

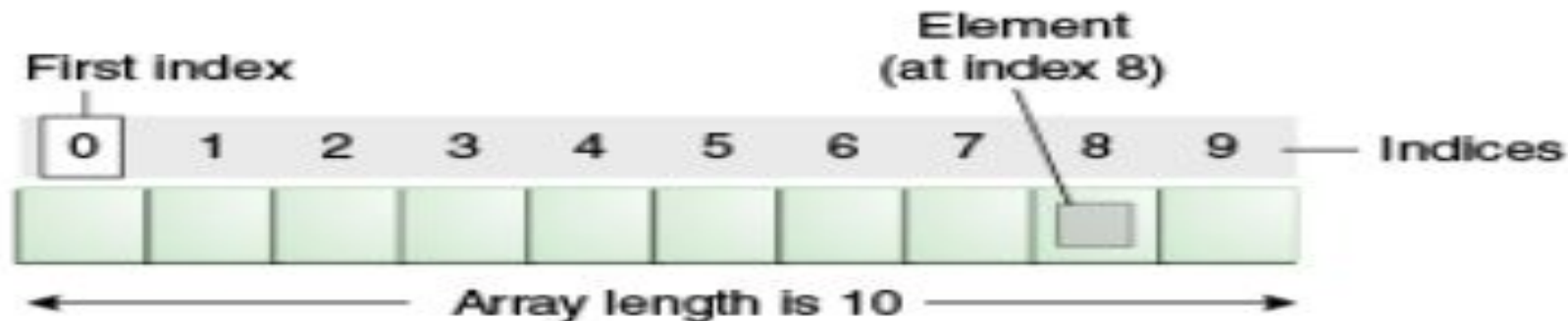
Unit-02

Object and Classes; Constructor; Data types; Variables; Modifier and Operators - Structural Programming Paradigm: Branching, Iteration, Decision making, and Arrays - Procedural Programming Paradigm: Characteristics; Function Definition; Function Declaration and Calling; Function Arguments - Object-Oriented Programming Paradigm: Abstraction; Encapsulation; Inheritance; Polymorphism; Overriding - Interfaces: Declaring, Implementing; Extended and Tagging - Package: Package Creation.

ARRAY IN JAVA

Array

- An array is a collection of similar type of elements which has contiguous memory.
- **Java array** is an object which contains elements of a similar data type.
- Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at 0 and so on.



Syntax to Declare an Array in Java

Single Dimensional Array in Java

- `dataType[] arr; (or)`
- `dataType []arr; (or)`
- `dataType arr[];`

Types of Array in javaSyntax

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Declaring Arrays as Objects

```
int myArray[];
```

declares *myArray* to be an array of integers

```
myArray = new int[8];
```

sets up 8 integer-sized spaces in memory, labelled *myArray[0]* to *myArray[7]*

```
int myArray[] = new int[8];
```

combines the two statements in one line

Assigning Values

- refer to the array elements by index to store values in them.

`myArray[0] = 3;`

`myArray[1] = 6;`

`myArray[2] = 3; ...`

- can create and initialise in one step:

`int myArray[] = {3, 6, 3, 1, 6, 3, 4, 1};`

Iterating Through Arrays

- *for* loops are useful when dealing with arrays:

```
for (int i = 0; i < myArray.length; i++) {  
    myArray[i] = getsomevalue();  
}
```


Arrays of Objects

- So far we have looked at an array of primitive types.
 - integers
 - could also use doubles, floats, characters...
- Often want to have an array of objects
 - Students, Books, Loans
- Need to follow 3 steps.

an array of objects stores and manipulates multiple instances of a particular class.

//Java Program to illustrate how to declare, instantiate
initialize

//and traverse the Java array.

```
1. class Testarray{  
2. public static void main(String args[]){  
3. int a[]=new int[5];//declaration and instantiation  
4. a[0]=10;//initialization  
5. a[1]=20;  
6. a[2]=70;  
7. a[3]=40;  
8. a[4]=50;  
9. //traversing array  
10. for(int i=0;i<a.length;i++)//length is the property of array  
11. System.out.println(a[i]);  
12. }}
```

Example:2

1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. **class** Testarray1{
4. **public static void** main(String args[]){
5. **int** a[]={33,3,4,5}; //declaration, instantiation and initialization
6. //printing array
7. **for**(**int** i=0;i<a.length;i++) //length is the property of array
8. System.out.println(a[i]);
9. }}

For-each Loop for Java Array

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one.

It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
1.for(data_type variable:array){  
2.//body of the loop  
3.}
```

Example3

1. //Java Program to print the array elements using for-each loop
2. **class** Testarray1{
3. **public static void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. //printing array using for-each loop
6. **for**(**int** i:arr)
7. System.out.println(i);
8. }}

```
1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. class Testarray2 {
4. //creating a method which receives an array as a parameter
5. static void min(int arr[]) {
6. int min=arr[0];
7. for(int i=1;i<arr.length;i++)
8. if(min>arr[i])
9.   min=arr[i];
10.
11. System.out.println(min);
12. }
13.
14. public static void main(String args[]) {
15. int a[]={33,3,4,5}; //declaring and initializing an array
16. min(a); //passing array to method
17. }}
```

JAVA USER INPUT

Java User Input

- The Scanner class is used to get user input, and it is found in the `java.util` package.
- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.


```
import java.util.Scanner; // Import the Scanner class
```

```
class userinput
```

```
{
```

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

```
    {
```

```
        Scanner myObj = new Scanner(System.in); // Create a Scanner object
```

```
        System.out.println("Enter username");
```

```
        String userName = myObj.nextLine(); // Read user input
```

```
        System.out.println("Username is: " + userName); // Output user input
```

```
    }
```

```
}
```

Tamil Academy

In Java, the `public static void main(String[] args)` method is the entry point of any Java application. Here's a breakdown of why each part is used:

public: This keyword means that the method is **accessible from anywhere**, which is necessary since the JVM (Java Virtual Machine) needs to access this method to start the program.

static: This allows the **method to be called without creating an instance of the class**. The JVM can directly invoke this method without needing to instantiate the class.

void: This means that the method **does not return any value**. The main method serves to start the program, not to return data.

main: This is the name of the method that the JVM looks for when running a Java application. It's a predefined name and must be used exactly as it is.

String[] args: This is an array of String objects, and it allows the program to accept arguments from the command line. Each element in the array corresponds to an argument passed when the program is run.

So, `public static void main(String[] args)` is essential in Java because it serves as the standard entry point for the application, allowing the JVM to start the program and optionally pass command-line arguments.

CA Command Prompt

Microsoft Windows [Version 10.0.19043.1826]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SAI>cd\

C:\>cd jdk-8\bin

C:\jdk-8\bin>javac userinput.java

C:\jdk-8\bin>java userinput

Enter username

hemalatha

Username is: hemalatha

C:\jdk-8\bin>

Input Types

Method	Description
<code>nextBoolean()</code>	Reads a <code>boolean</code> value from the user
<code>nextByte()</code>	Reads a <code>byte</code> value from the user
<code>nextDouble()</code>	Reads a <code>double</code> value from the user
<code>nextFloat()</code>	Reads a <code>float</code> value from the user
<code>nextInt()</code>	Reads a <code>int</code> value from the user
<code>nextLine()</code>	Reads a <code>String</code> value from the user
<code>nextLong()</code>	Reads a <code>long</code> value from the user
<code>nextShort()</code>	Reads a <code>short</code> value from the user

```
import java.util.Scanner;

class userinput1
{
    public static void main(String[] args)
    {
        Scanner myObj = new Scanner(System.in);

        System.out.println("Enter name, age and salary:");

        // String input
        String name = myObj.nextLine();

        // Numerical input
        int age = myObj.nextInt();
        double salary = myObj.nextDouble();

        // Output input by user
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```

Microsoft Windows [Version 10.0.19043.1826]
 (c) Microsoft Corporation. All rights reserved.

