MCA FIRST SEMESTER- 2021
MCA20103-OBJECT ORIENTED PROGRAMMING USING JAVA
UNIT-4

1. Interfaces in Java:-

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

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 inheritances.
- It can be used to achieve loose coupling.

❖ 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
    // by default.
}
```

Eg:

```
// interface
  interface Animal {
 public void animalSound(); // interface method (does not have a body)
 public void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" by another class with the **implements** keyword (instead of extends). The body of the interface method is provided by the "implement" class:

Example program:

```
// Interface
 interface Animal {
 public void animalSound(); // interface method (does not have a body)
 public void sleep(); // interface method (does not have a body)
}
```

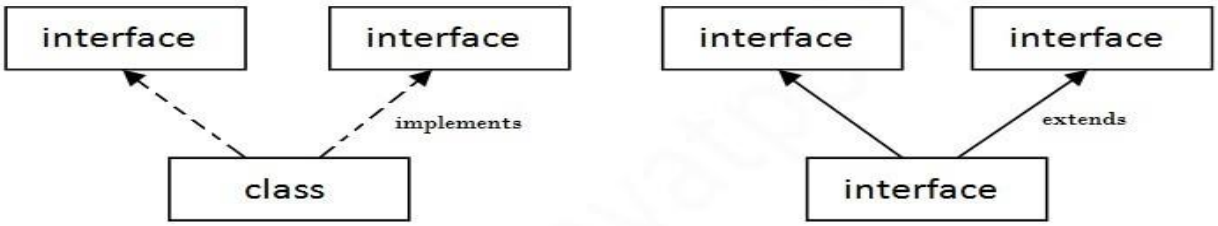
```
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

- **Rules for creating interface:**

1. Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
2. Interface methods do not have a body - the body is provided by the "implement" class
3. On implementation of an interface, we must override all of its methods
4. Interface methods are by default abstract and public
5. Interface attributes are by default public, static and final
6. An interface cannot contain a constructor (as it cannot be used to create objects)

2. Multiple inheritance in Java by interface:

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



Multiple Inheritance in Java

- **Extending Interfaces:** Like classes, interfaces can be extended that is an interface can be interfaced from other interface. A sub interface will inherits all the members from super

interface as similar to classes. This is achieved by using keyword "extends". The general form of defining a sub interface from super interface is as follows:

Syntax:

Interfacename2 extends name1
{
Body of sub interfacename 2
}

Eg: interface A
{
}
interface B extends A
{
}

- In the above example "interface B" is created from "interface A". So the properties of interface A are inherited into interface B
- In java **multiple inheritance** can be implemented on interfaces that is creating a sub interface by using more than one interface.

Eg: interface A
{
}
interface B
{
}
interface C extends A,B
{
……
……
}

In the above example interface C is created from interface A and interface B. So the members of interface A and interface B are inherited into interface C

Example program:

```
interface A {
   void funcA();
}
interface B extends A {
   void funcB();
}
class C implements B {
   public void funcA() {
      System.out.println("This is funcA");
   }
   public void funcB() {
      System.out.println("This is funcB");
```

```
   }
}
public class Demo {
   public static void main(String args[]) {
      C obj = new C();
      obj.funcA();
      obj.funcB();
   }
}
```

Output

```
This is funcA
This is funcB
```

❖ Implementing interfaces:

Any class can implement an interface. The class must be implemented with all the abstract methods of that interface. Otherwise the class will become an abstract class.

Syntax:

class CName implements IName
{
Body of the class
}

- In the above syntax "implements" is a keyword "CName" represents class name,"IName" represents interface name
- A class can implement any number of interfaces. When a class can implementing more than one interface. The interface names are separated with comma operator

Eg:

```
interface area
{
final static float PI=3.144f;
public float compute(float x,float y);
}
class rectangle implements area
{
public float compute(float x,float y)
{
return x*y;
}
}
class circle implements area
{
public float compute(float x,float y)
{
return PI*x*x;
```

