Java Programming

JVM (Java Virtual Machine) Architecture

JVM (Java Virtual Machine) is an abstract machine.

It is a specification that provides runtime environment in which java bytecode can be executed.

What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

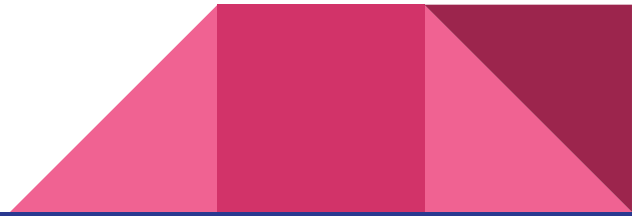
What it does

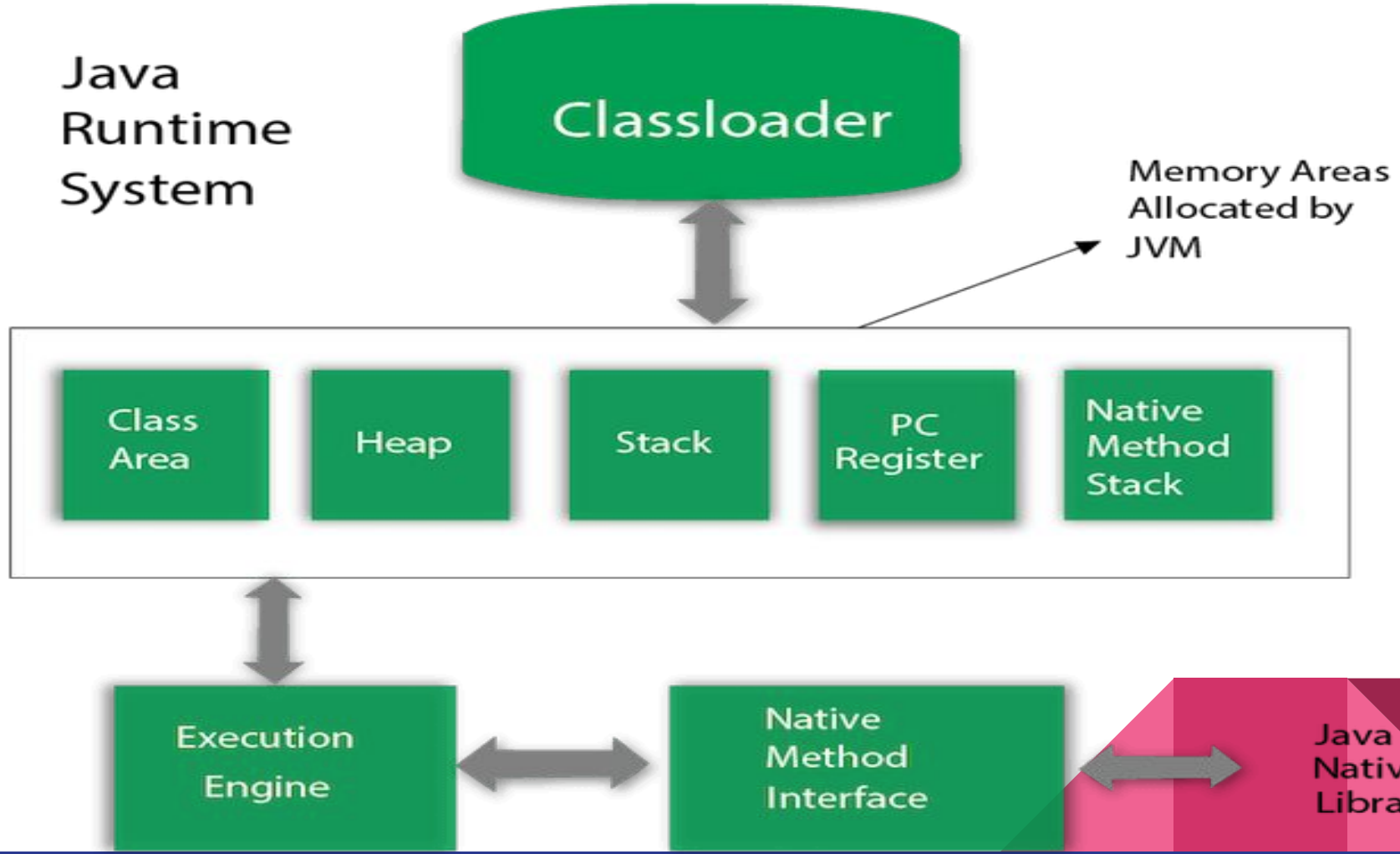
It does

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.




Java
Runtime
System
Classloader
Memory Areas
Allocated by
JVM
Class
Area
Heap
Stack
PC
Register
Native
Method
Stack
Execution
Engine
Native
Method
Interface
Java
Native
Libraries

1) Classloader

Classloader is a subsystem of JVM which is used to load class files.

Whenever we run the java program, it is loaded first by the classloader.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

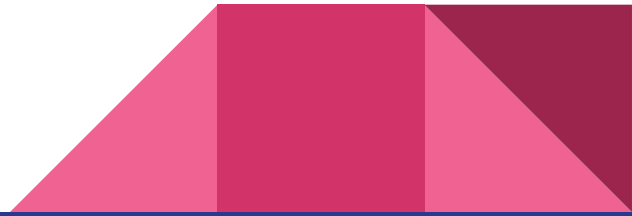
6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

It contains:

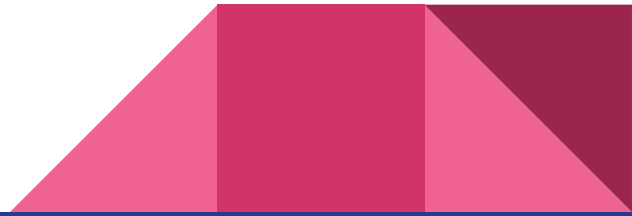
1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:**



8) Java Native Interface

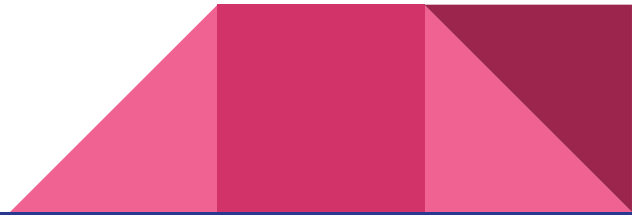
Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc.

Java uses JNI framework to send output to the Console or interact with OS libraries.



Constructors in Java

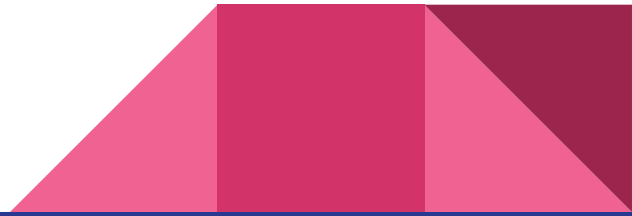
- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object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

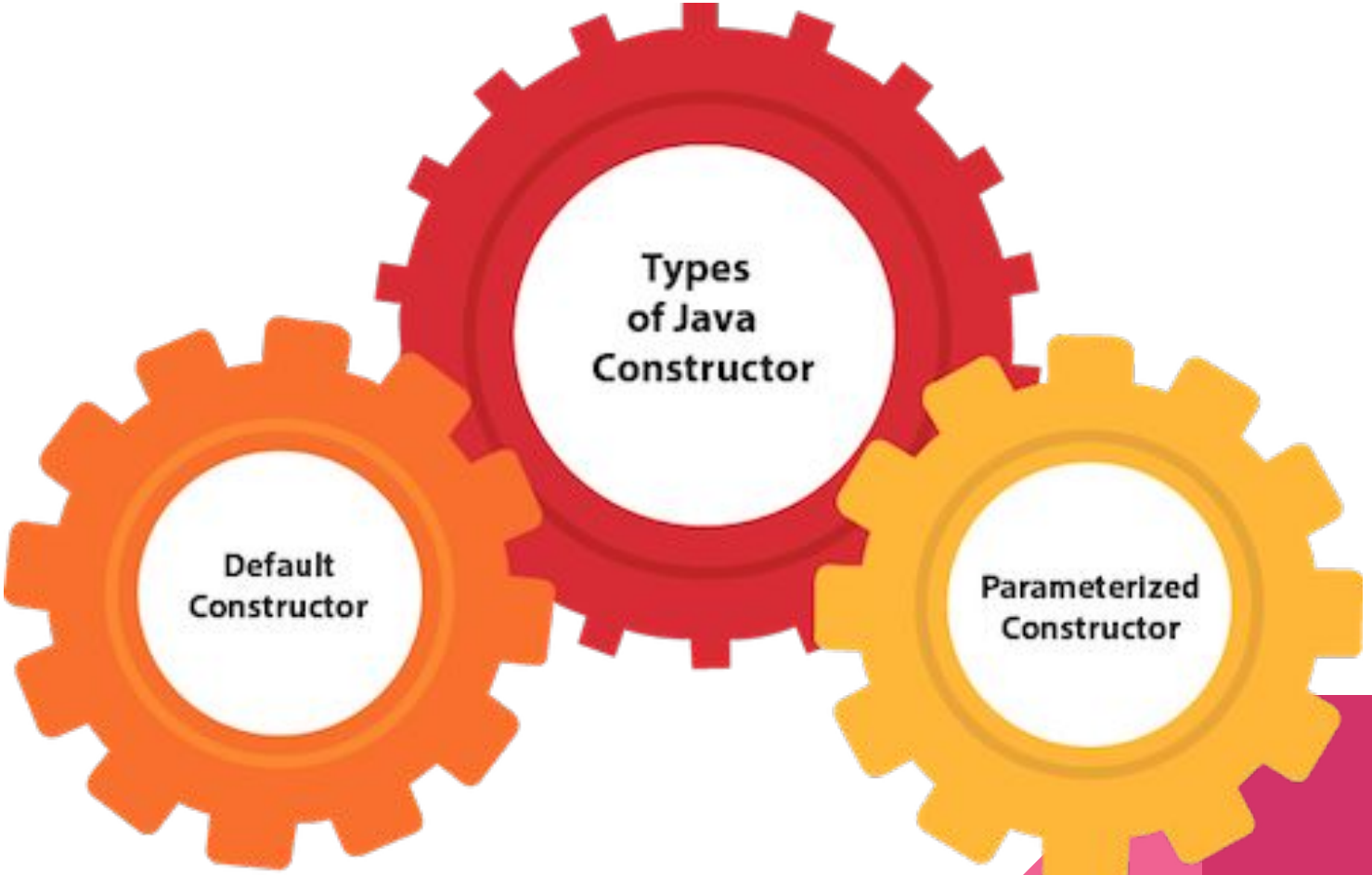


Rules for creating Java constructor

There are two rules defined for the constructor.

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



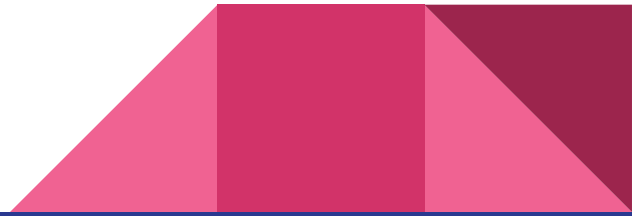

Types
of Java
Constructor
Default
Constructor
Parameterized
Constructor

Java Default Constructor

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

Syntax of default constructor:

```
<class_name>()

    { }
```



```
public class Bike1
{
Bike1()
{
System.out.println("Bike is created");
}
public static void main(String args[])
{
Bike1 b=new Bike1();
}
}
```

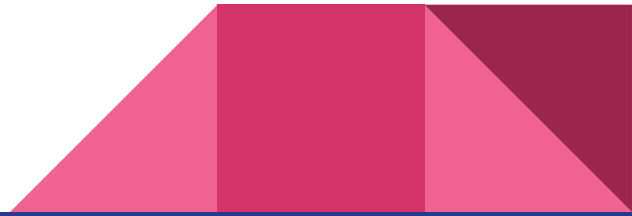
Bike is created

Java Parameterized Constructor

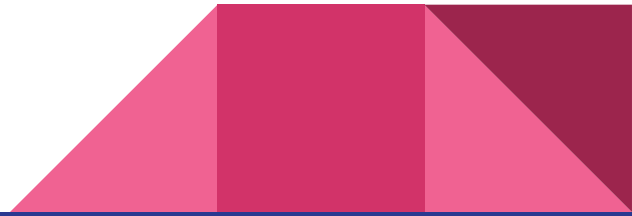
- A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

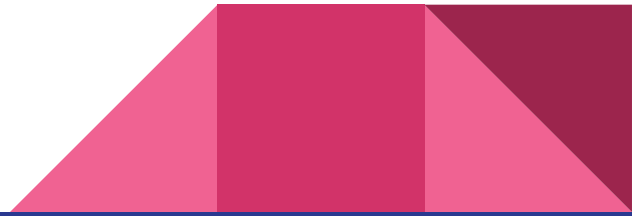
- The parameterized constructor is used to provide different values to distinct objects.
- However, you can provide the same values also.
-



```
class Rectangle
{
   int area, length, breadth;
   // Parameterized constructor
   Rectangle(int l, int b)
   {
      length = l;
      breadth = b;
   }
   void getArea()
   {
      area = length * breadth;
      System.out.println("Area of rectangle : " + area);
   }
}
```

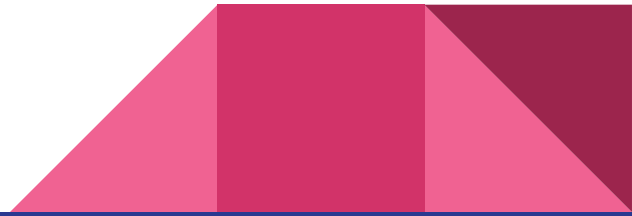


```
public class FindArea
{
   public static void main(String[] args)
    {

      Rectangle rs = new Rectangle(10, 20);
      rs.getArea();
    }
}
```

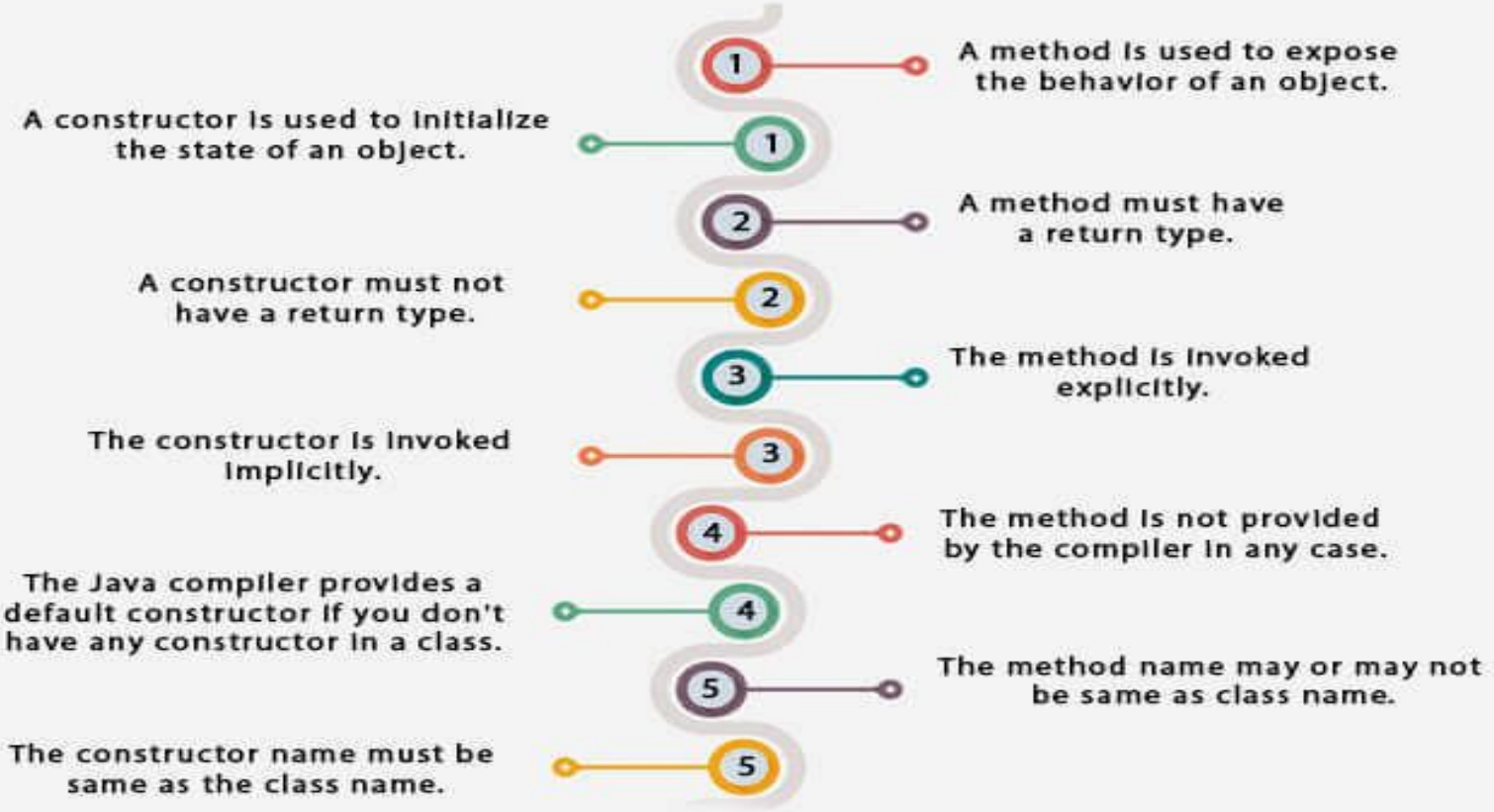


Constructor Overloading in Java

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



Difference between constructor and method in Java



Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

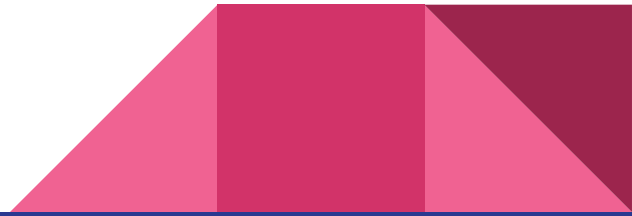
Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class / Derived Class :** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class/ Base Class :** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Syntax :

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

The **extends keyword** indicates that you are making a new class that derives from an existing class.