C:\Users\SAI>cd\

C:\>cd jdk-8\bin

C:\jdk-8\bin>javac userinput.java

C:\jdk-8\bin>java userinput

Enter username

hemalatha

Username is: hemalatha

C:\jdk-8\bin>javac userinput1.java

C:\jdk-8\bin>java userinput1

Enter name, age and salary:

hemalatha

34

18000

Name: hemalatha

Age: 34

Salary: 18000.0

C:\jdk-8\bin>_


```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
Your first argument is: hema
```

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
Your first argument is: hema
```

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
    at CommandLineExample.main(CommandLineExample.java:5)
```

```
C:\jdk-8\bin>javac A.java
```

```
C:\jdk-8\bin>java A java c c++ python
java
c
c++
python
```

```
C:\jdk-8\bin>java A java c c++ python php oracle
java
c
c++
python
php
oracle
```

```
C:\jdk-8\bin>
```

Command Line Arguments

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.


```
class CommandLineExample
{
    public static void main(String args[])
    {
        System.out.println("Your first argument is: "+args[0]);
    }
}
```

1.compile by > javac CommandLineExample.java

2.run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo



Command Prompt

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
```

```
Your first argument is: hema
```

```
C:\jdk-8\bin>
```

Tamil Academ

```
class CommandLineExample
{
    public static void main(String ar[])
    {
        System.out.println("Your first argument is: "+ar[0]);
    }
}
```



Command Prompt

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema  
Your first argument is: hema
```

```
C:\jdk-8\bin>javac CommandLineExample.java Academy
```

```
C:\jdk-8\bin>java CommandLineExample hema  
Your first argument is: hema
```

```
C:\jdk-8\bin>
```

```
class CommandLineExample  
{  
public static void main(String ar[])  
{  
System.out.println("Your first argument is: "+ar[1]);  
}  
}
```

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
```

```
Your first argument is: hema
```

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
```

```
Your first argument is: hema
```

```
C:\jdk-8\bin>javac CommandLineExample.java
```

```
C:\jdk-8\bin>java CommandLineExample hema
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1  
    at CommandLineExample.main(CommandLineExample.java:5)
```

```
C:\jdk-8\bin>_
```



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

for(int i=0;i<args.length;i++)
System.out.println(args[i]);

}
}
```

1.compile by > javac A.java

2.run by > java A sonoo jaiswal 1 3 abc

Output: sonoo jaiswal 1 3 abc

OOPs (Object-Oriented Programming System)

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Tamil Academy

- Class

- *Collection of objects* is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class doesn't consume any space.

Tamil Academy

- Object

- Any entity that has state and behavior is known as an object.
- An Object can be defined as an instance of a class.
- Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

- Encapsulation

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation.* For example, a capsule, it is wrapped with different medicines.

Tamil Academy



Capsule

- Inheritance

- *When one object acquires all the properties and behaviours of a parent object, it is known as inheritance. It provides code reusability.*

Small Example

- Polymorphism

- If *one task is performed in different ways*, it is known as polymorphism.



ACCESS MODIFIERS IN JAVA

Access Modifiers in Java

- Access modifiers 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

2. Private Access Modifier

- 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
 - private means “only visible within the enclosing class”.
 - protected means “only visible within the enclosing class and any subclasses”
- Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes
- In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try


```
class A{  
private int data=40;  
private void msg(){System.out.println("Hello java");}  
}
```

```
public class Simple{  
public static void main(String args[]){  
A obj=new A();  
System.out.println(obj.data);//Compile Time Error  
obj.msg();//Compile Time Error  
}  
}
```

```
class A{  
private int data=40;  
private void msg()  
{  
System.out.println("Hello java");  
}  
public static void main(String args[])  
{  
A obj=new A();  
System.out.println(obj.data);  
//Compile Time Error  
obj.msg();//Compile Time Error  
}  
}
```

```
// Private Modifier
package p1;
class A {
    private void display()
    {
        System.out.println("hello....");
    }
}
```

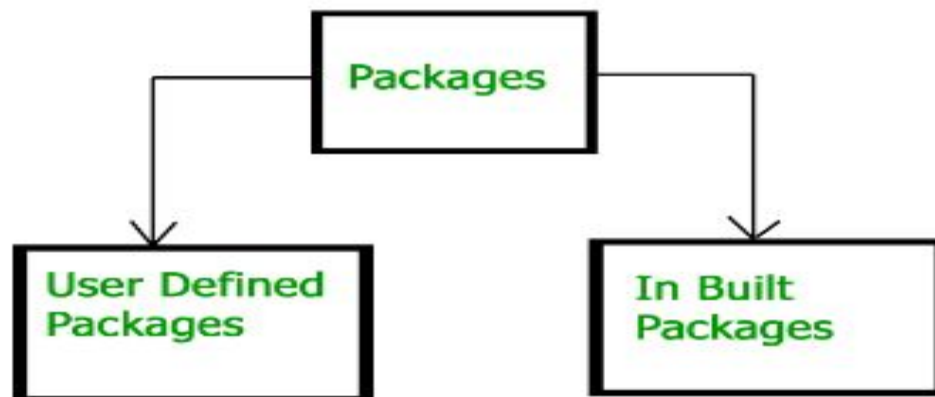
```
class B {
    public static void main(String args[])
    {
        A obj = new A();
        // Trying to access private method
        // of another class
        obj.display();
    }
}
```

**O/P - error: display() has private access in A
obj.display();**

A **package** is a container of a **group of related classes** where some of the classes are accessible are exposed and others are kept for internal purpose.

We can reuse existing classes from the packages as many time as we need it in our program.