```
}
}
class test
{
public static void main (String args[ ])
{
rectangle r=new rectangle();
circle c=new circle();
area a;
a=r;
System.out.println("area of rectangle="+a.compute(10,20));
a=c;
System.out.println("area of circle="+a.compute(10,0));
}
}
```

Output:

Area of rectangle=200.0

Area of circle=314.4

3. Exception handling in Java:

Exception: An exception is a condition that is caused by a run time error in the program. When the java interpreter finds an error it creates an exception object and throws it. If we want to continue the program execution we should try to catch the exception object and process it. This process is known as exception handling

The process of exception handling is to direct and report exceptional conditions so that appropriate action can be taken. The basic concepts of exception handling are throwing an exception and catching it.

The general form of exception handling is as follows

There are 5 keywords used in java exception handling

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

- **try and catch statements:**

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

Syntax:

```
try
{
……
…..
The statement that generates an exception
…….
…..
}
catch(ExceptionType)
{
…….
……
The statements that Handle exception
…….
…..
}
```

In the above syntax the try block can have one or more statements that may generates an exception. If any statement generates an exception the remaining statements in try block are skipped and execution jumps to the catch block.

The catch block can have one or more statements that process the exception. The catch statement has one argument which refers to an exception type. If the catch statement then the statements in the catch block will be executed otherwise the exception will terminate.

Eg:

/* usage of exception*/

```
class Example
{
public static void main(String args[ ])
{
int a=10,b=5,c=5,x,y;
try
{
x=a/(b-c);
}
catch(ArithmeticException e)
{
System.out.println("Division by zero");
}
y=a/(b+c);
System.out.println("Y="+y);
}
}
```

Output:

Division by zero

Y=1

- **Multiple catch statements:**

It is possible to have more than one catch statement in the catch block. The syntax for multiple catch statement is as follows.

syntax:

```
try
{
Statements;
}
catch(ExceptionType1   e)
{
Statements;
}
catch(ExceptionType2   e)
{
Statements;
}
catch(ExceptionTypen   e)
{
Statements ;
}
```

When we try block statements generates an exception the exception object compared with exception type in multiple catch statements. If a match is found the statements in that catch block will be executed. If no match is found the program execution terminates immediately.

- **finally statement:**

Java supports finally statement that can be used to handle an exception that is not caught by any of the previous catch statements. Finally statements can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block as shown below.

Syntax 1:	**Syntax 2:**

try
{
…….
……
}
finally
{
……
……
}
	try
	{
	……..
	}
	catch(ExceptionType1 e)
	{
	…….
	}
	catch(ExceptionType2 e)
	{
	…….

	}
	finally
	{
	…….
	……..
	}

Example:

```
public class MyClass {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

Output:

```
Somethingwentwrong.
The 'try catch' is finished.
```

- **The throw keyword:-**

The throw statement allows us to create a custom error. The throw statement is used together with an exception type. There are many exception types available in java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

The `throw` keyword is used to explicitly throw a single exception.When an exception is thrown, the flow of program execution transfers from the `try` block to the `catch` block. We use the `throw` keyword within a method.

Its syntax is:

```
throw throwableObject;
```

Ex:

```
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...");
    }
      }
```

Test it Now

Output:

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

❖ **The throws keyword:**

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax:

```
type method_name(parameter_list) throws exception_list
{
  // definition of method
}
```

Example:

```
class Test
{
  static void check() throws ArithmeticException
  {
    System.out.println("Inside check function");
    throw new ArithmeticException("demo");
  }

  public static void main(String args[])
  {
    try
    {
      check();
    }
    catch(ArithmeticException e)
    {
      System.out.println("caught" + e);
    }
  }
}
```

Output:

Inside check function
caughtjava.lang.ArithmeticException: demo

Difference between throw and throws

Throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

❖ Finally clause

A finally keyword is used to create a block of code that follows a try block. A finally block of code is always executed whether an exception has occurred or not. Using a finally block, it lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

Example finally Block

In this example, we are using finally block along with try block. This program throws an exception and due to exception, program terminates its execution but see code written inside the finally block executed. It is because of nature of finally block that guarantees to execute the code.