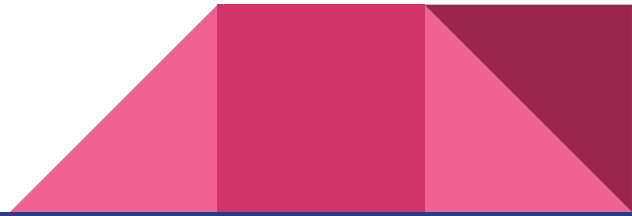


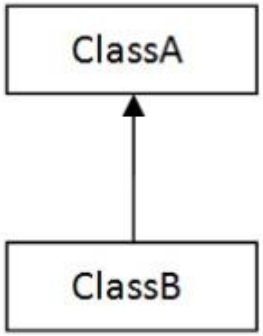
Types of inheritance in java

On the basis of class, there can be two types of inheritance in java:

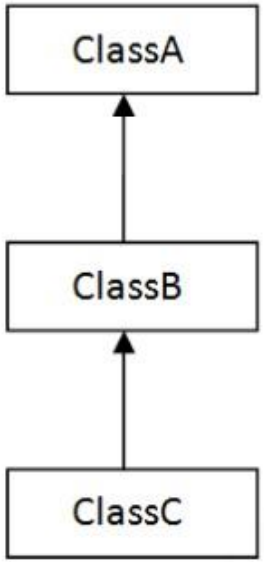
- Single,
- multilevel

In java programming, multiple and hybrid inheritance is supported through interface only.

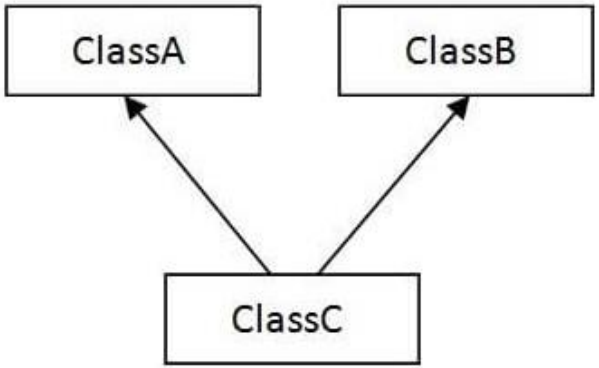




Single



Multilevel



Multiple

Single Inheritance

When a class inherits another class, it is known as a *single inheritance*.

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

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

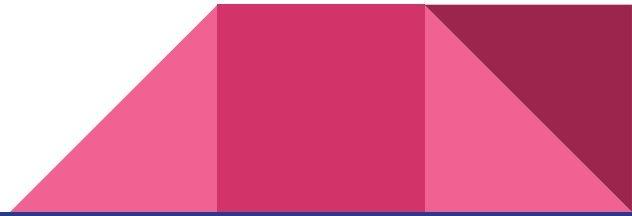
Multilevel Inheritance

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

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

```
class BabyDog extends Dog
{
        void weep()
        {
        System.out.println("weeping...");
        }
}
```

```
class TestInheritance2
{
public static void main(String args[])
        {
        BabyDog d=new BabyDog();
        d.weep();
        d.bark();
        d.eat();
        }
}
```



Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading

Method overloading increases the readability of the program.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

```
class Adder
{
static int add(int a,int b)
        {
        return a+b;
        }
static int add(int a,int b,int c)
        {
        return a+b+c;
        }
}
```

```
class TestOverloading1
{
public static void main(String[] args)
        {
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(11,11,11));
        }
}
```

OUTPUT :

22

33

```
class Adder
{
static int add(int a, int b)
        {
        return a+b;
        }
static double add(double a, double b)
        {
        return a+b;
        }
}
```

```
class TestOverloading2
{
public static void main(String[] args)
        {
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(12.3,12.
        6));
        }
}
```

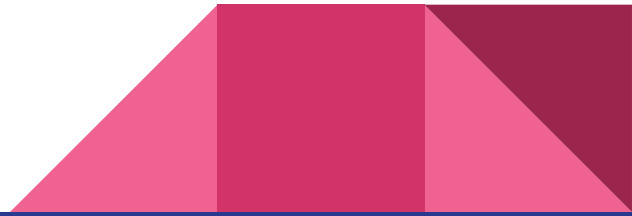
OUTPUT :
22
24.9

Super Keyword

- The **super keyword** refers to superclass (parent) objects.
- It is used to call superclass methods, and to access the superclass constructor.
- The most common **use** of the **super keyword** is to eliminate the confusion between superclasses and subclasses that have methods with the same name.
- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

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

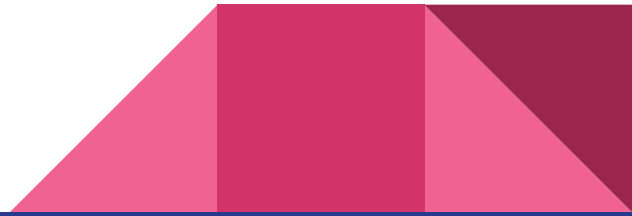
- We can use super keyword to access the data member or field of parent class.
- It is used if parent class and child class have same fields.



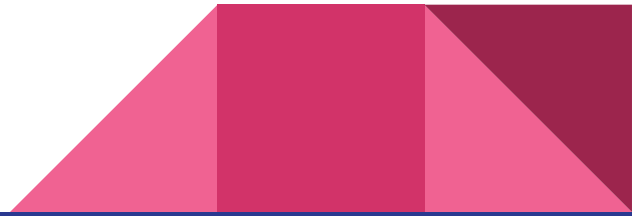
```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}
}
```

2) super can be used to invoke parent class method

- The super keyword can also be used to invoke parent class method.
- It should be used if subclass contains the same method as parent class.
- In other words, it is used if method is overridden



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

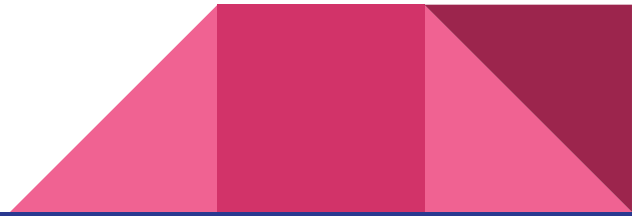


```
void work()
{
super.eat();
bark();
}
}
class TestSuper2
{
public static void main(String args[])
{
Dog d=new Dog();
d.work();
}
}
```

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

Let's see a simple example:



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

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

Output:

```
animal is created

 dog is created
```

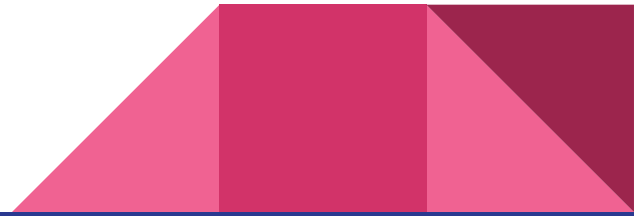
final Keyword In Java

The **final keyword** in java is used to restrict the user.

The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

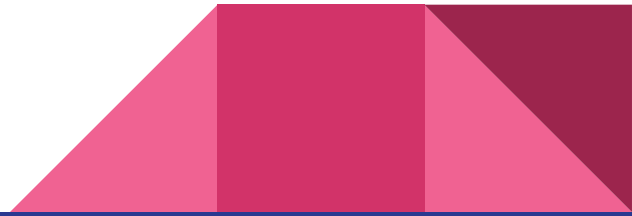
The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.



1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant)

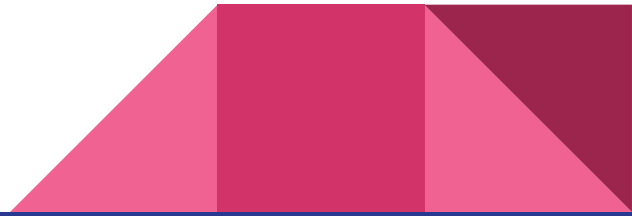
```
final int speedlimit=90;//final variable
```



2) Java final method

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

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



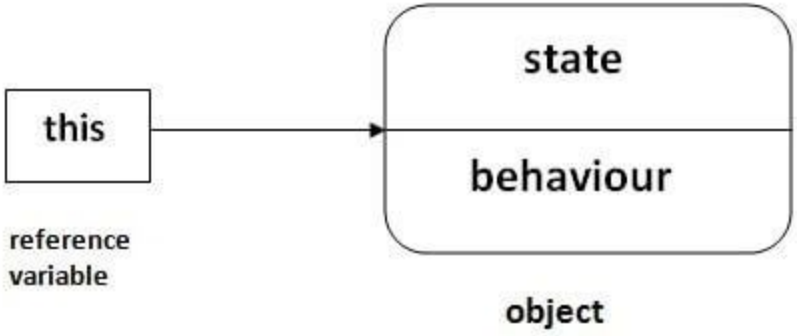
3) Java final class

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

```
class Bike{}
final class Honda1 extends Bike{
 void run()
{
    System.out.println("running safely with 100kmph");
}
 public static void main(String args[])
{
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

this keyword in Java

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



Usage of Java this Keyword

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

01 this can be used to refer current class instance variable.

02 this can be used to invoke current class method (implicity)

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

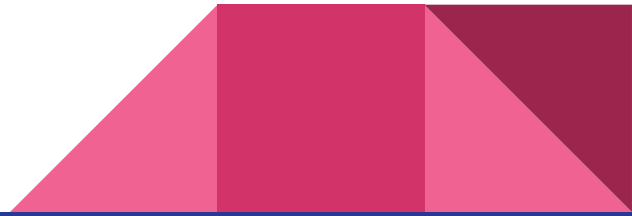
04 this can be passed as an argument in the method call.

05 this can be passed as argument in the constructor call.

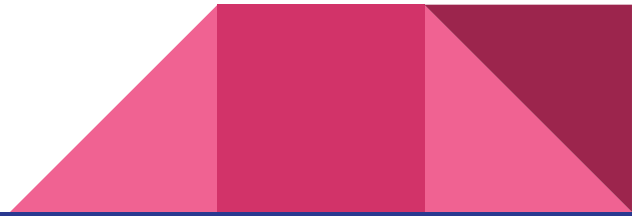
06 this can be used to return the current class instance from the method

1) this: to refer current class instance variable

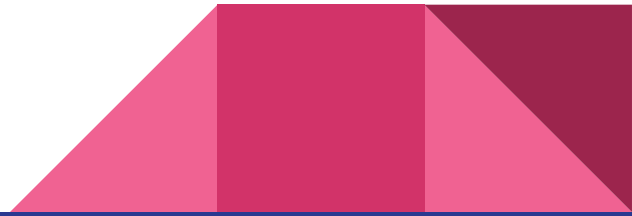
- The this keyword can be used to refer current class instance variable.
- If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity



```
class Student

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

}
void display()
 {
   System.out.println(rollno+" "+name+" "+fee);
 }
}
```

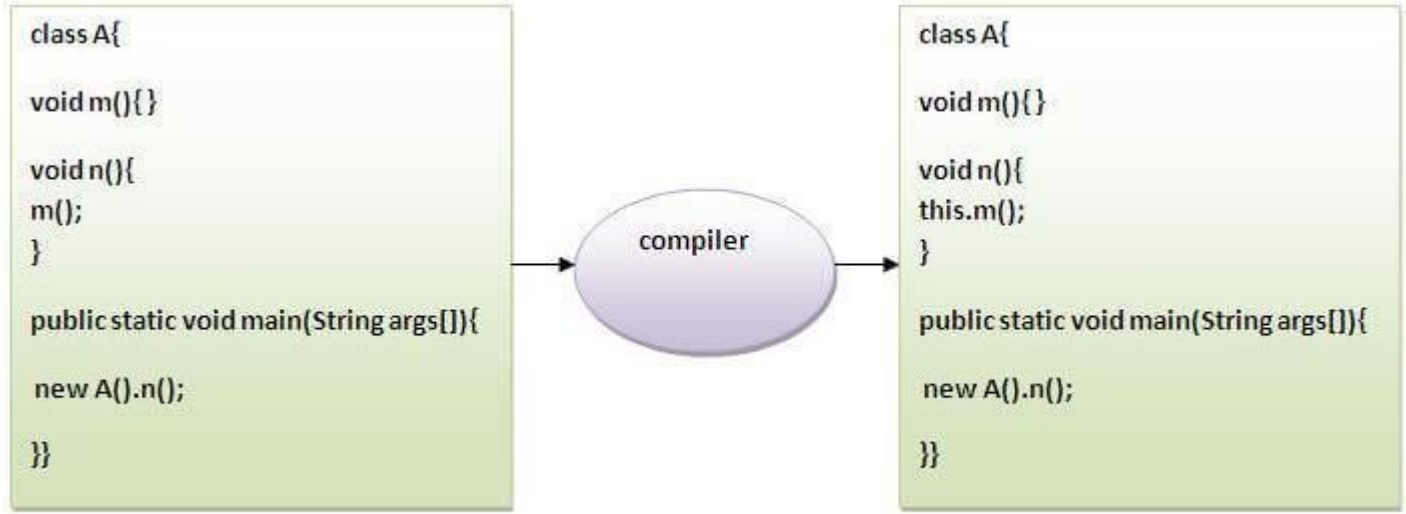


```
class TestThis2

{
public static void main(String args[])
{
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}
```



2) this: to invoke current class method

- You may invoke the method of the current class by using the this keyword.
- If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.
- Let's see the example



```
class A
{
        void m()
        {
        System.out.println("hello m");
        }
        void n()
        {
        System.out.println("hello n");
        //m();//same as this.m()
        this.m();
        }
}
class TestThis4
{
        public static void main(String args[])
        {
            A a=new A();
            a.n();
        }
}
```

OUTPUT :

hello n
hello m

3) this() : to invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor.
- It is used to reuse the constructor. In other words, it is used for constructor chaining.
- **Calling default constructor from parameterized constructor:**

```
class A
{
        A()
        {
        System.out.println("hello a");
        }
        A(int x)
        {
        this();
        System.out.println(x);
        }
}
```

```
class TestThis5
{
        public static void main(String args[])
        {
        A a=new A(10);
        }
}
```

```
hello a
10
```

4) this: to pass as an argument in the method

- The this keyword can also be passed as an argument in the method.
- It is mainly used in the event handling.
- Let's see the example:

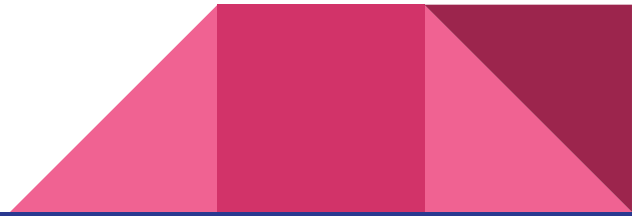
```
class S2
{
        void m(S2 obj)
        {
        System.out.println("method is invoked");
        }
        void p()
       {
        m(this);
        }
```

```
public static void main(String args[])
{
  S2 s1 = new S2();
  s1.p();
  }
}
```

```
method is invoked
```

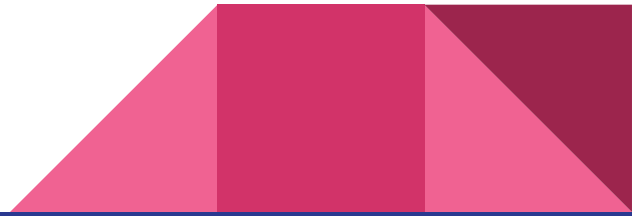
5) this: to pass as argument in the constructor call

- We can pass the this keyword in the constructor also.
- It is useful if we have to use one object in multiple classes. Let's see the example:



```
class B
{
  A4 obj;
          B(A4 obj)
        {
           this.obj=obj;
         }
         void display()
        {
           System.out.println(obj.data);//using data member of A4 class
         }
}
```

```
class A4
{
  int data=10;
            A4()
          {
            B b=new B(this);
            b.display();
           }
  public static void main(String args[])
          {
            A4 a=new A4();
           }
}
```



6) this keyword can be used to return current class instance

- We can return this keyword as an statement from the method.
- In such case, return type of the method must be the class type (non-primitive).
- Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name()

{

return this;

}
```

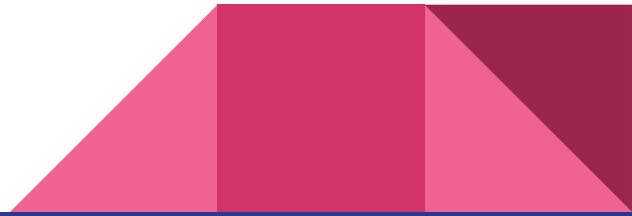
```
class A
{
        A getA()
        {
        return this;
        }
void msg()
        {
        System.out.println("Hello java");
        }
}
```

```
class Test1
{
        public static void main(String args[])
        {
        new A().getA().msg();
        }
}
```

Hello java

Interface in Java

- An **interface in Java** is a blueprint of a class.
- It has static constants and abstract methods.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables.
- It cannot have a method body.

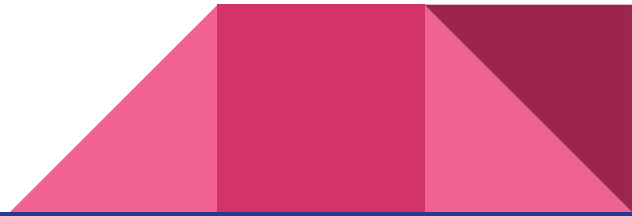


Why use Java interface?

There are mainly three reasons to use interface.

They are given below.

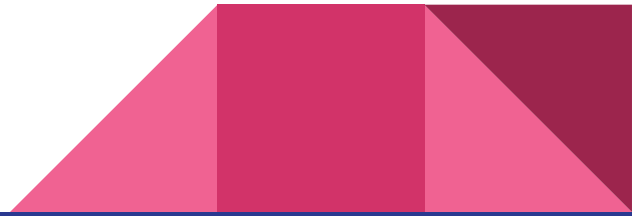
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.




