Object Oriented Programming Using Java

Code: PCS3I102

3rd Semester B. Tech (CSE)

3rd Semester B. Tech (CSE)
(2018-19)

Bapuji Rao, B.Sc(Hons.), DISM, M.Sc(IT), M.Tech(CS), (Ph.D)
Mobile: +91 9437825480

IGIT, Sarang, Dhenkanal Dist., Odisha.

Syllabus

Module 2: -

Chapter 1-: Introduction to Classes and Objects.

Classes, Methods, Objects, Description of data hiding and data encapsulation, Constructors, Use of static Keyword in Java, Use of this Keyword in Java, Array of Objects, Concept of Access Modifiers (Public, Private, Protected, Default).

Chapter 2-: Inheritance

Understanding Inheritance, Types of Inheritance and Java supported Inheritance, Significance of Inheritance, Constructor call in Inheritance, Use of super keyword in Java, Polymorphism, Understanding Polymorphism, Types of polymorphism, Significance of Polymorphism in Java, Method Overloading, Constructor Overloading, Method Overriding, Dynamic Method Dispatching.

Chapter 3-: String Manipulations.

Introduction to different classes, String class, String Buffer, String Builder, String Tokenizer, Concept of Wrapper Classes, Introduction to wrapper classes, Different predefined wrapper classes, Predefined Constructors for the wrapper classes. Conversion of types from one type (Object) to another type (Primitive) and Vice versa, Concept of Auto boxing and unboxing.

Chapter 4:- Data Abstraction

Basics of Data Abstraction, Understanding Abstract classes, Understanding Interfaces, Multiple Inheritance Using Interfaces, Packages, Introduction to Packages, Java API Packages, User-Defined Packages, Accessing Packages, Error and Exception Handling, Introduction to error and exception, Types of exceptions and difference between the types, Runtime Stack Mechanism, Hierarchy of Exception classes, Default exception handling in Java, User defined/Customized Exception Handling, Understanding different keywords (try, catch, finally, throw, throws), User defined exception classes, Commonly used Exceptions and their details.

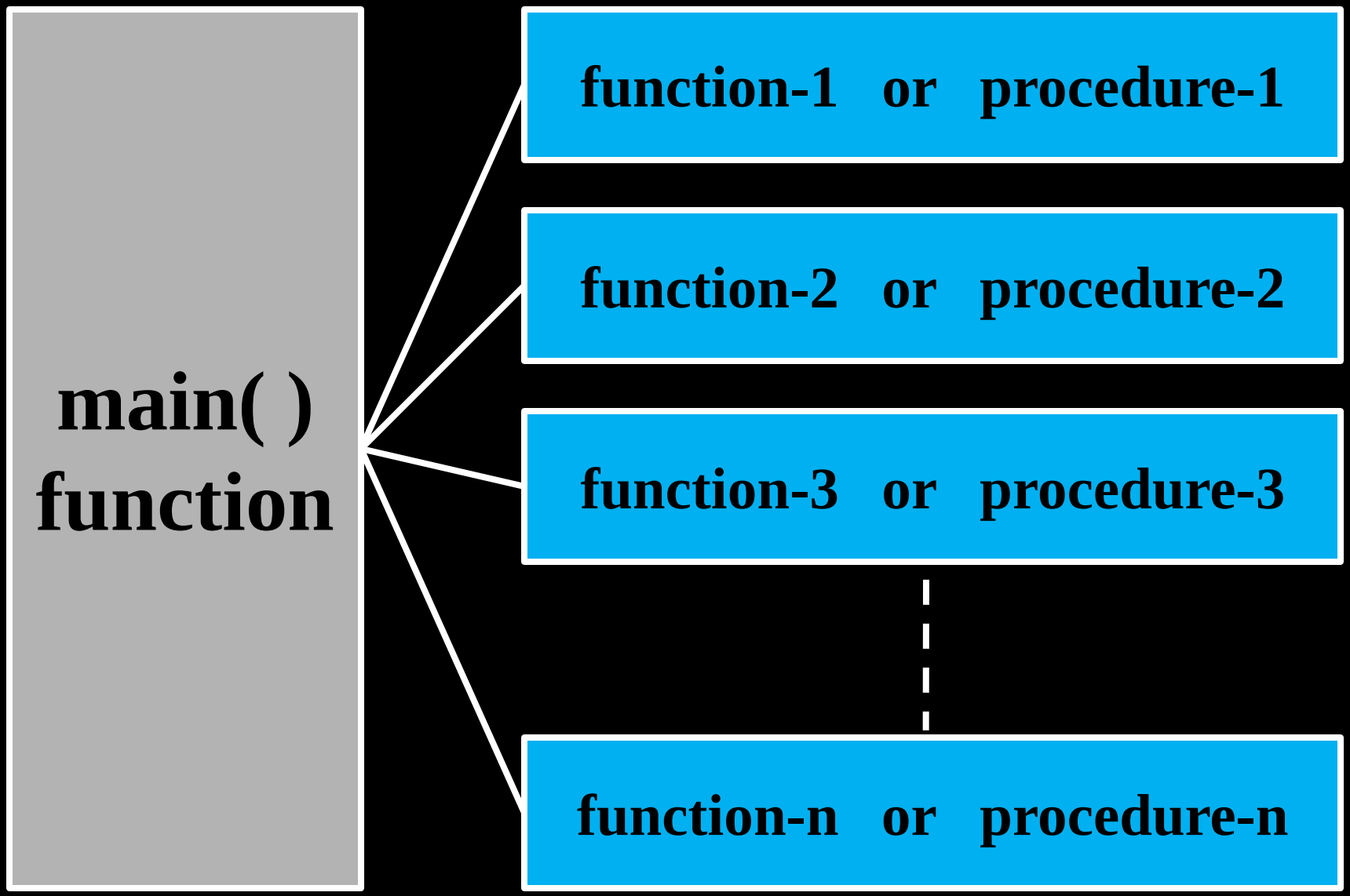
Chapter 5:- Multithreading

Introduction of Multithreading/Multitasking, Ways to define a Thread in Java, Thread naming and Priorities, Thread execution prevention methods. (yield(), join(), sleep()), Concept of Synchronisation, Inter Thread Communication, Basics of Deadlock, Demon Thread, Improvement in Multithreading, Inner Classes, Introduction, Member inner class, Static inner class, Local inner class, Anonymous inner class.

Procedure Oriented Approach

- The programmer uses procedures or functions to perform a task.
- The main function is divided into so many functions or procedures depending upon the situation.
- The approach is called as procedure oriented approach.
- Examples: C, Pascal, Fortran etc.

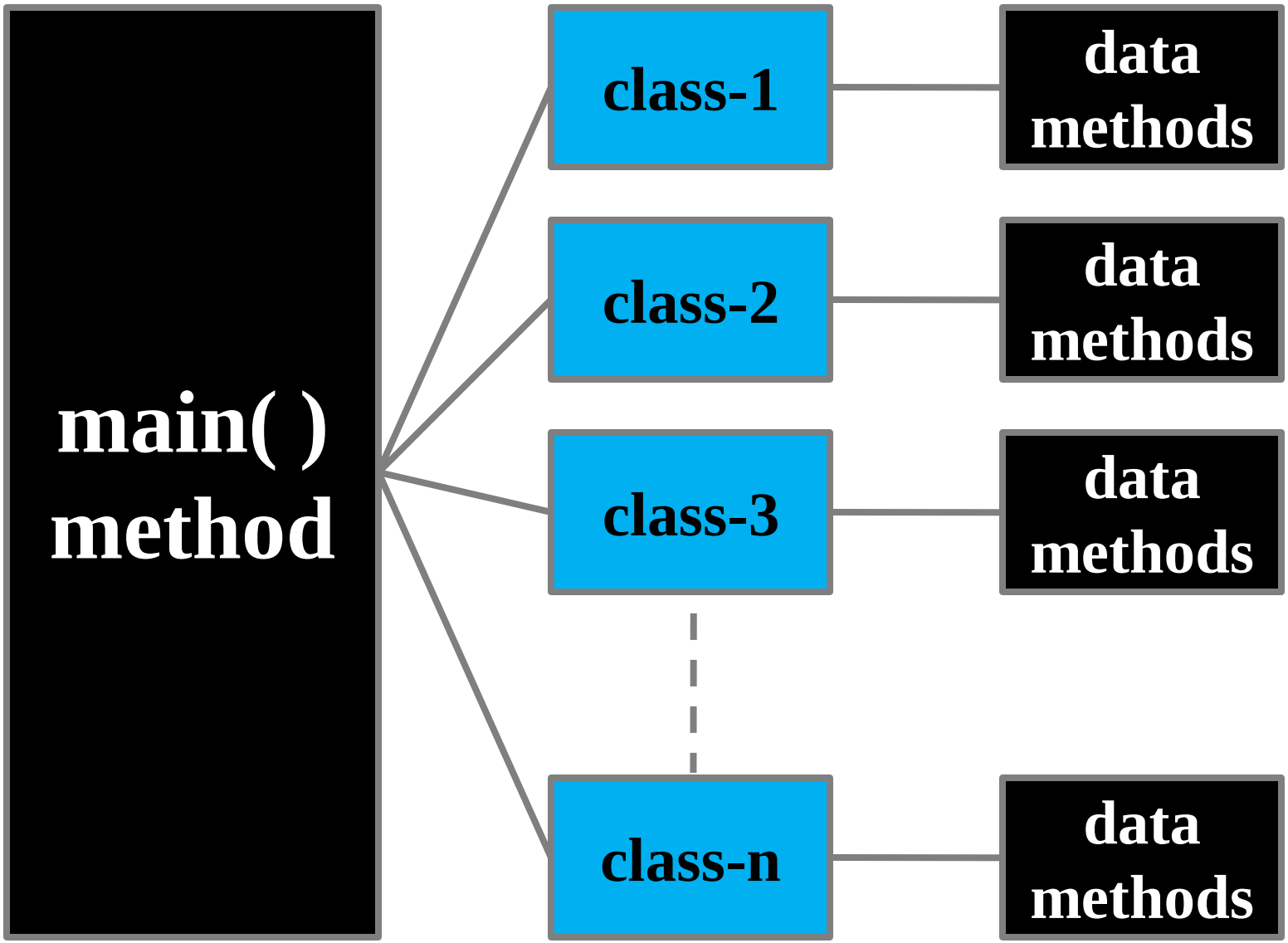
Procedure Oriented Approach



Object Oriented Approach

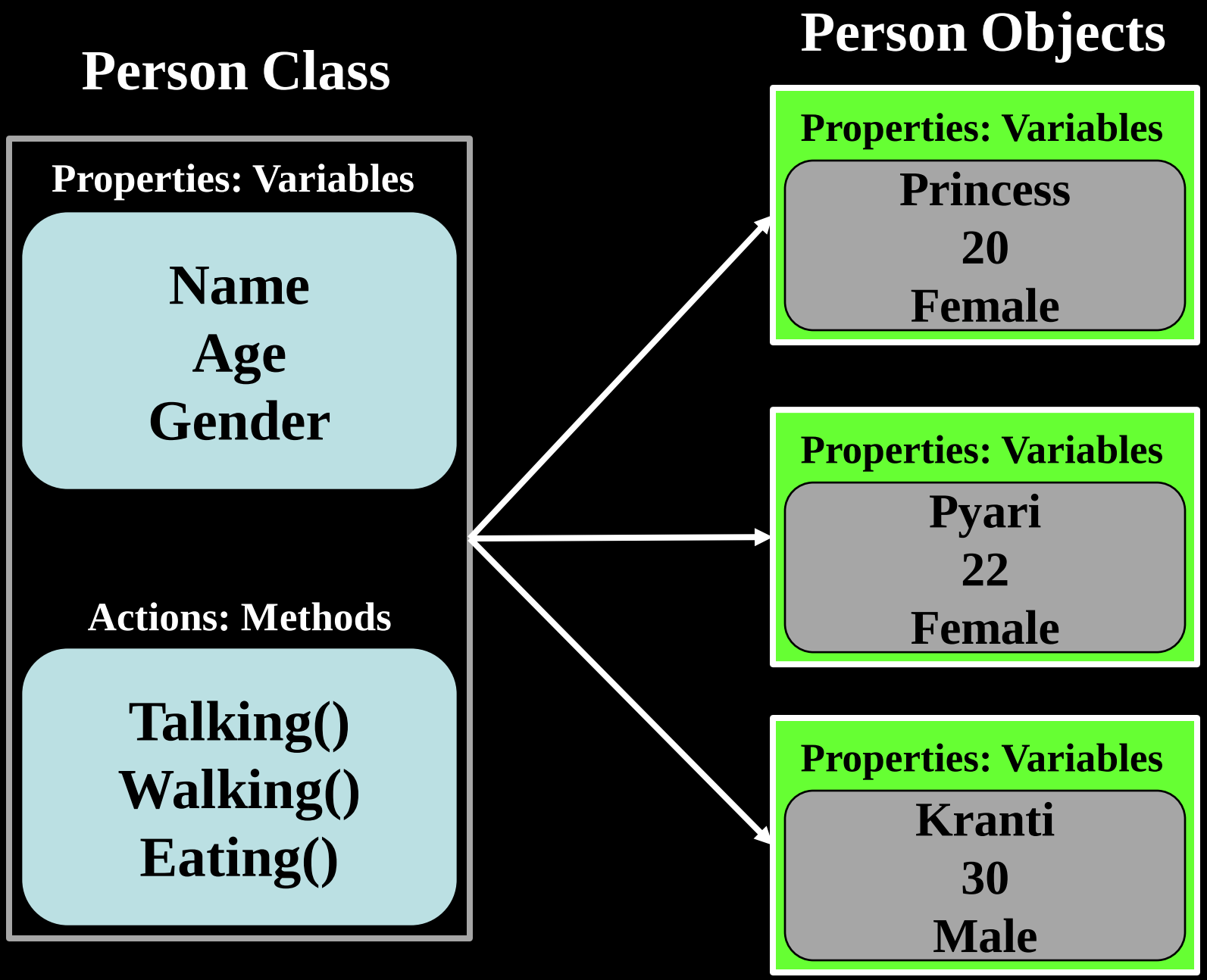
- The programmer uses classes and objects in the program.
- A class is a module which itself contains data and methods (functions) to achieve the task.
- The main task is divided into several methods, and these are represented as classes.
- Each class performs some tasks through its available methods.
- This approach is called as object oriented approach.
- Examples: C++, Java, C# etc.

Object Oriented Approach



Object Oriented Programming System (OOPS)

- Object oriented approach programming will have several modules.
- Each module represents a **“class”**.
- These classes are reusable. Hence maintenance of code becomes easy.
- The entire program is built on a single root concept called as **“object”**.
- **Examples:** All human beings are objects.
  All animals are objects.
- Programming in object oriented approach is called as **OOPS**.


Person Class
Properties: Variables
Name
Age
Gender
Actions: Methods
Talking()
Walking()
Eating()
Person Objects
Properties: Variables
Princess
20
Female
Properties: Variables
Pyari
22
Female
Properties: Variables
Kranti
30
Male

CLASS

- It is an user-defined data type.
- It's template defines class properties.

Syntax of Declaration of Class:

```
class  <class_name>
{
        variable(s) declaration;
        method(s) declaration;
}
```

So a class is a combination of variables and methods. Variables are called **member variables** and methods are known as **member methods**.

Examples-1: Empty class declaration.

```
class xyz
{

}
```

Example-2: Class with 2 instance or member variables a and b.

```
class pqr
{
    int a, b;
}
```

Example-3: Class with one member method and two member variables p and q.

```
class hello
{
        int p,q;
        void assign( )
        {
          p=1;  q=2;
        }
}
```

Syntax of Method Declaration

```
return_type method_name (arguments(s), if any)
{
        // body of the method
}
```

Method declaration has four parts:

- **Name of method**
- **The return data type**
- **A list of argument(s), if any**
- **Body of the method**

Example-1:

```
void result( int a , int b )
{
 int c=a+b;
 int d=a*b;
System.out.println("Sum=" + c);
System.out.println("Product=" + d);
}
```

Example-2:

```
class values
{
int a, b;
void assign (int p, int q)
{
   a=p;
   b=q;
 }
int  sum( )
{
return(a+b);
}
}
```

Note:

- In class, we may have many methods and variables.
- Only these are accessible by all the methods in the **same class**.
- But **a method cannot access the variables declared in other methods.**

```
class access
 {    int  x;
      void method1( )
     {
       x = 10;
       int  y = x;   }
     void method2( )
     {
        x = 5;
        int z = 10;
        y  = 1; // illegal because it belongs to method1( )
      }
}
```

OBJECTS

- Class variable creation is known as Objects and the process is known as **instances of classes.**

- An object is a block of memory that contains space to store all the **instance variables**.

- Objects are created using **new** operator.

- It creates an object of a specified class and returns a **reference** to the object reference.

Syntax-1:

```
class_name    object_name;
object_name = new  class_name( );
```

OR

Syntax-2:

```
class_name   object_name = new  class_name( );
```

AREA obj; // declaration

During declaration an object reference is created called **obj** which holds a null value.

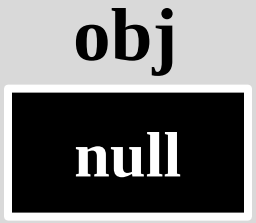
obj = new AREA( ); // class gets instantiated

After instantiate, AREA object is created and its reference is returned to the object reference **obj**.

