1.Git

2. Core Java ->

2.1 -> Conditional Statements i.e if else if else

Current Bill Generation

10 units -> 10 per unit = 10\*10 =100

15 units -> 15 per unit

20 units -> 20 per unit

Above -> 30 per unit

3. Looping Statements

For loop -> for( int i=1;i<=5;i++ ) {

// statements

for(int j=1;j<=5;j++) {

}

}

For Each => we can discuss later.

While Loop =>

While(condition) {

}

Do while loop:

Do {

}while(condition);

4.Arrays.

Int I = 250;

Int []a = {1,2,3,4,5}; a.length = 5;

Int []a = new int[5];

a[0] = 1;

a[1] = 2;

a[2] = 3;

a[3] = 4;

a[4] = 5;

Arrays tasks:

int arr[] = {1,5,3,2,4};

asc = [1,2,3,4,5];

des = [5,4,3,2,1];

String arr[] = {“Naresh”, “Triveni”,”Veera”, “Sai”};

asc = [“Naresh”, “Sai”, “Triveni”, “Veera”];

desc = [“Veera”, “Triveni”,”Sai”,”Naresh”];

5. Packages

6. Object

If we want to create the memory and save some values we can create object.

7.Methods

Syntax For Method

accessModifier returnType methodName(Parameters) {}

Create Arithmetic class and create methods with return types.

OOPS Concepts:

1.Class

2.Object

3.Inheritence

4.Polymorphism

5.Encapsulation

6.Abstraction

1.Class

Class is Template or Bluprint Or UserDefined datatype => class Human {

String name;

String color;

String gender;

float height;

int weight;

}

2.Object

Object is a instance of class

It is physically exist.

Human h = new Human();

h.name = “naresh”;

h.color = “white”;

h.gender = “m”;

h,height = 5.8;

h.weight = 70;

3.Inheritence

Class Arithmetic {

Addition();

Substraction();

Multiplication();

Division();

}

Class Arithmetic2 extends Arithmetic {

Reminder();

}

Access Modifiers:

1.public: It is acceptable from one package to another package.it contains global access.public we can use at class level, method level and variable level.

2.private: private modifier we can access within the class only. private only methods and variables.

3.protected: protected modifier we can access within the same package and sub class. protected we can use at method level and variable level.

4.default: default modifier within the same package only. default we can use at class level, method level and variable level.

Encapsulation

Encapsulation is nothing but a properties and behaviours we can take as single unit or we can combine both.

26/11/2020

Abstraction:

Abstraction is nothing hiding internal implementation.

Abstraction we can achieve by 2 ways.

1. Interfaces -> 100%
2. Abstract classes -> 0 – 100%

Is it possible to create concrete methods in interface?NO

Is it possible to create concrete methods in abstract classes?Yes

Is it possible to create abstract methods in abstract classes?Yes

Task: Payroll task

Constructors

Constructor is used for to initialize the values into the object we can constructor.

Syntax for constructor:

accessModifier classname(){}

Access Modifiers for Constructor

We can private, protected, default and public as accessmodifiers.

static vs non-static, string

static is a key word, we can use static as a variable, method and block.

Static Variable

* Static variable -> The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.
* Advantages of static variable
* It makes your program **memory efficient** (i.e., it saves memory).

Static Block

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

### Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the [main method](https://www.javatpoint.com/java-main-method).