It is used to achieve abstraction.
1
2
By interface, we can support the functionality of multiple inheritance.
It can be used to achieve loose coupling.
3

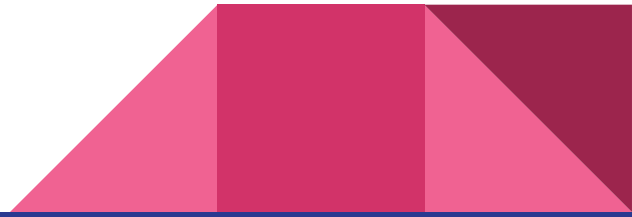
How to declare an interface?

- An interface is declared by using the **interface** keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.



Syntax:

```
interface  interface_name

{

  // declare constant fields

    // declare methods that abstract

 }
```

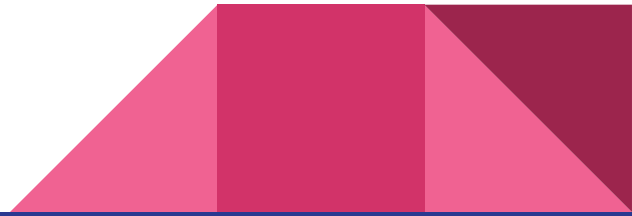


Example

```
interface printable
{
void print();
}
class A6 implements printable{
        public void print()
        {
        System.out.println("Hello");
        }
public static void main(String args[]){
        A6 obj = new A6();
        obj.print();
 }
}
```

Example

```
interface Bank
{
float rateOfInterest();
}
class SBI implements Bank
{
        public float rateOfInterest()
        {
        return 9.15f;
        }
}
```



```
class PNB implements Bank
{
        public float rateOfInterest()
        {
        return 9.7f;
        }
}
class TestInterface2{
public static void main(String[] args)
        {
        Bank b=new SBI();
        System.out.println("ROI: "+b.rateOfInterest());
        }
}
```

Example

```
import java.io.*;
 interface Vehicle

{
   void changeGear(int a);
   void speedUp(int a);
   void applyBrakes(int a);
}

  class Bicycle implements Vehicle
{
   int speed;
   int gear;
   public void changeGear(int newGear)
   {

       gear = newGear;

   }

public void speedUp(int increment)
{
      speed = speed + increment;
   }

   public void applyBrakes(int decrement)
{
       speed = speed - decrement;
   }

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

```
class Bike implements Vehicle
{
    int speed;
    int gear;
    public void changeGear(int newGear)
{
        gear = newGear;
    }
    public void speedUp(int increment)
{
        speed = speed + increment;
    }
public void applyBrakes(int decrement)
{
        speed = speed - decrement;
    }
  public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}
```

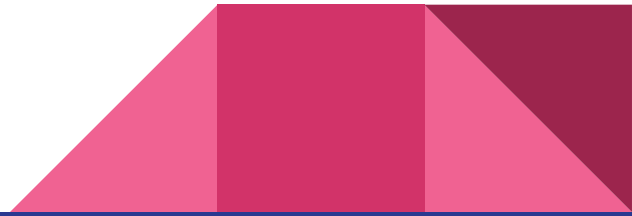
```
class GFG {
  public static void main (String[] args)

 {
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);
        System.out.println("Bicycle present state :");
        bicycle.printStates();
        //creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);
        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

OUTPUT :

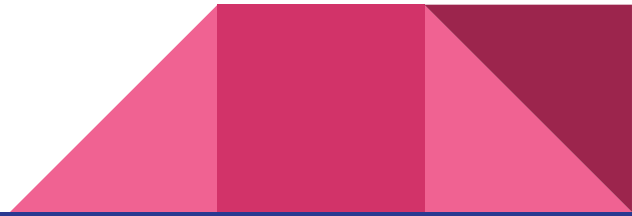
Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1

Multiple inheritance using Interface

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

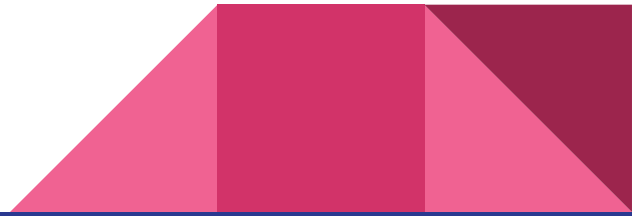


```
class TestInterface3 implements Printable, Showable{

public void print()

{

System.out.println("Hello");

}

public static void main(String args[])

{

TestInterface3 obj = new TestInterface3();

obj.print();

 }

}
```

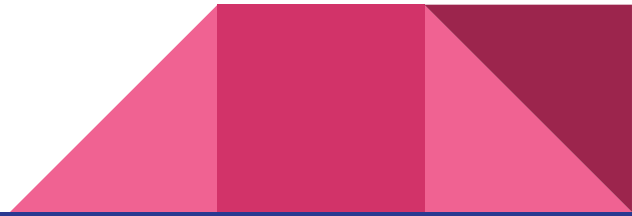


Abstract class in Java

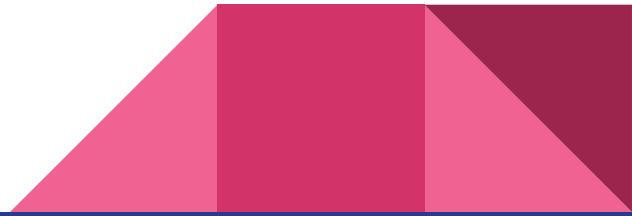
- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract and non-abstract methods (method with the body).
- Abstraction is a process of hiding the implementation details and showing only functionality to the user.



- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- It cannot be instantiated.



- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.



Rules for Java Abstract class

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run()
{
System.out.println("running safely");
}
public static void main(String args[])
{
 Bike obj = new Honda4();
 obj.run();
}
}
```

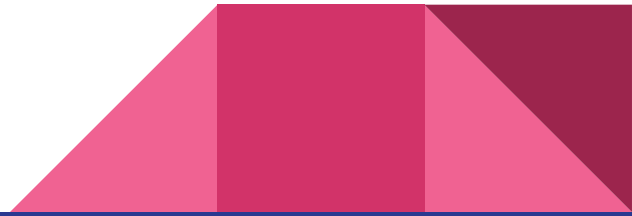
Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.

5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Wrapper classes in Java

- The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.
- Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically.
- The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing



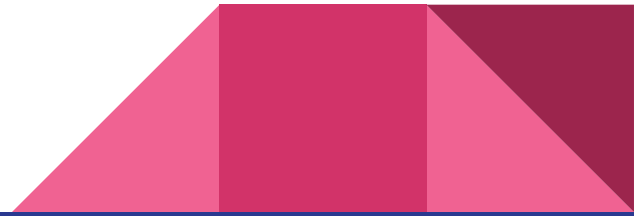
Use of Wrapper classes in Java

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

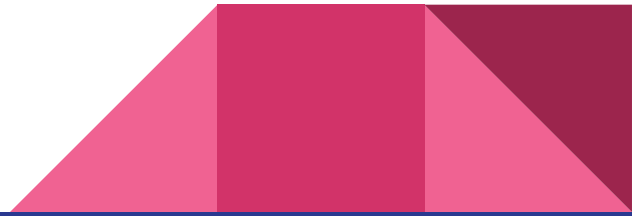
Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.



```
public class WrapperExample1
{
public static void main(String args[])
	{
		//Converting int into Integer
		int a=20;
		Integer i=Integer.valueOf(a);//converting int into Integer explicitly
		Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
		System.out.println(a+" "+i+" "+j);
	}
}
```

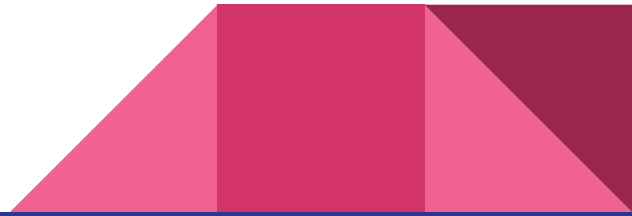


Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.

It is the reverse process of autoboxing.

Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.



```
public class WrapperExample2{
public static void main(String args[])
{
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}
}
```

Package in Java

Package in Java is a mechanism to encapsulate a group of classes, sub **packages** and interfaces.

A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Package in java can be categorized in two form, built-in package and user-defined package.

Some of the existing packages in Java are −

- java.lang − bundles the fundamental classes
- java.io − classes for input , output functions are bundled in this package

It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

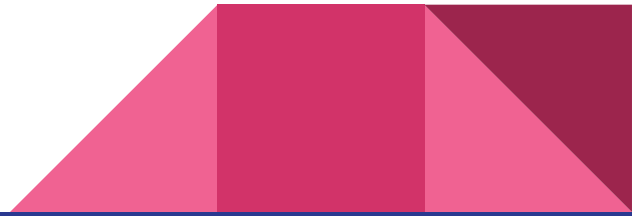
Advantage of Java Package

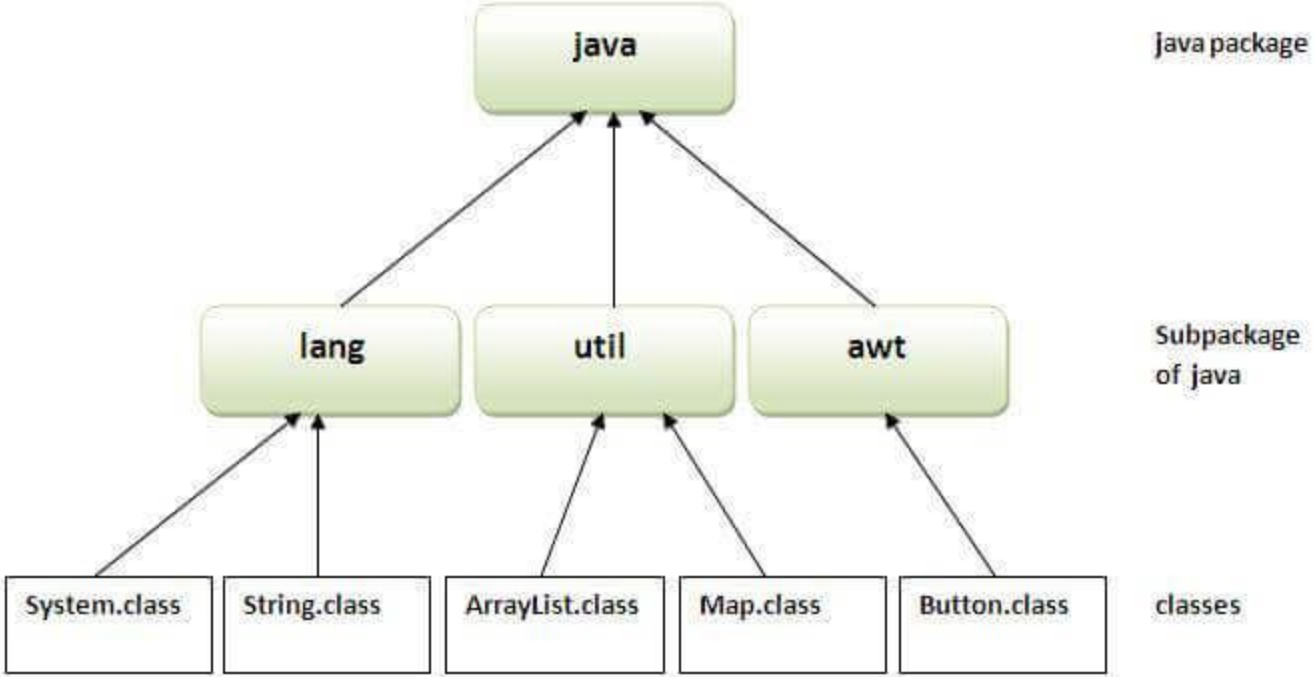
1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

The **package** keyword is used to create a package in java.




java
java package
lang
util
awt
Subpackage
of java
System.class
String.class
ArrayList.class
Map.class
Button.class
classes

Example of java package

```
package mypack;
public class Simple
{
 public static void main(String args[])
{
   System.out.println("Welcome to package");
  }
}
```

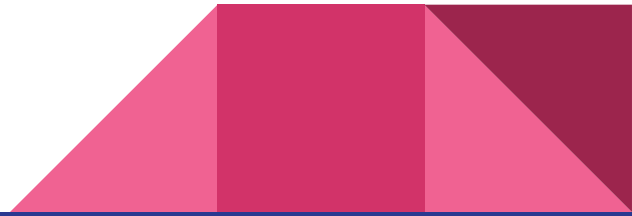
How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```



How to run java package program

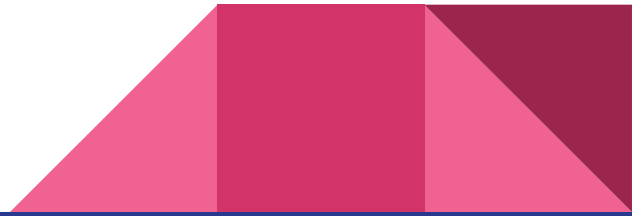
You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination.

The . represents the current folder.

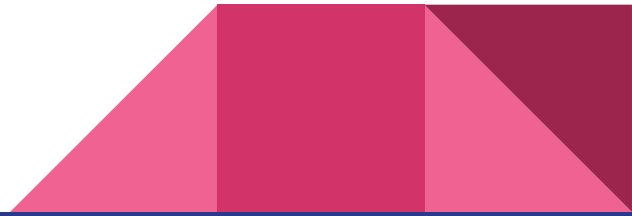
To Run: java mypack.Simple



How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.



1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

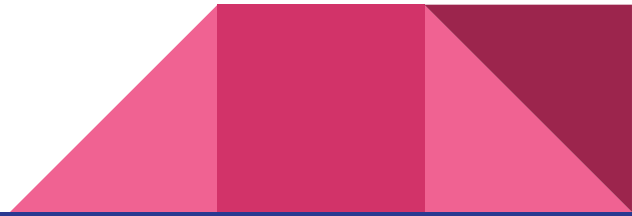
The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java

package pack;

public class A{

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

}
```

```
//save by B.java
package mypack;
import pack.*;

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

2) Using packagename.classname

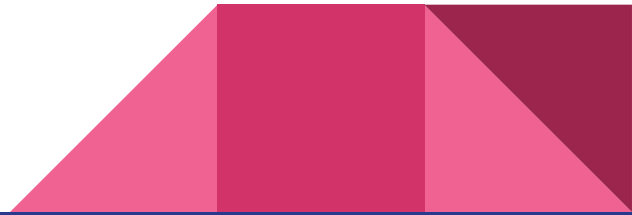
If you import package.classname then only declared class of this package will be accessible.



```
//save by A.java
 package pack;
public class A
{
  public void msg(){System.out.println("Hello");}
}

package mypack;
import pack.A;
  class B
{
  public static void main(String args[])
{
   A obj = new A();
   obj.msg();
  }
}
```

3) Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible.
- Now there is no need to import.
- But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

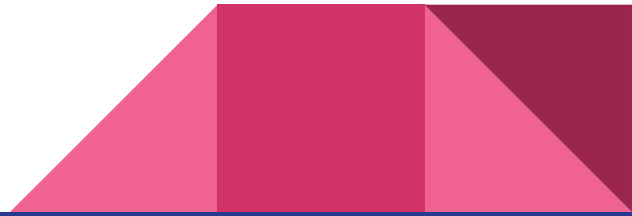


Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A
{
  public void msg()
   {
        System.out.println("Hello");
    }
}
```