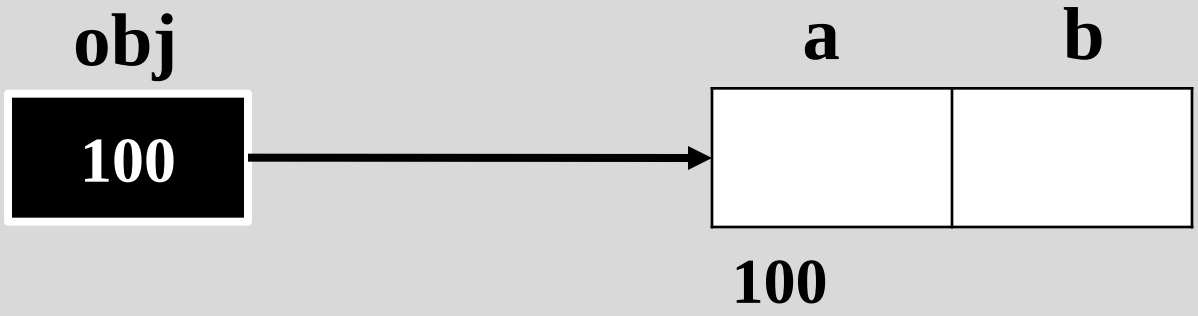
```
// Display Sum and difference of 12 and 7.
 class  numbers
 {   int a, b;
     void assign( int x, int y)
     {
          a = x;  b = y;
      }
    void result( )
    {
   System.out.println("Sum=" + (a + b ) );
   System.out.println("Difference=" + (a - b ) );
   }
  }
```

```
class result
{
public static void main( String args[ ])
{
   numbers obj = new numbers( );
  //  numbers  obj;
 //  obj = new numbers( );
obj.assign(12, 7);
obj.result( );
}
}
```

numbers obj;

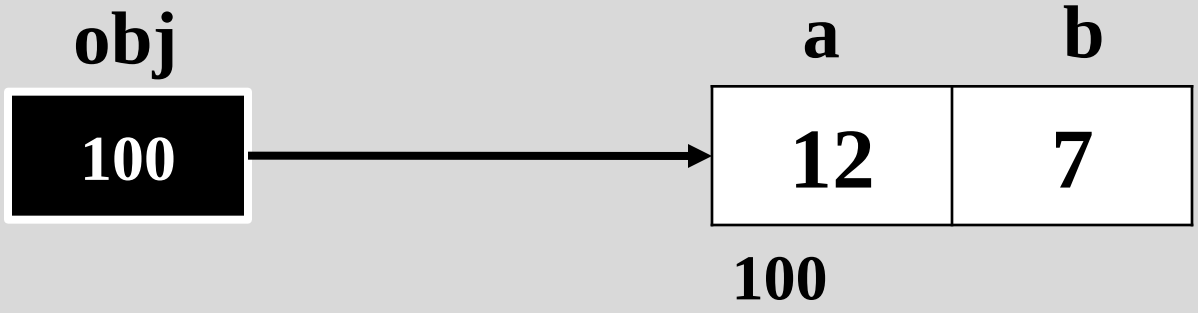


obj = new numbers( );

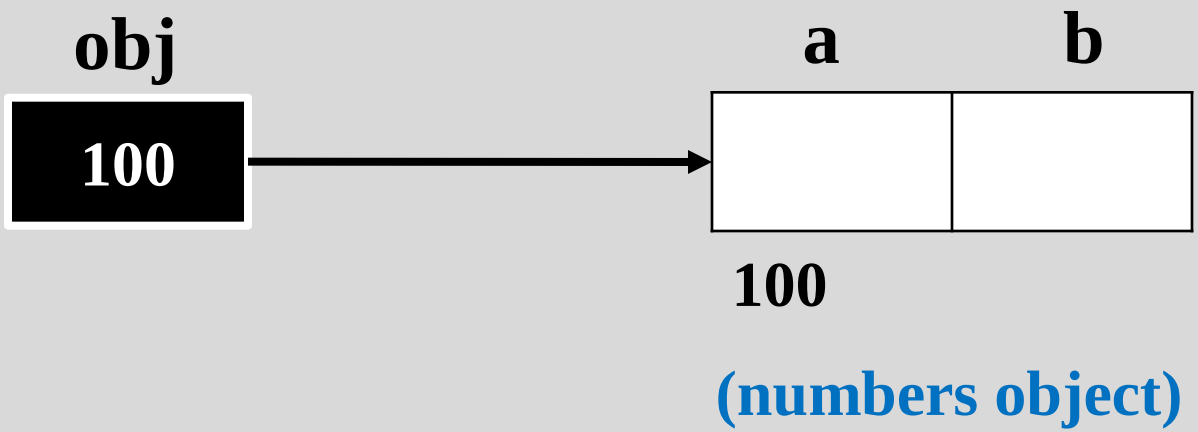


(numbers object)

obj.assign(12, 7);

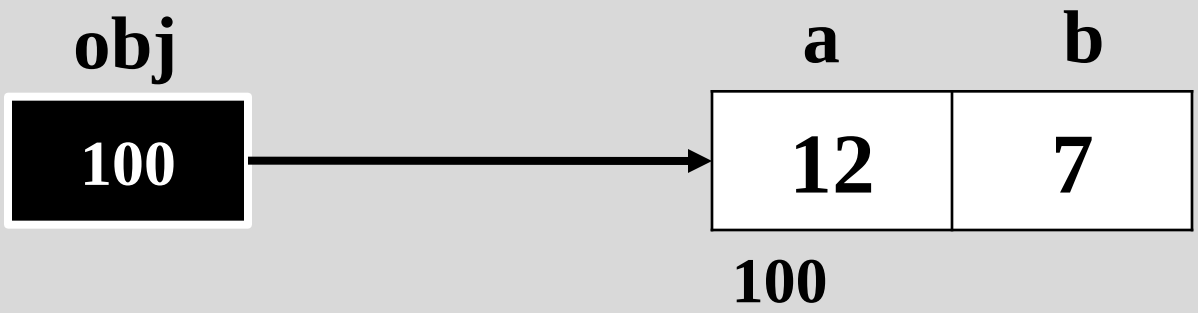


numbers obj = new numbers( );



(numbers object)

obj.assign(12, 7);

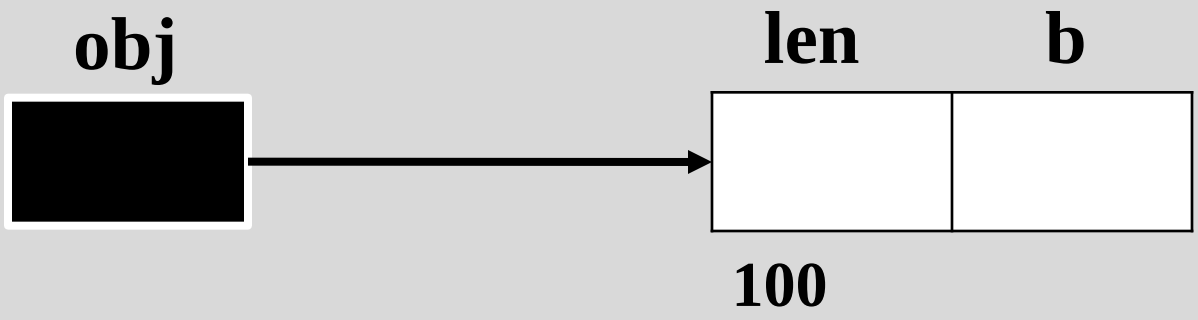


```
// Find area and perimeter of a rectangle.
class  rectangle
{
double len, b;
void init(double x, double y)
{
len = x;  b = y;
}
double area( )
{
   return (len*b);
 }
 double perimeter( )
 {
   return (2*(len+b));
  }
}
```

```
class AreaPer
{
public static void main( String args[ ] ) throws
   IOException
{
  DataInputStream in = new
   DataInputStream(System.in);

  rectangle obj = new rectangle( );

  System.out.print("Enter length and breadth");
  double a = Double.parseDouble(in.readLine() );
  double b = Double.parseDouble(in.readLine() );
```

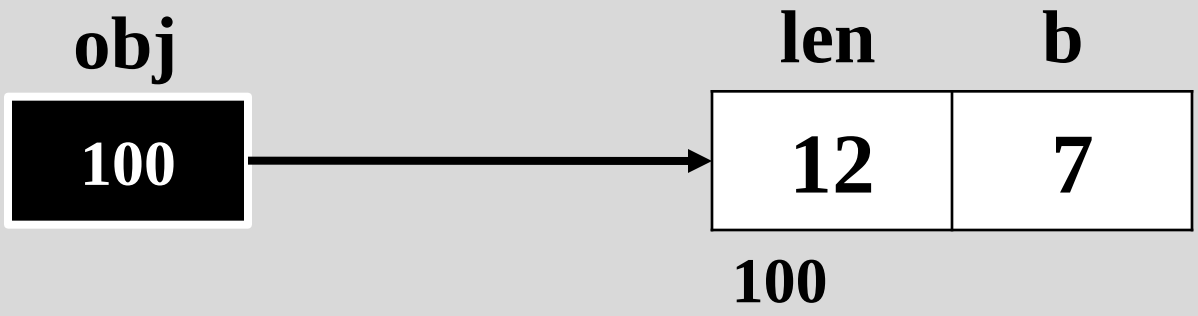
```
    obj.init( a, b);

    double r1 = obj.area( );
    double r2 = obj.perimeter( );
    System.out.println("Area=" + r1);
    System.out.print("Perimeter =" + r2);
    }
}
```

rectangle obj = new rectangle( );



(rectangle object)

obj.init(a, b);



METHOD OVERLOADING

- Methods having same name but different parameter lists and definitions.
- It is required when objects perform similar tasks but uses different parameters.
- This process is known as **polymorphism**.

Syntax:

```
class  class_name
{
   return_type method(argument)
   {
      // statement(s);
   }
   return_type method(arg1, arg2, ……..)
   {
     // statement(s);
   }
}
```

```
// Find area of a square, rectangle, and
// circle using method overloading.
class  AREAS
 {
    int  area (int s)
    {
      return (s*s);
     }
```

```
int  area ( int l, int b )
{
  return ( l * b );
 }

double area ( double r )
{
   return (3.142 * r * r );
 }
}
```

```
class  overload
 {
 public static void main(String args[ ])
 {
      AREAS  A = new AREAS( );
      int  r1 = A.area (10);
      int  r2 = A.area (4, 7);
      double  r3 = A.area (5.2);
System.out.print(r1 + “  “ + r2 + “  “ + r3 );
 }
}
```

Array of Objects

Syntax-1:

```
ClassName objectReference[ ];
objectReference  = new ClassName[dimension];
```

Syntax-2:

```
ClassName objectReference[ ]
                    = new ClassName[dimension];
```

Question:

Read ‘n’ employees name, gender, post, and salary in an array of object. Then display all the details.

Method-1

```
import java.io.*;
class employee
 {  String name, gen, post;
    double sal;
  void assign_data(String n, String g, String p, double s)
  {
    name=n;
    gen=g;
    post=p;
    sal=s;
   }
  void out_data()
   {
    System.out.println("Name     = "+name+"\nGender  = "+gen);
    System.out.println("Post      = "+post+"\nSalary    = "+sal);
   }
}
```

```
class array_object1
{
 public static void main(String args[]) throws IOException
  {
 DataInputStream in=new DataInputStream(System.in);
 System.out.print("Enter number of employees ?");
 int ne=Integer.parseInt(in.readLine());
 employee emp[ ]=new employee[ne];
 for(int i=0;i<ne;i++)
 {
    // reading data in main() method
     System.out.print("Enter name....");
     String name=in.readLine();
     System.out.print("Enter gender....");
     String gen=in.readLine();
     System.out.print("Enter post....");
     String post=in.readLine();
```

```
System.out.print("Enter salary....");
double sal=Double.parseDouble(in.readLine());

employee obj = new employee();
obj.assign_data(name, gen, post, sal);

emp[i]=obj;
  }

 System.out.println("All "+ne+" employee details");
 for(int i=0;i<ne;i++)
   emp[i].out_data();
  }
}
```

Question:

Read ‘n’ employees name, gender, post, and salary in an array of object. Then display all the details.

Method-2

```
import java.io.*;
class employee
{
    String name, gen, post;
    double sal;
    void getdata() throws IOException   // reading data in getdata()
    {
      DataInputStream in=new DataInputStream(System.in);
      System.out.print("Enter name....");
      name=in.readLine();
      System.out.print("Enter gender....");
      gen=in.readLine();
      System.out.print("Enter post....");
      post=in.readLine();
      System.out.print("Enter salary....");
      sal=Double.parseDouble(in.readLine());
     }
```

```
void outdata()
{
    System.out.println("Name     = "+name);
    System.out.println("Gender  = "+gen);
    System.out.println("Post       = "+post);
    System.out.println("Salary    = "+sal);
}
  }
 // main class
class array_object
{
    public static void main(String args[]) throws IOException
   {
  DataInputStream in=new DataInputStream(System.in);
  System.out.print("Enter number of employees ?");
  int ne=Integer.parseInt(in.readLine());
  employee emp[ ]=new employee[ne];
```

```
for(int i=0;i<ne;i++)
{
   employee e=new employee();
   e.getdata();
   emp[i]=e;
}

System.out.println("All "+ne+" employee details");
 for(int i=0;i<ne;i++)
   emp[i].outdata();
  }
}
```

Object as Argument in Member Method

When more than one objects of a class are to be manipulated, then such case there must be a member method having object as argument to be defined in the class.

Syntax:

```
class   class_name
{
// member variable(s) declaration;
return_type  method_name  (class_name object1, ……)
{
   // statement (s);
}
}
```

Question:

Addition and Subtraction of matrices a[2][3]={ {1,2,3},{4,5,6}} and b[2][3]={{7,8,9},{2,1,7}} using object as argument in a member method.

```
import java.io.*;
class Matrix
{
  private int mat[ ][ ]=new int[2][3];
  void assign (int a[ ][ ])
  {
    for(int i=0; i<2; i++)
      for(int j=0; j<3; j++)
        mat[i][j]=a[i][j];
  }
```

```
void Show()
{
  for(int i=0; i<2; i++)
  {
    for(int j=0; j<3; j++)
      System.out.print(mat[i][j]+"  ");
    System.out.println(" ");
  }
}

void Add(Matrix obj1, Matrix obj2)
{
  for(int i=0; i<2; i++)
   for(int j=0; j<3; j++)
     mat[i][j] = obj1.mat[i][j] + obj2.mat[i][j];
}
```

```
void Sub(Matrix obj1, Matrix obj2)
 {
   for(int i=0; i<2; i++)
    for(int j=0; j<3; j++)
      mat[i][j] = obj1.mat[i][j] - obj2.mat[i][j];
 }
}
// main program
class MatrixAddSub
{
public static void main(String args[])
{
  int a[][] = {{1,2,3},{4,5,6}};
  int b[][] = {{7,8,9},{2,1,7}};
  Matrix mat1 = new Matrix( );
  Matrix mat2 = new Matrix( );
```

```
Matrix Res1 = new Matrix();
Matrix Res2 = new Matrix();
mat1.Assign(a);
mat2.Assign(b);

Res1.Add(mat1,mat2);
Res2.Sub(mat1,mat2);

System.out.println("Matrix-1");
mat1.Show();
System.out.println("Matrix-2");
mat2.Show();
System.out.println("Matrix Addition");
Res1.Show();
System.out.println("Matrix Subtraction");
Res2.Show();
 }
}
```

Object as Return Value from Member Method

Syntax:

```
class   class_name
{
// member variable(s) declaration;
class_name  method_name  (class_name object1, ……)
{
   // statement (s);
   return (object of class_name);
}
}
```

Question:

Addition and Subtraction of matrices a[2][3]={{1,2,3},{4,5,6}} and b[2][3]={{7,8,9},{2,1,7}} using object as argument and return object method.

```
import java.io.*;
class Matrix
{
  private int mat[ ][ ]=new int[2][3];
  void assign (int a[ ][ ])
  {
    for(int i=0; i<2; i++)
      for(int j=0; j<3; j++)
        mat[i][j]=a[i][j];
  }
```

```
void Show()
{
  for(int i=0; i<2; i++)
  {
    for(int j=0; j<3; j++)
      System.out.print(mat[i][j]+"  ");
    System.out.println(" ");
  }
}
Matrix Add(Matrix obj)
{
  Matrix temp = new Matrix( );
  for(int i=0; i<2; i++)
   for(int j=0; j<3; j++)
     temp.mat[i][j] = mat[i][j] + obj.mat[i][j];
  return(temp);
}
```

```
Matrix Sub(Matrix obj)
 {
   Matrix temp = new Matrix( );
   for(int i=0; i<2; i++)
    for(int j=0; j<3; j++)
      temp.mat[i][j] = mat[i][j] - obj.mat[i][j];
   return(temp);
 }
}
class MatrixAddSub   // main class
{
public static void main(String args[])
{
  int a[][] = {{1,2,3},{4,5,6}};
  int b[][] = {{7,8,9},{2,1,7}};
  Matrix mat1 = new Matrix( );
  Matrix mat2 = new Matrix( );
```

```
 Matrix Res1 = new Matrix();
 Matrix Res2 = new Matrix();
 mat1.Assign(a);
 mat2.Assign(b);

 Res1 = mat1.Add(mat2);
 Res2 = mat1.Sub(mat2);

 System.out.println("Matrix-1");
 mat1.Show();
 System.out.println("Matrix-2");
 mat2.Show();
 System.out.println("Matrix Addition");
 Res1.Show();
 System.out.println("Matrix Subtraction");
 Res2.Show();
 }
}
```

Addition and subtraction of two complex numbers.

```
import java.util.*;
 class complex
  {
  private int real, img;
  void assign(int a, int b )
   {
      real = a;
      img = b;
   }
  void show( )
 {
char ch;
if(img < 0) ch = '-';  else ch = '+';
System.out.println( real + " " + ch + " i " + Math.abs(img) );
 }
```

```
void add(complex obj1, complex obj2)
   {
      real = obj1.real + obj2.real;
      img = obj1.img + obj2.img;
   }
void sub( complex obj1, complex obj2)
   {
      real = obj1.real - obj2.real;
      img = obj1.img - obj2.img;
   }
}
```

```
class complex_add_sub  // main program
 {
 public static void main(String args[ ]) throws IOException
 {
  Scanner sc = new Scanner(System.in);
  complex c1, c2, c3, c4;
  c1 = new complex( );
  c2 = new complex( );
  c3 = new complex( );
  c4 = new complex( );
System.out.print("Enter 1st complex number's real and imaginary part:");
int r = sc.nextInt( );
int i = sc.nextInt( );
c1.assign( r , i );
```

```
System.out.print("Enter 2nd complex number's real and imaginary part:");
r = sc.nextInt( );
i = sc.nextInt( );
c2.assign( r , i );
c3.add(c1,c2);     c4.sub(c1,c2);
System.out.println("1st complex Number");
c1.show( );
System.out.println("2nd complex Number");
c2.show( );
System.out.println("Complex Addition");
c3.show( );
System.out.println("Complex Subtraction");
c4.show( );
 }  }
```

CONSTRUCTORS

- To assign initial value to the instance variables, which is done through an **instance method** or using the **dot (.)** operator to access the instance variables. It is a tedious approach to initialize all the member variables of all the objects.

- When the object is created automatically the instance variables are initialized, which is done through **constructors**.

- A constructor is having the name of class itself.
- A constructor may or may not have argument(s) but no return value.
- There are **two** types of constructors.
- They are:

  Automatic or Default Constructor
  Parameterized Constructor

1. Automatic or Default Constructor:

When class gets instantiated, the unnamed object invokes the constructor automatically (**or implicitly**).

Syntax:

```
class class_name
{
  class_name ( )
  {
     // statement (s);
  }
}
```

```
class constructor
{
   constructor()    // default or automatic constructor
   {
      System.out.println("I am Java Automatic Constructor");
   }
 }
class con1
{
public static void main(String args[])
{
   constructor obj=new constructor();
   new constructor();
 }
}
```

Output

I am Java Automatic Constructor
I am Java Automatic Constructor

```
class constructor
{
   String st;
   constructor()  // default or automatic constructor
   {
      st = "I am Java Automatic Constructor";
   }
  void show()
  {
    System.out.println(st);
  }
 }
```

```
class con2
{
public static void main(String args[])
{
   constructor obj=new constructor();
   obj.show();
   new constructor().show();
 }
}
```

Output

I am Java Automatic Constructor
I am Java Automatic Constructor

2. Parameterized Constructor:

During class instantiation, the required numbers of parameters are to be passed.

Syntax:

```
class  class_name
{
  class_name (argument(s) list )
  {
    // statement (s);
  }
}
```

```
class constructor
{
   constructor(String st)    // parameterized constructor
   {
      System.out.println(st);
   }
}
```

```
class con3
{
public static void main(String args[])
{
   constructor friend1 = new constructor("Smiley");
   constructor friend2 = new constructor("Sweety");
   // new constructor("Smiley");
   // new constructor("Sweety");
 }
}
```

Output

Smiley
Sweety

```
class constructor
{
   String name;
   constructor(String st)  // parameterized constructor
   {
      name = st;
   }
   void show()
   {
     System.out.println(name);
    }
}
```

```
class con4
{
public static void main(String args[])
{
 constructor friend1 = new constructor("Smiley");
 constructor friend2 = new constructor("Sweety");

 friend1.show();
 friend2.show();
 }
}
```

Output

Smiley
Sweety

Constructor Overloading

Definition:

To define a constructor more than once with a different set of parameter list in a class.