```
// import the Vector class from util package.  
import java.util.vector;  
// import all the classes from util package  
import java.util.*;
```



Default Access Modifier

- The data members, classes, or methods that are not declared using any access modifiers .
- default access modifiers are accessible **only within the same package**.
- In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

```
// Java program to illustrate default  
modifier
```

```
package p1;
```

```
// Class Geek is having Default access  
modifier
```

```
class Geek
```

```
{
```

```
    void display()
```

```
    {
```

```
        System.out.println("Hello World!");
```

```
    }
```

```
}
```

O/p: Compile time error

```
// Java program to illustrate error while
```

```
// using class from different package with
```

```
// default modifier
```

```
package p2;
```

```
import p1.*;
```

```
// This class is having default access modifier
```

```
class GeekNew
```

```
{
```

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

```
    {
```

```
        // Accessing class Geek from package p1
```

```
        Geek obj = new Geek();
```

```
        obj.display();
```

```
    }
```

```
}
```

3. Protected Access Modifier

- 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.

```
// Java Program to Illustrate
// Protected Modifier
package p1;
```

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

Output:

GeeksforGeeks

```
// Java program to illustrate
// protected modifier
package p2;
```

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

// Class B is subclass of A
class B extends A {
    public static void main(String args[])
    {
        B obj = new B();
        obj.display();
    }
}
```

Public Access modifier

- 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.

	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

CLASS AND OBJECTS SAMPLE PROGRAM

- **What is a class in Java**

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
 - Fields
 - Methods
 - Constructors
 - Blocks
 - Nested class and interface

Syntax to declare a class:

- class <class_name>
 {
 field;
 method;
 }

new keyword in Java

- The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Main within Class

Object and Class Example: main within the class

1. //Defining a Student class.
2. **class** Student{
3. //defining fields
4. **int** id;//field or data member or instance variable
5. String name;
6. //creating main method inside the Student class
7. **public static void** main(String args[]){
8. //Creating an object or instance
9. Student s1=**new** Student();//creating an object of Student
0. //Printing values of the object
1. System.out.println(s1.id);//accessing member through reference variable
2. System.out.println(s1.name);
3. }

main outside the class

Object and Class Example: main outside the class

```
1. //Java Program to demonstrate having the main method in
2. //another class
3. //Creating Student class.
4. class Student{
5.     int id;
6.     String name;
7. }
8. //Creating another class TestStudent1 which contains the main metho
9. class TestStudent1{
10.    public static void main(String args[]){
11.        Student s1=new Student();
12.        System.out.println(s1.id);
13.        System.out.println(s1.name);
14.    }
15. }
```

3 Ways to initialize object

- There are 3 ways to initialize object in Java.

1.By reference variable

2.By method

3.By constructor

Initialization through reference

```
1. class Student{
2.   int id;
3.   String name;
4. }
5. class TestStudent2{
6.   public static void main(String args[]){
7.     Student s1=new Student();
8.     s1.id=101;
9.     s1.name="Sonoo";
10.    System.out.println(s1.id+" "+s1.name);//printing members with
      a white space
11.  }
12. }
```

Initialization through method

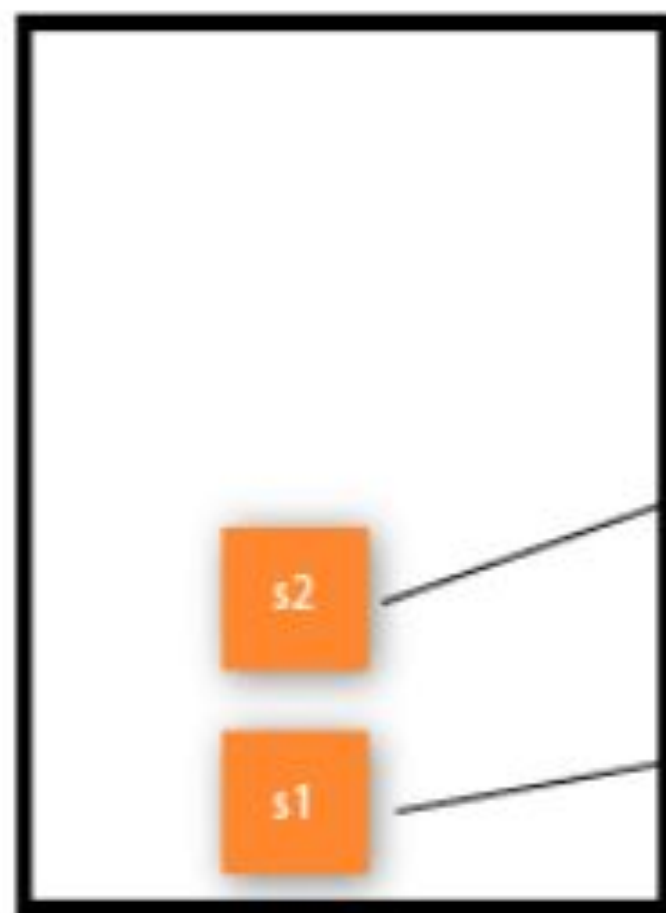
```
class Student
{
    int rollno;
    String name;
    void insertRecord(int r, String n)
    {
        rollno=r;
        name=n;
    }
    void displayInformation()
    {
        System.out.println(rollno+" "+name);
    }
}
```

```
class TestStudent4{
    public static void main(String args[])
    {
        Student s1=new Student();
        Student s2=new Student();
        s1.insertRecord(111,"Karan");
        s2.insertRecord(222,"Aryan");
        s1.displayInformation();
        s2.displayInformation();
    }
}
```