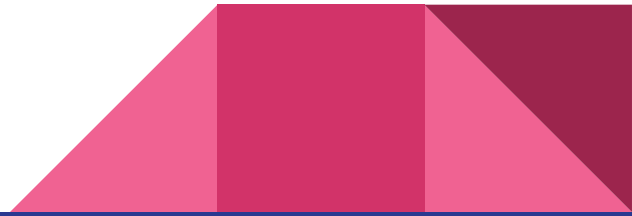
```
//save by B.java
package mypack;
class B{
  public static void main(String args[])
   {
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output:Hello



Multithreading in Java

- **Multithreading in Java is a process of executing multiple threads simultaneously.**
- **A thread is a lightweight sub-process, the smallest unit of processing.**
- **Multiprocessing and multithreading, both are used to achieve multitasking.**
- **However, we use multithreading than multiprocessing because threads use a shared memory area.**

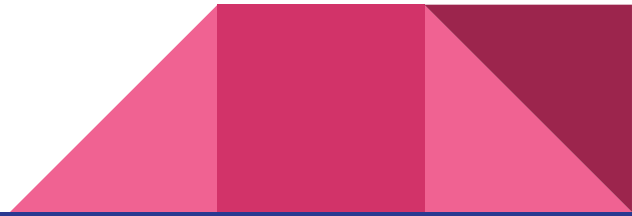


Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You can perform many operations together, so it saves time.

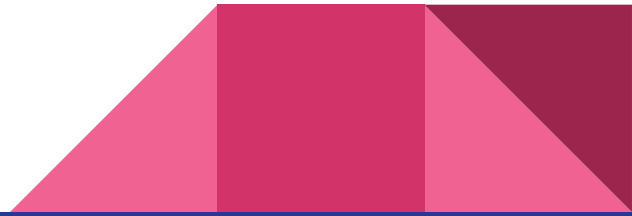
3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread



Multitasking

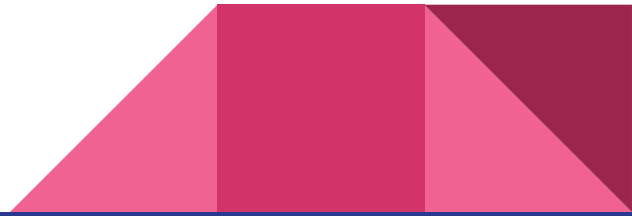
Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- **Process-based Multitasking (Multiprocessing)**
- **Thread-based Multitasking (Multithreading)**



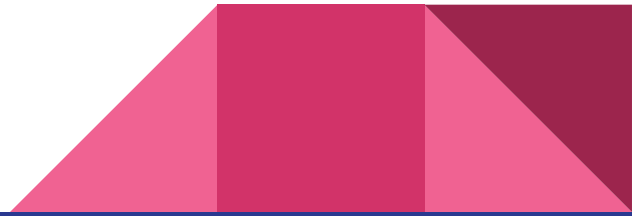
1) Process-based Multitasking (Multiprocessing)

- **Each process has an address in memory. In other words, each process allocates a separate memory area.**
- **A process is heavyweight.**
- **Cost of communication between the process is high.**
- **Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.**



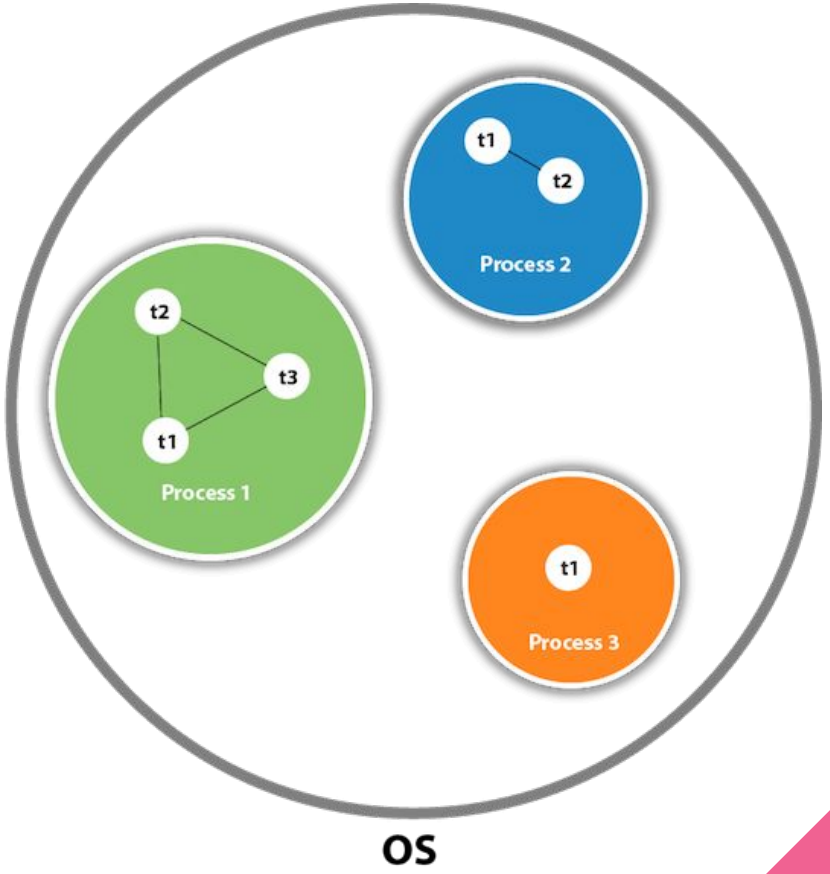
2) Thread-based Multitasking (Multithreading)

- **Threads share the same address space.**
- **A thread is lightweight.**
- **Cost of communication between the thread is low.**



What is Thread in java

- **A thread is a lightweight subprocess, the smallest unit of processing.**
- **It is a separate path of execution.**
- **Threads are independent.**
- **If there occurs exception in one thread, it doesn't affect other threads.**
- **It uses a shared memory area.**

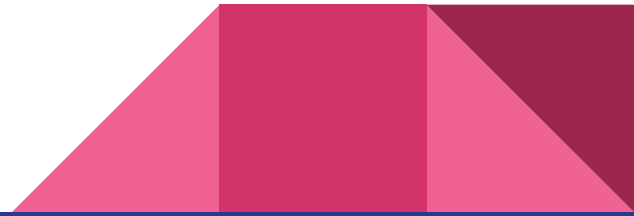

t1
t2
Process 2
t2
t3
t1
Process 1
t1
Process 3
OS

Java Thread class

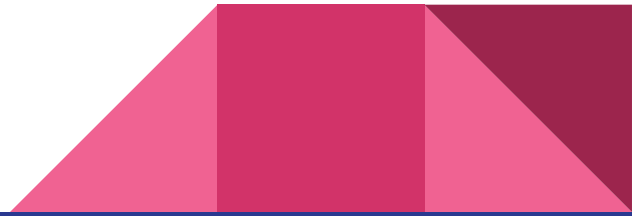
- **Java provides Thread class to achieve thread programming.**
- **Thread class provides constructors and methods to create and perform operations on a thread.**
- **Thread class extends Object class and implements Runnable interface**

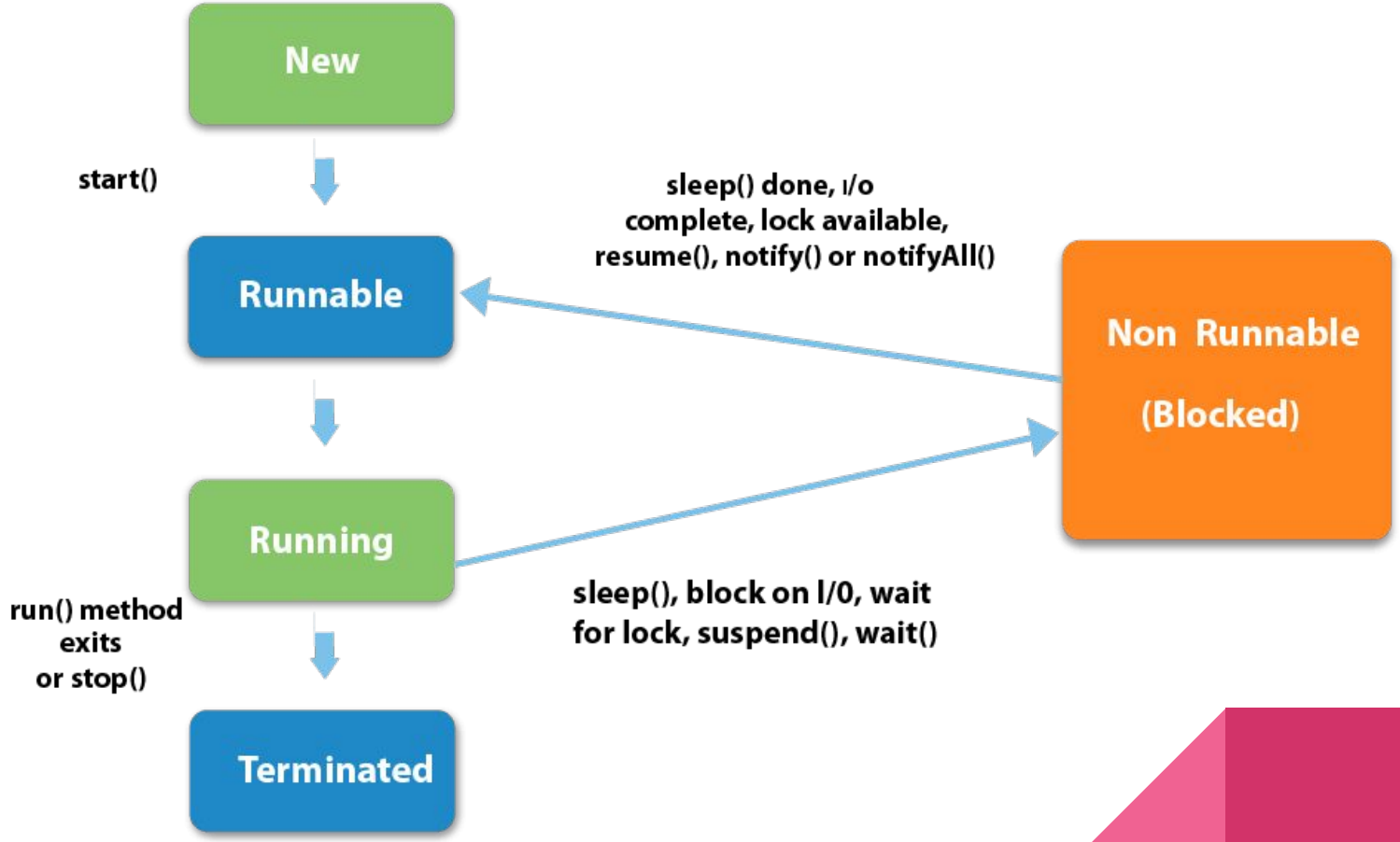
Life cycle of a Thread (Thread States)

- **A thread can be in one of the five states.**
- **According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated.**
- **There is no running state.**
- **But for better understanding the threads, we are explaining it in the 5 states.**
- **The life cycle of the thread in java is controlled by JVM.**
- **The java thread states are as follows:**



1. **New**
2. **Runnable**
3. **Running**
4. **Non-Runnable (Blocked)**
5. **Terminated**




New
start()
Runnable
Running
run() method
exits
or stop()
Terminated
sleep() done, I/o
complete, lock available,
resume(), notify() or notifyAll()
Non Runnable
(Blocked)
sleep(), block on I/0, wait
for lock, suspend(), wait()

1) New

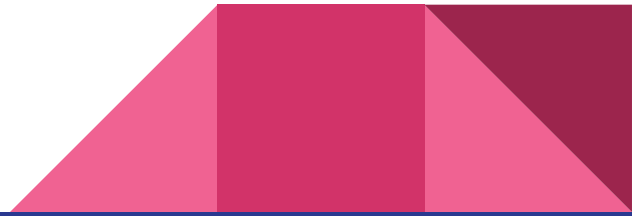
The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

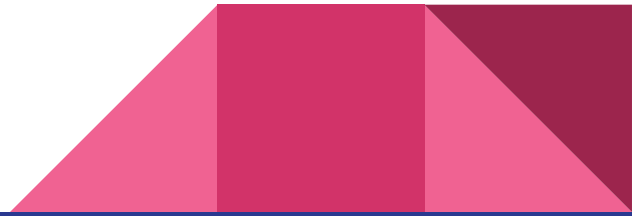


4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits



How to create thread

There are two ways to create a thread:

1. **By extending Thread class**
2. **By implementing Runnable interface.**

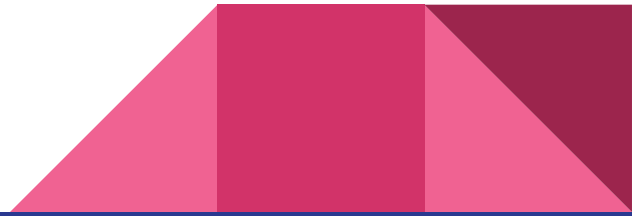
Thread class

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

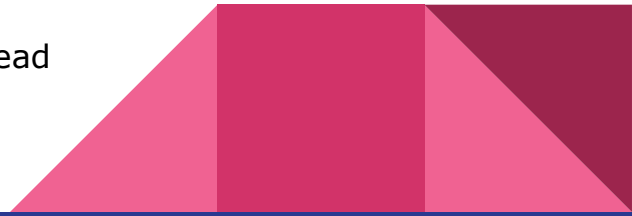
Commonly used Constructors of Thread class:

- **Thread()**
- **Thread(String name)**
- **Thread(Runnable r)**
- **Thread(Runnable r,String name)**



Commonly used methods of Thread class

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread

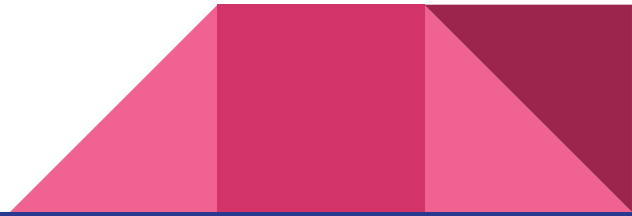


Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.

Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

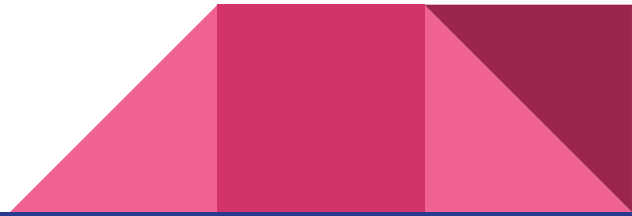


Starting a thread:

start() method of Thread class is used to start a newly created thread.

It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run



Thread Example by extending Thread class

```
class Multi extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

Output: thread is running..

Thread Example by implementing Runnable interface

```
class Multi  implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}
  public static void main(String args[])
{
Multi m1=new Multi();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output : thread is running...

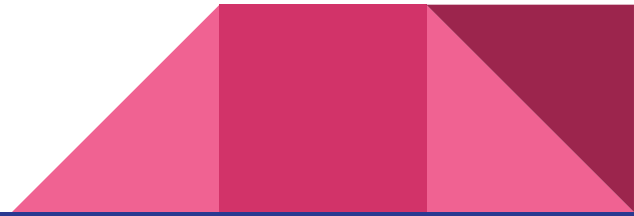
Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- **public static void sleep(long miliseconds)throws InterruptedException**
- **public static void sleep(long miliseconds, int nanos)throws InterruptedException**



```
class TestSleepMethod1 extends Thread
{
 public void run()
{
  for(int i=1;i<5;i++)
   {
    try{
        Thread.sleep(500);
        }
     catch(InterruptedException e)
         {
           System.out.println(e);
          }
   System.out.println(i);
 }
}
```

```
public static void main(String args[])
 {
 TestSleepMethod1 t1=new TestSleepMethod1();
 TestSleepMethod1 t2=new TestSleepMethod1();
 t1.start();
 t2.start();
 }
}
```

Output :

1

1

2

2

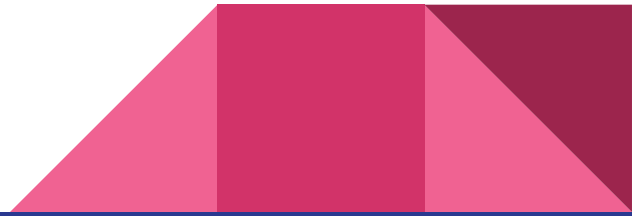
3

3

4

4

- **As you know well that at a time only one thread is executed.**
- **If you sleep a thread for the specified time,the thread scheduler picks up another thread and so on.**

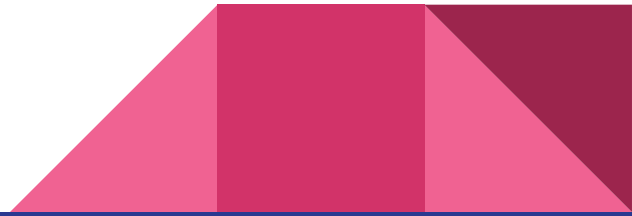


Naming Thread and Current Thread

Naming Thread

- **The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on.**
- **By we can change the name of the thread by using setName() method.**
- **The syntax of setName() and getName() methods are given below:**

1. **public String getName(): is used to return the name of a thread.**
2. **public void setName(String name): is used to change the name of a thread.**



```
class TestMultiNaming1 extends Thread{
  public void run()
  {
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  t1.start();
  t2.start();
  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

Output:

Name of t1:Thread-0

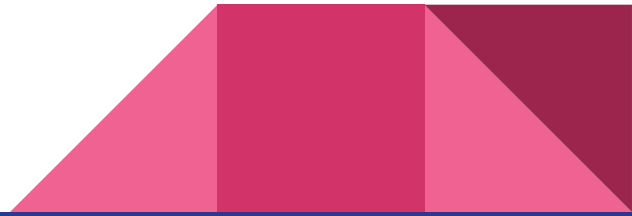
Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:Sonoo Jaiswal

running...



Priority of a Thread (Thread Priority):

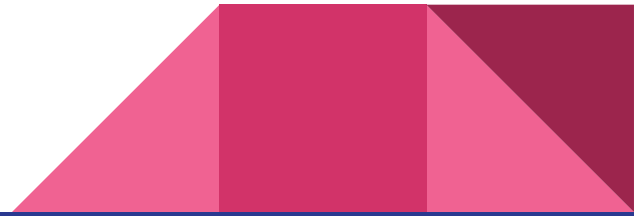
- Each thread have a priority.
- Priorities are represented by a number between 1 and 10.
- In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY).

The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.



```
class TestMultiPriority1 extends Thread
 {
 public void run()
  {
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
   }
 public static void main(String args[])
{
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
 }
}
```

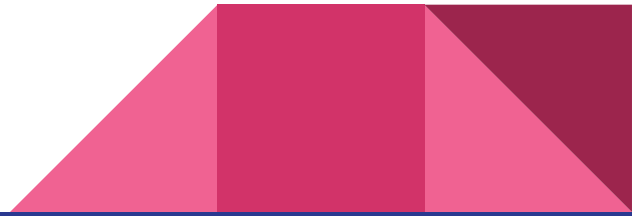
Output:

running thread name is:Thread-0

running thread priority is:10

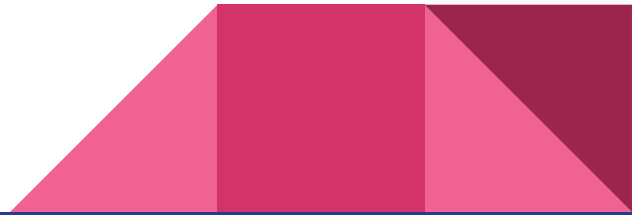
running thread name is:Thread-1

running thread priority is:1



Daemon Thread in Java

- **Daemon thread in java** is a service provider thread that provides services to the user thread.
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.
- You can see all the detail by typing the jconsole in the command prompt.
- The jconsole tool provides information about the loaded classes, memory usage, running threads etc
- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.



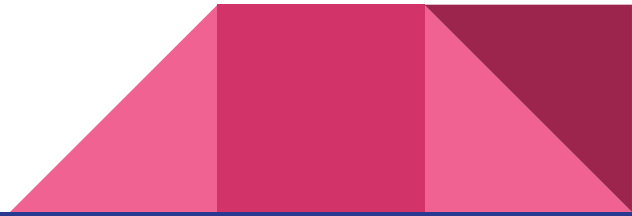
The java.lang.Thread class provides two methods for java daemon thread.

1) public void setDaemon(boolean status)

used to mark the current thread as daemon thread or user thread.