Syntax:

```
class className
{
  member variable(s) declaration;
  className()
  {
    statement(s);
  }
  className(dataType variable)
  {
    statement(s);
  }
  ------------
  ------------
}
```

```
// Program using constructor overloading
class construct
{
     String st;
    construct()
     {
        st="Automatic";
       System.out.println(st+" constructor invoked.....");
     }
    construct(String d)
     {
        st=d;
       System.out.println(st+" constructor invoked.....");
     }
```

```
construct(String a, String b)
{
    st = a + " and " +b;
    System.out.println(st+" constructor invoked.....");
 }
}  // close of class construct
```

```
// main class
 class constructor_overloaded
 {
  public static void main(String args[])
 {
  construct c1=new construct();
  construct c2=new construct("Parameter");
  construct c3=
                  new construct("Parameter1", "Parameter2");
  new construct();
  new construct("Parameter");
  new construct("Parameter1","Parameter2");
 }
 }
```

Output

```
Automatic constructor invoked.....
Parameter constructor invoked.....
Parameter1 and Parameter2 constructor invoked.....
Automatic constructor invoked.....
Parameter constructor invoked.....
Parameter1 and Parameter2 constructor invoked.....
```

finalize( )

- Sometimes an object needs to perform some action when it is destroyed.
- Java provides a mechanism called ***finalization***.
- To add a finalizer in a class, define finalize( ) method in that class.
- Java run time calls that method whenever it is about to recycle an object of that class.

Syntax:

```
protected void finalize ( )
{
   // statement (s);
 }
```

Note: It is not called when an object goes out of scope. It is only called just prior to garbage collection.

Static Members

- To define a member which is common to all the objects and accessed without using a particular object is known as static members.

- It is accessed through **class** name.

- If a member variable is declared as static kind, it is known as **static variable** or **class variable**.

- **Syntax of static variable:**

  static data_type var1, var2, …….;

- If a member method is declared as static kind, then it is termed as **static method** or **class method**.

- **Syntax of static method:**

```
static return_type method_name (Argument(s) declaration, if any)
{
    // statement (s);
}
```

```
// Program using static members

class StaticKind
{
   static float multiply (float x, float y)
   {
       return (x*y);
   }

  static float divide (float x, float y)
  {
       return (x/y);
  }
}
```

```
// main program
class mainStatic
{
  public static void main (String args[ ])
{
  float res1 = StaticKind.multiply (5.0f, 4.0f);
  float res1 = StaticKind.divide (5.0f, 2.0f);

  System.out.println ("Product = " + res1);
  System.out.println ("Division = " + res2);
 }
}
```

Note:

- It is called using class names. No objects are required for accessing static method.

- It has some restrictions:
  - It can call other static methods.
  - It can access only static data.
  - It cannot refer to ***super*** and ***this***.

this keyword

- Sometimes a method will need to refer to the object that invoked it.

- This can be used inside any method to refer to the current object on which the method was invoked.

Syntax: this.member_variable

```
class  this_ex
 {
   int a , b;
this_ex( int a, int b )
   {
      a = a; // invalid
      b = b; // invalid
    }
 }
```

```
class  this_ex
 {
   int a , b;
this_ex( int a, int b )
   {
      this.a = a;
      this.b = b;
    }
 }
```

ACCESS SPECIFIERS

- To restrict the access to certain variables and methods from outside the class, we require **visibility modifiers** to the instance variables and methods.

- These **visibility modifiers** are known as access specifiers.

- There are four types of access specifiers. They are:
  - **private**
  - **public**
  - **protected**
  - **default**

private access

- It enjoys the highest degree of protection.
- It is accessible only with in their own class.
- It cannot be inherited by sub-classes.
- A method defined as private acts as final method, which prevents from being sub-classed.
- We **cannot override** a private method.

protected access

- These kind of members are visible to all classes and sub-classes in the same package but also to sub-classes in other packages.

- The other sub-classes not related to any package cannot use protected fields.

public access

- Any variable or method is visible to the entire class in which it is defined.

- By default every member of a class are treated as public.

default access

- If no access specifier is mentioned by the programmer, then the Java Compiler uses a “default” access specifier.

- “default” members are accessible outside the class, but generally within the directory.

Syntax of Member Variables Declaration

```
class Class_Name
{
   private  data_type var1, var2,……;

   public  data_type var1, var2,……;

   protected  data_type var1, var2,……;

   data_type  var1, var2,……;
}
```

Syntax of Member Methods Declaration

```
private return_type  method(data_type var,…….)
{
        // statement(s);
}

protected return_type  method(data_type var,…….)
{
        // statement(s);
}
```

Syntax of Member Methods Declaration

```
public return_type  method(data_type var,…….)
{
        // statement(s);
}

return_type  method(data_type var,…….)
{
        // statement(s);
}
```

Example

```
class PRIVATE
{
  private int a=10;

  void display( )
   {
  System.out.println("A="+ a);
   }
 }
```

```
class  private_main
{
  public static void main(String args[ ])
    {
      PRIVATE obj = new PRIVATE( );
      obj.display( );
}
}
```

Output: A = 10

```
class  private_main
{
  public static void main(String args[ ])
    {
      PRIVATE obj = new PRIVATE( );
      obj.display( );
      obj.a = 15;  // error
      obj.display( );
}
}
```

Error: a has private access in PRIVATE

Nested Classes

To define a class within another class. Such a class is called a ***nested class***.

Syntax:

```
class OuterClass
{
    .....
   class NestedClass
   {
     .....
   }
 }
```

- Nested classes are divided into two categories: **static** and **non-static**.
- Nested classes that are declared static are simply called ***static nested classes***.
- Non-static nested classes are called ***inner classes***.

Syntax: **class** OuterClass

```
{
        ........
        static class StaticNestedClass
        {
          .......
        }
        class InnerClass
        {
          ........
        }
}
```

- A **nested class** is a member of its **enclosing class**.

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared **private**.

- Static nested classes do not have access to other members of the enclosing class.

- As a member of the OuterClass, a nested class can be declared as **private**, **public**, or **protected**.

Inner Classes

- It is a class defined within another class.

- It is basically a safety mechanism since it hides from other outer classes.

- An object of inner class cannot be created in other classes.

- Object of inner class can be created only in its outer class.

- Inner class can access the members of outer class directly.

- The memory allocated for inner class object and outer class object separately.

- Inner class object contains an invisible additional field **“this$0”** that refers to the outer class object.

- When same member names are used in outer class as well as inner class, the accessing of members in this way:
  - **outerClassName.this.member** for accessing the member of outer class.
  - **this.member** for accessing the member of inner class.

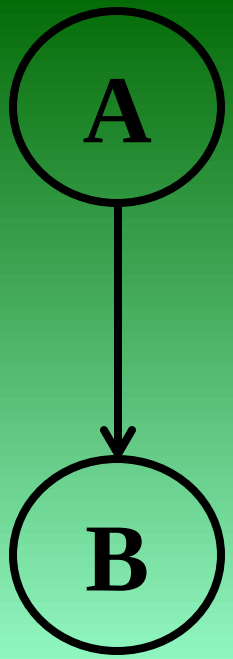
INHERITANCES

- Java classes can be reused in several ways.
- This is basically done by creating a new classes, reusing the properties of existing ones.
- The mechanism of deriving a new class from an old class is called inheritance.
- The old class is called ***base class*** or ***super class*** or ***parent class***.
- The new class is called ***sub-class*** or ***derived*** or ***child class***.

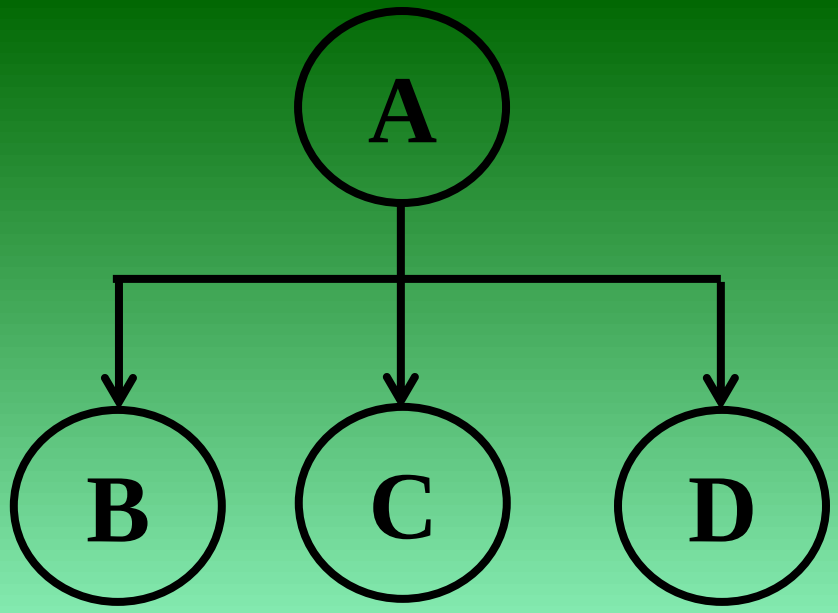
The inheritance allows sub-classes to inherit all the variables and methods of their parent classes.

It is of different forms:

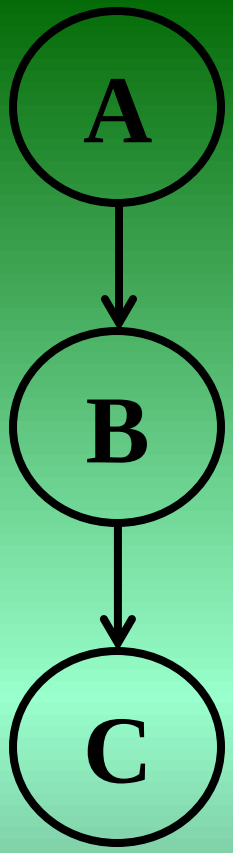
i. **Single Inheritance** (only one super class)
ii. **Multiple Inheritance** (Several Super Classes)
iii. **Hierarchical Inheritance** (One Super Class, many sub-classes)
iv. **Multilevel Inheritance** (Derived from a derived class)



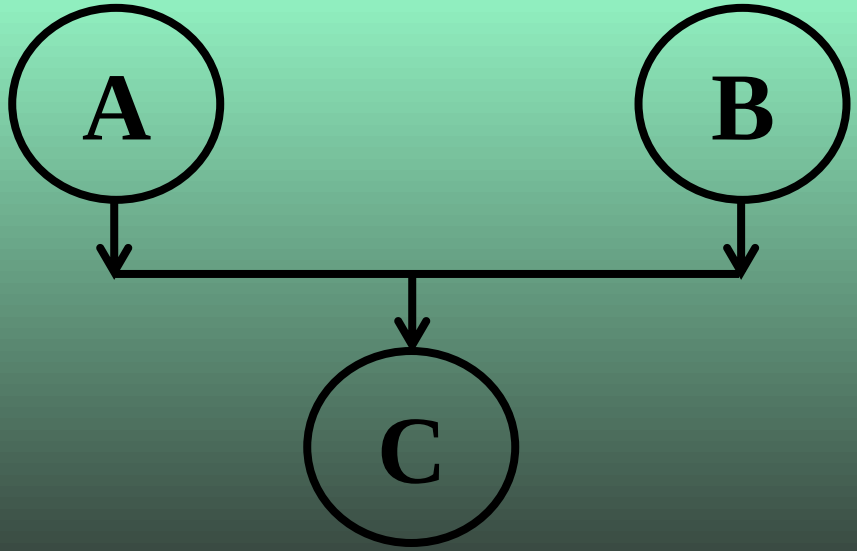
Single



Hierarchical



Multilevel



Multiple

Single Inheritance

A new class is derived from **one base class** (super class).

Syntax:

```
class sub_class_name extends super_class_name
{
    variable(s) declaration;
    method(s) declaration;
}
```

```
// single inheritance example
   class BOX    // base class
      {
       protected  int l , b;
       void Assign1( int x, int y )
       {
            l = x;  b = y;
       }
       int area( )
        {
            return(l*b);
        }
    }
```

```
class VOLUME extends BOX   // derived class
{
    private int h;
    void Assign2( int a)
    {
        h = a;
     }
    int volume( )
    {
         return(l*b*h);
     }
   }
```

```
class single_inheritance
{
      public static void main(String args[ ] )
      {
       VOLUME obj  = new VOLUME( );
        obj.Assign1(5, 10);
        obj.Assign2(3);
        int r1 = obj.area( );
        int r2 = obj.volume( );
        System.out.print("Area=" + r1 + "Volume =" + r2);
       }
    }
```

```
// Multilevel inheritance example
class ONE   // base class
{
    void display( )
    {
     System.out.println("Inside class ONE");
    }
}
```

```
class TWO extends ONE   // intermediate base class
{
    void display( )
    {
     super.display();
     System.out.println("Inside class TWO");
     }
}
```

```
class THREE extends TWO   // derived class
{
    void display( )
    {
     super.display();
     System.out.println("Inside class THREE");
     }
}
```

```
class Multuilevel
{
   public static void main(String args[ ])
    {
      THREE obj = new THREE( );
      obj.display( );
     }
}
```

Output

```
Inside class ONE
Inside class TWO
Inside class THREE
```

```
// Hierarchical inheritance example
class ONE   // base class
{
    void display( )
    {
     System.out.println("Inside class ONE");
    }
}
```

```
class TWO extends ONE   // derived class
{
    void display( )
    {
      super.display();
      System.out.println("Inside class TWO");
     }
}
```

```
class THREE extends ONE   // derived class
{
   void display( )
   {
     super.display();
     System.out.println("Inside class THREE");
    }
}
```

```
class Hierarchical
{
   public static void main(String args[ ])
    {
      TWO obj1 = new TWO( );
      THREE obj2 = new THREE( );
      obj1.display( );
      obj2.display( );
     }
}
```

Output

```
Inside class ONE
Inside class TWO
Inside class ONE
Inside class THREE
```

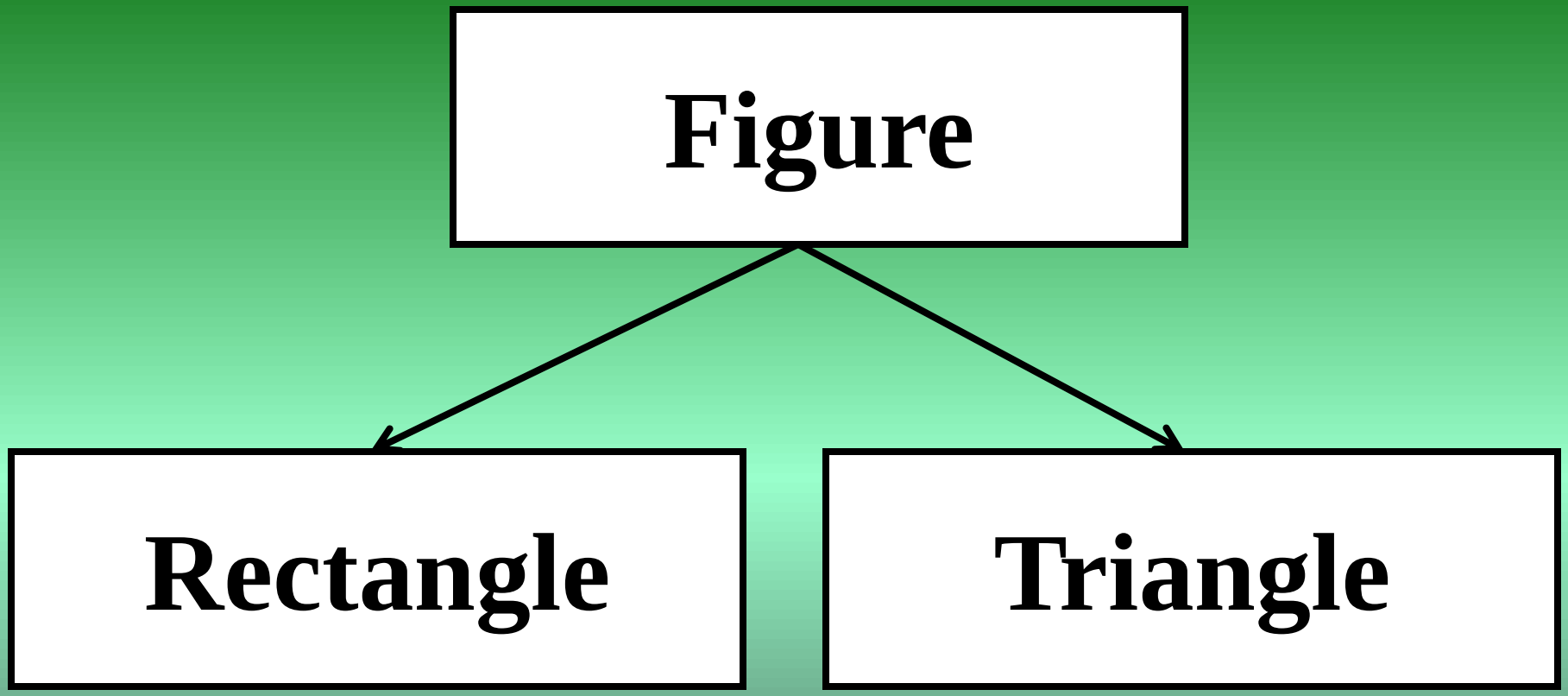


Figure
Rectangle
Triangle

```
// Hierarchical Inheritance
class Figure  // super class
{
  protected float a, b;
  void Assign( float x, float y)
  {
     a = x;
     b = y;
   }
}
```

```
class Rectangle extends Figure
{
  void area()
  {
   floar res = a*b;
   System.out.println("Area of Rectangle="+r);
   }
}
```

```
class Triangle extends Figure
{
 void area()
 {
   float res = a*b/2;
   System.out.println("Area of Triangle="+res);
 }
}
```

```
// main program for calculation of area
 class Area_Calculation
 {
 public static void main(String args[ ])
 {
   Rectangle robj = new Rectangle( );
   Triangle tobj = new Triangle( );

    robj.Assign(9, 15);
    tobj.Assign(6, 12);

    tobj.Area();
    robj.Area();
}
}
```

Output

```
Area of  Triangle = 36.000000
Area of Rectangle = 135.000000
```

```
// Hierarchical Inheritance with constructor
class Figure  // super class
{
  protected float a, b;
  Figure ( float x, float y)
  {
     a = x;
     b = y;
   }
}
```

```
class Rectangle extends Figure
{
   Rectangle(float x, float y)
   {
     super (x,y);
   }
  float area()
  {
   System.out.println("Inside Rectangle");
   return(a*b);
  }
}
```

```
class Triangle extends Figure
{
   Triangle(float x, float y)
   {
     super (x,y);
   }
 float area()
 {
 System.out.println("Inside Triangle");
 return(a*b/2);
 }
}
```

```
// main program for calculation of area
 class Area_Calculation
 {
 public static void main(String args[ ])
 {
   Triangle tobj = new Triangle(6, 12);
   Rectangle robj = new Rectangle(9, 15);

   System.out.println("Area =" + tobj.area());
   System.out.println("Area =" + robj.area());
 }
}
```

Output

```
Inside Triangle
Area = 36.000000
Inside Rectangle
Area = 135.000000
```

Method Overriding

If a method in a sub-class has the same name and signature as a method in its super class, then the method in the sub-class is said to be override the method in super-class.

The overridden method is called from a sub-class always.

The method of sub-class always overrides the method of super-class if both the methods are having same name and signature.

To avoid such situation the keyword ***super*** is used.