Static methods

## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

### Example of static method

1. //Java Program to demonstrate the use of a static method.
2. **class** Student{
3. **int** rollno;
4. String name;
5. **static** String college = "ITS";
6. //static method to change the value of static variable
7. **static** **void** change(){
8. college = "BBDIT";
9. }
10. //constructor to initialize the variable
11. Student(**int** r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. **void** display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. **public** **class** TestStaticMethod{
20. **public** **static** **void** main(String args[]){
21. Student.change();//calling change method
22. //creating objects
23. Student s1 = **new** Student(111,"Karan");
24. Student s2 = **new** Student(222,"Aryan");
25. Student s3 = **new** Student(333,"Sonoo");
26. //calling display method
27. s1.display();
28. s2.display();
29. s3.display();
30. }
31. }

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

### Another example of a static method that performs a normal calculation

1. //Java Program to get the cube of a given number using the static method
3. **class** Calculate{
4. **static** **int** cube(**int** x){
5. **return** x\*x\*x;
6. }
8. **public** **static** **void** main(String args[]){
9. **int** result=Calculate.cube(5);
10. System.out.println(result);
11. }
12. }

Output:125

### Restrictions for the static method

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

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.
3. **class** A{
4. **int** a=40;//non static
6. **public** **static** **void** main(String args[]){
7. System.out.println(a);
8. }
9. }

Output:Compile Time Error

### Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

HashCode

Hashcode is a method it will return address of the object.

String class

# Java String

In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

1. **char**[] ch={'n’,’a’,’r’,’e’,’s’,’h’};
2. String s=**new** String(ch);

is same as:

1. String s="naresh";

**Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence [interfaces](https://www.javatpoint.com/interface-in-java).



## CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class) and [StringBuilder](https://www.javatpoint.com/StringBuilder-class) classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

### 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



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

#### Note: String objects are stored in a special memory area known as the "string constant pool".

### Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

### 2) By new keyword

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

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) 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).

### Java String Example

1. **public** **class** StringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java";//creating string by java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}

java

strings

example

Type Casting

## Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

(or)

The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

## Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

### Example

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

## Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

### Example

public class Main {

public static void main(String[] args) {

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

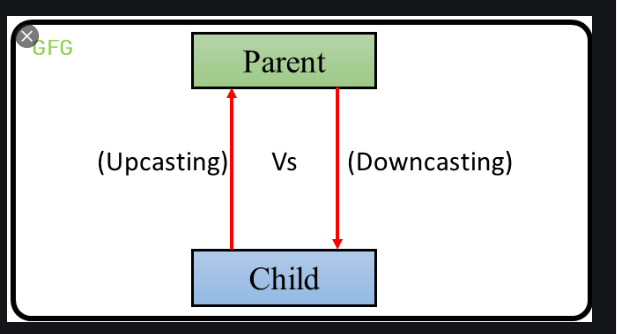
System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

Upcasting and Downcasting



Converting an object from one type to another is a very important aspect of Java which is popularly known as Typecasting. Let’s understand the concepts of Upcasting and Downcasting in Java in the following manner:

## ****What are Upcasting and Downcasting in Java?****

Upcasting (Generalization or Widening) is casting to a parent type in simple words casting individual type to one common type is called upcasting while downcasting (specialization or narrowing) is casting to a child type or casting common type to individual type.



Upcasting can be done but for downcasting, we need to check the types or else we may get

ClassCastException

## ****Hierarchy Example****

Let’s take this example.

**Food -> Fruit -> Apple, Orange**

Food is the interface which is at the topmost level

|  |  |
| --- | --- |
| 1  2  3  4 | public interface Food {     public float getTotalCalories();     public String getOrigin();  } |

The fruit is the abstract class

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public abstract class Fruit implements Food {  public float getTotalCalories(){        return 0.50f;  }     public String getOrigin();  } |

Apple and Orange are the two concrete subclasses

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Apple extends Fruit {  public float getTotalCalories() {      return 0.40f  }  public String getOrigin() {      return "someCity";    }  }  public class Orange extends Fruit {  public float getTotalCalories() {      return 0.30f  }  public String getOrigin() {      return "someOtherCity";  }  } |

In your case, a cast from an Apple to a Fruit is an upcast, because of an Apple is-a Fruit. whenever there is an inheritance i.e is-a relationship between two classes we can do upcasting.

