cannot simply be ignored at the time of compilation

Some checked exceptions are as follows:

- ClassNotFoundException
- IllegalAccessException
- NoSuchFieldException
- EOFException

3. Unchecked Exceptions :

- Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not
- These exceptions need not be included in any method's throws list
- because compiler does not check to see if a method handles or throws these exceptions.

Some unchecked exceptions are as follows:

- ArithmaticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- NegativeArraySizeException

- **Exception handling**

The exception handling is one of the powerful mechanism to handle the runtime errors.

An exception is a run time error.

An exception is an abnormal condition that arises in a code sequence at run time.

- **Advantage of Exception Handling:**

The core advantage of exception handling is to maintain the normal flow of the application.

Let's take a scenario:

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

```
statement 1;  
statement 2;  
statement 3;  
statement 4;  
statement 5;//exception occurs  
statement 6;  
statement 7;  
statement 8;  
statement 9;  
statement 10;
```

Different types of exceptions :

(1)

```
int x = 10;
```

```
int y = 0;
```

```
int result = x/y; -----> ArithmeticException:
```

Can't divide a number by 0

(2)

```
String name = null;
```

```
System.out.println(name.charAt(0));
```

---> NullPointerException

(3)

```
String name = "HaMeem" // length is 6
```

```
System.out.println(name.charAt(9));
```

-----> StringIndexOutOfBoundsException

(4)

```
int num = Integer.parseInt ("anis");
```

-----> NumberFormatException

(5)

```
File file = new File ("D: //file.txt");
```

-----> FileNotFoundException

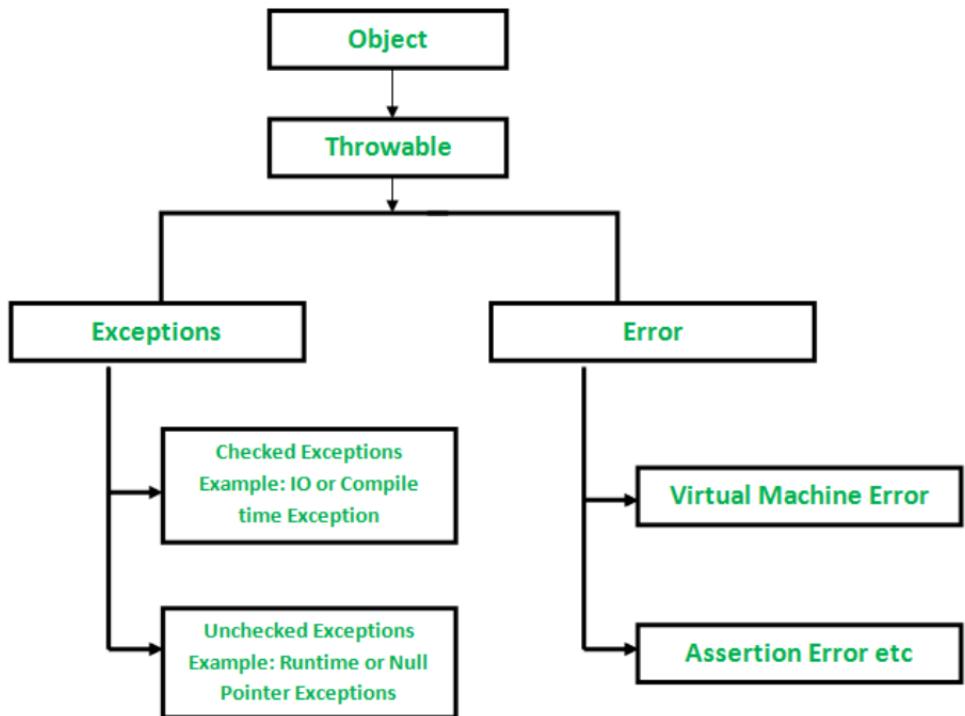
(6)

```
int a[ ] = new int[5]  
a[5] = 32;
```

-----> `ArrayIndexOutOfBoundsException`

Some other exceptions :

```
ClassNotFoundException  
IOException  
NoSuchMethodException  
Etc.
```



Types of Exceptions

User-Defined Exception

Built-in Exception

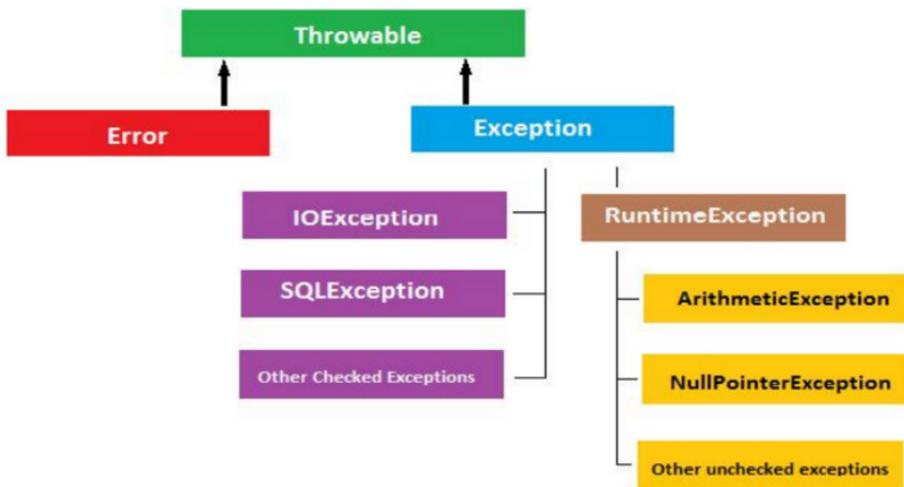
Checked Exceptions

- > ClassNotFoundException
- > InterruptedException
- > IOException
- > InstantiationException
- > SQLException
- > FileNotFoundException

Unchecked Exceptions

- > ArithmeticException
- > ClassCastException
- > NullPointerException
- > ArrayIndexOutOfBoundsException
- > ArrayStoreException
- > IllegalThreadStateException

Exception classes



Exception handling is managed by 5 keywords:

1. try
2. catch
3. finally
4. throw
5. throws

```
try{  
    //code you want to monitor  
} catch (ExceptionType1 e1) {  
    //exception handler for exception  
} catch (ExceptionType2 e2) {  
    //exception handler for exception  
}  
.....  
finally{  
    //block of code to be executed after try block  
}
```

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

public class Test {
    public static void main(String[] args){

        try{
            int x = 10;
            int y = 0;
            int result = x / y;
            System.out.println("Result : "+result);
        }

        catch(ArithmaticException a){
            System.out.println("Exception : "+a);
        }

        System.out.println("Last line of the program");
    }
}
```

Output:

Exception : java.lang.ArithmaticException: / by zero
Last line of the program

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

public class Test {
    public static void main(String[] args){

        try{
            int x = 10;
            int y = 0;
            int result = x / y;
            System.out.println("Result : "+result);
        }

        catch(Exception a){
            System.out.println("Exception : "+a);
        }

        System.out.println("Last line of the program");
    }
}
```

Output:

Exception : java.lang.ArithmaticException: / by zero
Last line of the program

Here,

In Exception method all Exception handling method are included that's it can check all exception.