This ***super*** keyword can invoke method of super-class.

The statement becomes 1st statement in the method of sub-class.

```
// Program using overridden method
 class A
    {
     int i, j;
    void Assign1( int x, int y )
     {
       i = x;  j = y;
     }
  void show( )
    {
     System.out.println(" I =" + i + " J =" + j );
    }
```

```
class B extends A
{
      int k;
    void Assign2(int d )
    {
        k = d;
     }
   void show( )
   {
     System.out.println(" " K =" + k );
   }
}
```

```
class override
{
   public static void main(String args[ ])
  {
      B  obj  = new B( );
      obj.Assign1(12,5);
      obj.Assign2(7);
      obj.show( );
   }
}
```

Output:

K = 7

Because the sub-class method show() overrides the method show() in super-class which has same name and signature.

```
// Program using without-overridden method
class A
{
     int i, j;
    void Assign1( int x, int y )
    {
      i = x;  j = y;
     }
  void show( )
  {
     System.out.println(" I =" + i + " J =" + j );
   }
}
```

```
class B extends A
{
      int k;
    void Assign2(int d )
    {
        k = d;
     }
    void show( )
    {
      super.show( );
      System.out.println( " K =" + k );
    }
}
```

```
class without_override
{
public static void main(String args[ ])
{
      B  obj  = new B( );
      obj.Assign1(12,5);
      obj.Assign2(7);
      obj.show( );
}
}
```

Output:

I = 12 J = 5
K = 7

When obj invokes show( ) method of class **B**, then it finds the first statement **super.show( )**, is the method show( ) of super/base class **A**. So first the show() method of base class A gets invoked. Then the statements of show( ) method of derived class **B** gets executed. Hence no overridden of method of base class takes place.

Sub-class Constructor

- It is used to construct the instance variables of both the sub-class and the super-class.

- The sub-class constructor uses the method **super( )** to invoke the constructor method of super-class.

The keyword super has some **limitations**:

i. It is only used in sub-class constructor method.

ii. It must appear as the first statement with in the **sub-class constructor**.

iii. The parameters or arguments of **super( )** must match the order and type of variables declared in the super-class constructor.

```
// Program for single inheritance and constructor
class BOX   // base class
{
      protected  int l , b;
     // parameterized constructor in base class
       BOX( int x, int y)
        {
           l = x;  b = y;
        }
      int area( )
       {
          return(l*b);
       }
}
```

```
class VOLUME extends BOX    // derived class
{
    private int h;
   // parameterized constructor in derived class
    VOLUME( int a, int b, int c)
    {
        super(a, b); // to invoke the constructor of super class
        h = c;
     }
    int volume( )
       {
         return( l*b*h);
       }
}
```

```
class single_constructor
{
   public static void main(String args[ ] )
  {
     VOLUME obj  = new VOLUME(5, 10, 2);
     int r1 = obj.area( );
     int r2 = obj.volume();
    System.out.print("Area=" + r1 + "Volume =" + r2);
  }
}
```

final

All methods and variables can be overridden by default in sub-class. If we wish to prevent the sub-classes from overriding the members of the super class, declare them by using the keyword ***final***.

The final has the following advantages:

i) It is used to create the equivalent of a named constant.

ii) To prevent method overriding in inheritance.

iii) To prevent inheritance.

The final keyword used in:
variable, method, and class

Final Variables

- A variable can be declared as final.
- It prevents from modification of contents.
- To declare a variable with some initial value with the keyword **final** now acts as a constant (similar to **const** in C/ C++).

Syntax: **final data_type variable_name = value;**

Example: **final int a = 5;**
final float PI = 3.142f;

Variable with final declaration does not occupy memory because it becomes a constant.

Final Methods

To **prevent method overriding**, then the method must be declared as final.

Syntax:

```
final return_type method_name (Argument(s),if any)
{
    // statement(s);
}
```

```
class xyz
{
        final void display( )
        {
         System.out.println(“ I Am in Super-Class”);
        }
}
class pqr extends xyz
{
   void display( )
   {
     System.out.println(“ I Am in Derived-Class”);
    }
}
```

Since display( ) is declared as **final**.

It cannot be overridden in **pqr**.

If you do so, then a **compile-time error** results.

final class

- To prevent a class from being inherited.
- To do so, precede the keyword **final** before class name.
- A class cannot be sub-classed further is called **final class**.

Syntax:

```
final  class class_name
{
   // statement(s);
}
```

Example:

```
final  class A
{
   // statement(s);
 }
class  B extends A
 {
    // statement(s);
 }
```

Illegal operation since ***A*** is declared as ***final***. So it cannot be inherited to ***B***.

Dynamic Method Dispatch

- It is known as **run-time polymorphism** in JAVA.

- It is the mechanism by which a call to an **overridden method** is resolved at **run-time** rather than compile-time.

- It is the type of object being referred to that determines which version of an overridden method will be executed.

```
// Program for dynamic method dispatch
class ONE   // super or base class
{
    void display( )
    {
     System.out.println( "Inside class ONE");
    }
}
```

```
class TWO extends ONE // intermediate base class
{    void display( )
     {
      System.out.println("Inside class TWO");
     }
}
```

```
class THREE extends TWO  // derived class
{   void display( )
    {
     System.out.println("Inside class THREE");
    }
}
```

```
class Dispatch
{
public static void main(String args[ ])
    {
      ONE ref;   // reference type of ONE kind
      ONE obj1 = new ONE( );
      TWO obj2 = new TWO( );
      THREE obj3 = new THREE( );
      ref = obj1;   ref.display( );
      ref = obj2;   ref.display( );
      ref = obj2;   ref.display( );
    }
}
```

Output

```
Inside class ONE
Inside class TWO
Inside class THREE
```

```
// Program using Dynamic Method Dispatch
class Figure  // super class
{
   protected float a, b;
  Figure ( float x, float y)
  {
    a = x;  b = y;
   }
   float area( )
  {
   System.out.println("Area of figure not Defined");
   return(0);
   }
}
```

```
class Triangle extends Figure
{
   Triangle(float x, float y)
   {
     super (x,y);
   }
  float area( )
  {
   System.out.println("Inside Triangle");
   return(a*b/2);
  }
}
```

```
class Rectangle extends Figure
{
   Rectangle(float x, float y)
   {
     super (x, y);
   }
   float area( )
  {
   System.out.println("Inside Rectangle");
   return(a*b);
   }
}
```

```
// main program for calculation of areas
 class Area_Calculation
 {
   public static void main(String args[ ])
  {
    Figure ref;

    Figure fobj = new Figure(4,2);
    Triangle tobj = new Triangle(6,12);
    Rectangle robj = new Rectangle(9,15);
```

```
ref = fobj;
System.out.println("Area =" + ref.area( ));
ref = tobj;
System.out.println("Area =" + ref.area( ));
ref = robj;
System.out.println("Area =" + ref.area( ));
 }
}
```

Output

```
Area of figure not Defined
Area = 0
Inside Triangle
Area = 36.000000
Inside Rectangle
Area = 135.000000
```

Interfaces

- Java does not support multiple inheritance i.e. classes in Java cannot have more than one super-class.

i.e. **class name1 extends name2 extends ……**

```
{
    // statement(s);
}
```

is invalid.

- An alternative of multiple inheritance is called **interfaces**, which supports the concept of multiple inheritance.

Defining Interfaces

- It is also a class kind.
- It also contains variables and methods.
- The method should not contain any code. Only methods prototype or declaration must be included.
- The variables must contain a constant value.

Syntax of Variable Declaration in Interfaces

final static data_type variable_name = value;

Example:

final static float PI = 3.142f;

final static int area = 10;

Note : All variables of interface are ***final*** and ***static*** kind.

Syntax of method declaration in Interfaces

return_type method_name (parameter(s) list, if any);

Examples: **void display( );**

int result( );

double maximum(double x, double y);

int sum(int, int, int);

Note: A method contain only the declaration without having body inside the interface.

Example-1:

```
interface area
{
   final static int l = 10, b = 5;
   void display( );
}
```

The class which implements the above interface **area** must define the code for the method **display( )**.

Example-2:

```
interface circle
{
    final static float PI = 3.142f;
    float area(float x, float y);
    void display( );
}
```

The above interface has 2 methods area( ) and display( ), whose codes must be defined in the class where it implements.

Implementation of Interfaces

Interfaces are used as super-class whose properties are inherited by classes. So it is necessary to create a class that inherits the given interfaces.

Syntax:

```
class class_name implements interface_name
{
    // statement(s);
 }
```

Example:

```
interface  AREA
{
    final static float PI = 3.142f;
    float result(float a, float b);
}
class rectangle  implements AREA
{
     public float result(float a, float b)
     {
        return(a*b);
      }
}
```

```
class circle implements AREA
{
    public float result( float x, float y)
    {
        return(PI* x * x);
    }
}
```

```
class INTERFACE
{
public static void main(String args[ ])
{
    rectangle robj = new rectangle( );
    circle cobj = new circle( );
    float r1 = robj.result(5, 15);
    float r2 = cobj.result(10, 0);
    System.out.println("Area of Rectangle = " + r1);
    System.out.println("Area of Circle = " + r2);
   }
  }
```

Extending Interfaces

Like class, interfaces can also be extended, i.e. interfaces can be sub-interfaced from other interfaces.

The new sub-interface inherits all the members of the super-interface similar to sub-classes.

Syntax:

```
interface interface_name extends interface_name
{
    statement(s);
}
```

```
// Program for extending interface

interface product
{
   final static int code = 786;
   final static String name = "COMPUTER";
}

interface details extends product
{
    void output( );
}
```

```
class productList implements details
{
  public void output()
  {
   System.out.println("….Computer Details….");
   System.out.println("Code = "+ code);
   System.out.println("Product Name = "+ name);
   }
}
```

```
class main_program
{
   public static void main(String args[ ])
   {
      productList obj = new productList( );
      obj.output();
   }
}
```

Here, product is the super-interface which has variables code and name.

detail is the sub-interface which has one member method called output( ).

```
// Program for extending interface
   interface ONE
   {
        void A( );
        void B( );
   }

    interface TWO extends ONE
   {
        void C( );
    }
```

```
// class implements all methods of interface ONE and TWO
class myclass implements TWO
{
        public void A()
        {
            System.out.println("Implementing method A()");
        }
       public void B()
       {
            System.out.println("Implementing method B()");
        }
      public void C()
      {
            System.out.println("Implementing method C()");
       }
}
```

```
// main program
class interface_extends
{
   public static void main(String args[ ])
     {
        myclass obj = new myclass( );
        obj.A( );
        obj.B( );
        obj.C( );
      }
}
```

```
// Program for multiple inheritance using interface
 interface Animal
 {
   void moves(); // by default every method of
                 //  interface is public and abstract
  }

  interface Bird
 {
   void fly();
 }
```

```
class multiInheritance implements Animal, Bird
{
 public void moves( )
{
   System.out.println("Animals move on land");
}
public void fly( )
{
   System.out.println("Birds fly in air");
}
```

```
 public static void main (String args[ ])
{
 multiInheritance obj = new multiInheritance( );
 obj.moves();
 obj.fly();
 }
}
```

String class

- Objects of String class are immutable i.e. its contents cannot be modified.

- Hence no methods are available in String class.

- Examples of classes of objects of immutable kind are:
  Character, Byte, Integer, Float, Double,
  Long etc. – wrapper classes
  Class
  BigInteger, BigDecimal etc.

Example:

String a = new String (“Hello”);

String b = new String (“Java”);

String c = a+b;

System.out.print(c); Output: HelloJava

String c = a.append(b); is a invalid statement since object “a” and object “b” immutable kind. Hence it is a compile-time error.

StringBuffer class

- Objects of StringBuffer class are mutable i.e. its contents can be modified.
- The modification of contents of the objects are done by using its available methods.
- It is synchronized by default.
- Several threads process on a StringBuffer class object one after another.
- Synchronizing the object is like locking the object when a thread acts on the object, it is locked and any other thread should wait till the current thread completes and unlocks the object.
- Synchronization does not allow more than one thread to act simultaneously.
- It takes more execution time.

- **Examples:**

- **StringBuffer a = new StringBuffer(“Hello”);**

- **StringBuffer b = new StringBuffer();**

The StringBuffer object **b** as an empty one having a default capacity of **16 characters**.

- **StringBuffer c = new StringBuffer(20);**

The StringBuffer object **c** as an empty one having a capacity of storing **20 characters**. More than 20 characters can be assigned since it is a mutable kind of object.

- **a.append(“Java”);**

  System.out.print(a); **Output: HelloJava**

Now “Java” is appened to “Hello” in object “a” since object “a” is a mutable one.

- **b.append(“Java”);**

System.out.print(b); **Output: Java**

StringBuilder class

- Added in jdk1.5.
- Objects of StringBuilder class are mutable i.e. its contents can be modified.
- The modification of contents of the objects are done by using its available methods.
- It is unshynchronized.
- Several threads process on a StringBuilder object simultaneosly.
- It may lead to inaccurate result in some cases.

- **Examples:**

i. StringBuilder a = new StringBuilder(“Hello”);

ii. StringBuilder b = new StringBuilder();

The StringBuffer object **b** as an empty one having a default capacity of **16 characters**.

iii. StringBuilder c = new StringBuilder(20);

The StringBuffer object **c** as an empty one having a capacity of storing **20 characters**. More than 20 characters can be assigned since it is a mutable kind of object.

a.append(“Java”);

System.out.print(a); **Output: HelloJava**

The StringBuilder object “a” is of mutable kind. Hence string literal “Java” is appened with object “a”.

length( ) : To find the length of a string.

Syntax: int string.length( )

Here string is an **object** or **string literals.**

Examples:

i. int a = “Hello”.length( );

ii. String st = “Java String”;
int len = st.length( );

iii. String ab = “I.G.I.T”;
System.out.print(ab.length( ));

iv. char z[ ] = {‘H’,’E’,’L’,’L’,’O’};
String m = new String(z);
System.out.print(“Length=“+ m.length( ));

concat( ): To concatenate two strings.

Syntax: String string1.concat(string2)

Example:

```
String a = “I Like”;
String b = “India”;
String c = a.concat(b);
System.out.print(a+”\n“+b+”\n”+c);
```

Output: I Like
India
I LikeIndia

toUpperCase(): To change a string or character to uppercase..

Syntax: String string. toUpperCase( )

Examples:

i. String a = “Hello”. toUpperCase( );
System.out.print(a); **Output: HELLO**

ii. String b = “hello”;
String c = b. toUpperCase( );
System.out.print(c); **Output: HELLO**

toLowerCase( ): To change a string or character to lowercase.

Syntax: **String string.toLowerCase( )**

Examples:

i. String a = “HELLO”.toLowerCase( );
System.out.print(a); **Output: hello**

ii. String b = “HellO”;
String c = b.toLowerCase( );
System.out.print(c); **Output: hello**

charAt(): To pick a particular positioned character from a string.

Syntax: char string.charAt(int position)

Example:

i. char a = "I.G.I.T".charAt(2);

System.out.print(a); **Output: G**

ii. String st = "Java Programming";

char b = st.charAt(6);

System.out.print(b + " " + st.charAt(0)); **Output: r J**

substring(n, m): To retrieve a substring from n^{th} index character to m^{th} index characters.

Syntax:

String string.substring(int n, int m)

Here **n** index must be less than **m** index.

Examples:

i. String a = “TECHNOLOGY”.substring(5, 7);
System.out.print(a); **Output: OLO**

ii. String d = “Talcher”;
String a =d.substring(4, 6);
System.out.print(a); **Output: her**

substring(n): To retrieve a substring from n^{th} character to *end* of the string.

Syntax: String string.substring(int position)

Examples:

i. String a = “TECHNOLOGY”.substring(5);
System.out.print(a); **Output: OLOGY**

ii. String d = “RAILWAY”;
String a =d.substring(4);
System.out.print(a); **Output: WAY**

replace(): It replaces all occurrences of one character in a string with the specified character.

Syntax: **String string.replace(char ac, char rc)**

Here **ac**: actual character.

rc: replacement character.

Examples:

i. String a = “INDIAN”.replace(‘I’,’?’);

System.out.print(a); **Output: ?ND?AN**

ii. String b =“INDIAN”;

String c = b.replace(‘I’,’*’);

System.out.print(c); **Output: *ND*AN**

equals( ): It checks for two strings are same or not, and returns a boolean value true or false.

Syntax: boolean string1.equals(string2)

Example:

```
String s1 = "Java", s2 = "JAVA";
boolean b = s1.equals(s2);
if (b==true)
 System.out.print("Equal");
else
 System.out.print("Not Equal");
```

Output: Not Equal

equalsIgnoreCase( ): It checks for two strings are same or not by ignoring its cases, and returns a boolean value true or false.

Syntax: **boolean string1.equalsIgnoreCase(string2)**

Example:

```
String s1 = "Java", s2 = "JAVA";
boolean b = s1.equalsIgnoreCase (s2);
if(b==true)
   System.out.print("Equal");
else
   System.out.print("Not Equal");
```

Output: Equal

compareTo( ): Compares 2 strings and returns an integer value.

Syntax: int string1.compareTo(string2)

if string1 > string2 then a +ve value returns.

if string1 < string2 then a -ve value returns.

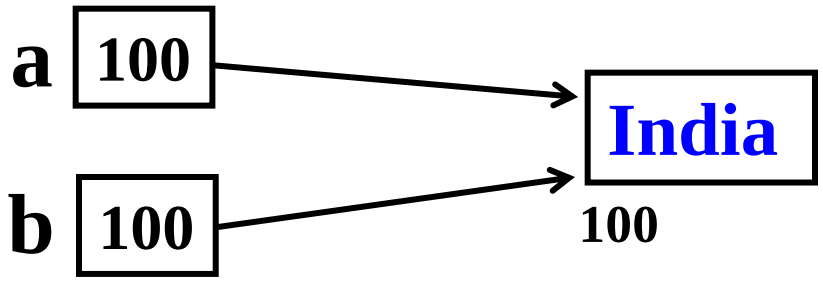
if string1 = string2 then a zero value returns.

Example:

```
String a = "Cool", b = "Don";
int t = a.compareTo(b);
if( t > 0 ) System.out.print("Bigger=" + a);
if( t < 0 ) System.out.print("Bigger=" + b);
```

Output: Bigger=Don

```
String a = "India", b = "India";
if(a==b)  System.out.print("Same");
else  System.out.print("Not");
```



Output: Same

```
String p = "Java", q = new String("Java");
if(p==q)  System.out.print("Same");
 else   System.out.print("Not");
```

Output: Not

String Tokenizer or Tokenizing a String

- The **split()** method in the **String cl**ass is specifically for splitting a string into tokens.

- It returns all the tokens from a string as an array of String objects.

- It passes two arguments.