Ex

```
class Demo
{
  public static void main(String[] args)
  {
    int a[] = new int[2];
    try
    {
      System.out.println("Access invalid element"+ a[3]);
      /* the above statement will throw ArrayIndexOutOfBoundException */
    }
    catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception caught");
    }
    finally
    {
      System.out.println("finally is always executed.");
    }
  }
```

```
}
```

Output:
Exception caught
finally is always executed.

4. Types of Exceptions:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled. An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exceptions can be categorized into two ways:

1. **Built-in Exceptions**
   - Checked Exception
   - Unchecked Exception
2. **User-Defined Exceptions**

❖ Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.

2. **ArrayIndexOutOfBoundsException**
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
4. **IOException**
   It is thrown when an input-output operation failed or interrupted.
5. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
6. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
7. **NumberFormatException**
   This exception is raised when a method could not convert a string into a numeric format.
8. **StringIndexOutOfBoundsException**
   It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

- **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

```
public class Unchecked_Demo {

   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8
```

Example:Arithmeticexception

```
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a/b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");
        }
    }
}
```

Output:

```
Can't divide a number by 0
```

❖ User-defined Exception

In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of **the** **try-catch** block and **throw** keyword.

Syntax:

```
class MyException extends Exception
```

To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

Ex:

```
MyException me = new MyException("Exception details");
throw me;
```

5. THREADS 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. Thread is nothing but a process executed by the computer. Executing of two or more Threads simultaneously by a single computer is known as multi Threading.

❖ **Creating Threads**:

There are two methods to create Threads in java
1. Extending Thread class
2. Implementing runnable interface

1.Extending Thread class:

By extending the class java.lang.Thread, it includes the following steps

- Define the class that extending the Thread class
- Over ride the run() method by placing the code of the process
- Create a Thread class object
- Call the start() method to invoke run () method

Commonly used Constructors of Thread class:

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

Some 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 int getPriority():** returns the priority of the thread.
5. **public int setPriority(int priority):** changes the priority of the thread.
6. **public void setName(String name):** changes the name of the thread.
7. **public Thread currentThread():** returns the reference of currently executing thread.
8. **public void suspend():** is used to suspend the thread(depricated).
9. **public void resume():** is used to resume the suspended thread(depricated).

10. **public void stop():** is used to stop the thread(depricated).

2. Implementing runnable interface:

- Define a class that implements runnable interface
- Implement the run () method
- Create a Thread class object and pass newly created implementation class object as a parameter
- Call the start() method to invoke run() method.

❖ **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.

Ex:1) Java 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...

Ex2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

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.

❖ 3.Stopping and blocking a Thread:

➢ Stopping a Thread:

Whenever we want to stop a Thread running further we may do by calling its stop() method
Ex: obj.stop();
Here obj is a Thread object
This statement causes the Thread to move to the dead state . a Thread will also move to the dead state automatically when it reaches the end of its method. The stop() method may be used when the thread to be stopped before its completion.

➢ Blocking a Thread:

A Thread can also be temporarily suspended or blocked from enterning from entering into the runnable and running state by using following Thread methods
1.sleep()
2.suspend()
3.wait()

These methods cause state when the specified time is elapsed in the case of sleep() the resume() method is invoked in the case of suspend() and the notify() method is in called in the case of wait() method.