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.lang.reflect.Array;

public class Test {
    public static void main(String[] args){

        try{
            int x = 10;
            int y = 0;
            int result = x / y;
            System.out.println("Result : "+result);
        }

        catch(ArrayIndexOutOfBoundsException a){
            System.out.println("Exception : "+a);
        }

        finally {
            System.out.println("Last line of the program");
        }

    }
}
```

Output:

Last line of the program

Exception in thread "main"

java.lang.ArithmetricException: / by zero
at FirstPackage.Test.main(Test.java:11)

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.lang.reflect.Array;

public class Test {
    public static void main(String[] args){

        try{
            int x = 10;
            int y = 0;
            int result = x / y;
            System.out.println("Result : "+result);
        }

        catch(ArrayIndexOutOfBoundsException a){
            System.out.println("Exception : "+a);
        }

        catch(ArithmeticException b){
            System.out.println("Exception : "+b);
        }

        finally {
            System.out.println("Last line of the program");
        }
    }
}
```

Output:

Exception : java.lang.ArithmaticException: / by zero
Last line of the program

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.lang.reflect.Array;

public class Test {
    public static void main(String[] args){

        try{
            int x = 10;
            int y = 0;
            int result = x / y;
            System.out.println("Result : "+result);
        }

        catch(ArithmaticException b){
            System.out.println("Exception : "+b);
            System.out.println("A");
        }

        catch(Exception a){
            System.out.println("Exception : "+a);
            System.out.println("B");
        }
    }
}
```

```
    finally {  
        System.out.println("Last line of the program");  
    }  
}  
}
```

Output:

```
Exception : java.lang.ArithmetricException: / by zero  
A  
Last line of the program
```

Here,

If we want to use other Exception handling method
then we must write them before the Exception
method.

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.lang.reflect.Array;

public class Test {
    public static void main(String[] args){

        try{
            int a[] = new int[4];
            a[4] = 10;
        }

        catch(ArithmetricException b){
            System.out.println("Exception : "+b);
        }

        catch(Exception a){
            System.out.println("Exception : "+a);
        }

        finally {
            System.out.println("Last line of the
program");
        }

    }
}
```

Output:

Exception :

java.lang.ArrayIndexOutOfBoundsException: Index
4 out of bounds for length 4

Last line of the program

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.util.Scanner;
import java.util.SortedMap;

public class Test {
    public static void main(String[] args){

        while(true){
            try{
                Scanner scan = new Scanner(System.in);
                System.out.print("Please enter num1 : ");
                int num1 = scan.nextInt();
                System.out.print("Please enter num2 : ");
                int num2 = scan.nextInt();
                int result = num1/num2;
                System.out.println("Result :
"+num1+"/"+num2+" = "+result);
            }

            catch(Exception a){
                System.out.println("Exception: "+a);
                System.out.println("You must enter an
integer. Please try again");
            }
        }
    }
}
```

Output:

```
Result : 100/7 = 14
Please enter num1 : 100
Please enter num2 : Hello
Exception: java.util.InputMismatchException
You must enter an integer. Please try again
Please enter num1 : 100
Please enter num2 : 0
Exception: java.lang.ArithmetricException: / by zero
You must enter an integer. Please try again
Please enter num1 :
```

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.util.Scanner;
import java.util.SortedMap;

public class Test {
    public static void main(String[] args){

        int count = 1;
        do {
            try{
                Scanner scan = new Scanner(System.in);
                System.out.print("Please enter num1 : ");
                int num1 = scan.nextInt();
                System.out.print("Please enter num2 : ");
            }
        }
    }
}
```

```
int num2 = scan.nextInt();
int result = num1/num2;
System.out.println("Result :
"+num1+"/"+num2+" = "+result);
count = 2;
}

catch(Exception a){
    System.out.println("Exception: "+a);
    System.out.println("You must enter an
integer. Please try again");
}
}while(count==1);

}
}
```

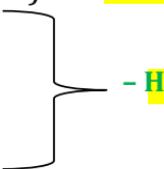
Output:

```
Please enter num1 : 2
Please enter num2 : 0
Exception: java.lang.ArithmetricException: / by zero
You must enter an integer. Please try again
Please enter num1 : 12
Please enter num2 : 4
Result : 12/4 = 3
```

21. UML

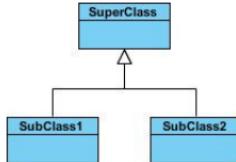
Contents

- **UML Basics**

- Generalization (Inheritance) - **IS-A Relationship** -
 - Association
 - Aggregation
 - Composition
- 
- **HAS-A Relationship**

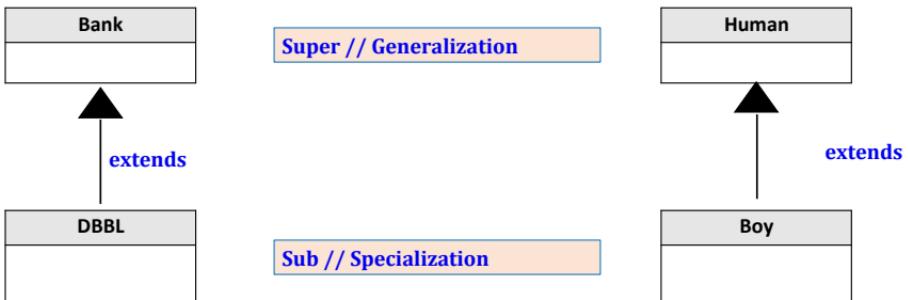
Topic - 1 : Inheritance (Generalization)

- A generalization is a taxonomic relationship between a **more general** classifier and a **more specific** classifier.
- Represents an "**is-a**" relationship.



- **SubClass1** and **SubClass2** are **specializations** of **SuperClass**.

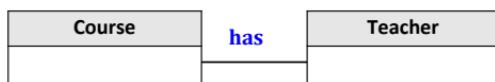
IS-A relationship



4

Topic - 2 : Association

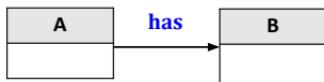
- **Relationship** between **two separate classes** which establishes through their Objects..
- Also known as "**has-a**" relationship.
- Each class is **independent**. They can exist without each other.



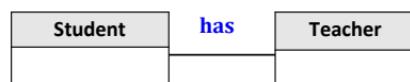
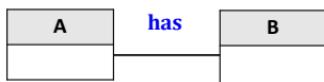
5

Two types of Association

- **Unary Association:** Class A has Class B; But Class B does not have Class A



- **Binary Association:** Both Classes know about each other.