- **Syntax:**

String[ ] object = StringObject.split(“[delimiters]”, LimitValue);
LimitValue: 0 or -1

Example-1:

```
class StringTokenizing2
{
public static void main(String[ ] args)
{
String text = "To be or not to be, that is the question.";
// Delimiters are comma, space, and period
String delimiters = "[, .]";
String[] tokens = text.split(delimiters, 0);
System.out.println("Number of tokens: " + tokens.length);
for(String token : tokens)
   System.out.println(token);
}
}
```

```
D:\JavaPro>java StringTokenizing
Number of tokens: 11
To
be
or
not
to
be

that
is
the
question

D:\JavaPro>javac StringTokenizing1.java
```

Example-2:

```
class StringTokenizing1
{
public static void main(String[ ] args)
{
String text = "To be or not to be, that is the question.";
// Delimiters are comma, space, and period
String delimiters = "[, .]";
String[] tokens = text.split(delimiters, -1);
System.out.println("Number of tokens: " + tokens.length);
for(String token : tokens)
   System.out.println(token);
}
}
```

```
D:\JavaPro>java StringTokenizing1
Number of tokens: 12
To
be
or
not
to
be

that
is
the
question

D:\JavaPro>javac StringTokenizing2.java
```

Extra blank line at the end

Example-3:

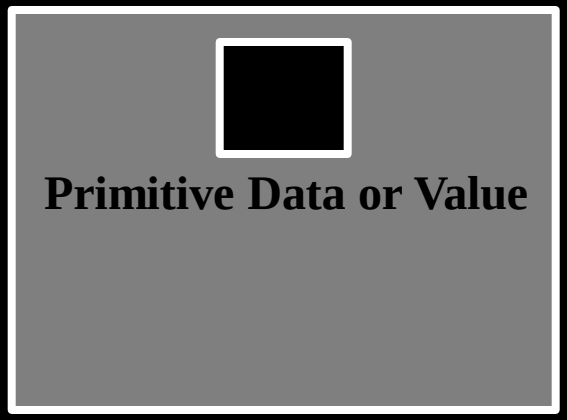
```
class StringTokenizing2
{
public static void main(String[] args)
{
String text = "To be or not to be, that is the question.";
String delimiters = "[abt]"; // Delimiters are a, b, and t

String[] tokens = text.split(delimiters, 0);
System.out.println("Number of tokens: " + tokens.length);
for(String token : tokens)
   System.out.println(token);
}
}
```

```
D:\JavaPro>java StringTokenizing2
Number of tokens: 10
To
e or no

o
e,
h

  is
he ques
ion.

D:\JavaPro>
```

Wrapper Classes

A wrapper class a class whose object wraps or contains a primitive data type. After creation of an object to a wrapper class, it contains a field where the primitive data stores.

In other words, we can wrap a primitive data into a wrapper class object.

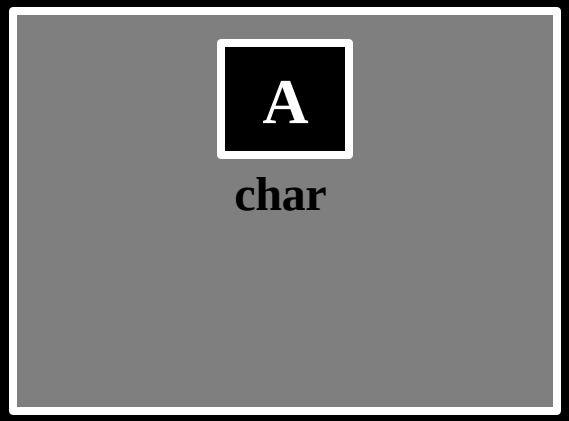


Wrapper Class Object

Example:

If we create an object to Character wrapper class, it contains a single field char and which can store a primitive character data say ‘A’.

So, **Character** is a wrapper class of **char** data type.



Character Object

- List of wrapper classes defined in **java.lang** package.
- It coverts the primitive data types into object form.

Primitive Data Types	Wrapper Classes
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Number Class

It is an abstract class whose sub-class are Byte, Short, Integer, Long, Float, and Double.

Number Class Methods	Purpose
byte byteValue()	It converts byte object into byte value.
short shortValue()	It converts short object into short value.
int intValue()	It converts integer object into int value.
long longValue()	It converts long object into long value.
float floatValue()	It converts float object into float value.
double doubleValue()	It converts double object into double value.

Character Class

- It wraps a value of the primitive type char in an object.
- Character class has only one constructor which accepts primitive data type.
- **Syntax:** **Character (char value)**
- **Example:** **Character obj = new Character ('A');**

Character Class Methods	Purpose
int Charater_Object.compareTo(Character Object)	It compares two Character objects for equal or greater than or less than, and returns an integer value.
String toString()	It converts Character object into String object and returns that String object.
static Character valueOf(char variable)	It converts a single character variable to Character object and return that object.
static boolean isDigit(char variable)	It returns a true value if the character is a digit; otherwise returns a false value.
static boolean isLetter(char variable)	It returns a true value if the character is an alphabet; otherwise returns a false value.

Character Class Methods	Purpose
static boolean isUpperCase(char variable)	It returns a true value if the character is an uppercase alphabet; otherwise returns a false value.
static boolean isLowerCase(char variable)	It returns a true value if the character is a lowercase alphabet; otherwise returns a false value.
static boolean isSpaceChar(char variable)	It returns a true value if the character is a space; otherwise returns a false value.
static boolean isWhiteSpace(char variable)	It returns a true value if the character is a white space character (tab key, enter key, or backspace key); otherwise returns a false value.
static boolean isLetterOrDigit(char variable)	It returns a true value if the character is either an alphabet or a digit; otherwise returns a false value.
static char toUpperCase(char variable)	It converts the character into uppercase and returns the uppercase character.
static char toLowerCase(char variable)	It converts the character into lowercase and returns the lowercase character.

Byte Class

- It wraps a value of the primitive type byte in an object.
- It has two constructors.
- **Syntax of 1st constructor: Byte(byte value)**
- **Example: Byte obj = new Byte(786);**
- **Syntax of 2nd constructor: Byte(String object)**
- **Example: Byte obj = new Byte("786");**

Byte Class Methods	Purpose
int byte_object.compareTo(Byte object)	It compares two Byte objects for equal or greater than or less than, and returns an integer value.
static byte parseByte(String object)	It returns the primitive byte number contained in the String object.
String toString()	It converts a Byte object into String object.
static Byte valueOf(String object)	It converts a String object that contain byte value into Byte class object.
static Byte valueOf(byte variable)	It converts the primitive byte data into Byte class object.

Short Class

- It wraps a value of the primitive type short in an object of Short class kind.

- It has two constructors.

- **Syntax of 1st constructor: Short(short value)**

- **Example: Short obj = new Short(7860);**

- **Syntax of 2nd constructor: Short(String object)**

- **Example: Short obj = new Short("7860");**

Short Class Methods	Purpose
int ShortObject1.compareTo(ShortObject2)	It compares two numerical values of Short class objects and returns 0, -ve value or +ve value.
boolean ShortObject1.equals(ShortObject2)	It compares two Short class objects. If same then returns true; otherwise false.
static short parseShort(String Object)	It returns the short value of the string object contains short value.
String ShortObject.toString()	It converts Short class object into String object and returns that String object.
Static Short valueOf(String object)	It converts a string object of short value to Short class object and return that object.

Integer Class

- It wraps a value of the primitive type ***int*** in an object of ***Integer*** class kind.
- It has two constructors.
- **Syntax of 1st constructor: Integer(int value)**
- **Example: Integer obj = new Integer(78607);**
- **Syntax of 2nd constructor: Integer(String object)**
- **Example: Integer obj = new Integer("78607");**

Integer Class Methods	Purpose
int IntegerObject1.compareTo(IntegerObject2)	It compares two numerical values of Integer class objects and returns 0, -ve value or +ve value.
boolean IntegerObject1.equals(IntegerObject2)	It compares two Integer class objects. If same then returns true; otherwise false.
static int parseInt(String Object)	It returns the int value of the string object contains int value.
String IntegerObject.toString()	It converts Integer class object into String object and returns that String object.
static Integer valueOf(String object)	It converts a string object of int value to Integer class object and return that object.

Integer Class Methods	Purpose
int IntegerObject.intValue()	It converts the Integer class object to primitive int data type.
static String toBinaryString(int)	It converts a decimal integer to its equivalent binary number and returns as String kind.
static String toOctalString(int)	It converts a decimal integer to its equivalent octal number and returns as String kind.
static String toHexaString(int)	It converts a decimal integer to its equivalent hexadecimal number and returns as String kind.

Long Class

- It wraps a value of the primitive type ***long*** in an object of ***Long*** class kind.

- It has two constructors.

- **Syntax of 1st constructor: Long(long value)**

- **Example: Long obj = new Long(12345678);**

- **Syntax of 2nd constructor: Long(String object)**

- **Example: Long obj = new Long("12345678");**

Long Class Methods	Purpose
int LongObject1.compareTo(LongObject2)	It compares two numerical values of Long class objects and returns 0, -ve value or +ve value.
boolean LongObject1.equals(LongObject2)	It compares two Long class objects. If same then returns true; otherwise false.
static long parseLong(String Object)	It returns the long value of the string object contains long value.
String LongObject.toString()	It converts Long class object into String object and returns that String object.
static Long valueOf(String object)	It converts a string object of long value to Long class object and return that object.

Float Class

- It wraps a value of the primitive type ***float*** in an object of ***Float*** class kind.
- It has three constructors.
- **Syntax of 1st constructor: Float(float value)**
- **Example: Float obj = new Float(12.345f);**
- **Syntax of 2nd constructor: Float(double value)**
- **Example: Float obj = new Float(12.345);**
- **Syntax of 3rd constructor: Long(String object)**
- **Example: Long obj = new Long("12.345");**

Float Class Methods	Purpose
int FloatObject1.compareTo(FloatObject2)	It compares two numerical values of Float class objects and returns 0, -ve value or +ve value.
boolean FloatObject1.equals(FloatObject2)	It compares two Float class objects. If same then returns true; otherwise false.
static float parseFloat(String Object)	It returns the float value of the string object contains float value.
String FloatObject.toString()	It converts Float class object into String object and returns that String object.
static Float valueOf(String object)	It converts a string object of float value to Float class object and return that object.

Double Class

- It wraps a value of the primitive type ***double*** in an object of ***Double*** class kind.
- It has two constructors.
- **Syntax of 1st constructor: Double(double value)**
- **Example: Double obj = new Double(12.3456);**
- **Syntax of 2nd constructor: Double(String object)**
- **Example: Double obj = new Double("12.3456");**

Double Class Methods	Purpose
int DoubleObject1.compareTo(DoubleObject2)	It compares two numerical values of Double class objects and returns 0, -ve value or +ve value.
boolean DoubleObject1.equals(DoubleObject2)	It compares two Double class objects. If same then returns true; otherwise false.
static double parseDouble(String Object)	It returns the double value of the string object contains double value.
String DoubleObject.toString()	It converts Double class object into String object and returns that String object.
static Double valueOf(String object)	It converts a string object of double value to Double class object and return that object.

Boolean Class

- It wraps a value of the primitive type ***boolean*** in an object of ***Boolean*** class kind.
- It has two constructors.
- **Syntax of 1st constructor: Boolean(boolean value)**
- **Example: Boolean obj = new Boolean(true);**
- **Syntax of 2nd constructor: Boolean(String object)**
- **Example: Boolean obj = new Boolean(“true”);**

Boolean Class Methods	Purpose
int BooleanObject1.compareTo(BooleanObject2)	It compares two boolean values of Boolean class objects and returns 0, -ve value or +ve value.
boolean BooleanObject1.equals(BooleanObject2)	It compares two Boolean class objects. If same then returns true; otherwise false.
static boolean parseBoolean(String Object)	It returns the boolean value of the string object contains boolean value.
String BooleanObject.toString()	It converts Boolean class object into String object and returns that String object.
static Boolean valueOf(String object)	It converts a string object of boolean value to Boolean class object and return that object.

Autoboxing

Conversions from a primitive type to the corresponding class type and vice-versa are called **boxing conversions**.

To convert a primitive type to the corresponding class type automatically is known as **autoboxing.**

Syntax:

ClassName ObjectRef = Primitive data/variable/expression;

Examples:

- Integer obj1 = 15;
- int a=10;
  Integer obj2 = a;
- Integer obj3 = a +15;

Unboxing

To convert from class type to a primitive data type automatically, is called as **unboxing**.

Syntax: PrimitiveDataType Variable = ObjectReference;

Examples:

- int num1 = obj1;
- int num2 = obj2;
- int num3 = obj3;

Here obj1, obj2, and obj3 are of Integer class kind.

Example-1 (Autoboxing and Unboxing):

```
class box_unbox
{
public static void main(String args[])
{
 // Autoboxing, which changes from primitive integer to
 // integer object
  Integer a=17;   // autoboxing

 // Unboxing, which changes from integer object to integer
   int b=a;   // unboxing
// int b=a.intValue();   // boxing
```

```
    Double  dobj = 6.78;   //  Autoboxing
    double d=dobj;          //  Unboxing

   System.out.println("Integer Object a=" + a);
   System.out.println("Integer Variable b=" + b);

   System.out.println("Double Object a=" + dobj);
   System.out.println("Double Variable d=" + d);
 }
}
```

Output

```
D:\JavaPro>java box_unbox
Integer Object a=17
Integer Variable b=17
Double Object a=6.78
Double Variable d=6.78
```

Example-2 (Autoboxing and Unboxing) :

```
class Autoboxing
{
public static void main(String []args)
{
int num[] = { 3, 97, 55, 22, 12345 };
// Array to store Integer objects
Integer obj[] = new Integer[num.length];

  // Call function to cause boxing conversions
  for(int i = 0 ; i<num.length ; i++)
  {
    obj[i] = Boxing(num[i]);
  }
```

```
 // Call function to cause unboxing conversions
  for(Integer Object : obj)
  {
    Unboxing(Object);
  }
}
  // Function to cause boxing conversion
  public static Integer Boxing(Integer Iobj)
  {
    return Iobj;
  }
  // Method to cause unboxing conversion
  public static void Unboxing(int n)
  {
    System.out.println("Integer Value = " + n);
  }
}
```

Output

```
D:\JavaPro>java Autoboxing
Integer Value  = 3
Integer Value  = 97
Integer Value  = 55
Integer Value  = 22
Integer Value  = 12345
```

Abstract Class

It is a class that contains zero or more abstract methods.

Syntax:

```
abstract  class className
{
    // declaration of abstract method(s)
}
```

Abstract Method

It is a method without method body. Its definition is to be given in the class where the abstract class is extended its features.

Syntax:

```
abstract  class className
{
 abstract returnType  methodName (Declaration of argument(s) if any);
}
```

```
// abstract class example
abstract class AbstractClass
{
   // abstract method
  abstract void Calculate(double a);
}

class Square extends AbstractClass
{
   void Calculate(double a)
   {
      System.out.println("Square of " + a + " = " + (a*a));
   }
}
```

```
class SquareRoot extends AbstractClass
{
   void Calculate(double a)
   {
      System.out.println("Square Root of " + a + " = " + Math.sqrt(a));
   }
}
class Cube extends AbstractClass
{
   void Calculate(double a)
   {
      System.out.println("Cube of " + a + " = " + (a*a*a));
   }
}
```

```
// main class
class DifferentCalculations
{
  public static void main (String args[])
  {
     Square s = new Square();
     SquareRoot sr = new SquareRoot();
     Cube c = new Cube();

     s.Calculate(12);
     sr.Calculate(625);
     c.Calculate(3);
   }
}
```

Output

```
D:\JavaPro>java DifferentCalculations
Square of 12.0 = 144.0
Square Root of 625.0 = 25.0
Cube of 3.0 = 27.0
```

Abstract Classes Vs. Interfaces

Abstract Classes	Interfaces
1. It is written when there is some common features shared by all the objects.	1. It is written when all the features are implemented differently by all the objects.
2. It is the duty of the programmer to provide sub-classes to it.	2. The programmer leaves the implementation part to the third party vendor.
3. It contains both abstract methods and concrete methods.	3. It only contains abstract methods.
4. It only contains instance variables.	4. It cannot contains instance variables.
5. All the abstract methods should be implemented in its sub-classes.	5. All the abstract methods should be implemented in its implementation classes.
6. A class is declared with the keyword abstract.	6. It is declared by using the keyword interface in place of class.

Packages

- Encapsulation is a technique in OOP language by which multiple related objects can be grouped under one object.

- Java implements encapsulation by the use of packages.

- A package is a collection of related classes and interfaces.

- It checks two things:

  - Reduce problems in name conflicts.
  - Control the visibility of classes, interfaces, methods, and data defined within them.

Reusability of Codes

- Generally reusability of code in a program by extending the classes and implementing the interfaces (Physically Copying).

- To use classes from other programs **without physically copying** them into the program. This can be achieved in **Java** by using ***packages***.

JAVA PACKAGES

JAVA API PACKAGES
(Application Programming Packages)

USER-DEFINED PACKAGES

i. java.lang: Language support classes which contains ***primitive data types***, ***strings***, ***math functions***, ***threads*** and ***exceptions***.

ii. java.io: Input/Output support classes which includes operation for ***input and output of data***.

iii. java.util: Language utility classes and interfaces which contains ***vectors***, ***random numbers***, ***date***, ***hash tables*** etc.

JAVA PACKAGES

JAVA API PACKAGES
(Application Programming Packages)

USER-DEFINED PACKAGES

iv. java.awt: awt means ***abstract window toolkit***. Contains graphical user interface (**GUI**) classes includes classes for ***windows***, ***buttons***, ***lists***, ***menus***, etc.

v. java.net: Classes for networking which include classes for communicating with ***local computers*** and ***internet servers***.

vi. java.applet: Classes for creation and implementation of ***applets***.

JAVA PACKAGES

JAVA API PACKAGES
(Application Programming Packages)

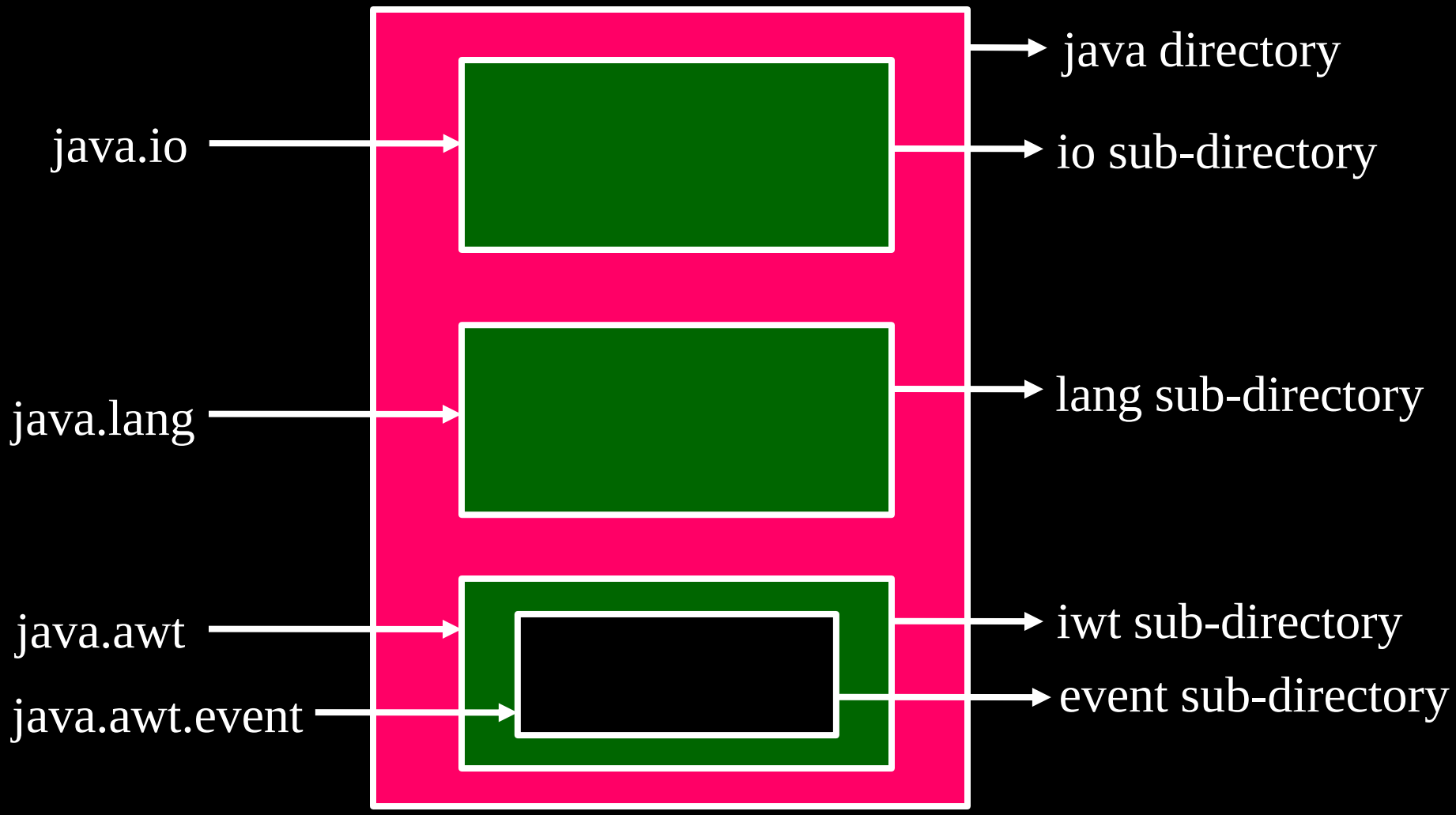
USER-DEFINED PACKAGES

vii. javax.swing: x means extended. It is the extended package of java.awt.

viii. java.text: It has classes **DateFormat** to format dates and times, and **NumberFormat** to format numeric values .

ix. java.sql: It helps to connect databases like Oracle or Sybase and retrieve data from it.