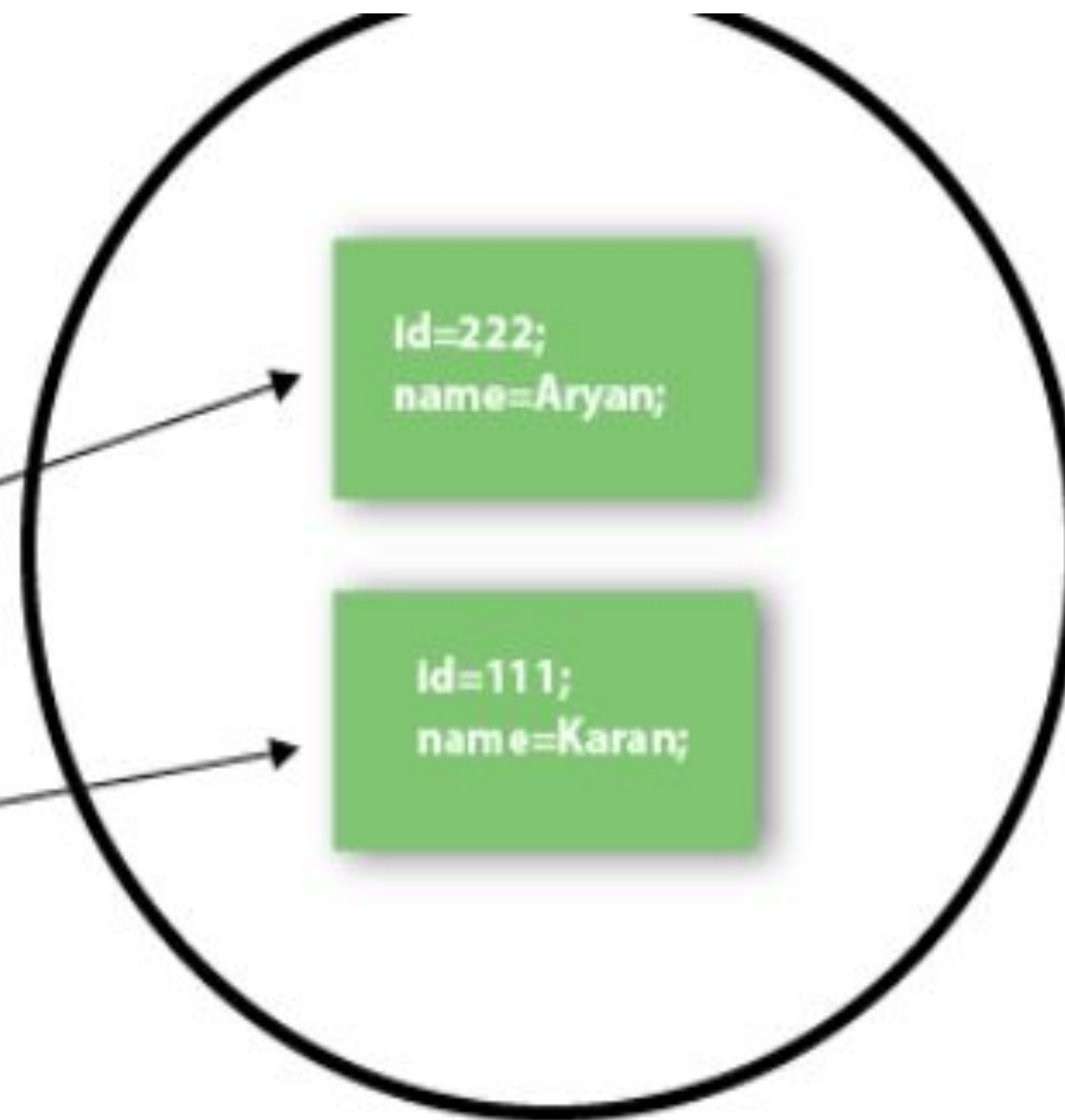
Compile by: javac TestStudent4.java

Run by: java TestStudent4

```
111 Karan
222 Aryan
```



Stack Memory



Heap Memory

Initialization through a constructor

```
class Rectangle{  
    int length;  
    int width;  
    void insert(int l, int w)  
    {  
        length=l;  
        width=w;  
    }  
    void calculateArea()  
    {  
        System.out.println(length*width);  
    }  
}
```

```
class TestRectangle1{  
    public static void main  
    (String args[])  
    {  
        Rectangle r1=new Rectangle();  
        Rectangle r2=new Rectangle();  
        r1.insert(11,5);  
        r2.insert(3,15);  
        r1.calculateArea();  
        r2.calculateArea();  
    }  
}
```

Constructors

- 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 constructor, memory for the object is allocated in the 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.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

Types of Constructor in JAva

- I. no-arg constructor
- II. parameterized constructor.
- III. Copy Constructor

Java Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
<class_name>()  
{  
}
```

It **will be invoked** at the time of object creation.

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

```
class Bike1{  
    //creating a default constructor  
    Bike1()  
    {   System.out.println("Bike is created");  
    }  
    //main method  
    public static void main(String args[])  
    {  
        //calling a default constructor  
        Bike1 b=new Bike1();  
    }  
}
```

Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.
- WHY: The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

```
class Student4{  
    int id;  
    String name;  
    //creating a parameterized constructor  
    Student4(int i,String n)  
    { id = i;  
      name = n;  
    }  
}
```

```
//method to display the values  
void display()  
{  
    System.out.println(id+" "+name);  
}  
public static void main(String args[])  
{  
    //creating objects and passing values  
    Student4 s1 = new Student4(111,"Karan");  
    Student4 s2 = new Student4(222,"Aryan");  
    //calling method to display the values of  
    object  
    s1.display();  
    s2.display();  
}  
}
```

Copy Constructor

- In Java, a copy constructor is a special type of constructor that **creates an object using another object of the same Java class**. It returns a duplicate copy of an existing object of the class.
- **Creating a Copy Constructor**
- Create a constructor that accepts an object of the same class as a parameter.

```
public class Fruits
{
    private double price;
    private String name;
    //copy constructor
    public Fruits(Fruits fruits)
    {
        //getters
    }
}
```

```

public class Fruit
{
private double fprice;
private String fname;
//constructor to initialize roll number and name of the student
Fruit(double fPrice, String fName)
{
    fprice = fPrice;
    fname = fName;
}
//creating a copy constructor
Fruit(Fruit fruit)
{
    System.out.println("\nAfter invoking the Copy Constructor:\n");
    fprice = fruit.fprice;
    fname = fruit.fname;
}

```

```

double showPrice()
{
return fprice;
}
//creating a method that returns the name of the fruit
String showName()
{
return fname;
}
//class to create student object and print roll number and name of the student
public static void main(String args[])
{
    Fruit f1 = new Fruit(399, "Ruby Roman Grapes");
    System.out.println("Name of the first fruit: " + f1.showName());

    System.out.println("Price of the first fruit: " + f1.showPrice());
    //passing the parameters to the copy constructor
    Fruit f2 = new Fruit(f1);
    System.out.println("Name of the second fruit: " + f2.showName());
    System.out.println("Price of the second fruit: " + f2.showPrice());
}

```

```

Name of the first fruit: Ruby Roman Grapes
Price of the first fruit: 399.0

After invoking the Copy Constructor:

```

o/p

```
Name of the first fruit: Ruby Roman Grapes
```

```
Price of the first fruit: 399.0
```

```
After invoking the Copy Constructor:
```

```
Name of the second fruit: Ruby Roman Grapes
```

```
Price of the second fruit: 399.0
```

Constructor Overloading in Java

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

//Java program to overload constructors