Two types of Association

1. Unary Association



Customer Class will have the object of Product

Two types of Association

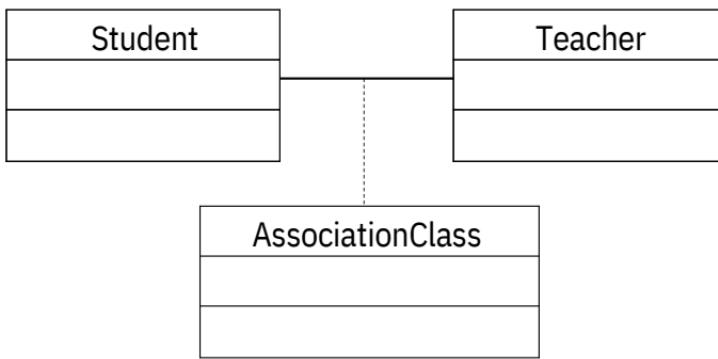
2. Binary Association



8

Two types of Association

2. Binary Association (Solved by Association Class)



9

Topic - 3 : Aggregation and Composition

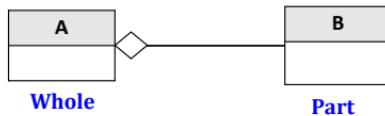
[Whole/Part Relationship]

3.1: Aggregation

- Specified form of Association

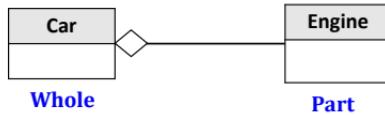
10

- Whole/Part Relationship



- Class B is a part of Class A.

- Both Classes are independent. Part class can exist without Whole Class.

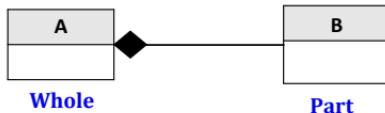


11

3.2: Composition

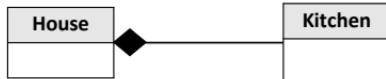
- Specified (**Stronger**) form of Association

- Whole/Part Relationship



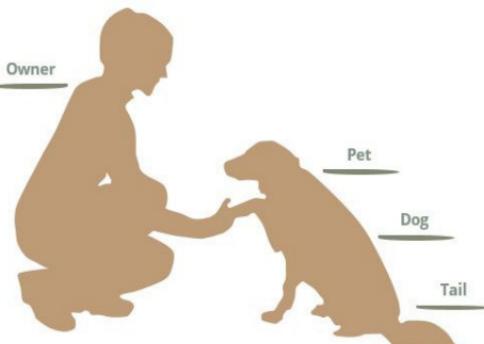
- Class B is a part of Class A.

- The existence of Part class depends on Whole Class.



The figure below shows the three types of association connectors: **association**, **aggregation** and **composition**

Association • Aggregation • Composition

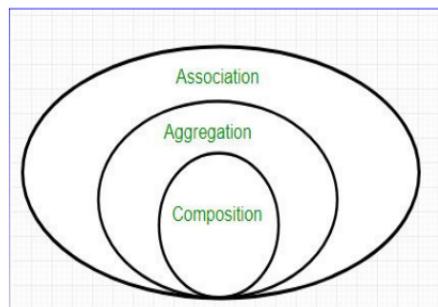
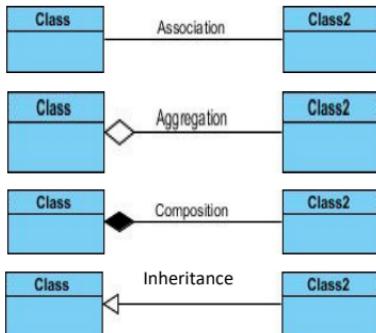


We see the following relationships:

- owners feed pets, pets please owners
(Association)
- a tail is a part of both dogs
(Aggregation / Composition)
- a dog is a kind of pet
(Inheritance / Generalization)

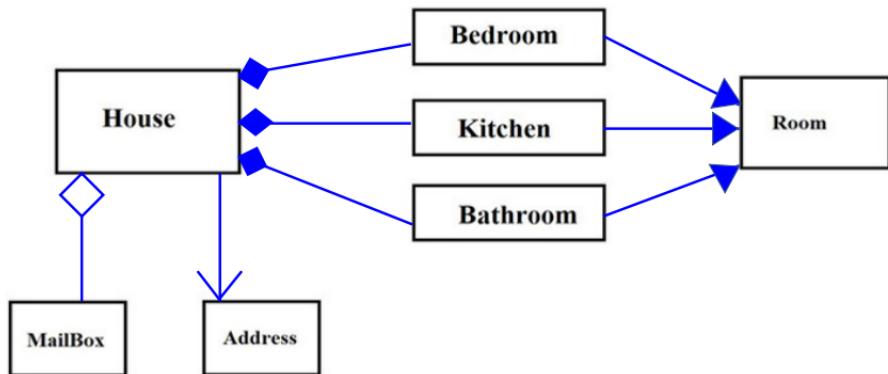
13

UML Notation



14

Example - 1



15

Example - 2: Description to UML

- Volvo is a Car.
- Every car has model and company of type string.
- Volvo has price of type double, productionYear and registrationNumber of type string. Volvo also has Engine and DashBoard.
- Engine has capacity of type double.
- DashBoard has size of type double.
- Car provides drive and stop as abstract service or method of type void.
- Volvo also provides changeFuel and checkBattery service or method of type void.

16

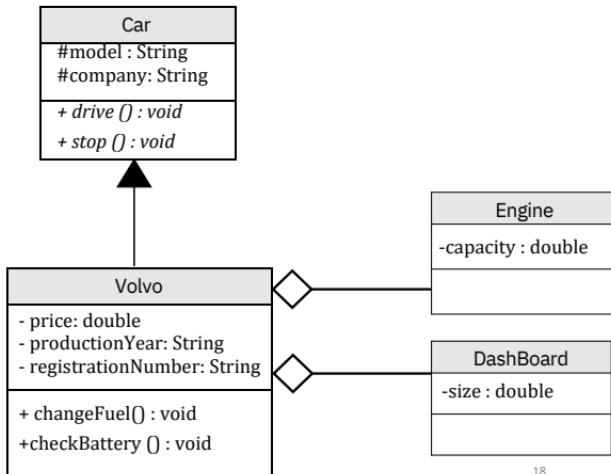
Example – 2: Description to UML

- Volvo **is a** Car.
- Every car **has** model and company of type **string**.
- Volvo **has** price of type **double**, productionYear and registrationNumber of type **string**.
- Volvo also **has** Engine and DashBoard.
- Engine **has** capacity of type **double**.
- DashBoard **has** size of type **double**.
- Car **provides** drive and stop as abstract service or method of type void.
- Volvo also **provides** changeFuel and checkBattery service or method of type void.