Java Package is a Directory



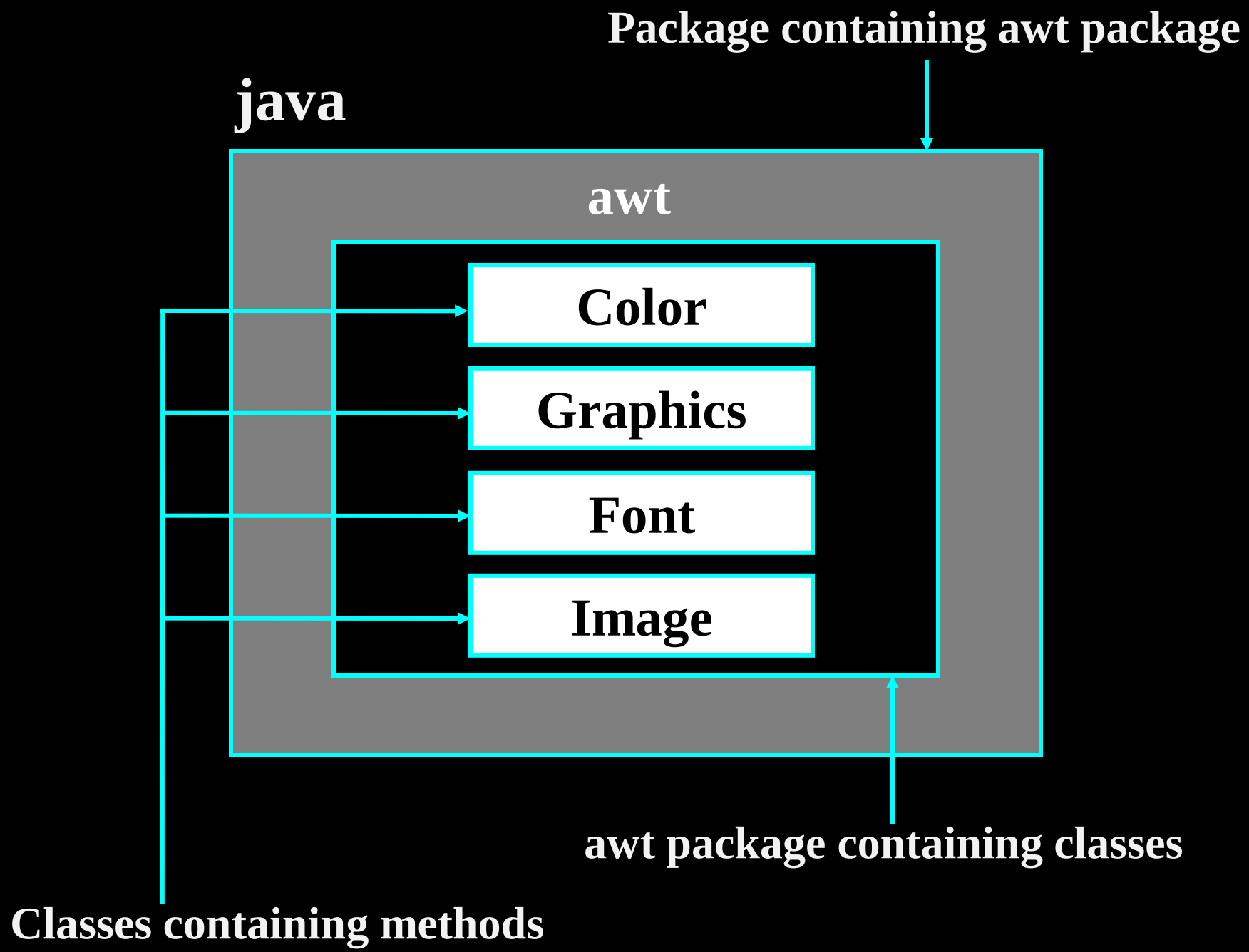
- To access Image class, the statement is **java.awt.Image**.

- To import in the program, it must appear at the top of the program before any class declaration. So, the statement is:

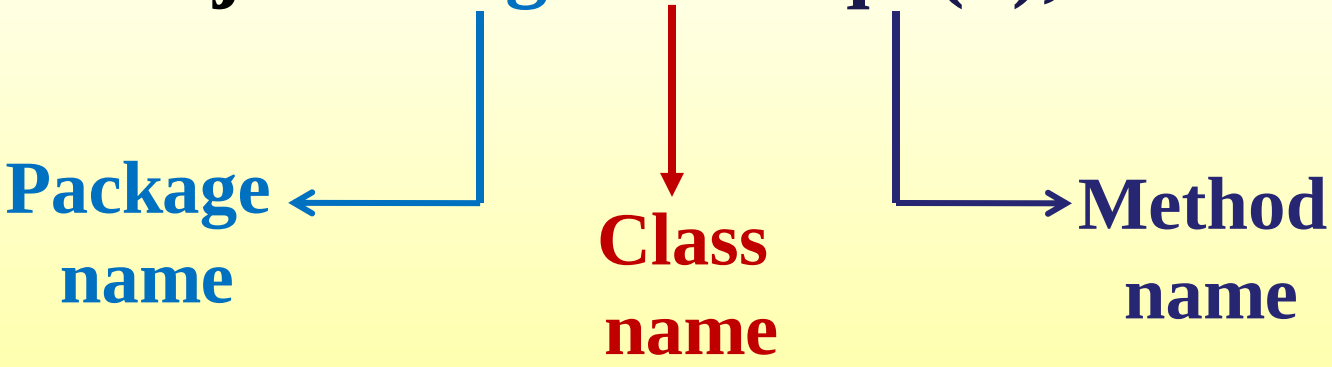
- **import java.awt.Image;**

- After it in the program any method belonging to ***Image class*** can be included directly by using **Image.method_name( );**

- The statement **import java.awt.*;** means inclusion of all classes of the package awt.


Package containing awt package
java
awt
Color
Graphics
Font
Image
awt package containing classes
Classes containing methods

double res = java.lang.Math.sqrt(n);



The above statement can also be written as:

```
import java.lang.Math;  // import java.lang.*;

double res = Math.sqrt(n);
```

USER-DEFINED PACKAGES

Syntax:

```
package package_name;  // package declaration
public class class_name
{
  // method(s) declarations;
 }
```

Example:

```
package mypackage;
 public class myclass
 {
   public void display( )
   {
     System.out.println(" My Package");
   }
  }
```

In the above package **mypackage** contains only one class called **myclass** which must be **public** kind.

The file name for saving must be the **class name** which is of public kind.

---Steps to follow for saving and compile a package---

- For the above example, the name of the file will be **myclass.java**.

- Then compile by using this **javac –d . Myclass.java**

- If no error then **myclass.class** will be created and saved under the package name as directory (here it is **mypackage**).

ACCESSING PACKAGE IN THE MAIN PROGRAM

In main program a package can be imported in this way:

import package_name.class_name;

OR

import package_name.*;

For the above package example:

import mypackage.myclass;

// import mypackage.*;

Note:

- If more than one classes are added in a package at a time, one class must be **public**. Remaining classes are declared as **non-public** kind.

- Save the program as **public class name** and store under the package (i.e. the sub-directory).

- After compilation it creates all classes class files.

- While importing the package in a program only **public class can access outside** but **not accessing non-public class**.

ACCESSING ALL CLASSES OF A PACKAGE IN A PROGRAM

- Declare at a time one class which must be public kind.

- To add one more class into the existing package, again declare a class of public kind and add to the sub-directory which is the package name.

- Then compile to create the class file.

ADDING CLASSES TO AN EXISTING PACKAGE

- Let us consider a package **page** with class **AA**:

```
package page;
public class AA
  {
    --------;
    --------;
  }
```

- Save as **AA.java** under the sub-directory called **page** (the package name).
- Then compile to get its class file as **AA.class**.

• To add another class called **BB** into the package **page**, then do the following:

```
package page;
public class BB
{
   --------;
   --------;
}
```

• Save as **BB.java** under the sub-directory called **page** (the package name). Then compile to get its class file as **BB.class**.

ADDING A PACKAGE IN AN EXISTING PACKAGE

Syntax:

package old_package_name.new_package_name;

Example: **package mypackage.pack1;**

Here **mypackage** is the existing one.

A new package called **pack1** is added in the above package and made as sub-package.

Interfaces in Package

- It is also possible to write interfaces in a package.

- The implementation classes must be defined in the package itself. Since the interface cannot create object.

Example:

```
// Create an interface dateDisplay in the package called pack

package pack;
public interface dateDisplay
{
    void showDate();
 }
```

```
// Create a class called dateImplement as implementation class of
// interface dateDisplay in the same package pack

package pack;
import pack.dateDisplay;

import java.util.*;

public class dateImplement implements dateDisplay
{
    public void showDate()
    {
        // default the object dd assigns with system date and time
        Date dd = new Date();
        System.out.print("The Current Date and Time = " + dd);
    }
}
```

```
// Write a program which uses implementation class
// dateImplement as for displaying date.

import pack.dateImplement;

class dateProgram
{
    public static void main(String args[])
    {
        dateImplement obj = new dateImplement();
        obj.showDate();
     }
 }
```

How to call garbage collector?

Ans: To call garbage collector of JVM for deletion of unused variables and unreferenced objects from memory using **gc( ) method**. This method is present in both runtime and system classes of java.lang package.

So garbage collector can be called as:

System.gc( );

Or

Runtime.getRuntime( ).gc( );

Exception Handling

- A wrong is known as exception.
- An exception may produce an incorrect output or may terminate the execution of the program abruptly or even causes the system to crash.
- There are two types of errors:
  - Compile-time errors
  - Run-time errors

Compile Time Error

- All syntax errors are detected and displayed by the java compiler known as compile-time errors.

- Some of the compile-time errors are:

(i) **Semicolon missing**.
(ii) **Misspelling of** identifiers and keywords.
(iii) **Missing of brackets** in classes and methods.
(iv) **Missing of double quotes** in string.
(v) Use of **undeclared variables**.
(vi) **Wrong initialization**.
(vii) **Bad references** to objects.
(viii) Use of = in place of == operator.

Example:

```
class ErrorProgram
{
public  static void main(String args[ ])
{
    System.out.println(“Java Program”)
 }
}
```

ErrorProgram.Java: 5 : ‘ ; ‘ expected

Run-Time Error

- Sometimes a program compiles successfully but may not run properly. This program gives wrong result due to wrong logic or may terminate due to errors such as stack overflow.
- Most common run-time errors are:

  (i) Dividing by 0.

  (ii) Accessing an element that is out of the bounds of an array.

  (iii) Trying to store a value into an array of an incompatible class or type.

  (iv) Passing invalid parameters for a method.

  (v) Converting invalid string to a number.

  (vi) Accessing a character that is out of bound of string etc.

Example:

```
class ErrorProgram1
{
   public class void main(String args[ ])
   {
      int a=10, b=4, c=4;
      int res = a/(b-c);   // dividing by zero
      System.out.println("Result = " + res);
    }
}
```

The error is / (divided) by zero.

Java.lang.ArithmeticException

After generating the error, the program stops.

The purpose of exception handling is to produce a means to detect and report an exceptional circumstance. So that appropriate action can be taken.

The following tasks are to be followed:

(i) Find the problem (hit the exception).
(ii) Detect the error (throw the exception).
(iii) Receive the error (catch the exception).
(iv)Take correct action (handle the exception).

Common Java Exception

- **ArithmeticException : Caused by math errors such as division by zero.**
- **ArrayIndexOutOfBoundsException : Caused by bad array indices.**
- **FileNotFoundException : Non-existence of file.**
- **IOException : General I/O failures, inability to read from a file.**
- **NumberFormatException : Caused when conversion between string and number fails.**

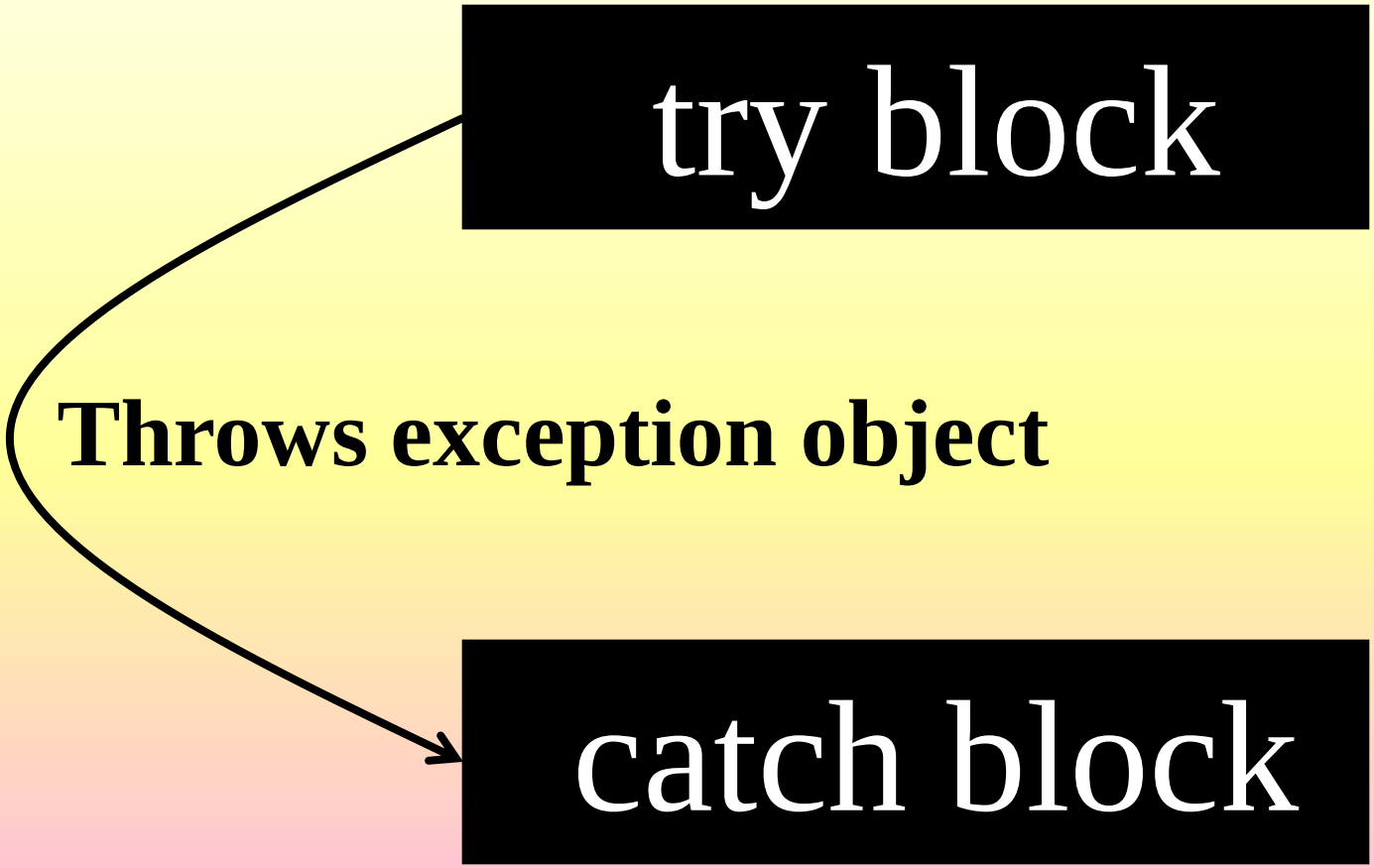
Common Java Exception

OutOfMemoryException : There is not enough memory to allocate a new object.

StringIndexOutOfBoundsException : Attempts to access non existent character position in a string.

StackOverflowException : When the system runs out of stack space.

ArrayStoreException : When an array attempts to store different data type.


try block
Throws exception object
catch block

Syntax:

```
try
{
    statement (s);
  }

catch (Exception_type  object_name)
 {
      statement (s);
 }
```

- **The try block could generate the exception. If any exception occurs then the control jumps to the catch block by skipping the remaining statements of the block.**

- **The catch block receives the exception and perform the statements present in it.**

- **The catch block always appear after the try block.**

- **The catch block passes single parameter which is reference to the exception object thrown by the try block.**

```
class error1
 {
public static void main (String args [ ])
 {
    int a = 10, b = 4, c = 4;
    int p, q;
     try
     {
       p = a / ( b – c );  // exception here
     }
```

```
catch (ArithmeticException e)
{
 System.out.println("Division by zero");
}
     q = a/(b+c);
     System.out.println("Q=" + q);
   }
 }
```

Multiple Catch Block

- We can have more than one catch block in a program.
- Each catch block for a particular exception.
- These catch blocks are written after the try block.
- Syntax:

```
try
  {
     statement (s);
   }
catch (exception_type1  object1)
{
   statement (s);
}
catch (exception_type2  object2)
{
  statement (s);
}
```

```
// Read two numbers. Find its division using exception
// handling method.
import java.io.*;
import java.util.*;

class Division_Exception
{
  public static void main (String args [ ])
  {
    DataInputStream in = new DataInputStream(System.in);
     int a, b, c;
```

```
try
  {
    System.out.print("Enter two numbers :");
    a = Integer.parseInt(in.readLine());
    b = Integer.parseInt(in.readLine());
    c = a/b;
    System.out.println ("Division = " + c);
   }
```

```
 // number input exception
catch (IOException ie)
{
    System.out.println("Wrong Data Input");
}
 // arithmetic division exception
catch (ArithmeticException e)
{
    System.out.println("Division by zero");
}
 }
}
```

Output

```
D:\JavaPro>java Division_Exception
Enter two numbers :12
3.0
Exception in thread "main" java.lang.NumberFormatException: For input string: "3
.0"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at Division_Exception.main(Division_Exception.java:14)

D:\JavaPro>java Division_Exception
Enter two numbers :12
3
Division = 4

D:\JavaPro>java Division_Exception
Enter two numbers :12
0
Division by zero
```

```
// With exception in a command line programming

class Arg_Add2
{
  public static void main (String args[ ])
  {
      int sum = 0;
     try
    {
        for ( String st : args )
        {
         sum += Integer.parseInt(st);
        }
     }
```

```
catch (NumberFormatException  object)
{
     System.out.println(" [ " + object + " ] is not an integer " +
" and will not be included in the sum.");
}
  System.out.println("Sum = " + sum);
  }
}
```

Output

```
D:\JavaPro>javac Arg_Add2.java

D:\JavaPro>java Arg_Add2 5 10 12 16 9
Sum = 52

D:\JavaPro>java Arg_Add2 5 10 12 16 9.7
 [ java.lang.NumberFormatException: For input string: "9.7" ] is not an integer
 and will not be included in the sum.
Sum = 43
```

```
// With exception in a command line programming
class Arg_Add3
{
  public static void main (String args[ ])
  {
      int sum = 0;

      for ( String st : args )
      {
                  try
                  {
                     sum += Integer.parseInt(st);
                   }
```

```
  catch (NumberFormatException  object)
  {
   System.out.println(" [ " + st + " ] is not an integer " + "
and will not be included in the sum.");
  }
}
      System.out.println("Sum = " + sum);
   }
}
```

Output

```
D:\JavaPro>javac Arg_Add3.java

D:\JavaPro>java Arg_Add3 4 6 12 15 7
Sum = 44

D:\JavaPro>java Arg_Add3 4 6 12.7 15 7
 [ 12.7 ] is not an integer  and will not be included in the sum.
Sum = 32
```

finally statement

It is used to handle an exception that is not caught by any of the previous catch statement.