```
class Student5{
```

```
    int id;
```

```
    String name;
```

```
    int age;
```

```
    //creating two arg constructor
```

```
    Student5(int i,String n){
```

```
        id = i;
```

```
        name = n;
```

```
    }
```

```
    //creating three arg constructor
```

```
    Student5(int i,String n,int a){
```

```
        id = i;
```

```
        name = n;
```

```
        age=a;
```

```
    }
```

```
    void display(){
        System.out.println(id+" "+name+" "+age);}
    public static void main(String args[]){
        Student5 s1 = new Student5(111,"Karan");
        Student5 s2 = new
        Student5(222,"Aryan",25);
        s1.display();
        s2.display();
    }
}
```

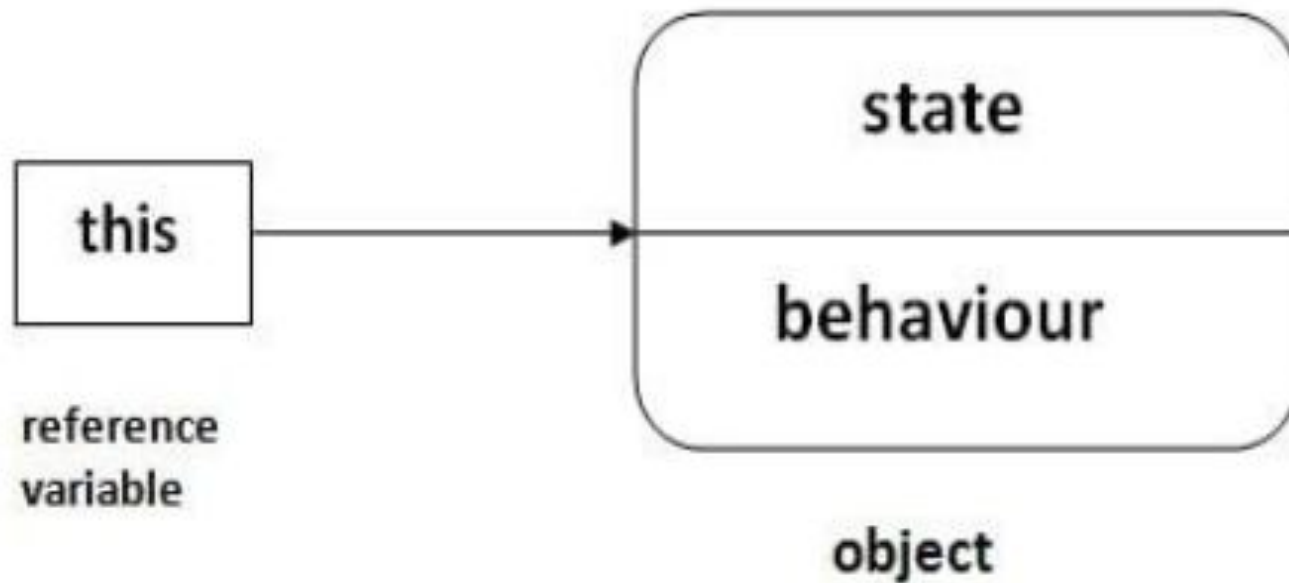
Output

```
java -cp /tmp/X1kyg7Ujte/Student5
111 Karan 0
222 Aryan 25
```


this keyword in Java

This keyword

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01

this can be used to refer current class instance variable.

02

this can be used to invoke current class method (implicit)

03

this() can be used to invoke current class Constructor.

04

this can be passed as an argument in the method call.

05

this can be passed as argument in the constructor call.

06

this can be used to return the current class instance from the method

1) this: to refer current class instance variable

- The this keyword can be used to refer **current class instance variable**. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
```

```
void display(){System.out.println(rollno+" "+
name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

Understanding the problem without this keyword

```
class Student{  
int rollno;  
String name;  
float fee;  
Student(int rollno,String name,float fee){  
this.rollno=rollno;  
this.name=name;  
this.fee=fee;  
}
```

```
void display()  
{System.out.println(rollno+" "+name+" "+fee);}  
}  
class TestThis2{  
public static void main(String args[]){  
Student s1=new Student(111,"ankit",5000f);  
  
Student s2=new Student(112,"sumit",6000f);  
  
s1.display();  
s2.display();  
}}
```

Output:

```
111 ankit 5000.0  
112 sumit 6000.0
```

Program where this keyword is not required

```
t{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display()
{System.out.println(rollno+" "+name+" "+fee);
}
}
```

```
class TestThis3
{
public static void main(String args[])
{
Student s1=new Student(111,"ankit",5000.0);
Student s2=new Student(112,"sumit",6000.0);
s1.display();
s2.display();
}
}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

WEEK2 TUTORIAL

1. Calculate and print the factorial of a given non-negative integer using a for loop.
2. Calculate and print the simple interest for a given principal, rate, and number of years using a for loop.
3. Print the multiplication table of a given number using While loop
4. Calculate the average of a series of numbers entered by the user using a while loop.
5. Find the sum of even numbers from 1 to a given number using a dowhile loop.
6. Continuously prompt the user to enter a positive number until they provide a valid input.

ACCESS MODIFIERS IN JAVA

Access Modifiers in Java

- Access modifiers 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

	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

```
class B{
private int data=40;
private void msg(){
System.out.println("Hello java");}
}
class First
{
public static void main(String args[]){
    B obj=new B();//Compile Time Error
    System.out.println(obj.data);//Compile Time Error
    obj.msg();//Compile Time Error
}
}
```

```
class B{
private int data=40;
private void msg(){System.out.println("Hello java");}
//}
//class First
//{
public static void main(String args[]){
    B obj=new B();//Compile Time Error
    System.out.println(obj.data);//Compile Time Error
    obj.msg();//Compile Time Error
}
}
```

STATIC Keyword in java

Java static keyword

- The static keyword in Java is used **for memory management mainly**.
- We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.
- The static can be:
 - Variable (also known as a class variable)
 - Method (also known as a class method)
 - Block
 - Nested class

Java static variable

- The static variable can be **used to refer to the common property of all objects** (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