17

Solution of Example – 2: Description to UML

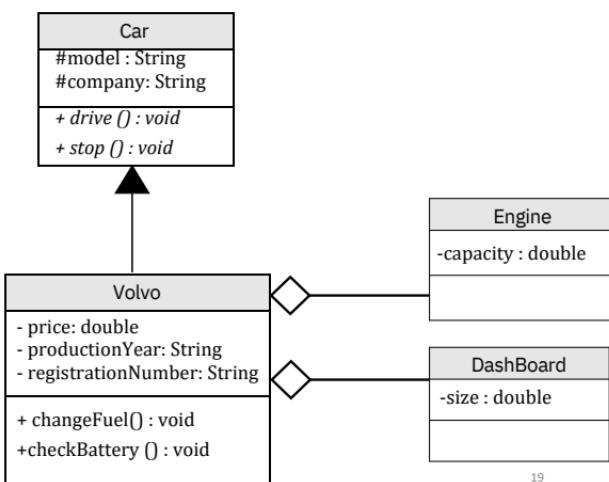
- Volvo **is a** Car.
- Every car **has** model and company of type **string**.
- Volvo **has** price of type **double**, productionYear and registrationNumber of type **string**.
- Volvo also **has** Engine and DashBoard.
- Engine **has** capacity of type **double**.
- DashBoard **has** size of type **double**.
- Car **provides** drive and stop as abstract service or method of type void.
- Volvo also **provides** changeFuel and checkBattery service or method of type void.



18

Solution of Example – 2: Description to UML

- Volvo **is a** Car.
- Every car **has** model and company of type string.
- Volvo **has** price of type double, productionYear and registrationNumber of type string.
- Volvo also **has** Engine and DashBoard.
- Engine **has** capacity of type double.
- DashBoard **has** size of type double.
- Car **provides** drive and stop as abstract service or method of type void.
- Volvo also **provides** changeFuel and checkBattery service or method of type void.



19

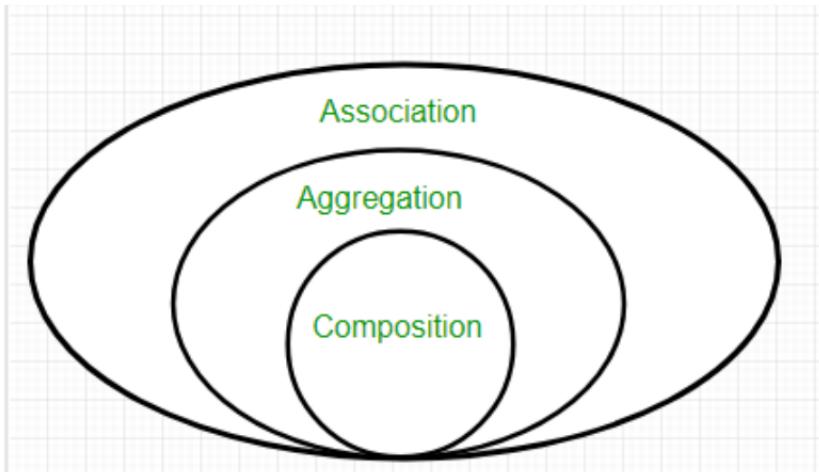
Example – 3: Try Yourself

- Apple is a fruit
- Apple has color and
- origin Apple has Sticker
- Sticker has logo

20

- **Association:**

Association is a relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.



- **Unary Association:**

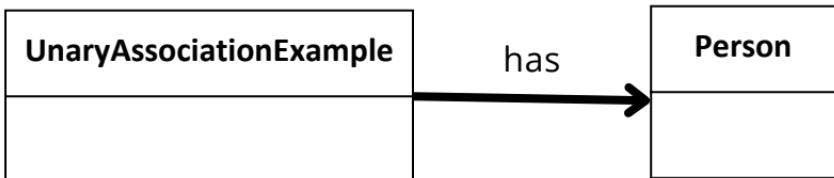
```
class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public void displayDetails() {  
        System.out.println("Name: " + name);  
        System.out.println("Age: " + age);  
    }  
}  
  
public class UnaryAssociationExample {  
    public static void main(String[] args) {  
        Person person = new Person("John Doe", 30);  
        person.displayDetails();  
    }  
}
```

Output:

Name: John Doe
Age: 30

Here,

Class UnaryAssociationExample has Class Person;
But Class Person does not have Class
UnaryAssociationExample



- Binary Association:**

```
class Car {  
    private String brand;  
    private String model;  
  
    public Car(String brand, String model) {  
        this.brand = brand;  
        this.model = model;  
    }  
  
    public String getBrand() {  
        return brand;  
    }  
  
    public String getModel() {  
        return model;  
    }  
}  
  
class Owner {  
    private String name;  
    private Car car;  
  
    public Owner(String name, Car car) {  
        this.name = name;  
        this.car = car;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public Car getCar() {  
    return car;  
}  
}  
  
public class BinaryAssociationExample {  
    public static void main(String[] args) {  
  
        Car car = new Car("Toyota", "Corolla");  
        Car car2 = new Car("Maruthi", "Suzuki");  
  
        Owner owner = new Owner("John Doe", car);  
        Owner owner2 = new Owner("Hamim", car2);  
  
        System.out.println("Car brand: "+car.getBrand());  
        System.out.println("Owner: " + owner.getName());  
        System.out.println("Car: " +  
owner.getCar().getBrand() + " " +  
owner.getCar().getModel());  
  
        System.out.println();  
  
        System.out.println("Car brand: "+car2.getBrand());  
        System.out.println("Owner: " + owner2.getName());  
        System.out.println("Car: " +  
owner2.getCar().getBrand() + " " +  
owner2.getCar().getModel());  
  
    }  
}
```

Output:

Car brand: Toyota

Owner: John Doe

Car: Toyota Corolla

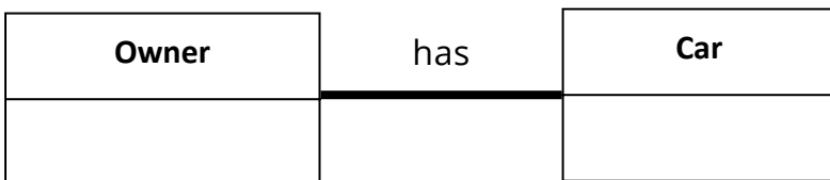
Car brand: Maruthi

Owner: Hamim

Car: Maruthi Suzuki

Here,

Both Classes know about each other.



- **What is the output of the following Java program fragment:**

```
class Department {  
    private String name;  
  
    public Department(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}  
  
class Employee {  
    private String name;  
    private Department department;  
  
    public Employee(String name, Department  
department) {  
        this.name = name;  
        this.department = department;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public Department getDepartment() {  
    return department;  
}  
}  
  
public class AssociationExample {  
    public static void main(String[] args) {  
        Department department = new  
        Department("Sales");  
        Employee employee = new Employee("John  
        Doe", department);  
  
        System.out.println("Employee name: " +  
        employee.getName());  
        System.out.println("Department name: " +  
        employee.getDepartment().getName());  
    }  
}
```

Output:

Employee name: John Doe
Department name: Sales

- **What is the output of the following Java program fragment:**

```
class Name {  
    String FirrstName;  
    String MiddleName;  
    String LastName;  
  
    public Name(String fname, String mname, String  
    lname){  
        this.FirrstName = fname;  
        this.MiddleName = mname;  
        this.LastName = lname;  
    }  
}  
  
public class EmployeeExam{  
    int age;  
    Name n;  
    public void display(int age, Name n){  
        System.out.println("Age: "+age);  
        System.out.println("Full Name: "+n.FirrstName+  
        " "+n.MiddleName+" "+n.LastName);  
    }  
    public static void main(String[] args) {  
        EmployeeExam e = new EmployeeExam();  
        Name m = new Name("Muhammed", "Hamim",  
        "Talukder");  
  
        e.display(23, m);  
    }  
}
```