Syntax-1:

```
try
{
    statement (s);
 }
finally
{
    statement (s);
}
catch (exception_type e)
{
    statement (s);
}
```

Syntax-2:

```
try
{
    statement (s);
 }
 catch (exception_type e)
 {
    statement (s);
 }
finally
{
    statement (s);
 }
```

```
// Example using finally
class FinallyExample
   {
  public static void main (String args[ ])
    {
     int i = 0;
    String greetings[ ] ={"Hello world", "I mean it",
                                                    "HELLO WORLD" };
    while (i < 4)
   {
        try
       {
            System.out.println (greetings[i]);
       }
```

```
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(“Array out of index");
}
finally
{
   System.out.println("This is always printed");
}
   i++;
  }  // while close
 }
}
```

Output

```
D:\JavaPro>javac FinallyExample.java

D:\JavaPro>java FinallyExample
Hello world
This is always printed
I mean it
This is always printed
HELLO WORLD
This is always printed
Array out of index
This is always printed
```

```
// Example of exception handling in command-line
// argument programming
class command_exception
{
  public static void main (String args[ ])
  {
      int wrong = 0, num, valid = 0;
      for ( int i=0; i<args.length; i++)
      {
         try
         {
            num = Integer.parseInt (args [i]);
          }
```

```
 catch (NumberFormatException ne)
 {
System.out.println (" Wrong Number Format = " + args[i]);
  wrong++;
  continue;
 }
    valid++;
}
  System.out.println ("Valid Numbers = " + valid);
  System.out.println (" Invalid Numbers = " + wrong);
  System.out.println (" Total Inputs = " + args.length);
 }
}
```

Output

```
D:\JavaPro>javac command_exception.java

D:\JavaPro>java command_exception 12 8 7.8 4.0f 20 30 0.7
Wrong Number Format = 7.8
Wrong Number Format = 4.0f
Wrong Number Format = 0.7
Valid Numbers = 4
Invalid Numbers = 3
Total Inputs = 7
```

User-Defined Exception

To throw our own exception then it can be created by using the keyword throw.

Syntax:

throw new your_own_exception_class_name;

Example

```
import java.lang.Exception;
class ownException extends Exception
{
    ownException (String m )
   {
      super (m);
    }
}
class Test_Myexception
{
   public static void main (String args[ ])
   {
```

```
try
{
  int a = 5, b = 10;
  if (a < b)
  {
    throw new ownException
 (“First number is smaller than 2nd number”);
  }
}
```

```
catch ( own_exception  e )
{
  System.out.println (“Caught my exception”);
  System.out.println (e.getMessage());
}
}
}
```

Note: **getMessage()** is a method belongs to **Exception** class, which shows the message residing in the object of **own_exception** class kind (in this example).

First number is smaller than 2nd number

new allocates an object of **ownException** kind and its reference is thrown to the exception handler **catch( )** and the object **e** assigns with the reference (here it is 100).

Output

```
Caught my exception

First number is smaller than 2nd number
```

Using user-defined exception handling, read numbers from the user and display +ve / -ve/ zero.

```
import java.lang.*;
import java.io.*;
class positive extends Exception
{
  positive(String st)
 {
    super (st);
  }
}
```

```
class negative extends Exception
{
    negative(String st)
    {
       super (st);
    }
}
```

```
class zero extends Exception
{
    zero(String st)
    {
      super (st);
    }
}
```

```
class multi_exception
{
  public static void main( String args[ ]) throws IOException
  {
  DataInputStream in = new DataInputStream(System.in);

  int n=0;
  String a = "yes";

  do
  {
```

```
try
{
    System.out.print ("Enter a number…. ");
    n = Integer.parseInt(in.readLine());
    if (n>0)  {  throw new positive("+ve");  }
    if (n<0)  {  throw  new  negative("-ve"”); }
    if (n==0)  { throw  new  zero("Zero");  }
}
catch (positive p)
{
  System.out.println (p.getMessage());
}
```

```
catch (negative n)
{
  System.out.print (n.getMessage());
}
catch (zero z)
{
  System.out.print (z.getMessage());
}
System.out.print ("Do you want to Continue[yes/No]?");
a = in.readLine();
} while(a.equalsIgnoreCase s("yes"));
}
}
```

Output

```
D:\JavaPro>java multi_exception
Enter a number ........ -8
Negative
Do you want to Continue [yes/No] ? Yes
Enter a number ........ 5
Positive
Do you want to Continue [yes/No] ? YES
Enter a number ........ 0
Zero
Do you want to Continue [yes/No] ? NO
```

Multithreaded Programming

Definition:

Running several programs simultaneously is known as multitasking.

In system technology, it is called **multithreading.**

Multitasking is of two types:

i. Process-based: In this multi tasking, **several programs** are executed at a time by the microprocessor.

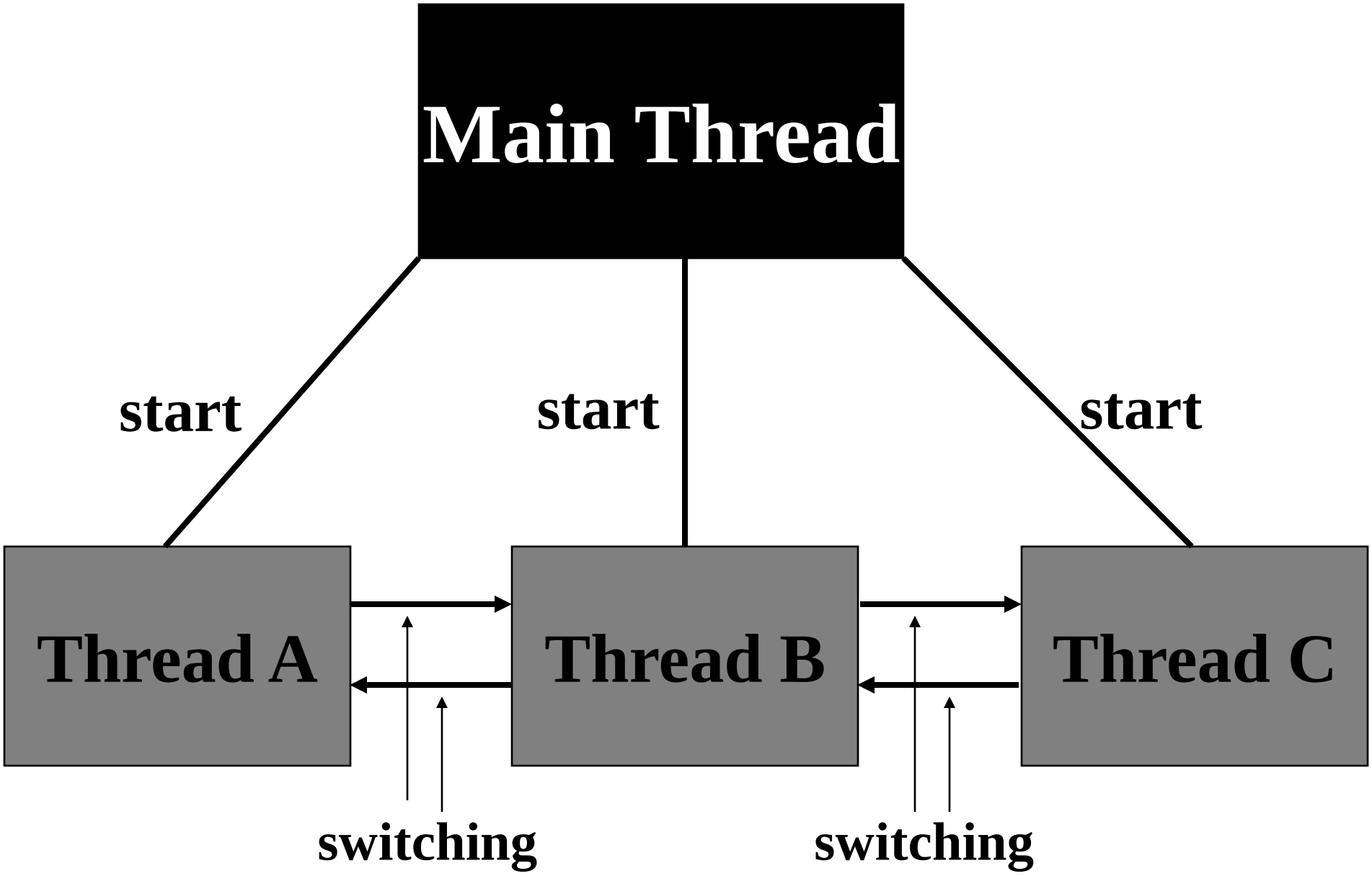
ii. Thread-based: In this multi tasking, **several parts of the same program** is executed at a time by the microprocessor.

- Java enables multiple flow of control.

- Each flow of control can be considered as a small program(or module) known as a **thread**.

- **Each thread** runs in parallel to others.

- Program containing **multiple flows of control** is known as **multithreaded program**.


Main Thread
start
start
start
Thread A
Thread B
Thread C
switching
switching

Uses of Threads

- It is mainly used in server-side programs to serve the needs of **multiple clients** on a **network** or **internet**. On internet, a server machine has to cater the needs of thousands of clients at a time.
- It is also used to create games and animation.

Creating a Thread

Create a class that **extends Thread class** or **implements Runnable interface**.

Both are belonged to the **java.lang** package.

Syntax of thread class using Thread extends

```
class class_name extends Thread
{
   public void run ( )
   {
   //statement(s) for implementing thread
   }
 //  statement(s);
}
```

Syntax of thread class using Runnable interface

```
Class  class_name  implements Runnable
{
    public void run ( )
    {
    // statement(s) for implementing thread
     }
  //  statement(s);
}
```

Running a Thread

Syntax

```
class_name  object_name  = new   class_name( );
object_name.start( );
```

OR

```
new   class_name.start( );
```

A thread is born with no name and brings to **runnable** state with the help of start( ) and moving to **running** state.

The thread invokes **run()** because it is inherited by the class **class_name**.

Example-1

```
// Write a program to create a thread and run it.

 class mythread  extends Thread // own thread class
{
    public void run()
    {
       for( int i=1; i<=100; i++)
          System.out.print( i + “  “);
     }
}
```

```
// main class
class thread_demo
{
 public static void main(String args[ ])
{
   mythread obj = new mythread();
   obj.start();
}
}
```

OR

```
// main class
class thread_demo
{
 public static void main(String args[ ])
{
   mythread obj = new mythread();
   Thread th = new Thread(obj);
   th.start();
}
}
```

Example-2

```
// Create three threads and run it.
class mythread1 extends Thread
{
    public void run()
    {
      System.out.println("1st Thread");
    }
  }
```

```
class mythread2 extends Thread
{
    public void run()
    {
      System.out.println("2nd Thread");
    }
  }
```

```
class mythread3 extends Thread
{
    public void run()
    {
      System.out.println("3rd Thread");
    }
  }
```

```
class three_threads   // main class
{
public static void main(String args[ ])
{
   mythread1 obj1 = new mythread1( );
   mythread2 obj2 = new mythread2( );
   mythread3 obj3 = new mythread3( );
   obj1.start( );
   obj2.start( );
   obj3.start( );
 }
}
```

OR

```
class three_threads    // main class
{
 public static void main(String args[ ])
 {
    new mythread1().start();
    new mythread2().start();
    new mythread3().start();
 }
}
```

Terminating the Thread

- A thread will terminate automatically when it comes **out of run( ) method**.
- To terminate a thread from the middle, the statement **return** is used with a **proper condition**.
- The **condition** should be placed in the **run( ) method**.
- Syntax:

```
public void run()
{
    // statement (s);
    if (condition) return;
}
```

```
// Create a thread , run it and stop by pressing enter key.
 import java.io.*;
class mythread extends Thread
{
        boolen stop = false;
        public void run()
        {
           for( int i=1; i<=100000; i++)
           {
              System.out.print( i + “  “);
              if (stop) return;
                }
       }
}
```

```
class thread_demo     // main class
{
public static void main(String args[ ])
{
    mythread obj = new mythread();
    obj.start();
    System.in.read();  // to come out of  thread press enter key
    obj.stop=true;
 }
}
```

Write a loop in thread-1. Read two numbers and swap it in thread-2.

```
import java.io.*;
class thread1  extends Thread
{
    public void run()
    {
      for( int i=1; i<=5; i++)
         System.out.println( “Thread One“);
      }
}
```

```
class thread2  extends Thread
{
     int a=0, b=0;
    public void run( )
    {
     read( );
     System.out.println("A="+a+"B="+b);
     int c = a;
     a = b;
     b = c;
     System.out.println("A="+a+"B="+b);
     }
```

```
void read( )
{
try
{
DataInputStream in = new DataInputStream(System.in);
System.out.print("Enter Two numbers: ");
a = Integer.parseInt(in.readLine( ));
b = Integer.parseInt(in.readLine( ));
}
catch(Exception e)
{                              }
}
}
```

```
class main_thread_program   // main class
{
 public static void main(String args[ ])
 {
   thread1 obj1 = new thread1();
   thread2 obj2 = new thread2();
   obj1.start();
   obj2.start();
 }
}
```

Output

```
Thread One
Thread One
Thread One
Thread One
Enter 2 numbers: Thread One
5
10
A=5 B=10
A=10 B=5
```

Or

```
Thread One
Thread One
Thread One
Thread One
Thread One
Enter 2 numbers: 6
9
A=6 B=9
A=9 B=6
```

Program to execute multiple task using one thread.

```
class my_thread_class implements Runnable
{
public void run()
{
    job1( );
    job2( );
    job3( );
}
```

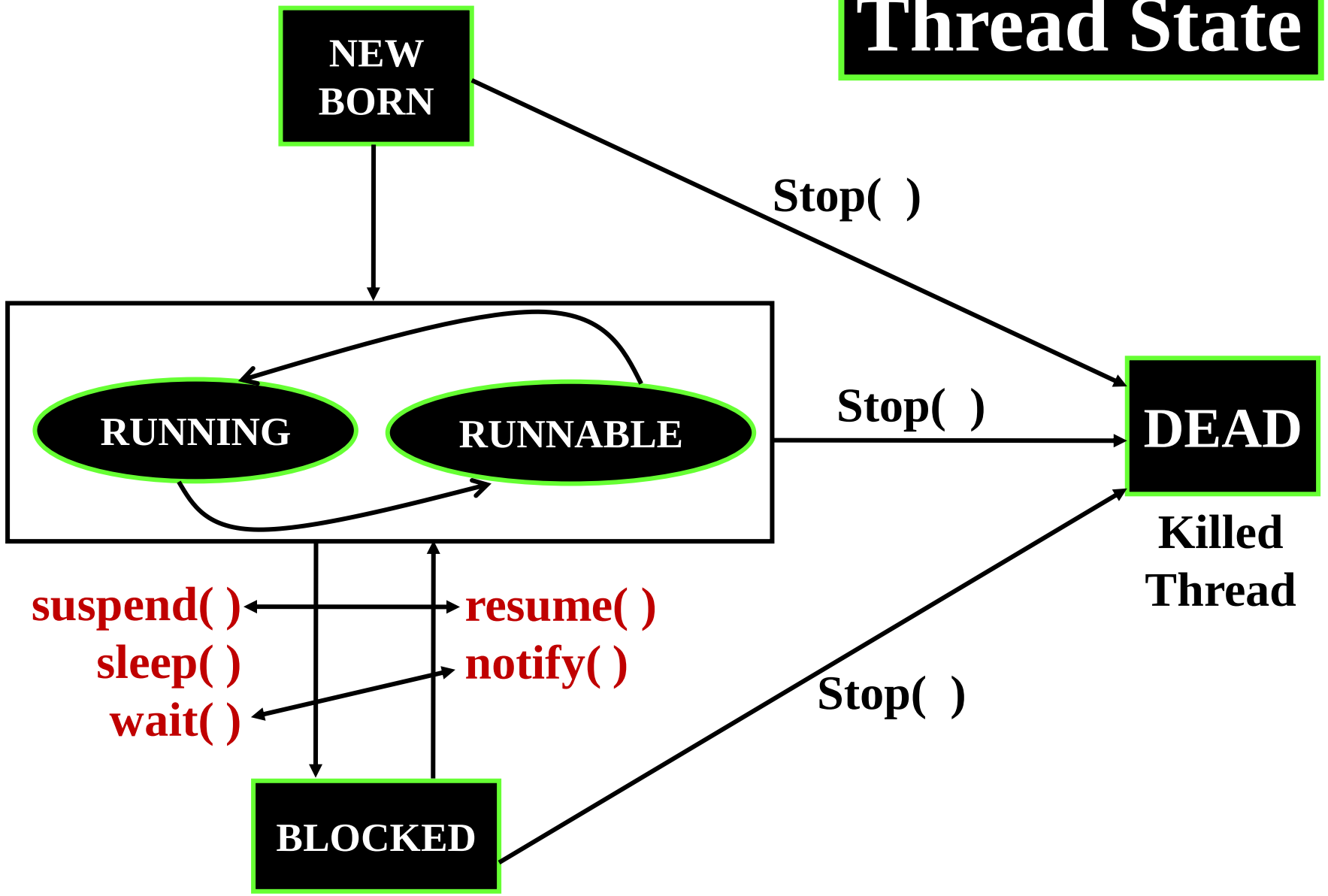
```
void job1( )
{
   System.out.println (“Job No. 1”);
}
void job2( )
{
   System.out.println (“Job No. 2”);
}
void job3( )
{
   System.out.println (“Job No. 3”);
}
}
```

```
class main_class
{
public static void main(String args[ ])
{
// create an object of  my_thread_class kind
 my_thread_class  obj = new my_thread_class( );
  // create a thread and link to the above object
 Thread t = new Thread(obj);

//execute the thread t on the above object's run( ) method
  t.start();
  }
}
```

Output

```
Job No. 1
Job No. 2
Job No. 3
```

Thread State



Idle thread or non-runnable thread

i. **stop ( ):** To stop a thread which moves to the dead state.