For example, if we cast Apple to Fruit it is upcasting because Apple is of type Fruit here we are generalizing from child type to parent type. So, if there is an is-a relationship (inheritance) between two classes we can upcast.

Let’s look at a downcasting example:

Here we are narrowing the type of object i.e we are converting common type to individual type.

|  |  |
| --- | --- |
| 1  2 | Fruit fruit = new Apple();  Apple castedApple = (Apple) fruit; |

Here we are casting common type to individual type i.e superclass to subclass which is not possible directly in Java so we explicitly do the casting and tell the compiler that what the runtime type of the object. It is possible in this case because the fruit is Apple even if the reference type is Fruit.

Now, Suppose if you do this:

|  |  |
| --- | --- |
| 1  2 | Fruit fruit = new Fruit();  Apple notApple = (Apple) Fruit; |

Above code will throw an exception because fruit’s runtime object is Fruit but not Apple it is not possible to cast superclass to subclass so, this will end up with ClassCastException.

If we want to invoke the method of superclass we can simply do this using super keyword as super.methodName() or we can upcast the object.

If we want to invoke subclass’s method then we will need to downcast the object but we can run into ClassCastException so, if you want to avoid this exception you can use a keyword instanceof which will check the runtime type of the object before we cast the object as in below code.

|  |  |
| --- | --- |
|  | Fruit fruit = getSomeFruit(); #we dont really know what getSomeFruit is returning so we can check the type of fruit using instanceof  if (fruit instanceof Apple) {      // the object can be casted and the code won't fail      Apple castedApple = (Apple) fruit;  } |

As a Java developer, you will come across this usually, you may need to cast objects depending on the requirements so, now you know how to do a casting. And it isn’t really that hard to do.

## ****Why do we need Upcasting in Java?****

Generally, upcasting is not necessary. We need upcasting when we want to write code that deals with only the parent class. Consider the following class

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class CalorieMeter{       public void readCalorie(Fruit fruit){            print("Calorie:" + fruit.getTotalCalories());       }  } |

Here we can pass any subtype of Fruit to readCalorie() method, thus it accepts both the objects of Apple and Orange class as they are the subtype of Fruit class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | Apple apple = new Apple();  Orange orange = new Orange();  Caloriemeter caloriemeter = new CalorieMeter();  caloriemeter.readCalorie(apple);  caloriemeter.readCalorie(orange); |

## ****Why do we need Downcasting in Java?****

We use downcasting whenever we want to access behaviors of the subtypes. This is used more frequently than that of upcasting. Consider the following example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class CalorieMeter{       public void readCalorie(Fruit fruit){            print("Calorie:" + fruit.getTotalCalories());            //if the fruit is orange the object should print city            if(fruit instanceof Orange){               Orange orange = (Orange) fruit;               System.out.println("City:"+fruit.getOrigin());            }       }  } |

Here in readCalorie() method, we have checked the object which is passed if that object is of type Orange we have to downcast it and invoke the method getOrigin() which will give the origin of that fruit.

## ****Important Topics****

* When we cast object only the reference type of the object is changed but not the actual object type.
* Upcasting is safe and it does not fail.
* We need to check the instance of the object when we downcast the object using instanceof operator or we might get ClassCastException.

In this article, we covered what is upcasting and downcasting in Java. What is the hierarchy needed so that the object can be upcasted and downcasted also, how to use instanceof operator in order to check the type of object while downcasting so that we can avoid getting ClassCastException. Few import points like while casting only reference of object are changed and while downcasting we should check the type of object and upcasting is safe.

String Buffer and String Builder

String Buffer

StringBuffer b = new StrngBuffer(“Naresh”);

b.apppend(“Kambala”);

Exception Handling

Exception Hirarchy:



Multiithreading

# Multithreading in Java

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) 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. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### 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](https://www.javatpoint.com/register-memory), 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.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

## Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

## Java Thread Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |

S.N. Modifier and Type Method Description

1) void start() It is used to start the execution of the thread.