Output:

Age: 23

Full Name: Muhammed Hamim Talukder

- What is the output of the following Java program fragment:

Professor views student's profile(scenario)

```
import java.util.Scanner;

class Student{
    String name;
    String regisNum;
    String phone;

    public Student(String name, String regisNum,
String phone){
        this.name = name;
        this.regisNum = regisNum;
        this.phone = phone;
    }

    public String getInfo(){
        return (name+" "+regisNum+" "+phone);
    }
}

class Professor {

    public void viewProfile(Student s){
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the registration number:");
        String regNum = input.nextLine();
    }
}
```

```
if(regNum.equals(s.regisNum)){
System.out.println(s.getInfo());
}
}
}

public class Test {
public static void main(String[] args) {
Student s1 = new Student("Hamim", "98954EJ",
"01731767273");

Professor Anisul = new Professor();
Anisul.viewProfile(s1);
}
}
```

Output:

Enter the registration number: 98954EJ
Hamim 98954EJ 01731767273

- **What is the output of the following Java program fragment:**

```
import java.io.*;
import java.util.*;

class Bank {

    private String name;

    private Set<Employee> employees;
    Bank(String name)
    {
        this.name = name;
    }

    public String getBankName()
    {
        return this.name;
    }

    public void setEmployees(Set<Employee> employees)
    {
        this.employees = employees;
    }

    public Set<Employee>
    getEmployees(Set<Employee> employees)
    {
        return this.employees;
    }
}
```

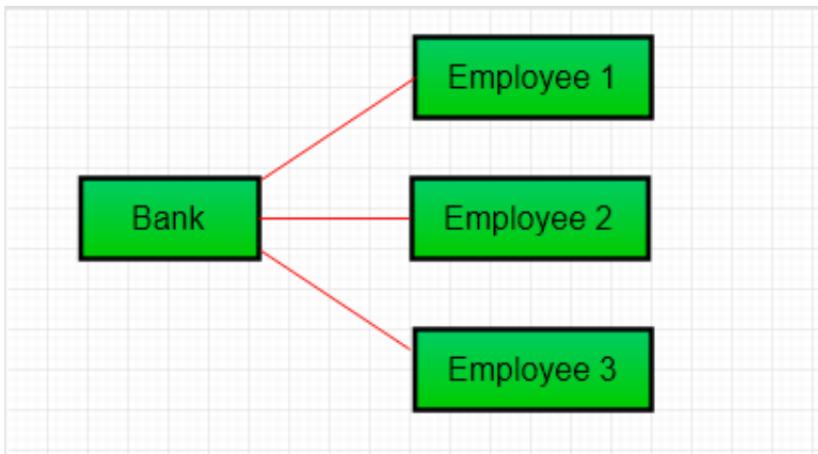
```
class Employee {  
    private String name;  
    Employee(String name)  
    {  
        this.name = name;  
    }  
  
    public String getEmployeeName()  
    {  
        return this.name;  
    }  
}  
  
public class Test {  
  
    public static void main(String[] args)  
    {  
  
        Bank bank = new Bank("ICICI");  
        Employee emp = new Employee("Ridhi");  
  
        Set<Employee> employees = new HashSet<>();  
        employees.add(emp);  
  
        bank.setEmployees(employees);  
  
        System.out.println(emp.getEmployeeName()  
            + " belongs to bank "  
            + bank.getBankName());  
    }  
}
```

Output:

Ridhi belongs to bank ICICI

Here,

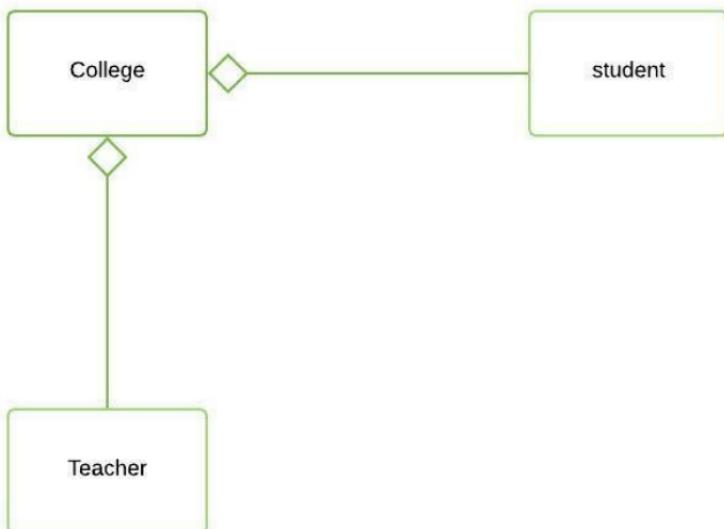
Output Explanation: In the above example, two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.



- **Aggregation:**

It is a special form of Association where:

- It represents Has-A's relationship.
- It is a unidirectional association i.e. a one-way relationship. For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, both entries can survive individually which means ending one entity will not affect the other entity.



- **What is the output of the following Java program fragment:**

```
class A {  
    private B b;  
  
    public A(B b) {  
        this.b = b;  
    }  
  
    public void performAction() {  
        b.doSomething();  
    }  
}  
  
class B {  
    public void doSomething() {  
        System.out.println("Class B is doing something.");  
    }  
}  
  
public class AggregationExample {  
    public static void main(String[] args) {  
        B b = new B();  
        A a = new A(b);  
        a.performAction();  
    }  
}
```

Output:

Class B is doing something.

Here,

In this updated example, Class A has a reference to an instance of Class B, which is passed through the constructor. This represents an aggregation relationship, where Class A aggregates Class B.

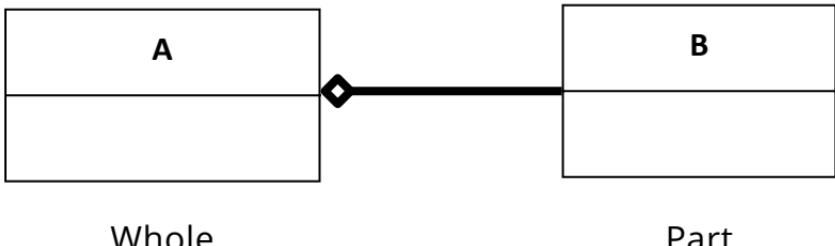
In the main() method, we first create an object of Class B named b. Then, we create an object of Class A named a, passing the b object as a parameter to the constructor. The a object now has a reference to the b object.

When we call the performAction() method on the a object, it internally invokes the doSomething() method of the associated b object.

In this example, Class A aggregates Class B, meaning that Class A contains a reference to an instance of Class B, but Class B can exist independently. The lifetime of the b object is not controlled by the a object, and the b object can be used by other parts of the program as well.