2) public boolean isDaemon()

used to check that current is daemon.



```
public class TestDaemonThread1 extends Thread
{
 public void run()
{
  if(Thread.currentThread().isDaemon())
  {
    //checking for daemon thread
   System.out.println("daemon thread work");
  }
  Else
 {
  System.out.println("user thread work");
  }
 }
```

```
public static void main(String[] args)
  {
 TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
 TestDaemonThread1 t2=new TestDaemonThread1();
 TestDaemonThread1 t3=new TestDaemonThread1();

 t1.setDaemon(true);//now t1 is daemon thread

 t1.start();//starting threads
 t2.start();
 t3.start();
 }
}
```

Output :

daemon thread work

user thread work

user thread work

Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

The synchronization is mainly used to

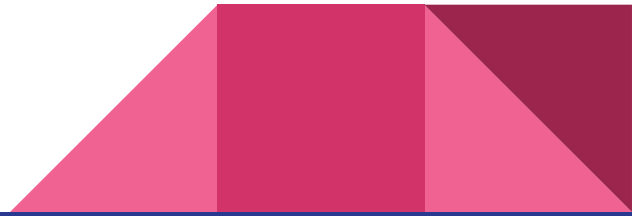
1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

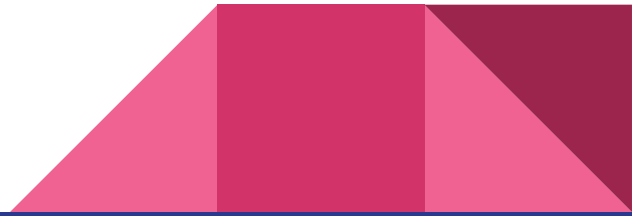
Here, we will discuss only thread synchronization.



Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

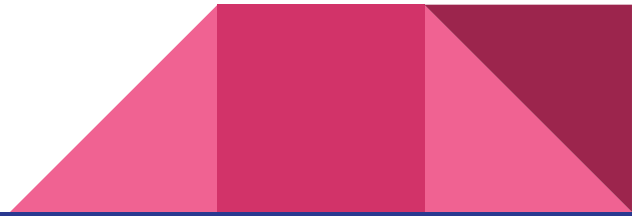


Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data.

This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

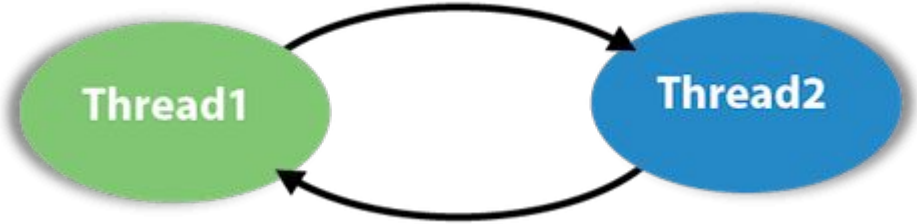


Deadlock in java

Deadlock in java is a part of multithreading.

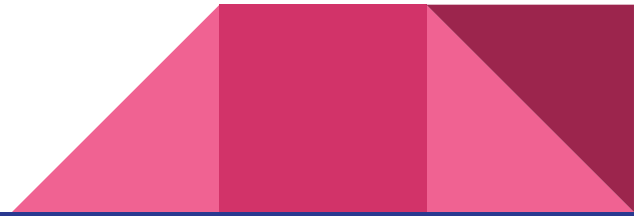
Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.

Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

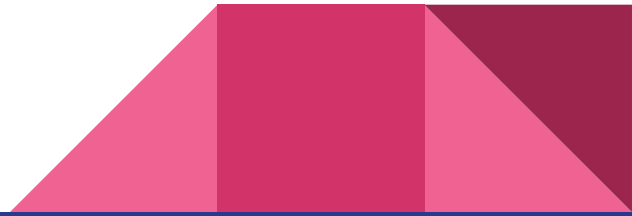


```
public class TestDeadlockExample1 {
  public static void main(String[] args) {
    final String resource1 = "ratan jaiswal";
    final String resource2 = "vimal jaiswal";
    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread()
    {
      public void run()
        {
          synchronized (resource1) {
           System.out.println("Thread 1: locked resource 1");
```

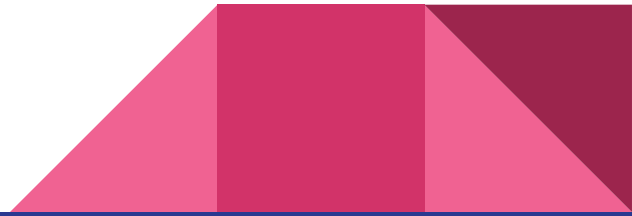
```
        try {
                Thread.sleep(100);
            } catch (Exception e) {}

        synchronized (resource2) {
         System.out.println("Thread 1: locked resource 2");
        }
      }
    }
};
```



```
// t2 tries to lock resource2 then resource1
 Thread t2 = new Thread() {
   public void run() {
     synchronized (resource2) {
       System.out.println("Thread 2: locked resource 2");


       try {
             Thread.sleep(100);
             } catch (Exception e) {}
```



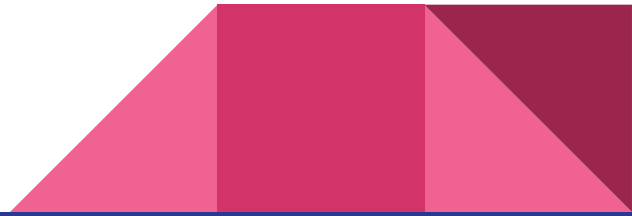
```
        synchronized (resource1) {
          System.out.println("Thread 2: locked resource 1");
        }
      }
    }
  };
   t1.start();
  t2.start();
 }
}
```



Output:

Thread 1: locked resource 1

Thread 2: locked resource 2



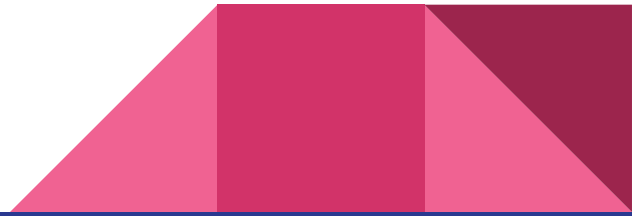
Inter-thread communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()



1) wait() method

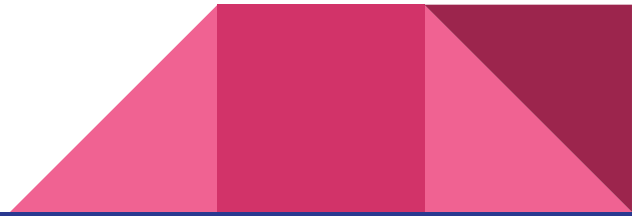
Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

public final void wait()throws InterruptedException

waits until object is notified.

public final void wait(long timeout)throws InterruptedException

waits for the specified amount of time.



2) notify() method

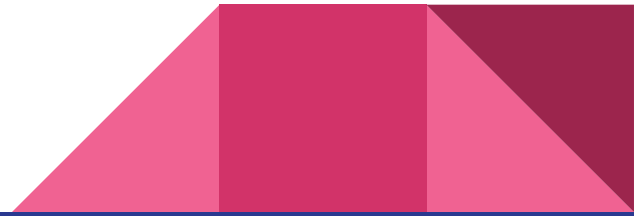
Wakes up a single thread that is waiting on this object's monitor.

If any threads are waiting on this object, one of them is chosen to be awakened.

The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

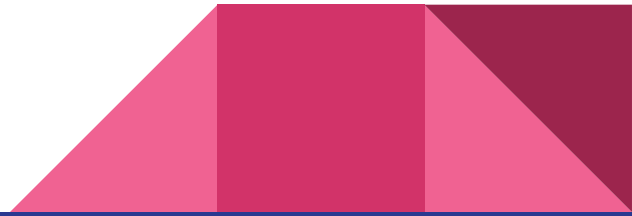


3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```



Difference between wait and sleep?

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Java String

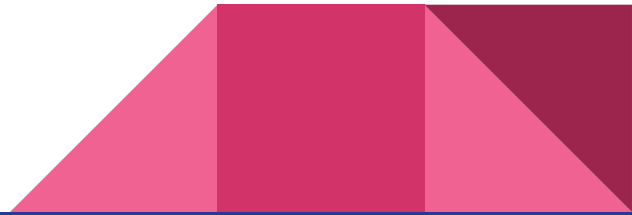
In Java, string is basically an object that represents sequence of char values.

An array of characters works same as Java string. For example:

```
1.  char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2.  String s=new String(ch);
```

is same as:

```
1.  String s="javatpoint";
```



What is String in java

Generally, String is a sequence of characters.

But in Java, string is an object that represents a sequence of characters.

The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

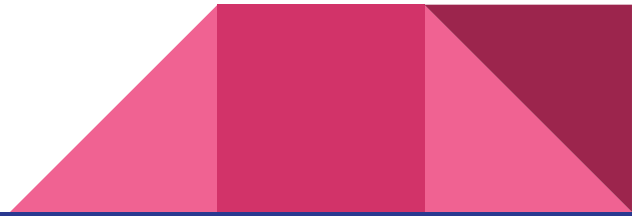
```
1.  String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned.

If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

```
1.  String s1="Welcome";
2.  String s2="Welcome";//It doesn't create a new instance
```

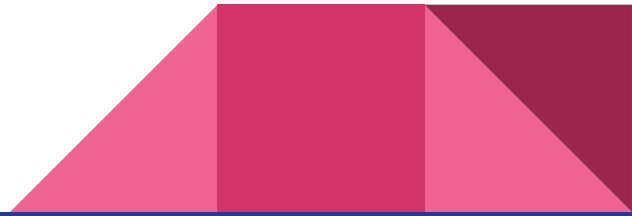


2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.

The variable s will refer to the object in a heap (non-pool).



```
public class StringExample
{
public static void main(String args[])
{
String s1="java";//creating string by java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}
}
```

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.

6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char value.

14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.

21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

```
class Testimmutablestring
{
 public static void main(String args[])
{
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Output: Sachin

String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality.

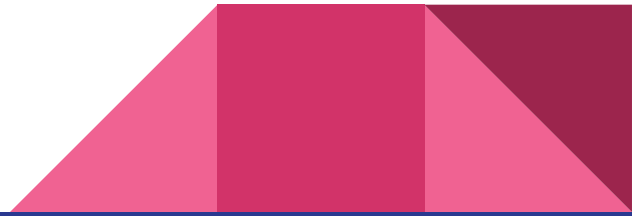
String class provides two methods:

- **public boolean equals(Object another) compares this string to the specified object.**
- **public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.**

Hello 1

HELLO 2

hello 3



```
class Teststringcomparison1
{
 public static void main(String args[])
{
  String s1="Sachin";
  String s2="Sachin";
  String s3=new String("Sachin");
  String s4="Saurav";
  System.out.println(s1.equals(s2));//true
  System.out.println(s1.equals(s3));//true
  System.out.println(s1.equals(s4));//false
 }
}
```

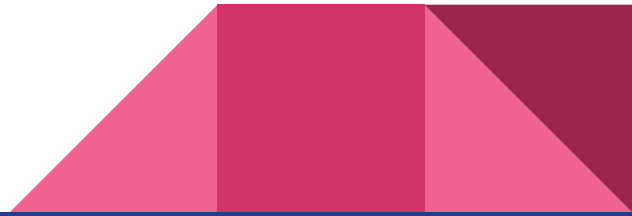
```
class Teststringcomparison1
{
 public static void main(String args[])
{
  String s1="Sachin";
  String s2="sachin";
  String s3=new String("Sachin");
  String s4="Saurav";
  System.out.println(s1.equalsIgnoreCase(s2));//true
  System.out.println(s1.equals(s2));//false
  System.out.println(s1.equalsIgnoreCase(s3));//true
  System.out.println(s1.equalsIgnoreCase(s4));//false
 }
}
```

String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2 :0**
- **s1 > s2 :positive value**
- **s1 < s2 :negative value**



```
class Teststringcomparison4
{
 public static void main(String args[])
{
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(because s1>s3)
   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 }
}
```

Output:

0
1
-1

Sachin
Sachon
Sachin 6

compareToIgnoreCase()

String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

```
public String concat(String another)
```

Example

```
class TestStringConcatenation3
{
 public static void main(String args[])
{
  String s1="Sachin ";
  String s2="Tendulkar";
  String s3=s1.concat(s2);
  //   String s3=s2.concat(s1);    // Tendulkar  Sachin
  System.out.println(s3);//Sachin Tendulkar
 }
}
```

OUTPUT :

```
Sachin Tendulkar
```

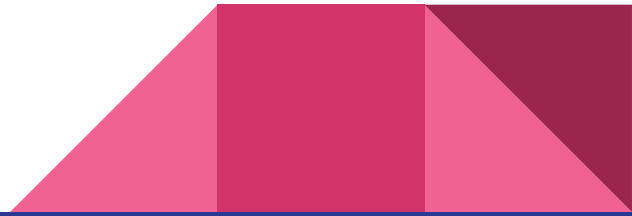
Java String toLowerCase()

The java string toLowerCase() method returns the string in lowercase letter.

In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method.

It internally uses the default locale.



```
public class StringLowerExample
{
public static void main(String args[])
{
String s1="JAVACLASS stARTING";
String s1lower=s1.toLowerCase();
// String s1lower=s1.toUpperCase();    // Make all the character upper case
System.out.println(s1lower);
}
}
```

OUTPUT:

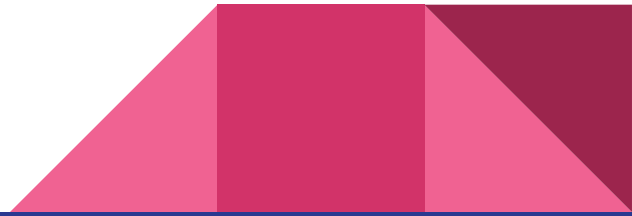
javaclass starting

trim() method

The string trim() method eliminates white spaces before and after string.

```
String s="  Sachin  ";

System.out.println(s);

System.out.println(s.trim());
```

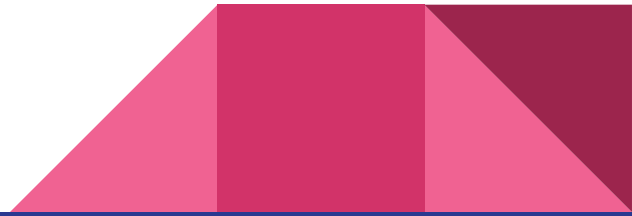
```
  Sachin

Sachin
```



startsWith() and endsWith() method

```
String s="Sachin";
 System.out.println(s.startsWith("Sa"));
 System.out.println(s.endsWith("n"));
```

```
true
true
```

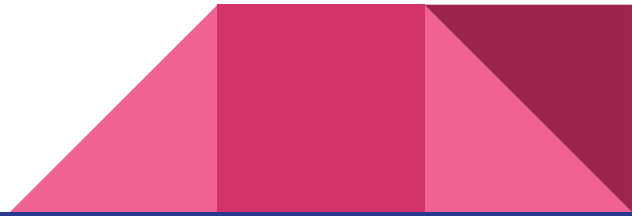


charAt() method

The string charAt() method returns a character at specified index.

```
String s="Sachin";

System.out.println(s.charAt(0));

System.out.println(s.charAt(3));
```

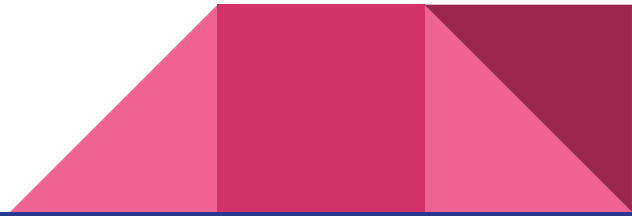
```
S

h
```



length() method

The string length() method returns length of the string.

```
1.  String s="Sachin";
2.  System.out.println(s.length());        //6
```



Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Constructor of StringBuffer

StringBuffer()

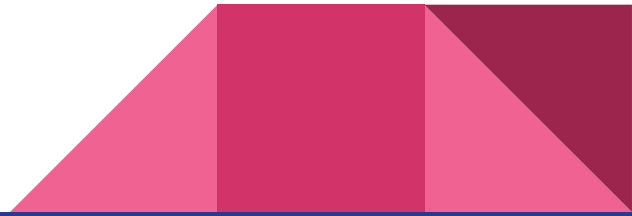
creates an empty string buffer with the initial capacity of 16.

StringBuffer(String str)

creates a string buffer with the specified string.

StringBuffer(int capacity)

creates an empty string buffer with the specified capacity as length.



Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Constructor of StringBuilder

StringBuilder()

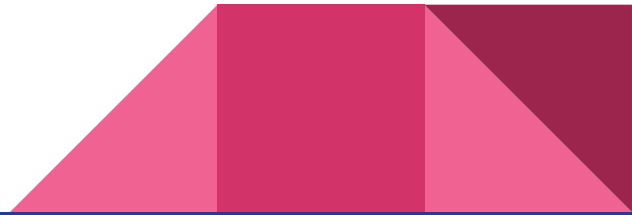
creates an empty string Builder with the initial capacity of 16.

StringBuilder(String str)

creates a string Builder with the specified string.

StringBuilder(int length)

creates an empty string Builder with the specified capacity as length.



Difference between String and StringBuffer

1)

String class is immutable.

StringBuffer class is mutable.

2)

String is slow and consumes more memory when you concat too many strings because every time it creates new instance.

StringBuffer is fast and consumes less memory when you cancat strings.

3)

String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.

StringBuffer class doesn't override the equals() method of Object class.

Difference between StringBuffer and StringBuilder

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

Exception Handling in Java

The Exception Handling in Java is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. **Checked Exception**
2. **Unchecked Exception**
3. **Error**

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.

Checked exceptions are checked at compile-time.