Syntax: **object_name.stop( )**

Blocked or non-runnable state of a thread

ii. sleep( ) : To block a thread for a specified time.

Syntax:

object_name.sleep (*int* time_in_milliseconds)

iii. suspend( ): To suspend a thread until further orders.

Syntax: **object_name.suspend( )**

iv. wait( ): To block a thread until certain condition occurs.

Syntax: object_name.wait( )

v. resume( ): It s used to get back to the running state if the thread is in **suspend ( )** state.

vi. notify ( ): It is used in case of the thread is in **wait( )** state.

Method Name	Purpose
getName()	To get the name of the thread
getPriority()	To get the priority of the thread
isAlive()	Determine if a thread is still running
join()	Wait for a thread to terminate or die
run()	Entry point for the thread
sleep()	Suspend a thread for a period of time
start()	Start a thread by calling its run method
setName()	To change or set the name of the method

```
// Display the maximum, minimum, and normal priority of
// main thread.

class Priority
{
public static void main(String args[ ])
{
 Thread t = Thread.currentThread( );

System.out.println("Priority of Main Thread:“ +
                                        t.getPriority( ));
```

```
System.out.println ("Normal Priority of  Main Thread:" +
t.NORM_PRIORITY);

System.out.println ("Minimum Priority of  Main Thread:" +
t.MIN_PRIORITY);

System.out.println ("Maximum Priority of  Main Thread:" +
t.MAX_PRIORITY);
  }
}
```

Output

```
Priority of Main Thread: 5

Normal Priority of  Main Thread: 5

Minimum Priority of Main Thread: 1

Maximum Priority of Main Thread: 10
```

```
// set and get the priorities of threads
class child extends Thread
{
    child(String n)
    {
       super (n);
      System.out.println("Thread Name: " + this);
     }
}
class Priority1
{
public static void main(String args[ ])
{
```

```
Thread t = Thread.currentThread( );
child T1 = new child("ONE");
child T2 = new child("TWO");
child T3 = new child("THREE");

System.out.println("Priority of Main Thread:" +
                                        t.getPriority( ));
System.out.println("Priority of Thread " + T1.getName()
                                + "=" + T1.getPriority( ));
System.out.println("Priority of Thread " + T2.getName()
                                  +"="+ T2.getPriority( ));
System.out.println("Priority of Thread " + T3.getName()
                                + "=" + T3.getPriority( ));
```

```
t.setPriority(Thread.MAX_PRIORITY);

T1.setPriority(Thread.MAX_PRIORITY-3);

T2.setPriority(Thread.MIN_PRIORITY+3);

T3.setPriority(Thread.MAX_PRIORITY-
                                    Thread.MIN_PRIORITY);

System.out.println("After set new priority to all 4 threads.");
```

```
t.setPriority(Thread.MAX_PRIORITY);

System.out.println("Priority of  Main Thread:" +
                                        t.getPriority());
System.out.println("Priority of Thread " +
                    T1.getName() + "=" +T1.getPriority());
System.out.println("Priority of Thread"  +
                    T2.getName()+"="+T2.getPriority());
System.out.println("Priority of Thread"  +
                    T3.getName()+"="+T3.getPriority());
  }
}
```

Output

```
Thread Name: Thread [ ONE , 5, main]
Thread Name: Thread [ TWO , 5, main]
Thread Name: Thread [ THREE , 5, main]
Priority of Main Thread: 5
Priority of Thread ONE : 5
Priority of Thread TWO  : 5
Priority of Thread THREE : 5
After set new priority to all 4 threads.
Priority of Main Thread: 10
Priority of Thread ONE : 7
Priority of Thread TWO  : 4
Priority of Thread THREE : 9
```

Thread Priorities

✓ Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

✓ Higher-priority threads get more CPU time than lower priority threads.

✓ Threads of equal priority should get equal access to the CPU.

✓ To set a thread's priority, use the **setPriority( )** method, which is a member of Thread.

The general form: **final void setPriority(int *level*)**

✓ Here, ***level*** specifies the new priority setting for the calling thread. The value of ***level*** must be within the range **MIN_PRIORITY** and MAX_PRIORITY. Currently, these values are **1** and **10**, respectively.

✓ To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **final variables** within Thread.

✓ You can obtain the current priority setting by calling the **getPriority( )** method of Thread.

The general form: **final int getPriority( )**

- ✓ One thread is set two levels above the normal priority, as defined by **Thread.NORM_PRIORITY**, and the other is set to two levels below it.
- ✓ The threads are started and allowed to run for **ten seconds**.
- ✓ Each thread executes a loop, counting the number of iterations.
- ✓ After ten seconds, the main thread stops both threads.
- ✓ The number of times that each thread made it through the loop is then displayed

```
// thread priorities program
import java.lang.*;
class thread_class extends Thread
{  int count = 0;
    public void run( )
    {
        for( int i=1; i<=10000; i++)
                count++;
        System.out.println("Completed Thread :" +
                        Thread.currentThread( ).getName( ));
        System.out.println("Its Priority:" +
                        Thread.currentThread( ).getPriority( ));
     }
}
```

```
class Priority2
{
public static void main(String args[ ])
{  thread_class obj = new thread_class( );
   Thread t1 = new Thread(obj, "One");
   Thread t2 = new Thread(obj, "Two");
   // set priority
   t1.setPriority(2);
   t2.setPriority(Thread.NORM_PRIORITY);
    // start first t1 and then t2
    t1.start( );
    t2.start( );
  }
}
```

1st Run

Completed Thread : One
Completed Thread : Two
Its Priority : 5
Its Priority : 2

2nd Run

Completed Thread : One
Its Priority : 2
Completed Thread : Two
Its Priority : 5

```
// Program for multi-tasking using threads
class my_thread implements Runnable
{
    String st;
    my_thread(String str)
    {
        this.st = str;
     // or st = str;
     }
     public void run()
     {
        for(int i=1; i<=4; i++)
          {
             System.out.println(st + " : " + i);
```

```
try
{
   //suspend execution for 2000 milliseconds
   // i.e. 1000 milliseconds = 1 second
   Thread.sleep (2000);
}
   catch(InterruptedException ie)
   {
        ie.printStackTrace();
    }
  } // loop close
 } // run() close
}
```

```
class cinema
{
public static void main(String args[ ])
{
my_thread obj1 = new my_thread("Get the Ticket");
my_thread obj2 = new my_thread("Show the Seat");

    Thread t1 = new Thread(obj1);
    Thread t2 = new Thread(obj2);
    t1.start( );
    t2.start( );
}
}
```

Output

```
Get the Ticket : 1
Show the Seat : 1
Get the Ticket : 2
Show the Seat : 2
Get the Ticket : 3
Show the Seat : 3
Get the Ticket : 4
Show the Seat : 4
```

```
// multiple threads acting on single object
class reservation implements Runnable
{
    int avail = 2;
    int wanted;
    reservation(int a)
      {
         wanted = a;
      }
    public void run()
    {
System.out.println("Available Seats =" + avail);
```

```
if(avail>=wanted)
{
String name = Thread.currentThread().getName();
System.out.println(wanted +  " Seat Reserved for " + name);
 try
    {
      Thread.sleep(1500);
      avail = avail - wanted;
     }
 catch(InterruptedException ie) {     }
 }
else
  System.out.println("Sorry, No seats Available");
 }
}
```

```
class unsafe
{    public static void main(String args[ ])
     {   reservation obj = new reservation(1);
       Thread t1 = new Thread(obj);
       Thread t2 = new Thread(obj);
       Thread t3 = new Thread(obj);
       t1.setName("Laxmi");
       t2.setName("Ram");
       t3.setName("Swati");
       t1.start();
       t2.start();
       t3.start();
     }
   }
```

Available Seats = 2
2 Seat Reserved for Laxmi
Available Seats = 2
2 Seat Reserved for Ram
Available Seats = 2
2 Seat Reserved for Swati

Note: Here all the three threads t1, t2, t3 act on the same object obj. The available seats never be updated after allotting to a particular person. This kind of situation is known as Thread Synchronization or Thread unsafe.

Thread Synchronization

When a **thread** is already acting on an object preventing any other thread from acting on the same object is called **thread synchronization**.

The **object** on which the threads are synchronized is called **synchronized object**.

Synchronized object is like a locked object, locked on a thread.

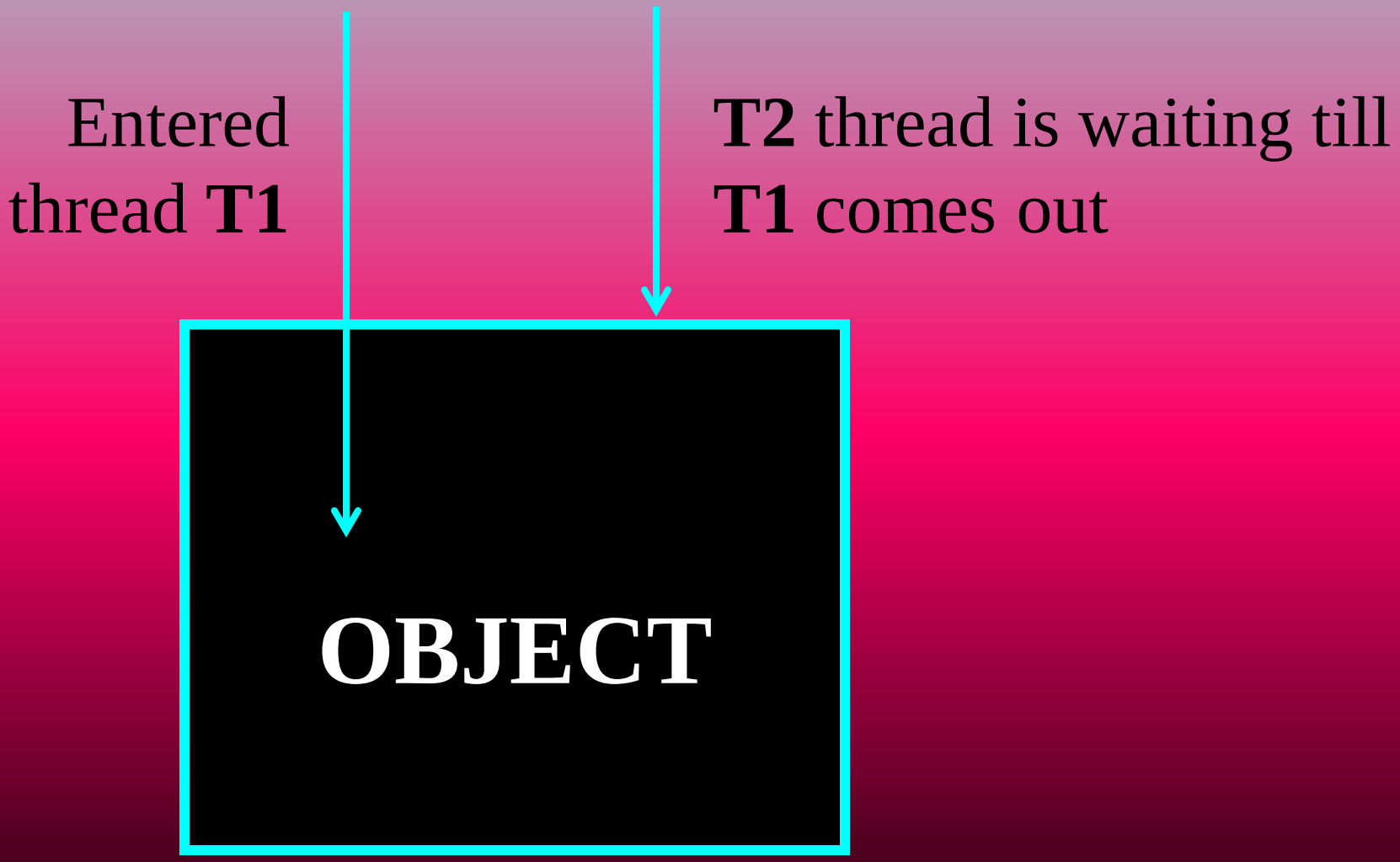
Example:

If a room has one door and a person entered into the room and locked. The second person who wants to enter the room should wait till the first person opened the door and comes out.

The same way an thread also locks the object after entering it. Then the next thread cannot enter it till the 1st thread comes out.

It means the object is locked mutually on threads. So this object is called ***mutex*** **(mutually exclusive lock)**.

Thread Synchronization



The **OBJECT** is called **MUTEX**

Two different ways an object can be synchronized.

i. Using synchronized method:

A group of statements of the object are kept in a synchronized block. That is these statements are kept inside run() method.

Syntax:
```
void run( )
{
   synchronized (object_name)
  {
     // statement(s);
  }
 }
```

Here, ***object_name*** represents the object to be locked or synchronized.

The statement(s) inside the synchronized block are available to only ***one thread at a time***.

ii. Using synchronized keyword:

An entire method can be synchronized by using the keyword **synchronized**.

Syntax:

```
synchronized void method_name( )
{
   // statement (s);
}
```

Here the statement(s) present inside **method_name( )** are available to only one thread at a time.

```
// Program for multiple threads acting on single
// object using synchronization.
class reservation implements Runnable
{
    int avail = 2;
    int wanted;
    reservation(int a)
    {
      wanted = a;
    }
    public void run( )
    {
```

```
synchronized (this)
{
  System.out.println("Available Seats =" + avail);
  if(avail>=wanted)
  {
String name = Thread.currentThread().getName();
System.out.println(wanted + " Seat Reserved for " +
                                                name);
  try
  {
     Thread.sleep(1500);
     avail = avail - wanted;
   }
```

```
catch(InterruptedException ie)
{       }

}
else
     System.out.println("Sorry, No seats Available");

    } // end of synchronized block
  }
}
```

```
class safe
{    public static void main(String args[])
     {   reservation obj = new reservation(1);
         Thread t1 = new Thread(obj);
         Thread t2 = new Thread(obj);
         Thread t3 = new Thread(obj);
          t1.setName("Kumar");
          t2.setName(“Smiley");
          t3.setName("Rani");
          t1.start();
          t2.start();
          t3.start();
  }
}
```

```
Available Seats = 2
1 Seat Reserved for Kumar
Available Seats = 1
1 Seat Reserved for Smiley
Available Seats = 0
Sorry, No seats Available
```

Output

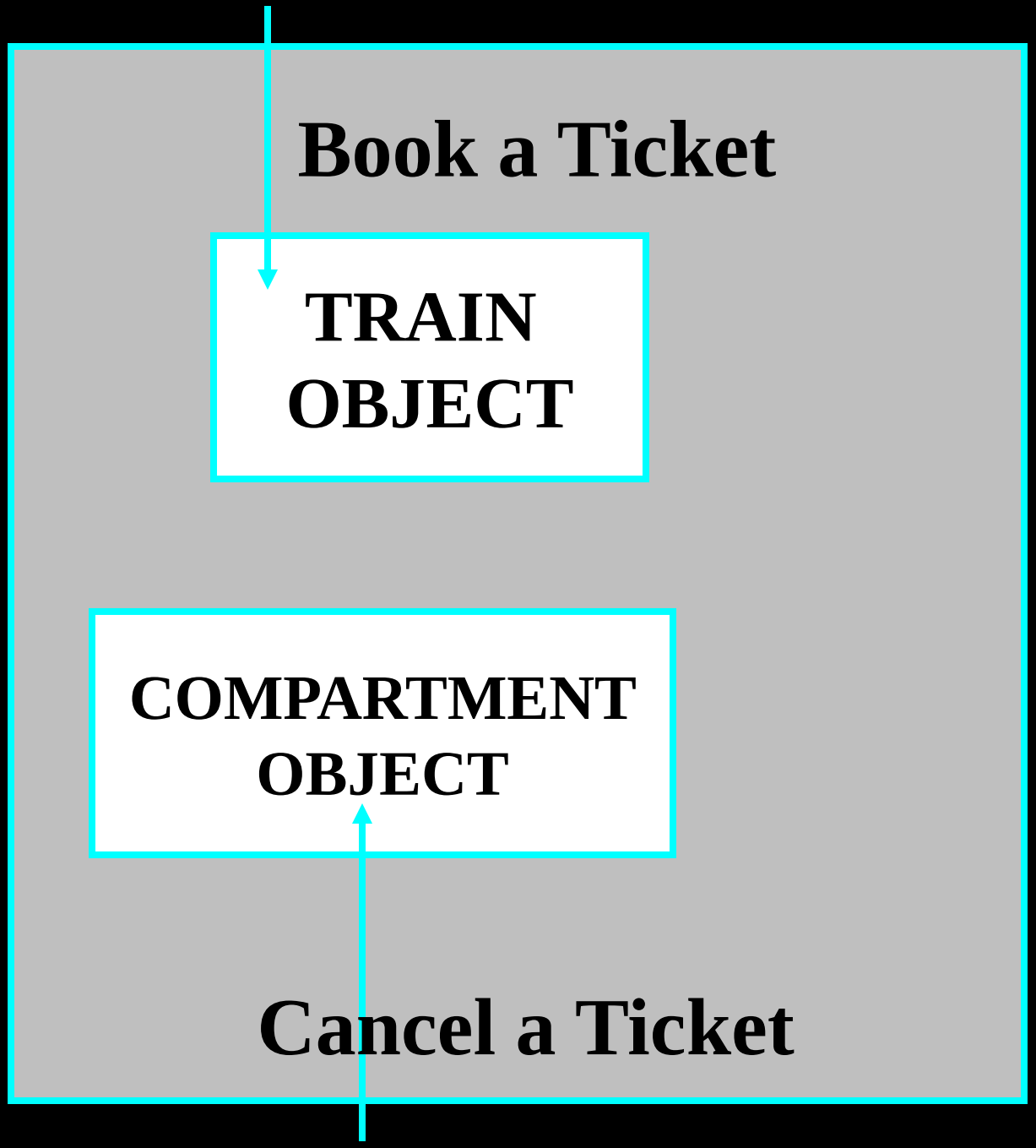
Output

```
Available Seats = 2
1 Seat Reserved for Kumar
Available Seats = 1
1 Seat Reserved for Rani
Available Seats = 0
Sorry, No seats Available
```

Thread Deadlock

When a thread has locked an object and waiting for another object to be released by another thread, and the other thread is also waiting for the first thread to release the first object.

So both the threads will continue waiting forever. This is called ***Thread Deadlock***.


Book a Ticket
TRAIN OBJECT
COMPARTMENT OBJECT
Cancel a Ticket

Thread Group

- Several threads can be grouped together under a common name.

- It can control the grouped threads by using the thread group name.

Syntax:

ThreadGroup tg = new **ThreadGroup**("group_name");

Example:

```
Thread t;
ThreadGroup tg = new  ThreadGroup ("Hello");
t = new Thread (tg, targetobject,  "threadname");
```

Here the thread **t** belongs to the thread group **tg** and acts on **targetobject**, which is the target object for the thread.

Adding a new thread group in an Existing thread group:

ThreadGroup **tg1** = new ThreadGroup(**tg**, "groupname");

Here a new thread group called **tg1** is created and added to the existing thread group **tg**.

ThreadGroup tg.getParent( ): It returns the parent of a thread or thread group .

t.getThreadGroup( ): It returns the parent thread group of a thread.

tg.activeCount( ): To know the number of threads actively running in the thread group.

tg.setMaxPriority( ): To change the maximum priority of a thread group.