Here,

- Class B is a part of Class A.
- Both Classes are independent. Part class can exist without Whole Class.



- Here's an example that demonstrates aggregation:

```
class University {  
    private String name;  
    private List<Student> students;  
  
    public University(String name) {  
        this.name = name;  
        students = new ArrayList<>();  
    }  
  
    public void addStudent(Student student) {  
        students.add(student);  
    }  
  
    public void removeStudent(Student student) {  
        students.remove(student);  
    }  
  
    public List<Student> getStudents() {  
        return students;  
    }  
}  
  
class Student {  
    private String name;  
    private int age;  
  
    public Student(String name, int age) {  
        this.name = name;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class AggregationExample {
    public static void main(String[] args) {
        University university = new University("ABC
University");

        Student student1 = new Student("John Doe", 20);
        Student student2 = new Student("Jane Smith", 22);

        university.addStudent(student1);
        university.addStudent(student2);

        List<Student> students = university.getStudents();
        for (Student student : students) {
            System.out.println("Student Name: " +
student.getName());
            System.out.println("Student Age: " +
student.getAge());
        }
    }
}
```

Output:

Student Name: John Doe
Student Age: 20
Student Name: Jane Smith
Student Age: 22

In this example, we have two classes: University and Student. The University class represents the whole, and the Student class represents the part.

The University class has an aggregation relationship with the Student class. It contains a list of students as its instance variable students, indicating that it aggregates multiple instances of the Student class.

In the University class, we have methods to add and remove students from the list, as well as a getter method to retrieve the list of students.

The Student class represents individual students. It has instance variables for name and age, along with getter methods to access these values.

In the main() method, we create an instance of the University class named university and two instances of the Student class named student1 and student2. We then add these students to the university using the addStudent() method.

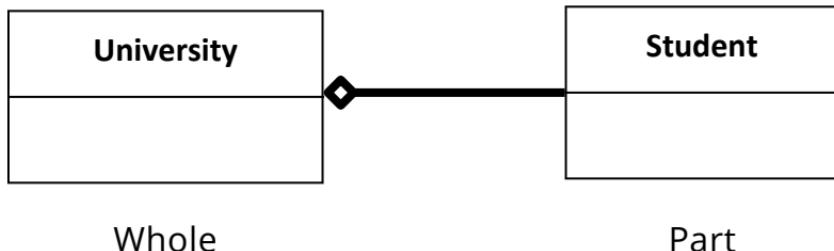
We retrieve the list of students using the `getStudents()` method and iterate over them to print their names and ages.

This example demonstrates an aggregation relationship where the `University` class aggregates instances of the `Student` class. The lifetime of the students is not managed by the `University` class, and they can exist independently.

In this example, the `Student` objects can exist outside the context of the `University` object, and adding or removing a student from the `University` object does not affect the existence of the `Student` object.

Here,

- Class `Student` is a part of Class `University`.
- Both Classes are independent. Part class can exist without Whole Class.



- **What is the output of the following Java program fragment:**

```
class Engine {  
    private String type;  
  
    public Engine(String type) {  
        this.type = type;  
    }  
  
    public String getType() {  
        return type;  
    }  
}  
  
class Car {  
    private String model;  
    private Engine engine;  
  
    public Car(String model, Engine engine) {  
        this.model = model;  
        this.engine = engine;  
    }  
  
    public String getModel() {  
        return model;  
    }  
}
```

```
public Engine getEngine() {  
    return engine;  
}  
}  
  
public class AggregationExample {  
    public static void main(String[] args) {  
        Engine engine = new Engine("V8");  
        Car car = new Car("Sports Car", engine);  
  
        System.out.println("Car Model: " +  
car.getModel());  
        System.out.println("Engine Type: " +  
car.getEngine().getType());  
    }  
}
```

Output:

Car Model: Sports Car
Engine Type: V8

- **What is the output of the following Java program fragment:**

```
import java.io.*;
import java.util.*;

class Student {

    String name;
    int id;
    String dept;

    Student(String name, int id, String dept)
    {

        this.name = name;
        this.id = id;
        this.dept = dept;
    }
}

class Department {
    String name;
    private List<Student> students;
    Department(String name, List<Student> students)
    {
        this.name = name;
        this.students = students;
    }
}
```

```
public List<Student> getStudents()
{
    return students;
}

}

class Institute {

String instituteName;
private List<Department> departments;

Institute(String instituteName,List<Department>
departments)
{
    this.instituteName = instituteName;
    this.departments = departments;
}

public int getTotalStudentsInInstitute()
{
    int noOfStudents = 0;
    List<Student> students;

    for (Department dept : departments) {
        students = dept.getStudents();

        for (Student s : students) {
            noOfStudents++;
        }
    }
}
```

```
    return noOfStudents;  
}  
}
```

```
public class Test {
```

```
    public static void main(String[] args)  
{
```

```
        Student s1 = new Student("Mia", 1, "CSE");
```

```
        Student s2 = new Student("Priya", 2, "CSE");
```

```
        Student s3 = new Student("John", 1, "EE");
```

```
        Student s4 = new Student("Rahul", 2, "EE");
```

```
        List<Student> cse_students = new ArrayList<Student>()
```

```
        cse_students.add(s1);
```

```
        cse_students.add(s2);
```

```
        List<Student> ee_students  
        = new ArrayList<Student>();
```

```
        ee_students.add(s3);
```

```
        ee_students.add(s4);
```

```
        Department CSE = new Department("CSE",  
        cse_students);
```

```
        Department EE = new Department("EE", ee_students);
```

```
        List<Department> departments = new  
        ArrayList<Department>();
```

```
departments.add(CSE);
departments.add(EE);

Institute institute = new Institute("BITS",
departments);

System.out.print("Total students in institute: ");

System.out.print(institute.getTotalStudentsInInstitute())
;
}
}
```

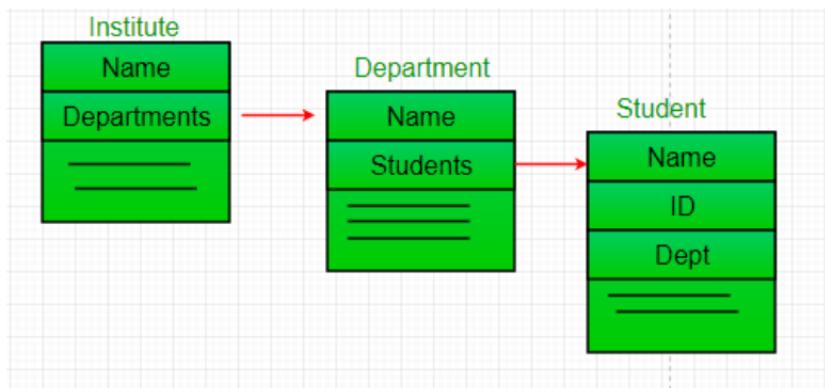
Output:

Total students in institute: 4

Here,

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make an Institute class that has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of the Student class means it is associated with the Student class through its Object(s).