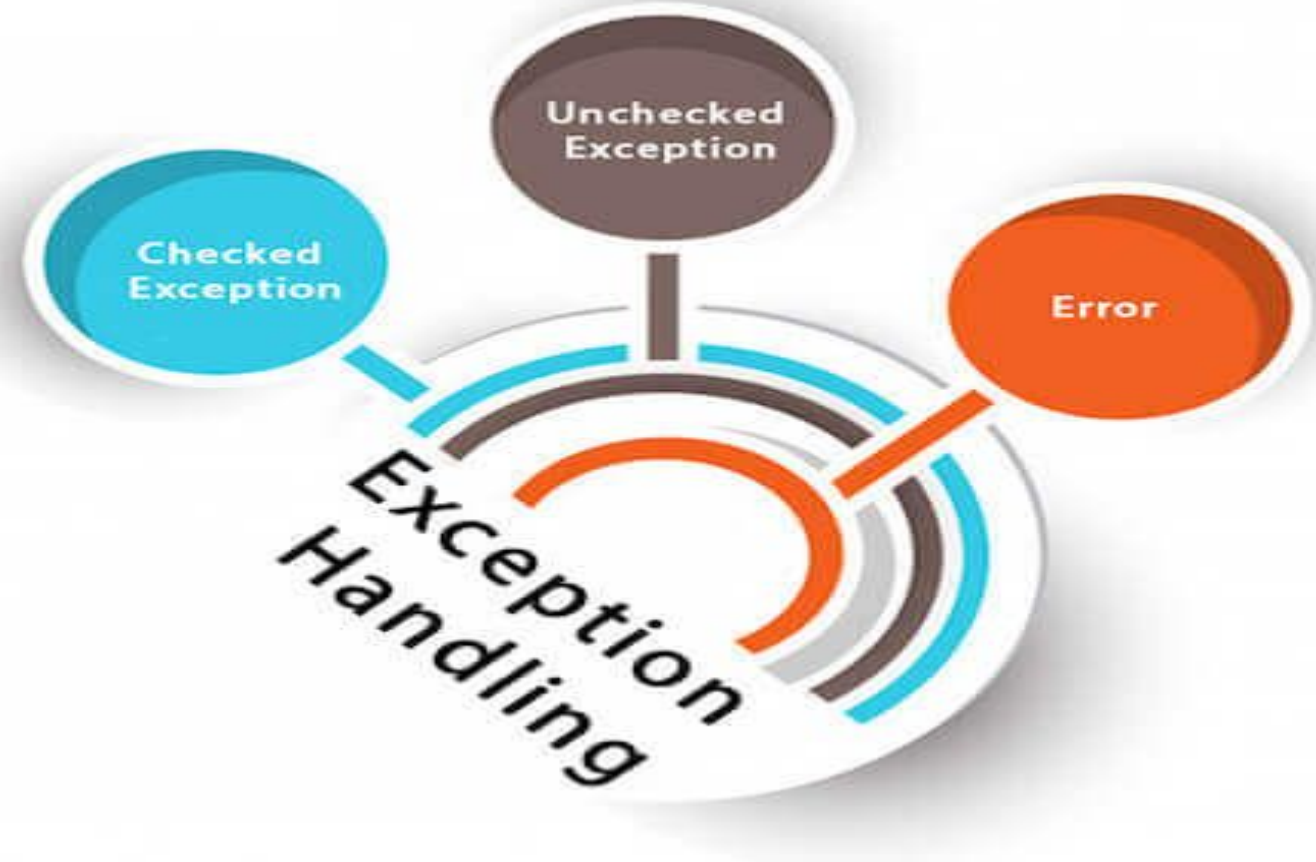
2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

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


Unchecked Exception
Checked Exception
Error
Exception Handling

Java Exception Keywords

try

- The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally.
- It means, we can't use try block alone.

catch

- The "catch" block is used to handle the exception.
- It must be preceded by try block which means we can't use catch block alone.
- It can be followed by finally block later.

finally

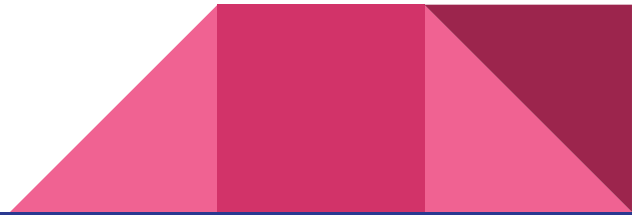
- The "finally" block is used to execute the important code of the program.
- It is executed whether an exception is handled or not.

throw

- **The "throw" keyword is used to throw an exception.**

throws

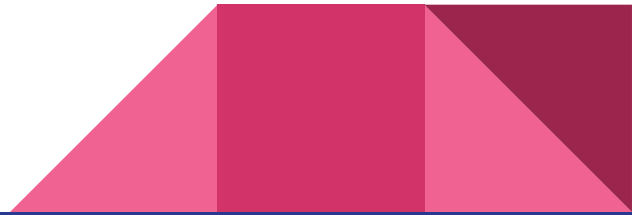
- **The "throws" keyword is used to declare exceptions.**
- **It doesn't throw an exception.**
- **It specifies that there may occur an exception in the method.**
- **It is always used with method signature.**



Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch



Exception Handling Example

```
public class JavaExceptionExample
{
  public static void main(String args[])
{
   try{       //code that may raise exception
     int data=100/0;
   }
catch(ArithmeticException e)
{
System.out.println(e);
}     //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

OUTPUT:-

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

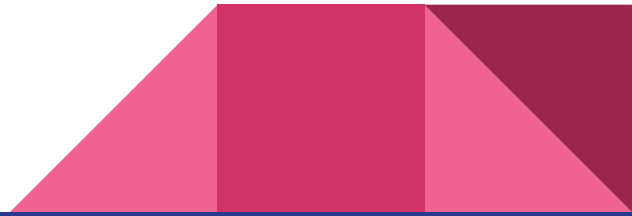
Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1.  int a=50/0;//ArithmeticException
```



2) A scenario where NullPointerException occurs

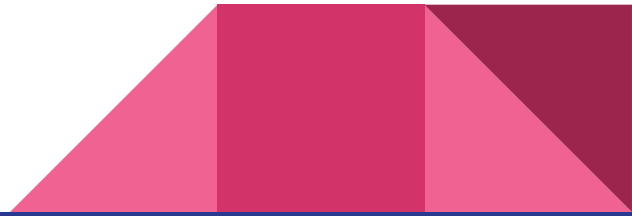
If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

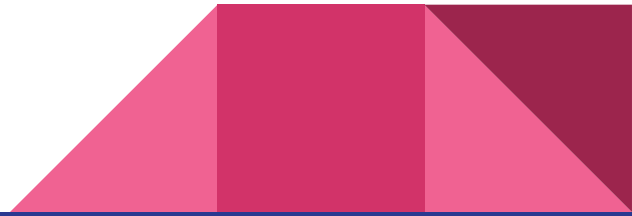
```
1.  String s="abc";
2.  int i=Integer.parseInt(s);//NumberFormatException
```



4) A scenario where ArrayIndexOutOfBoundsException occurs

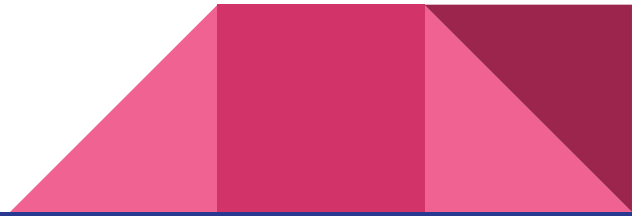
If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
1.  int a[]=new int[5];
2.  a[10]=50; //ArrayIndexOutOfBoundsException
```



Java try block

- Java **try** block is used to enclose the code that might throw an exception.
- It must be used within the method.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute.
- So, it is recommended not to keeping the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

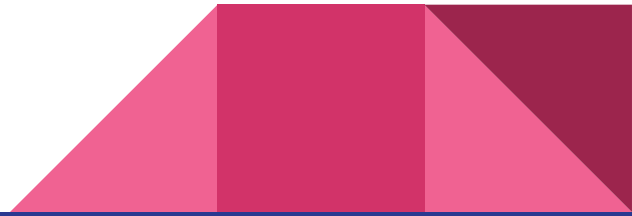


```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

- **Java catch block is used to handle the Exception by declaring the type of exception within the parameter.**
- **The declared exception must be the parent class exception (i.e., Exception) or the generated exception type.**
- **However, the good approach is to declare the generated type of exception.**
- **The catch block must be used after the try block only.**
- **You can use multiple catch block with a single try block.**



```
public class TryCatchExample1
{
   public static void main(String[] args)
 {
       int data=50/0; //may throw exception
       System.out.println("rest of the code");
  }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

```
public class TryCatchExample2
{
   public static void main(String[] args)
{
      try
      {
      int data=50/0; //may throw exception
      }           //handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }
}
```

```
java.lang.ArithmeticException: / by zero
rest of the code
```

```
public class TryCatchExample9 {

   public static void main(String[] args) {

      try

      {

      int arr[]= {1,3,5,7};

      System.out.println(arr[10]); //may throw exception

      }           // handling the array exception

      catch(ArrayIndexOutOfBoundsException e)

      {

         System.out.println(e);

      }

      System.out.println("rest of the code");

   }

}
```

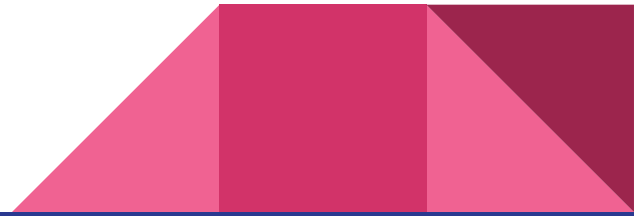
```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Java Nested try block

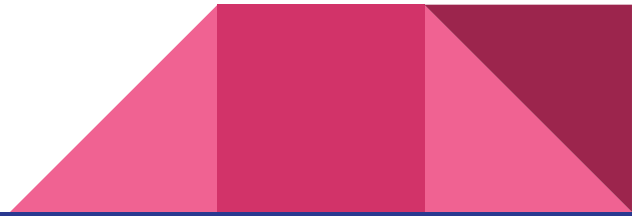
The try block within a try block is known as nested try block in java.

Why use nested try block

- **Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error.**
- **In such cases, exception handlers have to be nested.**



```
class Excep6{
 public static void main(String args[])
{
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e)
{
System.out.println(e);
}
```



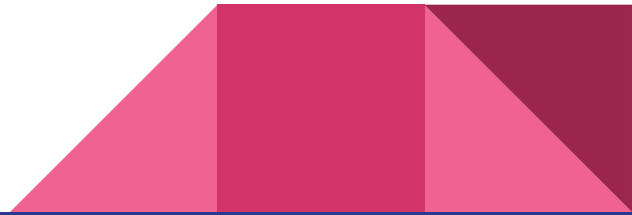
```
   try
{
   int a[]=new int[5];
   a[5]=4;
   }catch(ArrayIndexOutOfBoundsException e)
{
System.out.println(e);
}
   System.out.println("other statement);
  }catch(Exception e)
{
System.out.println("handeled");
}
  System.out.println("normal flow..");
 }
```

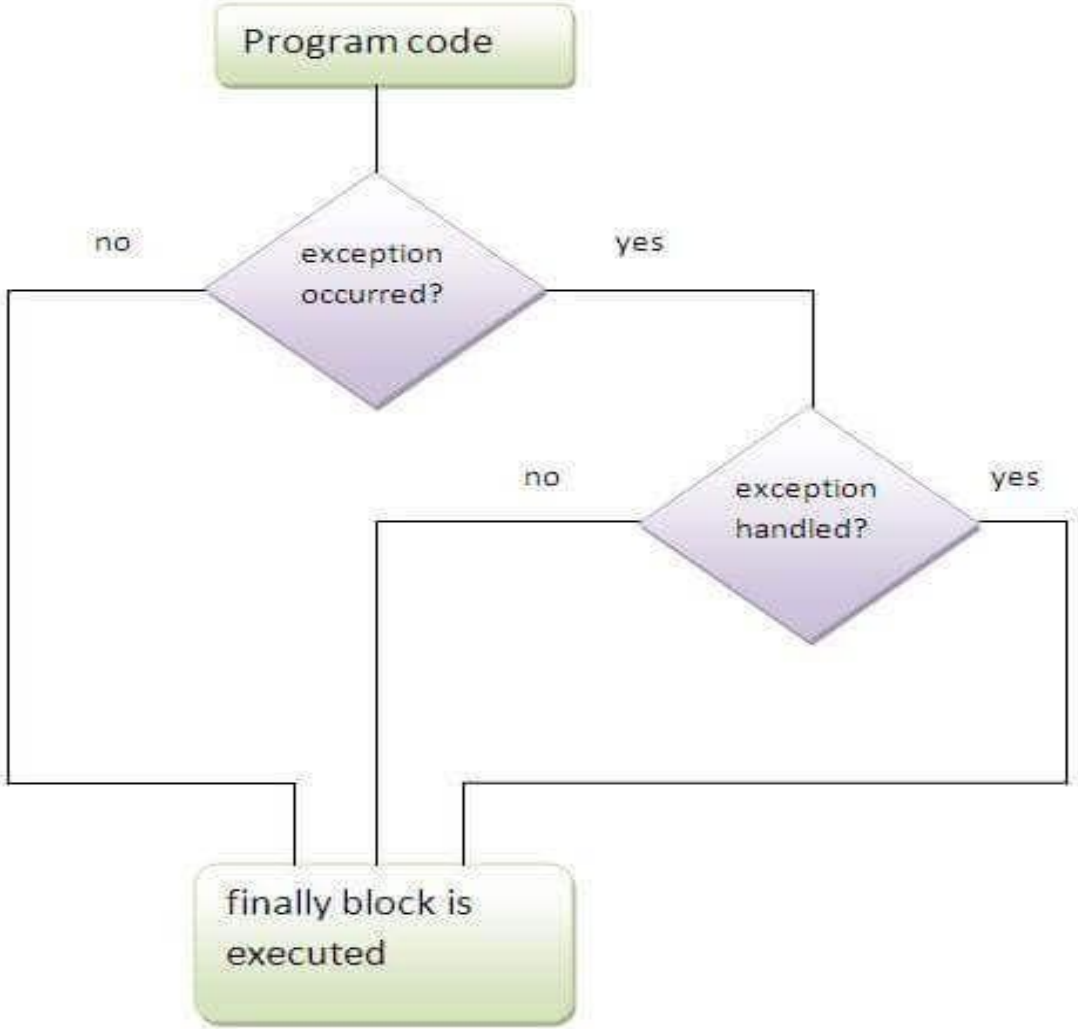
Java finally block

- **Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.**
- **Java finally block is always executed whether exception is handled or not.**
- **Java finally block follows try or catch block.**

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.




Program code
exception occurred?
no
yes
exception handled?
no
yes
finally block is executed

EXAMPLE

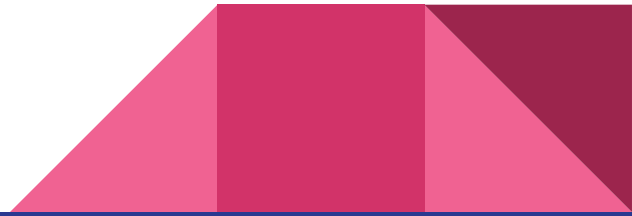
```
public class TestFinallyBlock2
{
  public static void main(String args[])
{
  try
  {
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e)
{
System.out.println(e);
}
finally
{
System.out.println("finally block is always
executed");
}
  System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by
zero

finally block is always executed

rest of the code...
```

Java throw keyword

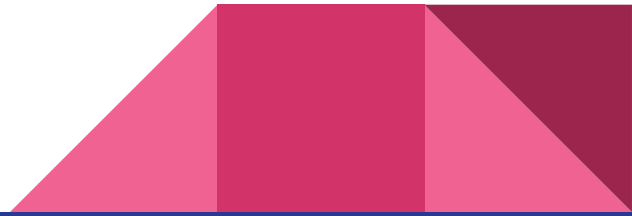
- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.
- We will see custom exceptions later.
- The syntax of java throw keyword is given below.



```
1.  throw exception;
```

Let's see the example of throw IOException.

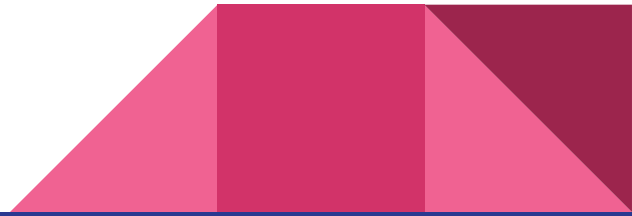
```
1.  throw new IOException("sorry device error);
```



Java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter.

If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.



```
public class TestThrow1
{
  static void validate(int age)
{
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[])
{
     validate(13);
     System.out.println("rest of the code...");
 }
}
```

OUTPUT :

```
Exception in thread main
java.lang.ArithmeticException:not valid
```

Java throws keyword

- **The Java throws keyword is used to declare an exception.**
- **It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.**
- **Exception Handling is mainly used to handle the checked exceptions.**
- **If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.**

Syntax of java throws

```
return_type method_name() throws exception_class_name{

//method code

}
```

EXAMPLE

```
import java.io.*;
class M
{
 void method()throws IOException
{
  throw new IOException("device error");
 }
}
```

```
public class Testthrows2
{
  public static void main(String args[])
{
   try
{
    M m=new M();
    m.method();
   }
catch(Exception e)
{
System.out.println("exception handled");
}
   System.out.println("normal flow...");
 } }
```

Difference between throw and throws

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Difference between final, finally and finalize

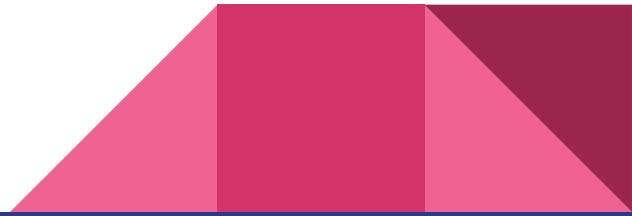
No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
class FinalExample
{
public static void main(String[] args)
{
final int x=100;
x=200;//Compile Time Error
}
}
```

Java finally example

```
class FinallyExample
{
public static void main(String[] args)
{
try{
int x=300;
}
catch(Exception e)
{
System.out.println(e);
}
Finally
{
System.out.println("finally block is executed");
}
}
}
```



Java finalize example

```
class FinalizeExample{
public void finalize()
{
System.out.println("finalize called");
}
public static void main(String[] args)
{
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}
}
```

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content.

It runs inside the browser and works at client side.

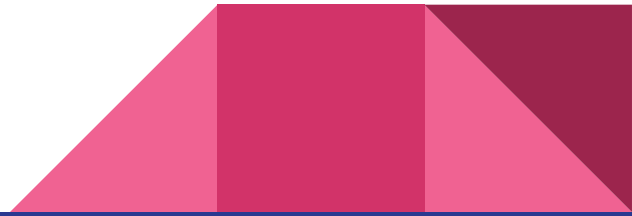
Advantage of Applet

There are many advantages of applet. They are as follows:

- **It works at client side so less response time.**
- **Secured**
- **It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.**

Lifecycle of Java Applet

1. **Applet is initialized.**
2. **Applet is started.**
3. **Applet is painted.**
4. **Applet is stopped.**
5. **Applet is destroyed**



Applet Lifecycle

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

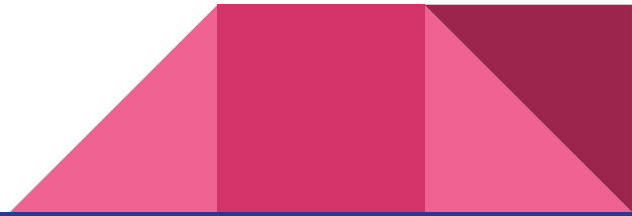
The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.**

How to run an Applet?

There are two ways to run an applet

1. **By html file.**
2. **By appletViewer tool (for testing purpose).**



Simple example of Applet by html file:

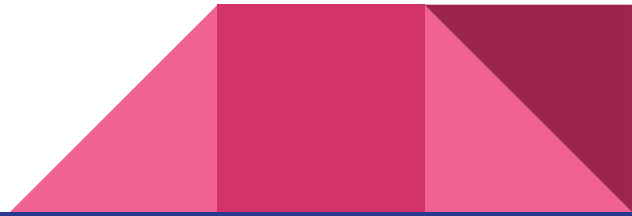
To execute the applet by html file, create an applet and compile it.

After that create an html file and place the applet code in html file. Now click the html file.

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
 public void paint(Graphics g)
{
g.drawString("welcome",150,150);
}
}
```

myapplet.html

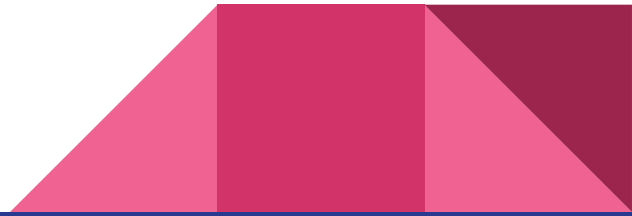
```
<html>

<body>

<applet code="First.class" width="300" height="300">

</applet>

</body>

</html>
```



Simple example of Applet by appletviewer tool:

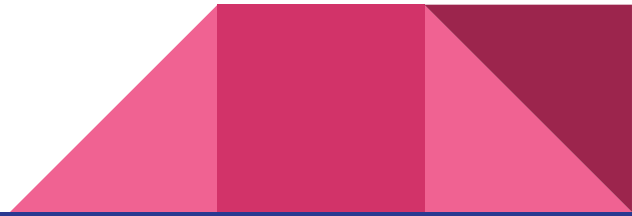
To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it.