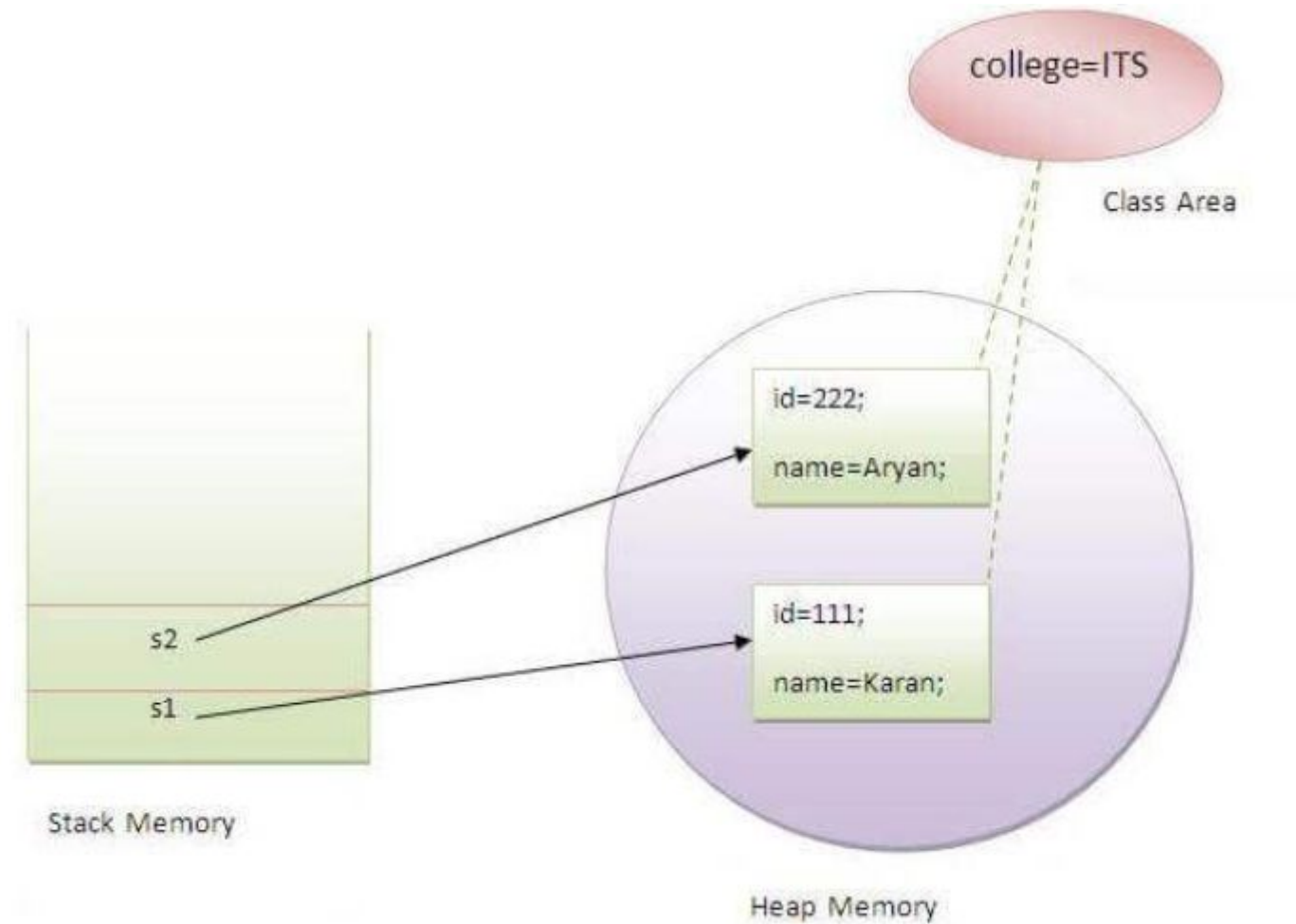
Java static variable

```
class Student{  
    int rollno;//instance variable  
    String name;  
    static String college ="ITS";//static  
variable  
    //constructor  
    Student(int r, String n){  
        rollno = r;  
        name = n;  
    }  
}
```

```
//method to display the values  
void display ()  
{  
    System.out.println(rollno+" "+name+" "+college);  
}  
//Test class to show the values of objects  
public class TestStaticVariable1{  
    public static void main(String args[]){  
        Student s1 = new Student(11,"MMM");  
        Student s2 = new Student(22,"NNN");  
  
        s1.display();  
        s2.display();  
    }  
}
```


Understanding the problem without static variable

```
class Student{  
    int rollno;  
    String name;  
    String college="ITS";  
}
```



With and without static

```
class Counter
{
int count=0;//will get memory each time when the
instance is created

Counter()
{
count++;//incrementing value

System.out.println(count);
}

public static void main(String args[]){
//Creating objects

Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
```

```
class Counter2
{
static int count=0;//will get memory only once and retain
its value

Counter2(){
count++;//incrementing the value of static variable

System.out.println(count);
}

public static void main(String args[]){
//creating objects

Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

2) Java static method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

//Java Program to demonstrate the use of a static method.

```
class Student{  
    int rollno;  
    String name;  
    static String college = "ITS";  
    //static method to change the value of static variable  
  
    static void change(){  
        college = "BBDIT";  
    }  
    //constructor to initialize the variable  
    Student(int r, String n){  
        rollno = r;  
        name = n;  
    }  
}
```

```
//method to display values  
    void display(){System.out.println(rollno+" "+name+" "+college);}  
}  
//Test class to create and display the values of object  
public class TestStaticMethod{  
    public static void main(String args[]){  
        Student.change();//calling change method  
  
        //creating objects  
        Student s1 = new Student(111,"Karan");  
        Student s2 = new Student(222,"Aryan");  
        Student s3 = new Student(333,"Sonoo");  
        //calling display method  
        s1.display();  
        s2.display();  
        s3.display();  
    }  
}
```

//Java Program to get the cube of a given number using the static method

```
class Calculate{  
    static int cube(int x){  
        return x*x*x;  
    }  
  
    public static void main(String args[]){  
        int result=Calculate.cube(5);  
        System.out.println(result);  
    }  
}
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

The static method can not use non static data member or call non-static method directly.

this and super cannot be used in static context.

```
class A {  
    int a=40;//non static  
  
    public static void main(String args[]){  
        System.out.println(a);  
    }  
}
```

Output:Compile Time Error

Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

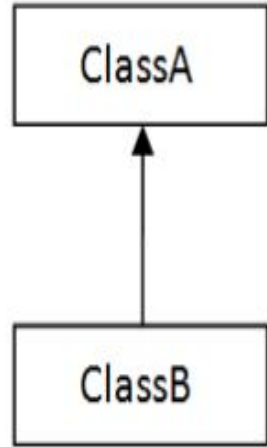
Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

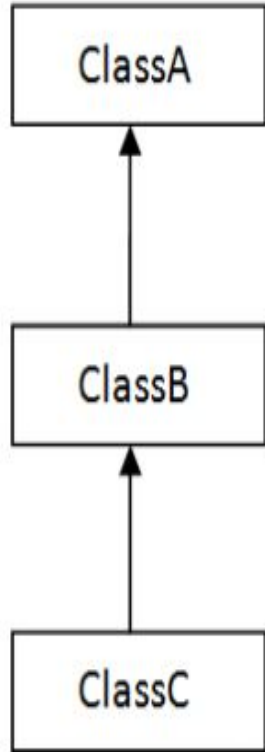
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name  
{  
    //methods and fields  
}
```

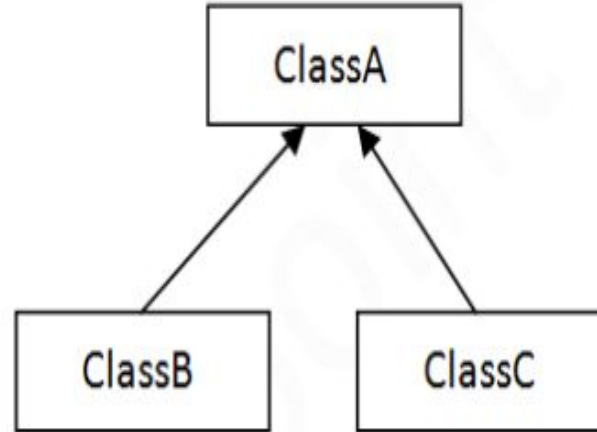
Types of inheritance in java



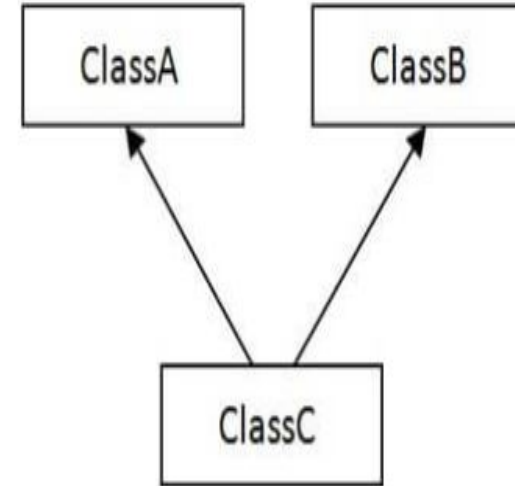
1) Single



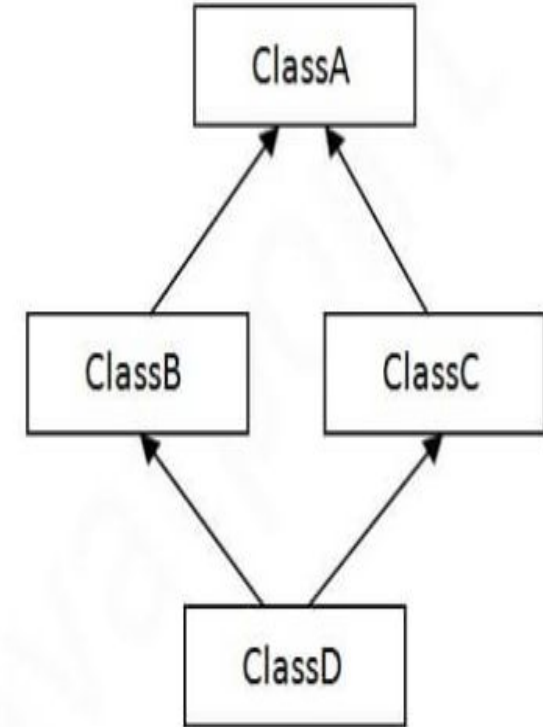
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