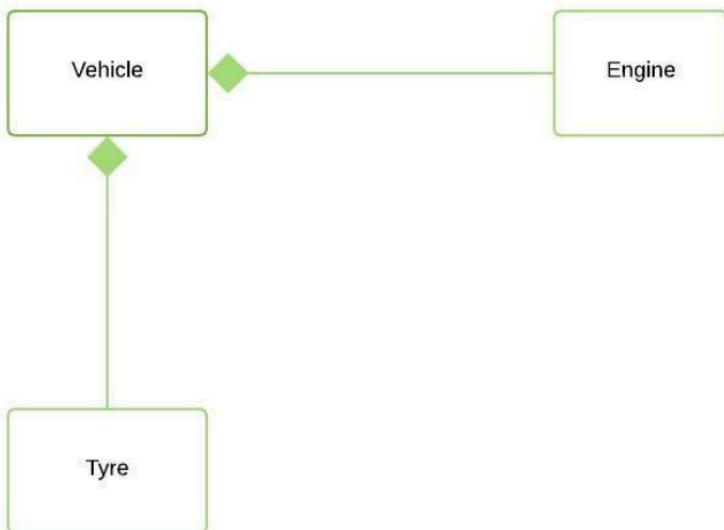
It represents a Has-A relationship. In the above example: Student Has-A name. Student Has-A ID. Student Has-A Dept. Department Has-A Students as depicted from the below media.



- **Composition:**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents part-of relationship.
- In composition, both entities are dependent on each other.
- When there is a composition between two entities, the composed object cannot exist without the other entity.



- **What is the output of the following Java program fragment:**

```
class A {  
    private B b;  
  
    public A() {  
        b = new B();  
    }  
  
    public void performAction() {  
        b.doSomething();  
    }  
  
    public void dispose() {  
        b = null;  
    }  
}  
  
class B {  
    public void doSomething() {  
        System.out.println("Class B is doing something.");  
    }  
}  
  
public class CompositionExample {  
    public static void main(String[] args) {  
        A a = new A();  
        a.performAction();  
  
        // After using A and B, dispose of A  
        a.dispose();  
    }  
}
```

Output:

Class B is doing something.

Here,

In this example, Class A creates an instance of Class B and manages its lifecycle. The existence of Class B is tied to Class A, as it is created in the constructor of Class A and released when needed.

Class A has a private instance variable b of type B, indicating that it has a composition relationship with Class B. In the constructor of Class A, a new instance of Class B is created and assigned to b.

The performAction() method in Class A calls the doSomething() method of Class B. Class A is responsible for utilizing the functionality provided by Class B.

Additionally, Class A provides a dispose() method that sets b to null, indicating the release of the associated Class B object.

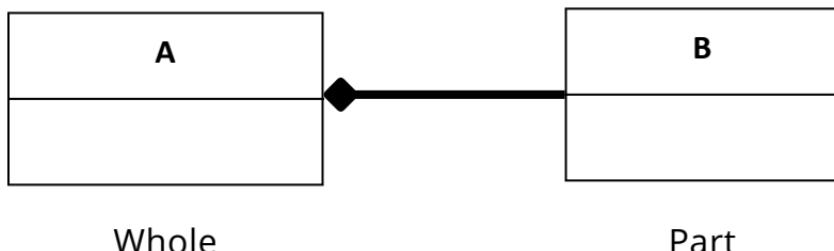
In the main() method, we create an instance of Class A named a. We can call performAction() to invoke the method in Class B. After using Class A and Class B, we can dispose of Class A by calling the dispose() method.

This example demonstrates a composition relationship, where the existence of Class B depends on Class A. When Class A is disposed, Class B is also released.

Please note that in this example, the dispose() method is a simplified representation to illustrate the release of Class B. In practice, the lifecycle management may involve more complex operations depending on the specific requirements of your application.

Here,

- Class B is a part of Class A.
- The existence of Part class depends on Whole Class.



- Here's an example that provides a more complex implementation of the dispose() method in the context of composition:

```
class A {  
    private B b;  
  
    public A() {  
        b = new B();  
    }  
  
    public void performAction() {  
        b.doSomething();  
    }  
  
    public void dispose() {  
        if (b != null) {  
            b.cleanup();  
            b = null;  
        }  
    }  
}  
  
class B {  
    private C c;  
  
    public B() {  
        c = new C();  
    }  
}
```

```
public void doSomething() {
    System.out.println("Class B is doing something.");
    c.performTask();
}

public void cleanup() {
    c.releaseResources();
}
}

class C {
    public void performTask() {
        System.out.println("Class C is performing a task.");
    }

    public void releaseResources() {
        System.out.println("Class C is releasing
resources.");
    }
}

public class CompositionExample {
    public static void main(String[] args) {
        A a = new A();
        a.performAction();

        // After using A and its associated objects, dispose
        // of A
        a.dispose();
    }
}
```

Output:

Class B is doing something.

Class C is performing a task.

Class C is releasing resources.

Here,

In this updated example, Class B contains an instance of Class C, and Class A has a composition relationship with both Class B and Class C.

In Class A, the dispose() method is enhanced to handle the cleanup process. When dispose() is called, it checks if b is not null, indicating that Class B exists. If so, it invokes the cleanup() method of Class B to release any resources it holds. Then, it sets b to null to indicate that Class B has been disposed of.

Class B also contains a cleanup() method, which in this case, delegates the resource release task to Class C by calling its releaseResources() method.

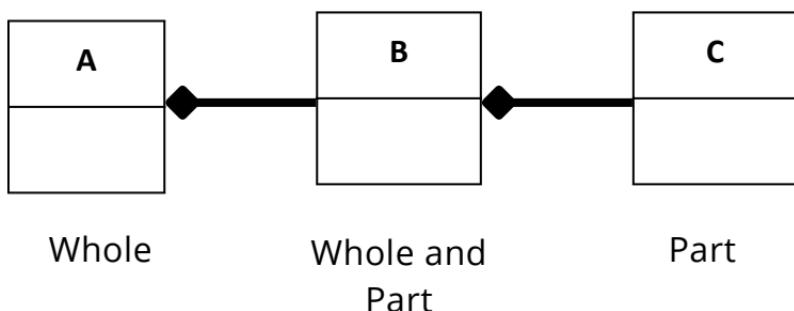
Class C represents another component in the composition hierarchy. It has a performTask() method to perform a specific task and a releaseResources() method to release any resources it holds.

In the main() method, we create an instance of Class A named a, call the performAction() method to invoke the actions of Class B and Class C, and finally, call the dispose() method to release resources and dispose of the objects.

In this example, the dispose() method demonstrates a more complex cleanup process where the resources held by Class B and Class C are properly released. The composition relationship ensures that the existence and lifecycle of Class B and Class C are tied to Class A, providing encapsulation and management of the associated objects.

Here,

- Class C is a part of Class B. Class B is a part of Class A.
- The existence of Part class depends on Whole Class.