After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.



```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
/*
<applet code="First.class" width="300" height="300">
</applet>
*/

public class First extends Applet{
public void paint(Graphics g){
g.drawString("Hello World",150,150);
}
}
```

```
c:\> javac First.java

c:\> appletviewer First.java
```




Applet Viewer: HelloWorld...
Applet
Hello World
Applet started.

```
*/
import java.awt.*;
import java.applet.*;
public class DisplayImageInApplet extends Applet
{
	Image img;
	public void init()
	{
		img = getImage(getCodeBase(),"Images/Juggler.gif");
	}
	public void paint(Graphics g)
	{
		g.drawImage(img,10,20,this);
	}
}
```


Applet Viewer: DisplayImageInApplet.class
Applet
Applet started

Displaying Image in Applet

Applet is mostly used in games and animation.

For this purpose image is required to be displayed.

The java.awt.Graphics class provide a method drawImage() to display the image.

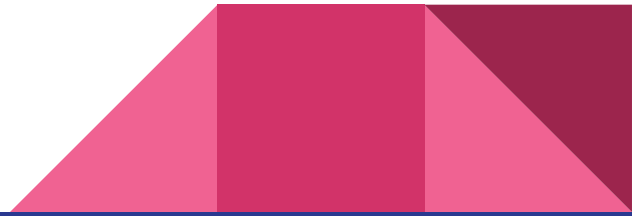
Syntax of drawImage() method:

1. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):**

   is used draw the specified image.

Other required methods of Applet class to display image:

1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded.
2. **public URL getCodeBase():** is used to return the base URL.



Example of displaying image in applet:

```
import java.awt.*;
import java.applet.*;
  public class DisplayImage extends Applet {
    Image picture;
    public void init() {
    picture = getImage(getDocumentBase(),"sonoo.jpg");
  }
     public void paint(Graphics g) {
    g.drawImage(picture, 30,30, this);
  }
  }
```

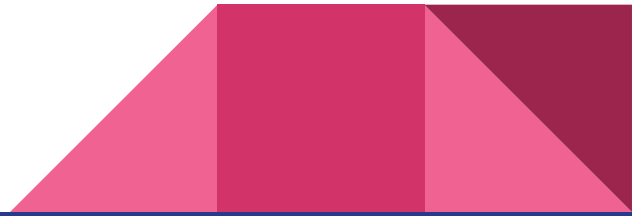
```
<html>
<body>
<applet code="DisplayImage.class" width="300" height="300">
</applet>
</body>
</html>
```

Animation in Applet

Applet is mostly used in games and animation. For this purpose image is required to be moved.

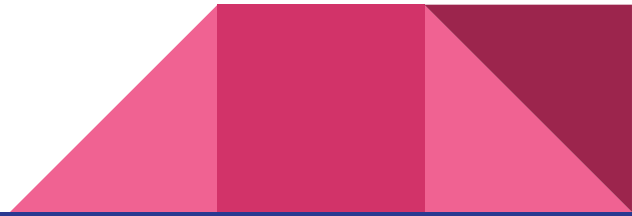
```
import java.awt.*;
import java.applet.*;
public class AnimationExample extends Applet {
   Image picture;
  public void init() {
    picture =getImage(getDocumentBase(),"bike_1.gif");
  }
```

```
public void paint(Graphics g) {
   for(int i=0;i<500;i++){
     g.drawImage(picture, i,30, this);
  try{Thread.sleep(100);}catch(Exception e){}
   }
 }
}
```



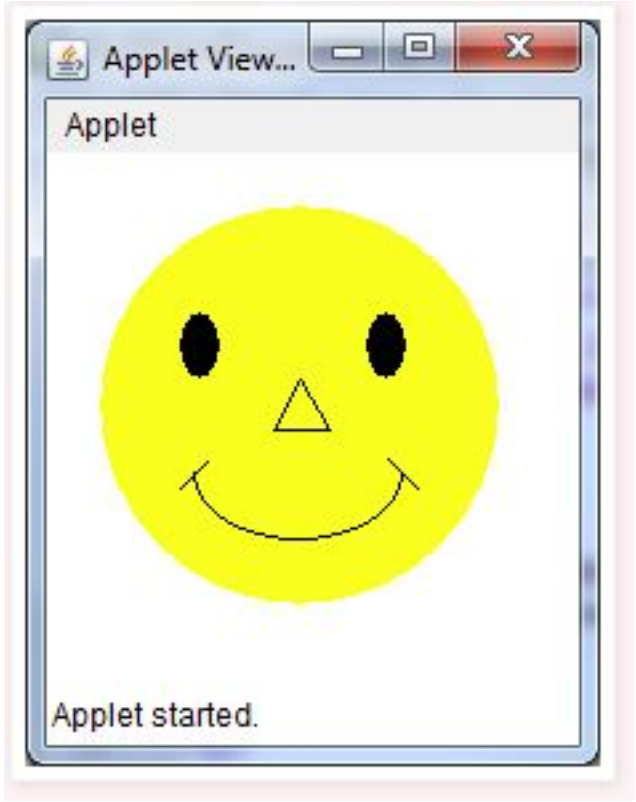
myapplet.html

```
<html>

<body>

<applet code="DisplayImage.class" width="300" height="300">

</applet>

</body>

</html>
```



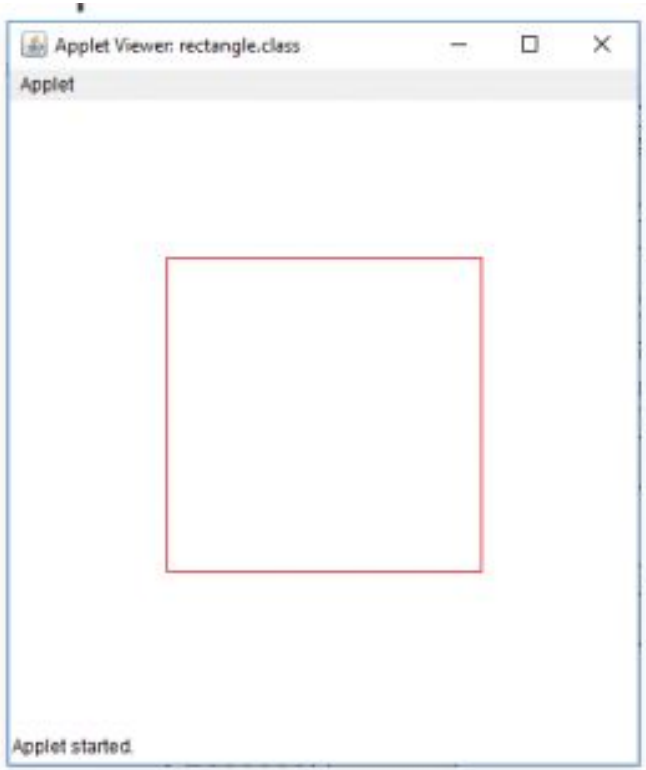
Smiling Face

```
import java.applet.Applet;

import java.awt.*;

/* <applet code="SmileyExc.class" width="200" height="200">

 </applet>

*/

 public class SmileyExc extends Applet {

 public void paint(Graphics g) {

 g.setColor(Color.yellow);

 g.fillOval(20,20,150,150);   // For face

 g.setColor(Color.black);

 g.fillOval(50,60,15,25);     // Left Eye

 g.fillOval(120,60,15,25);    // Right Eye

 int x[] = {95,85,106,95};

 int y[] = {85,104,104,85};

 g.drawPolygon(x, y, 4);      // Nose

 g.drawArc(55,95,78,50,0,-180);  // Smile

// drawArc(RectF oval, float startAngle, float sweepAngle,
boolean useCenter, Paint paint)

 g.drawLine(50,126,60,116);   // Smile arc1

 g.drawLine(128,115,139,126);  // Smile arc2

}

}
```

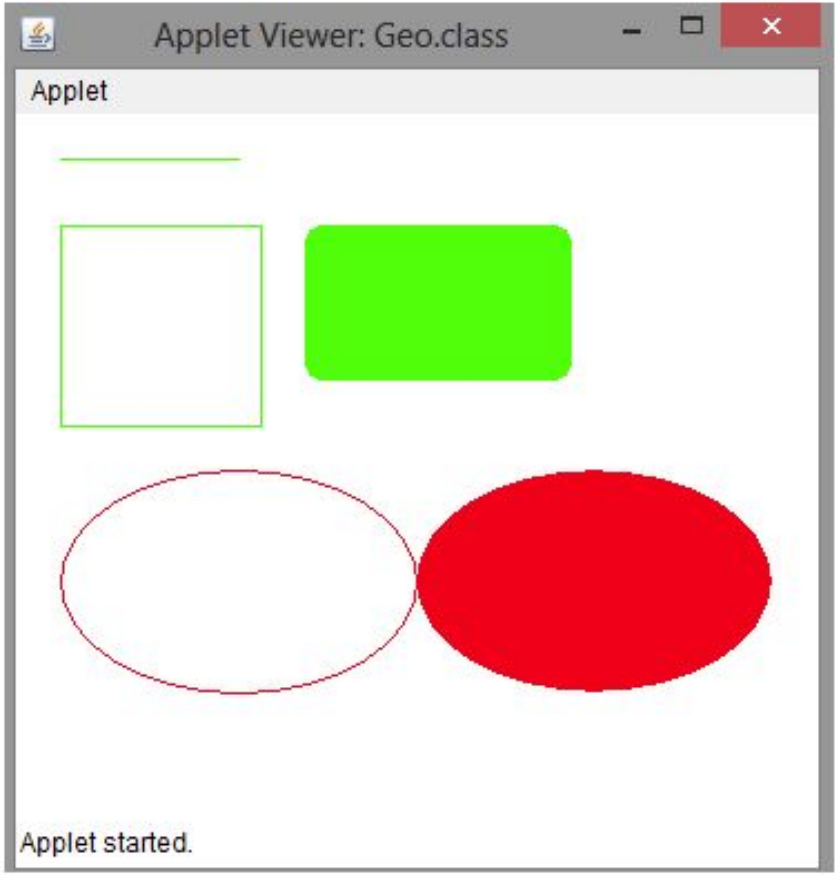

Applet View...
Applet
Applet started.

Draw a rectangle using drawRect(int x, int y, int width, int height)

```
import java.awt.*;
import javax.swing.*;
public class rectangle extends JApplet {

    public void init()
    {
        // set size
        setSize(400, 400);
        repaint();
    }

    // paint the applet
    public void paint(Graphics g)
        {
            // set Color for rectangle
            g.setColor(Color.red);
            // draw a rectangle
            g.drawRect(100, 100, 200, 200);
        }
}
```


Applet Viewer: rectangle.class
Applet
Applet started.

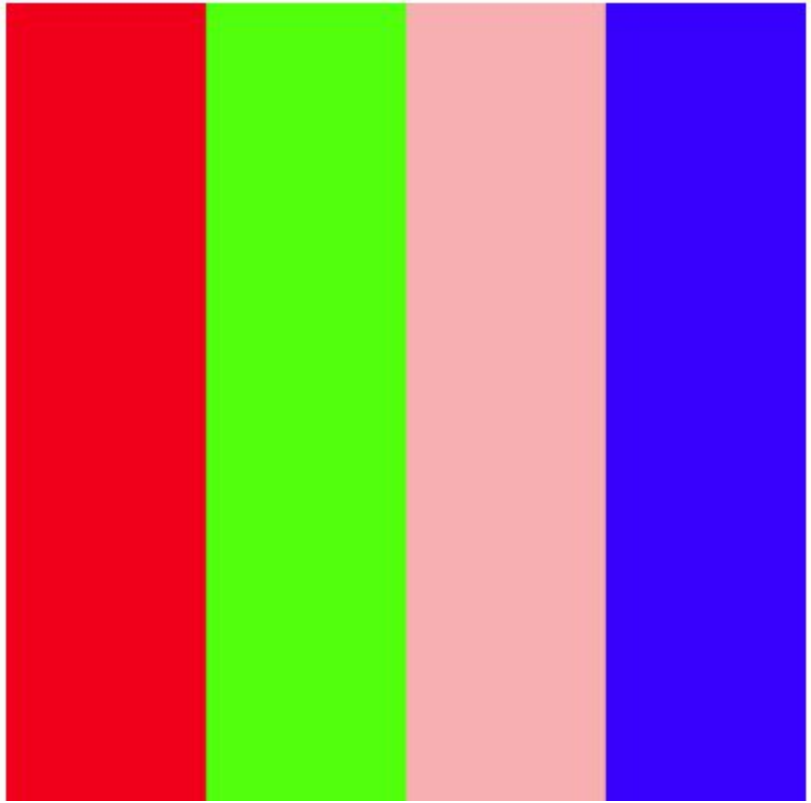
```
import java.applet.*;
import java.awt.*;
 /* <applet code="Geo.class" width =300  height=300>
   </applet>
*/
public class Geo extends Applet
{
 public void paint(Graphics g)
 {
   g.setColor(Color.GREEN);
   g.drawLine(20,20,100,20);
   g.drawRect(20,50,90,90);
   g.fillRoundRect(130,50,120,70,15,15);
   g.setColor(Color.RED);
   g.drawOval(20,160,160,100);
// drawOval( int X, int Y, int width, int height )
   g.fillOval(180,160,160,100);
  }
}
```


Applet Viewer: Geo.class
Applet
Applet started.