```
class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int benefits=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.benefits );
    }
}
```

Single Inheritance Example

```
class Animal{  
    void eat(){  
        System.out.println("eating...");  
    }  
    class Dog extends Animal{  
        void bark(){  
            System.out.println("barking...");  
        }  
    }  
    class TestInheritance{  
        public static void main(String args[]){  
            Dog d=new Dog();  
            d.bark();  
            d.eat();  
        }  
    }  
}
```

Output:

```
barking...  
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*.

```
class Animal{  
void eat(){System.out.println("eati  
ng...");}  
}  
class Dog extends Animal{  
void bark(){System.out.println("ba  
rking...");}  
}
```

```
class BabyDog extends Dog{  
void weep(){System.out.println("w  
eeping...");}  
}  
class TestInheritance2{  
public static void main(String args  
[]){  
    BabyDog d=new BabyDog();  
    d.weep();  
    d.bark();  
    d.eat();  
}}
```

Output:

```
weeping...  
barking...  
eating...
```

Hierarchical Inheritance Example

- When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

```
class Animal{
void eat(){System.out.println("eating...");
}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");
}
}
```

```
class Cat extends Animal{
void meow(){System.out.println("meowing...");
}
}
class TestInheritance3{
public static void main(String args
[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Multiple Inheritance not supported in Java

```
class A
```

```
{
```

```
void msg(){System.out.println("Hello");}
```

```
}
```

```
class B
```

```
{
```

```
void msg(){System.out.println("Welcome");}
```

```
}
```

```
class C extends A,B
```

```
{//suppose if it were
```

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

```
{
```

```
    C obj=new C();
```

```
    obj.msg();//Now which msg() method would be invoked?
```

```
}
```


Super Keyword in Java

Super Keyword in Java

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Super Keyword

1

Super can be used to refer immediate parent class instance variable.

2

Super can be used to invoke immediate parent class method.


3

super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

```
class Animal{  
    String color="white";  
}  
class Dog extends Animal{  
    String color="black";  
    void printColor(){  
        System.out.println(color);//prints  
        color of Dog class  
        System.out.println(super.color);//  
        /prints color of Animal class  
    }  
}
```

```
class TestSuper1{  
    public static void main(String  
        args[]){  
        Dog d=new Dog();  
        d.printColor();  
    }  
}
```



black
white

2) super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eatin
g...");}
}
class Dog extends Animal{
void eat(){System.out.println("eatin
g bread...");}
void bark(){System.out.println("bar
king...");}
void work(){
super.eat();
bark();
}
}
```

```
class TestSuper2{
public static void main(String a
rgs[]){
Dog d=new Dog();
d.work();
}}
```

Output:

```
eating...
barking...
```

3) super is used to invoke parent class constructor.

```
class Animal{  
    Animal()  
    {  
        System.out.println("animal is created  
");  
    }  
}  
  
class Dog extends Animal{  
    Dog(){  
        super();  
        System.out.println("dog is created");  
    }  
}
```

```
class TestSuper3  
{  
    public static void main(String args[  
    ])  
    {  
        Dog d=new Dog();  
    }  
}
```

Output:

```
animal is created  
dog is created
```

Another example of super keyword where `super()` is provided by the compiler implicitly.

```
class Animal
{
    Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog()
    {
        System.out.println("dog is created");
    }
}
class TestSuper4{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}
```

Output:

```
animal is created
dog is created
```

super example: real use

```
class Person{
    int id;
    String name;
    Person(int id,String name){
        this.id=id;
        this.name=name;
    }
}

class Emp extends Person{
    float salary;
    Emp(int id,String name,float salary){
        super(id,name);//reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+ " "+name
+ " "+salary);
}
}
```

```
class TestSuper5
{
    public static void main(String[] ar
gs)
    {
        Emp e1=new Emp(1,"ankit",4500
Of);
        e1.display();
    }
}
```


Final Keyword In Java

- The **final keyword** in java is **used to restrict the user**.
- The java final keyword can be used in many context. Final can be:
 1. variable
 2. method
 3. class
- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.

1) Java final variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant).
- **Example of final variable**
- There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

```
class Bike9{  
  final int speedlimit=90;//final variable  
  void run(){  
    speedlimit=400;  
  }  
  public static void main(String args[]){  
    Bike9 obj=new Bike9();  
    obj.run();  
  }  
}//end of class
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{  
    final void run(){System.out.println("running");}  
}
```

```
class Honda extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda honda= new Honda();  
        honda.run();  
    }  
}
```

Test it Now

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
```

```
class Honda1 extends Bike{
```

```
    void run(){System.out.println("running safely with 100kmph");}
```

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

```
        Honda1 honda= new Honda1();
```

```
        honda.run();
```

```
    }
```

```
}
```

Interface in Java

Interface in Java

- An interface in Java is **a blueprint of a class**.
- It has 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 method body. It is used to achieve abstraction and multiple inheritance in Java.
- Java Interface also represents the IS-A relationship.

Why use Java interface?

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

- Syntax:

```
interface <interface_name>{
```

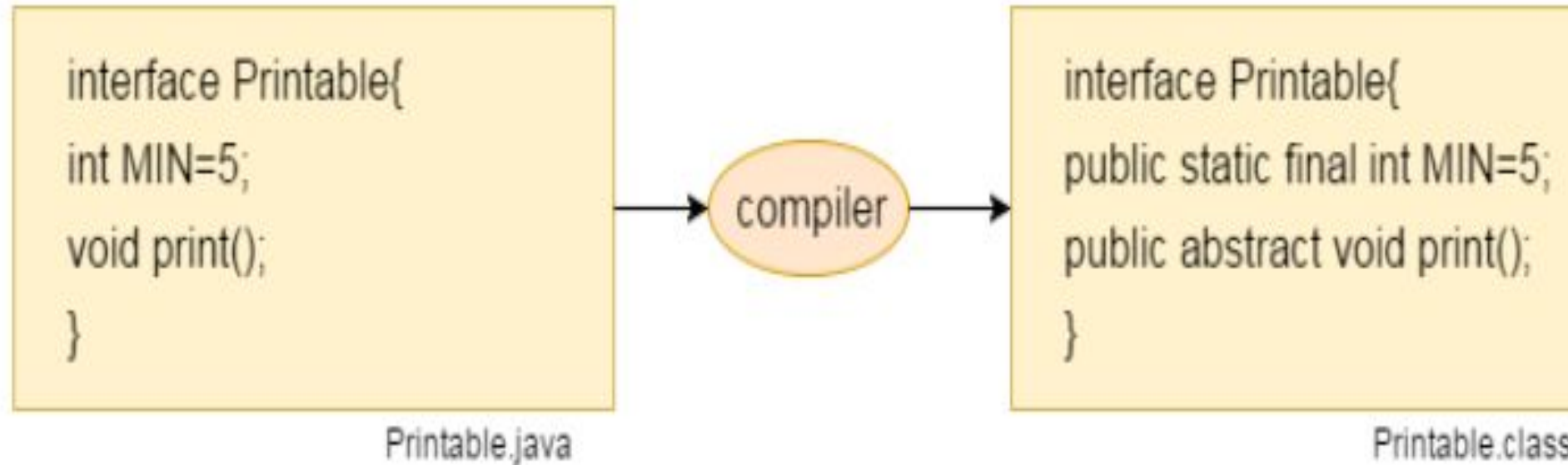
```
    // declare constant fields
```

```
    // declare methods that abstract
```

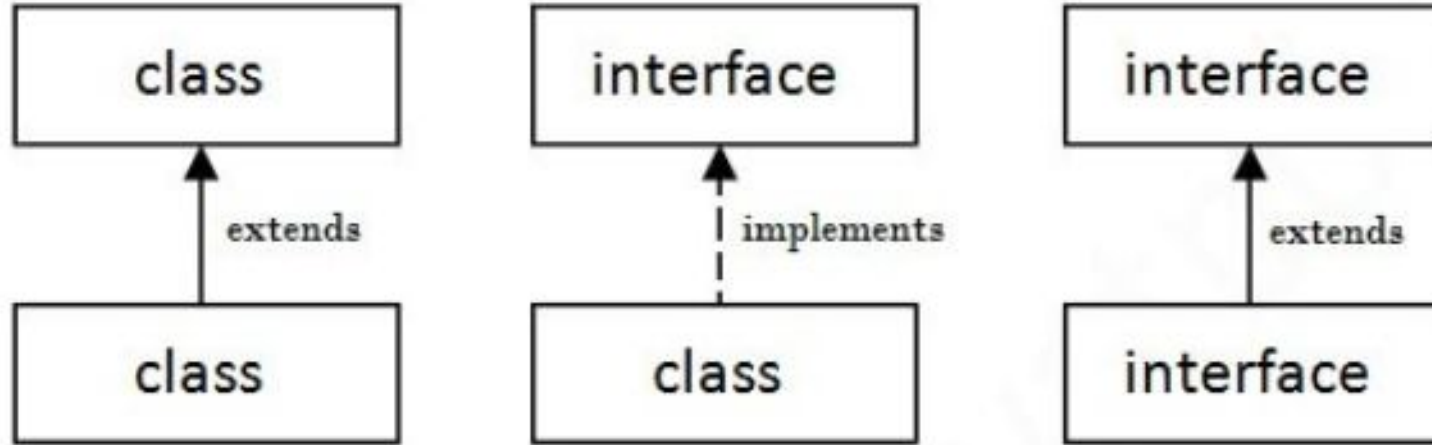
```
    // by default.
```

```
}
```


Internal addition by the compiler



The relationship between classes and interfaces



```
import java.io.*;
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

Output:
Hello

//Interface declaration: by first user

1. **interface** Drawable

2. {

3. **void** draw(); }

4. //Implementation: by second user

5. **class** Rectangle **implements** Drawable{

6. **public void** draw(){

7. System.out.println("drawing rectangle");}

8. }

1. **class** Circle **implements** Drawable{

2. **public void** draw(){

3. System.out.println("drawing circle");}

4. }

5. //Using interface: by third user

6. **class** TestInterface1{

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

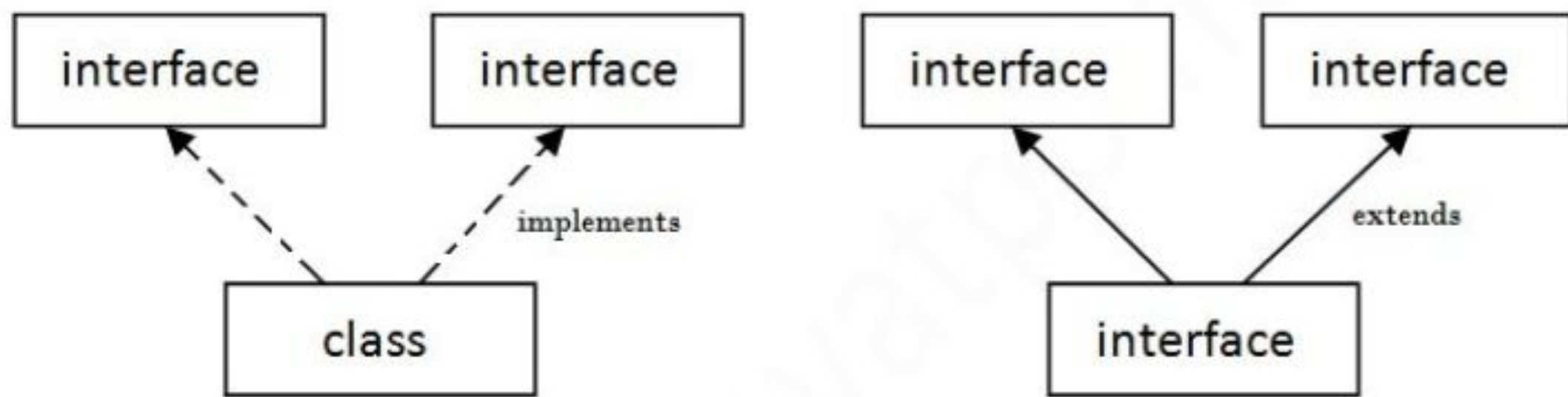
8. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()

9. d.draw();

10. }}

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{  
void print();  
}  
interface Showable{  
void show();  
}  
class A7 implements Printable,Showable{  
public void print()  
{System.out.println("Hello");}  
public void show()  
{System.out.println("Welcome");}  
public static void main(String args[]){  
A7 obj = new A7();  
obj.print();  
obj.show();  
}  
}
```