- **What is the output of the following Java program fragment:**

```
class Engine {  
    private String type;  
  
    public Engine(String type) {  
        this.type = type;  
    }  
  
    public String getType() {  
        return type;  
    }  
}  
  
class Car {  
    private String model;  
    private Engine engine;  
  
    public Car(String model, String engineType) {  
        this.model = model;  
        this.engine = new Engine(engineType);  
    }  
  
    public String getModel() {  
        return model;  
    }  
}
```

```
public Engine getEngine() {  
    return engine;  
}  
}  
  
public class CompositionExample {  
    public static void main(String[] args) {  
        Car car = new Car("Sports Car", "V8");  
  
        System.out.println("Car Model: " + car.getModel());  
        System.out.println("Engine Type: " +  
car.getEngine().getType());  
    }  
}
```

Output:

Car Model: Sports Car

Engine Type: V8

- **What is the output of the following Java program fragment:**

```
class Book {  
    private String title;  
    private Author author;  
  
    public Book(String title, String authorName) {  
        this.title = title;  
        this.author = new Author(authorName);  
    }  
  
    public String getTitle() {  
        return title;  
    }  
  
    public Author getAuthor() {  
        return author;  
    }  
}  
  
class Author {  
    private String name;  
  
    public Author(String name) {  
        this.name = name;  
    }  
}
```

```
public String getName() {  
    return name;  
}  
}  
  
public class CompositionExample {  
    public static void main(String[] args) {  
        Book book = new Book("The Great Gatsby", "F.  
Scott Fitzgerald");  
  
        System.out.println("Book Title: " +  
book.getTitle());  
        System.out.println("Author Name: " +  
book.getAuthor().getName());  
    }  
}
```

Output:

Book Title: The Great Gatsby
Author Name: F. Scott Fitzgerald

- **What is the output of the following Java program fragment:**

```
import java.io.*;
import java.util.*;

class Book {
    public String title;
    public String author;

    Book(String title, String author)
    {
        this.title = title;
        this.author = author;
    }
}

class Library {

    private final List<Book> books;

    Library(List<Book> books)
    {
        this.books = books;
    }

    public List<Book> getTotalBooksInLibrary()
    {
        return books;
    }
}
```

```
public class Test {  
  
    public static void main(String[] args)  
    {  
  
        Book b1  
        = new Book("Effective Java", "Joshua Bloch");  
        Book b2  
        = new Book("Thinking in Java", "Bruce Eckel");  
        Book b3 = new Book("Java: The Complete Reference",  
                           "Herbert Schildt");  
  
        List<Book> books = new ArrayList<Book>();  
  
        books.add(b1);  
        books.add(b2);  
        books.add(b3);  
  
        Library library = new Library(books);  
  
        List<Book> bks = library.getTotalBooksInLibrary();  
  
        for (Book bk : bks) {  
  
            System.out.println("Title : " + bk.title  
                               + " and "  
                               + " Author : " + bk.author);  
        }  
    }  
}
```

Output:

Title : Effective Java and Author : Joshua Bloch

Title : Thinking in Java and Author : Bruce Eckel

Title : Java: The Complete Reference and Author : Herbert Schildt

Here,

In the above example, a library can have no. of books on the same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. books can not exist without libraries. That's why it is composition. Book is Part-of Library.

- **Aggregation vs Composition:**

1. Dependency: Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don't exist separate to a Human
2. Type of Relationship: Aggregation relation is "has-a" and composition is "part-of" relation.
3. Type of association: Composition is a strong Association whereas Aggregation is a weak Association.

- **What is the output of the following Java program fragment:**

```
import java.io.*;  
  
class Engine {  
  
    public void work()  
    {  
  
        System.out.println(  
            "Engine of car has been started ");  
    }  
}  
  
final class Car {  
  
    private final Engine engine;  
  
    Car(Engine engine)  
    {  
        this.engine = engine;  
    }  
  
    public void move()  
    {  
  
        engine.work();
```

```
        System.out.println("Car is moving ");
    }
}
}

public class Test {
    public static void main(String[] args)
    {
        Engine engine = new Engine();
        Car car = new Car(engine);
        car.move();
    }
}
```

Output:

Engine of car has been started
Car is moving

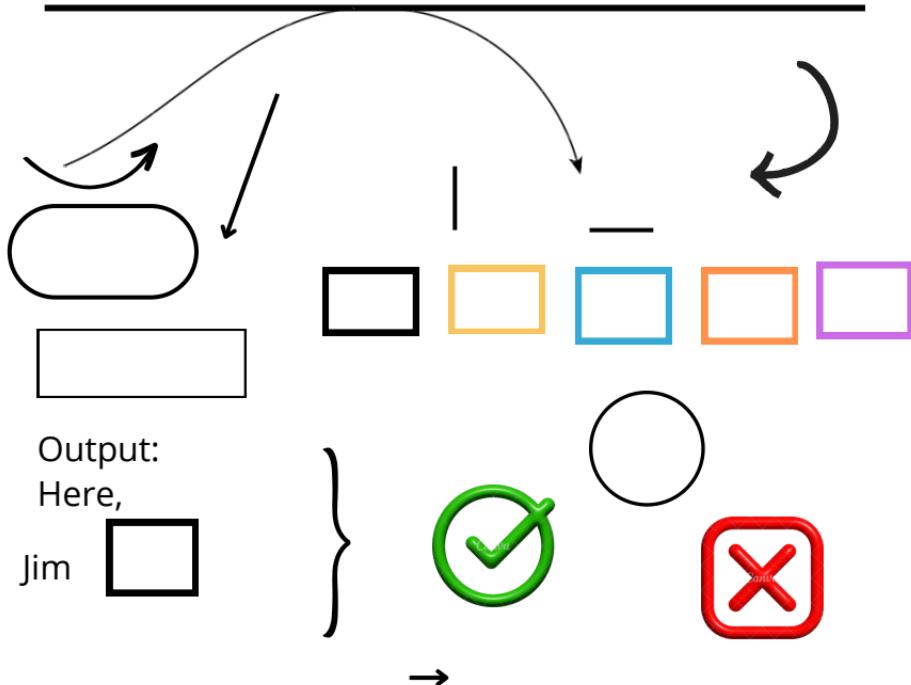
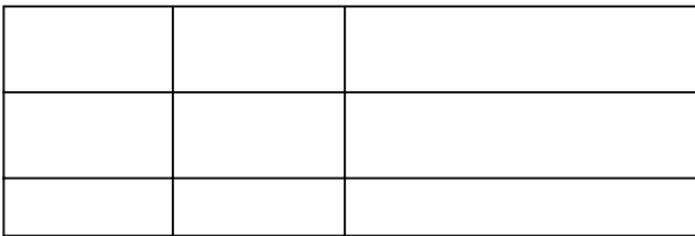
Here,

In case of aggregation, the Car also performs its functions through an Engine. but the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That's why we make The Engine type field non-final.

4. ArrayList

Operator Precedence and Associativity:

- Make a program that will show String palindrome.



- What is the output of the following Java program fragment:

Operator



JAVA
(SECOND PART)
T.I.M. HA-MEAM

ABC
PROKASHONI