Write an applet that draws four Vertical bars of equal size & of different colors such that they cover up the whole applet area

```
import java.applet.*;

import java.awt.*;
 /*
<applet code="equalBars.class" width="500" height="500">
</applet>
*/
 public class equalBars extends Applet
{
    Color[] clr = {Color.red, Color.green, Color.pink, Color.blue};
    int cnt = 0;
```

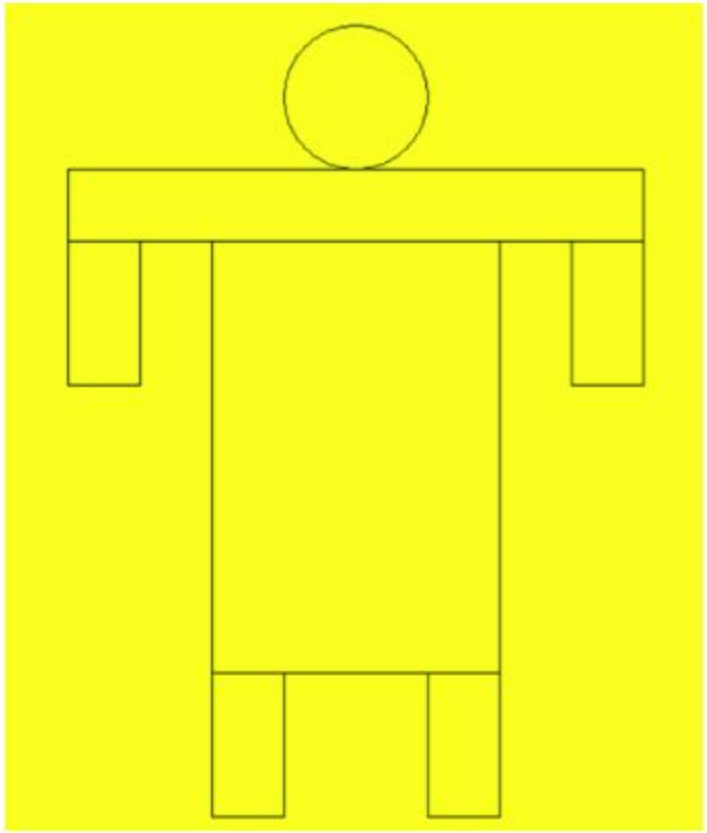
```
public void init()
   {
      setBackground(Color.black);
   }

   public void paint(Graphics g)

   {
      int w = getWidth();
      int h = getHeight();
       for(int i=0;i<4;i++)

      {
         g.setColor(clr[i]);
         g.fillRect(w/4*i,0,w/4,h);     //   fillRect(int x, int y, int width, int height)

      }

   }

}
```



Cartoon Character – Moving Hands

```
import java.applet.*;
import java.awt.*;
 /*
<applet code="Cartoon.class" width="500" height="700">
</applet>
*/
public class Cartoon  extends Applet
{
        int y = 200;
        public void init()
        {
      setBackground(Color.YELLOW);
        }
```

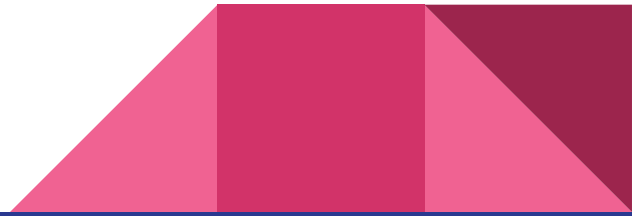
```
  public void stop()
  {
      if (y==200)
           y = 50;
      else
           y = 200;
  }

  public void paint(Graphics g)
  {
 g.drawOval(200,50,100,100);
 g.drawRect(50,150,400,50);
 g.drawRect(50,y,50,100);
 g.drawRect(400,y,50,100);
 g.drawRect(150,200,200,300);
 g.drawRect(150,500,50,100);
g.drawRect(300,500,50,100);

  }
```



Analog clock in Applet

```
import java.applet.*;
import java.awt.*;
import java.util.*;
import java.text.*;
  public class MyClock extends Applet implements Runnable {
    int width, height;
Thread t = null;
   boolean threadSuspended;
   int hours=0, minutes=0, seconds=0;
   String timeString = "";
```

```
public void init() {
    width = getSize().width;
    height = getSize().height;
    setBackground( Color.black );   }
public void start() {
    if ( t == null ) {
      t = new Thread( this );
      t.setPriority( Thread.MIN_PRIORITY );
      threadSuspended = false;
      t.start();
    }
```

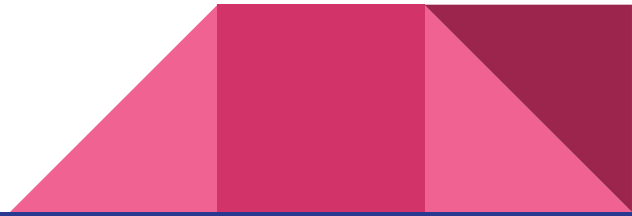


```
else {
        if ( threadSuspended ) {
           threadSuspended = false;
           synchronized( this ) {
              notify();    }
        }   }   }
public void stop() {
     threadSuspended = true;
  }
  public void run() {
```

```
Calendar cal = Calendar.getInstance();
        hours = cal.get( Calendar.HOUR_OF_DAY );
        if ( hours > 12 ) hours -= 12;
        minutes = cal.get( Calendar.MINUTE );
        seconds = cal.get( Calendar.SECOND );
SimpleDateFormat formatter
          = new SimpleDateFormat( "hh:mm:ss", Locale.getDefault() );
        Date date = cal.getTime();
        timeString = formatter.format( date );
```

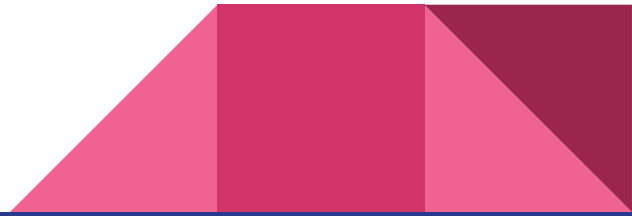
```
// Now the thread checks to see if it should suspend itself
        if ( threadSuspended ) {
          synchronized( this ) {
            while ( threadSuspended ) {
              wait();
            }     }   }
        repaint();
        t.sleep( 1000 );  // interval specified in milliseconds
      }   }
    catch (Exception e) { }
  }
```

```
void drawHand( double angle, int radius, Graphics g ) {
    angle -= 0.5 * Math.PI;
    int x = (int)( radius*Math.cos(angle) );
    int y = (int)( radius*Math.sin(angle) );
    g.drawLine( width/2, height/2, width/2 + x, height/2 + y );
  }
void drawWedge( double angle, int radius, Graphics g ) {
    angle -= 0.5 * Math.PI;
    int x = (int)( radius*Math.cos(angle) );
    int y = (int)( radius*Math.sin(angle) );
```



```
angle += 2*Math.PI/3;
    int x2 = (int)( 5*Math.cos(angle) );
    int y2 = (int)( 5*Math.sin(angle) );
    angle += 2*Math.PI/3;
    int x3 = (int)( 5*Math.cos(angle) );
    int y3 = (int)( 5*Math.sin(angle) );
    g.drawLine( width/2+x2, height/2+y2, width/2 + x, height/2 + y );
    g.drawLine( width/2+x3, height/2+y3, width/2 + x, height/2 + y );
    g.drawLine( width/2+x2, height/2+y2, width/2 + x3, height/2 + y3 );
  }
```

```
public void paint( Graphics g ) {
    g.setColor( Color.gray );
    drawWedge( 2*Math.PI * hours / 12, width/5, g );
    drawWedge( 2*Math.PI * minutes / 60, width/3, g );
    drawHand( 2*Math.PI * seconds / 60, width/2, g );
    g.setColor( Color.white );
    g.drawString( timeString, 10, height-10 );
  }
}
```



myapplet.html

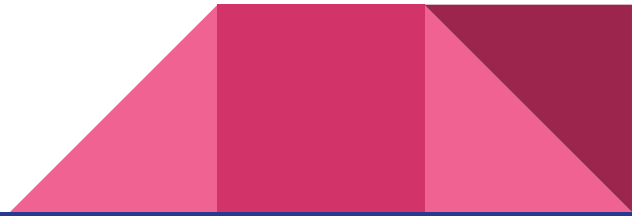
```
<html>
<body>
<applet code="MyClock.class" width="300" height="300">
</applet>
</body>
</html>
```

Applet Communication

java.applet.AppletContext class provides the facility of communication between applets. We provide the name of applet through the HTML file. It provides getApplet() method that returns the object of Applet.

Syntax:

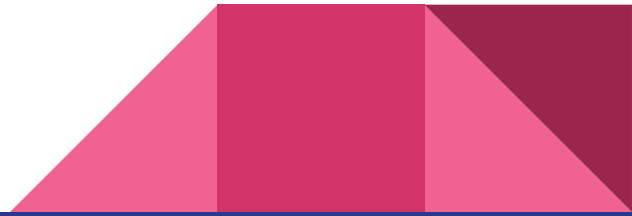
```
1.  public Applet getApplet(String name){}
```



Write a program to generate a Bouncing Ball on Click

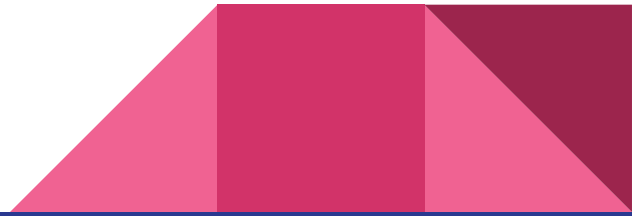
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ContextApplet extends Applet implements ActionListener{
Button b;
  public void init(){
b=new Button("Click");
b.setBounds(50,50,60,50);
  add(b);
b.addActionListener(this);
}
```

```
public void actionPerformed(ActionEvent e){
  AppletContext ctx=getAppletContext();
Applet a=ctx.getApplet("app2");
a.setBackground(Color.yellow);
}
}
```



myapplet.html

```
<html>
<body>
<applet code="ContextApplet.class" width="150" height="150" name="app1">
</applet>
  <applet code="First.class" width="150" height="150" name="app2">
</applet>
</body>
</html>
```



Java Swing

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is used to create window-based applications.

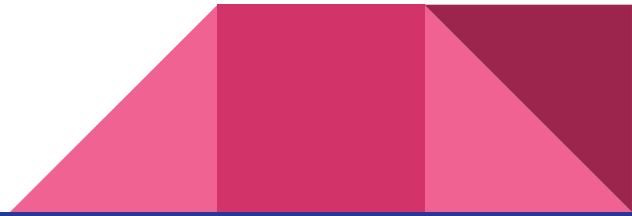
It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Java Swing provides platform-independent and lightweight components.

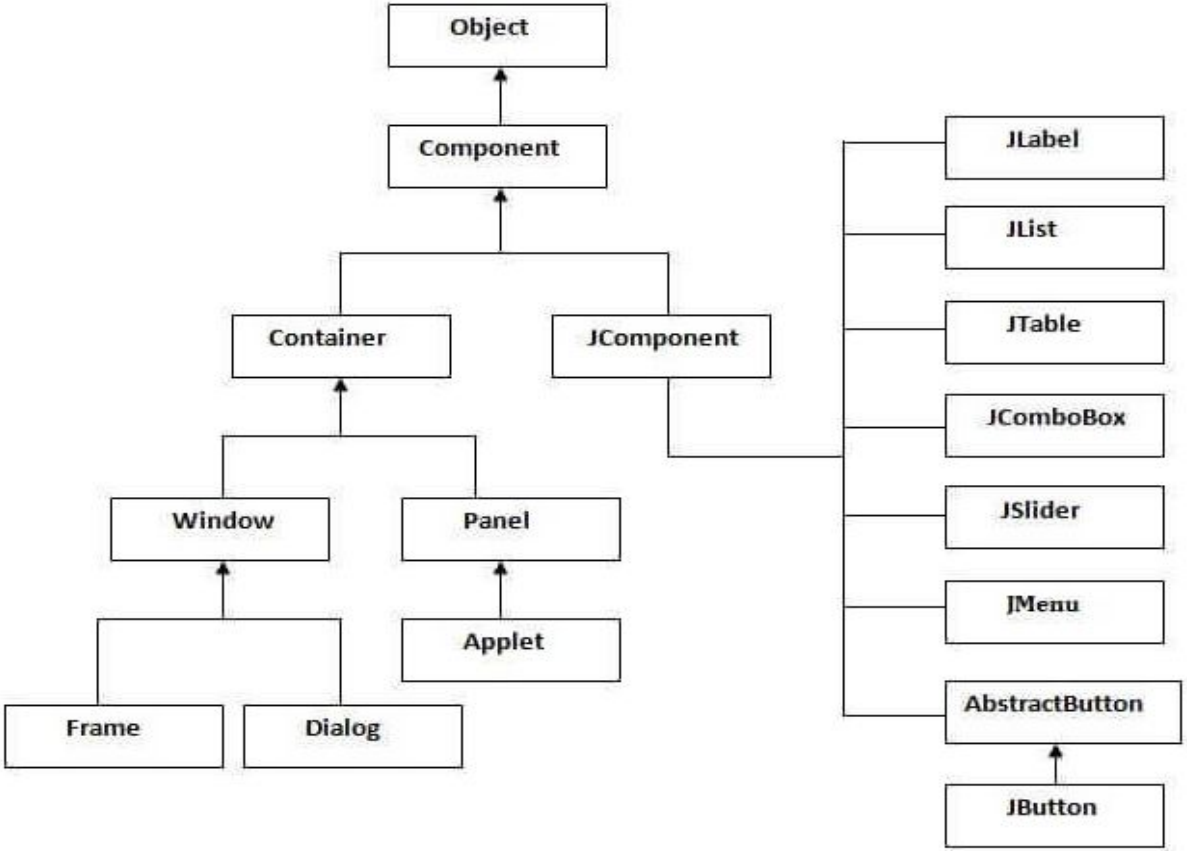
Swing is more flexible and robust

Some of the basic classes are

JApplet
JFame
JButton

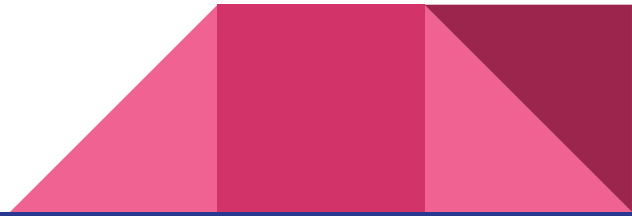


Hierarchy of Java Swing classes



Features of Swing

- Swing components are Lightweight
- Having a number of built in controls
- We can change the look and feel of the GUI
- Swing is not thread safe



A simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

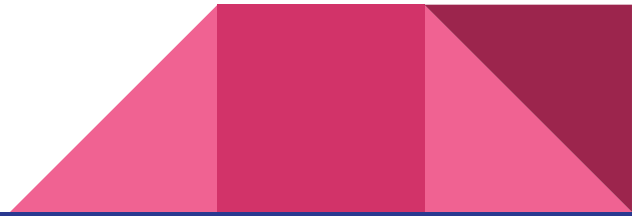
```
import javax.swing.*;

public class FirstSwingExample

{

public static void main(String[] args)

{

JFrame f=new JFrame();//creating instance of JFrame

JButton b=new JButton("click");//creating instance of JButton

b.setBounds(130,100,100, 40);//x axis, y axis, width, height

f.add(b);//adding button in JFrame

f.setSize(400,500);//400 width and 500 height

f.setLayout(null);//using no layout managers

f.setVisible(true);//making the frame visible

}

}
```

```
javac  FirstSwingExample .java

java  FirstSwingExample
```


click

Java JRadioButton

- The JRadioButton class is used to create a radio button.
- It is used to choose one option from multiple options.
- It is widely used in exam systems or quiz.
- It should be added in ButtonGroup to select one radio button only.

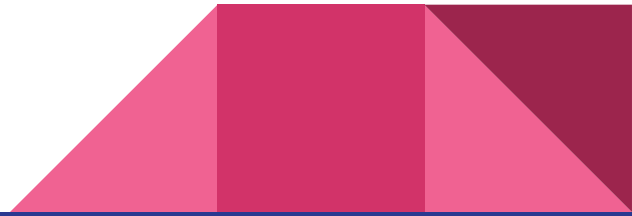


```
import javax.swing.*;
public class RadioButtonExample {
JFrame f;
RadioButtonExample(){
f=new JFrame();
JRadioButton r1=new JRadioButton("A) Male");
JRadioButton r2=new JRadioButton("B) Female");
r1.setBounds(75,50,100,30);
r2.setBounds(75,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(r1);bg.add(r2);
f.add(r1);f.add(r2);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String[] args)
{
    new RadioButtonExample();
}
}
```


A) Male
B) Female

Java JComboBox

- The object of Choice class is used to show popup menu of choices.
- Choice selected by user is shown on the top of a menu.
- It inherits JComponent class.

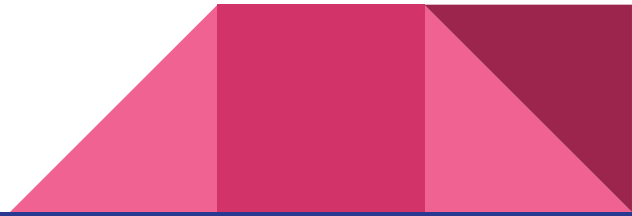


```
import javax.swing.*;
public class ComboBoxExample {
JFrame f;
ComboBoxExample(){
    f=new JFrame("ComboBox Example");
    String
country[]={"India","Aus","U.S.A","England","Newzealand"};
    JComboBox cb=new JComboBox(country);
    cb.setBounds(50, 50,90,20);
    f.add(cb);
    f.setLayout(null);
    f.setSize(400,500);
    f.setVisible(true);
}
public static void main(String[] args)
{
    new ComboBoxExample();
}
}
```

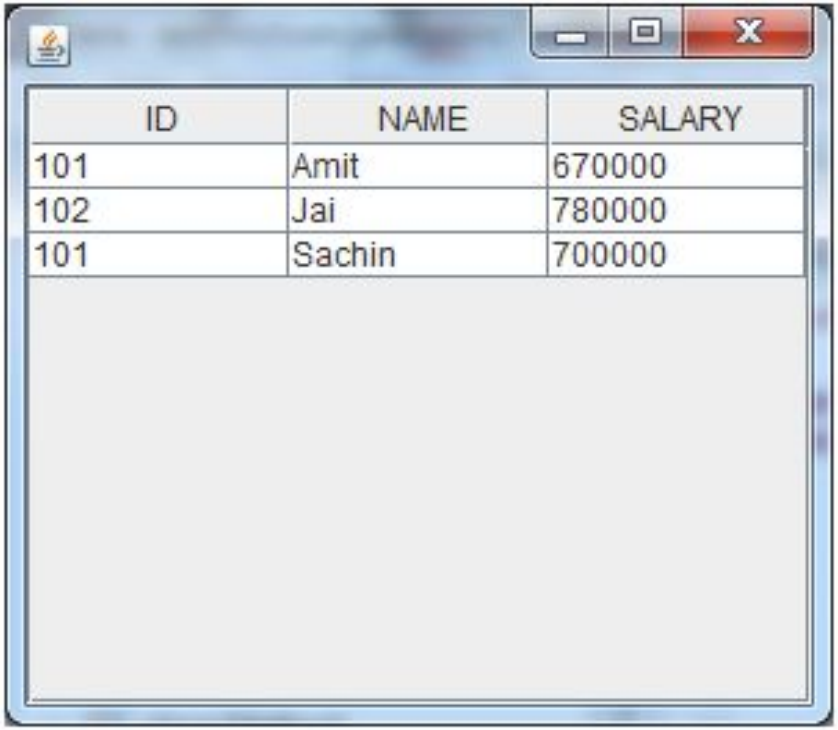

ComboBox Example
U.S.A
India
Aus
U.S.A
England
Newzealand

Java JTable

- The JTable class is used to display data in tabular form.
- It is composed of rows and columns.
- Let's see the declaration for javax.swing.JTable class.



```
import javax.swing.*;
public class TableExample {
    JFrame f;
    TableExample(){
    f=new JFrame();
    String data[][]={ {"101","Amit","670000"},
                      {"102","Jai","780000"},
                      {"101","Sachin","700000"}};
String column[]={"ID","NAME","SALARY"};
    JTable jt=new JTable(data,column);
    jt.setBounds(30,40,200,300);
    JScrollPane sp=new JScrollPane(jt);
    f.add(sp);
    f.setSize(300,400);
    f.setVisible(true);
}
public static void main(String[] args) {
    new TableExample();
}
}
```


ID
NAME
SALARY
101
Amit
670000
102
Jai
780000
101
Sachin
700000