2) void run() It is used to do an action for a thread.

3) static void sleep() It sleeps a thread for the specified amount of time.

4) static Thread currentThread() It returns a reference to the currently executing thread object.

5) void join() It waits for a thread to die.

6) int getPriority() It returns the priority of the thread.

7) void setPriority() It changes the priority of the thread.

8) String getName() It returns the name of the thread.

9) void setName() It changes the name of the thread.

10) long getId() It returns the id of the thread.

11) boolean isAlive() It tests if the thread is alive.

12) static void yield() It causes the currently executing thread object to pause and allow other threads to execute temporarily.

13) void suspend() It is used to suspend the thread.

14) void resume() It is used to resume the suspended thread.

15) void stop() It is used to stop the thread.

16) void destroy() It is used to destroy the thread group and all of its subgroups.

17) boolean isDaemon() It tests if the thread is a daemon thread.

18) void setDaemon() It marks the thread as daemon or user thread.

19) void interrupt() It interrupts the thread.

20) boolean isinterrupted() It tests whether the thread has been interrupted.

21) static boolean interrupted() It tests whether the current thread has been interrupted.

22) static int activeCount() It returns the number of active threads in the current thread's thread group.

23) void checkAccess() It determines if the currently running thread has permission to modify the thread.

24) static boolean holdLock() It returns true if and only if the current thread holds the monitor lock on the specified object.

25) static void dumpStack() It is used to print a stack trace of the current thread to the standard error stream.

26) StackTraceElement[] getStackTrace() It returns an array of stack trace elements representing the stack dump of the thread.

27) static int enumerate() It is used to copy every active thread's thread group and its subgroup into the specified array.

28) Thread.State getState() It is used to return the state of the thread.

29) ThreadGroup getThreadGroup() It is used to return the thread group to which this thread belongs

30) String toString() It is used to return a string representation of this thread, including the thread's name, priority, and thread group.

31) void notify() It is used to give the notification for only one thread which is waiting for a particular object.

32) void notifyAll() It is used to give the notification to all waiting threads of a particular object.

33) void setContextClassLoader() It sets the context ClassLoader for the Thread.

34) ClassLoader getContextClassLoader() It returns the context ClassLoader for the thread.

35) static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler() It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.

36) static void setDefaultUncaughtExceptionHandler() It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

# Life cycle of a Thread (Thread States)

1. [Life cycle of a thread](https://www.javatpoint.com/life-cycle-of-a-thread)
   1. [New](https://www.javatpoint.com/life-cycle-of-a-thread#threadstatenew)
   2. [Runnable](https://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunnable)
   3. [Running](https://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunning)
   4. [Non-Runnable (Blocked)](https://www.javatpoint.com/life-cycle-of-a-thread#threadstateblocked)
   5. [Terminated](https://www.javatpoint.com/life-cycle-of-a-thread#threadstateterminated)

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



|  |
| --- |
| 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. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |
| [**next>>**](https://www.javatpoint.com/thread-scheduler-in-java)[**<<prev**](https://www.javatpoint.com/life-cycle-of-a-thread)  **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. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |   **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. |   **1) Java Thread Example by extending Thread class**   1. **class Multi extends Thread{** 2. **public void run(){** 3. **System.out.println("thread is running...");** 4. **}** 5. **public static void main(String args[]){** 6. **Multi t1=new Multi();** 7. **t1.start();** 8. **}** 9. **}**   **Output:thread is running...**  **2) Java Thread Example by implementing Runnable interface**   1. **class Multi3 implements Runnable{** 2. **public void run(){** 3. **System.out.println("thread is running...");** 4. **}** 6. **public static void main(String args[]){** 7. **Multi3 m1=new Multi3();** 8. **Thread t1 =new Thread(m1);** 9. **t1.start();** 10. **}** 11. **}**   **Output:thread is running...**   |  | | --- | | If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute. | |

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.