Example: Multithreading in JAVA

```
class SleepThread extends Thread  {
    //run method for thread
    public void run()    {
        for(int i=1;i<5;i++)   {
            try {
                //call sleep method of thread
                Thread.sleep(1000);
            }catch(InterruptedException e){System.out.println(e);}
            //print current thread instance with loop variable
       System.out.println(Thread.currentThread().getName() + "   : " + i);
        }
    }
}
class Main{
    public static void main(String args[])
    {
        //create two thread instances
        SleepThread thread_1 = new SleepThread();
        SleepThread thread_2 = new SleepThread();
        //start threads one by one
        thread_1.start();
        thread_2.start();
    }
}
```

Output:

```
Thread-0    : 1
Thread-1    : 1
Thread-0    : 2
Thread-1    : 2
Thread-0    : 3
Thread-1    : 3
Thread-0    : 4
Thread-1    : 4
```

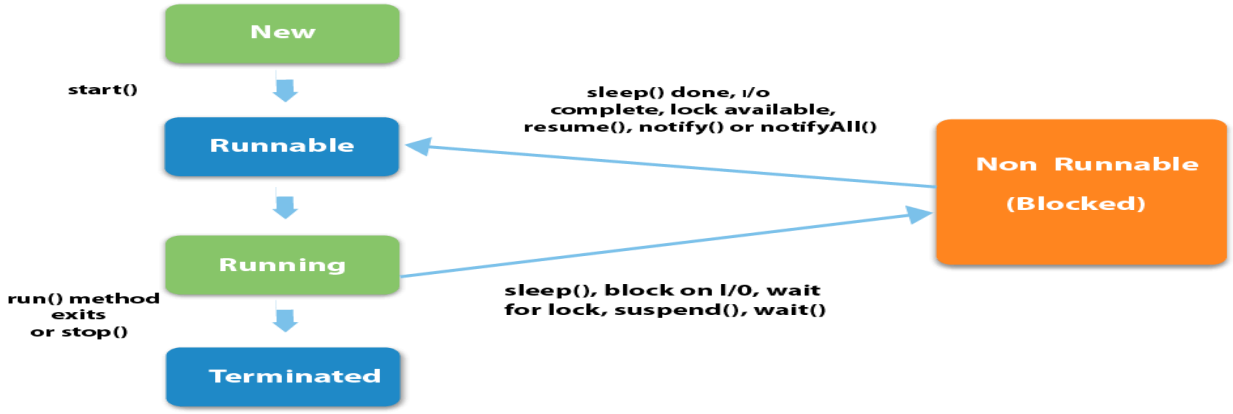
4. Thread life cycle:

A thread can be in one of the five states. .But, 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 Born state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state(Terminated)



1. NewBorn state:

When we create a Thread object then Thread is in new born state . at this state we can do only one of the following things

- Scheduled it for running using start() method
- Kill it using stop() method

2. Runnable state: In this state the Thread Is ready for execution and is waiting for the availability of the processor. That is the Thread has joined the queue of Threads.

3. Running state: Running state means that the processor has given its time to the Thread for execution. The running Thread may stop its execution by calling any one of the following methods

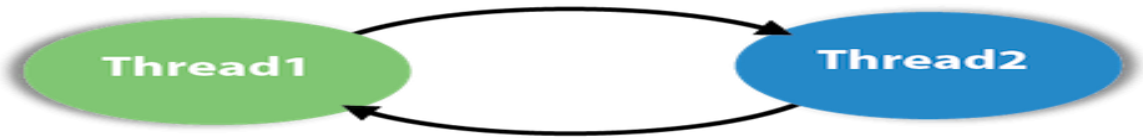
- **suspend ():** The Threads has been suspended using suspend"() method and again starts with resume() method.
- **sleep():** The Threads has been sleep for a specified time using sleep() method and execution continues after the specified time
- **wait():**the Threads has been wait for occurring of some events using wait() method and again starts with the notify() method

4. Blocked state: a Thread is said to be blocked state whenever the Thread is suspended or sleeping or waiting.

5. Dead state: A Thread is said to be dead state when its run() method completed the execution or by calling stop() method.

5. 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.



6. 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.The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
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()

7. Java Thread Priorities

Priority of a thread describes how early it gets execution and selected by the thread scheduler. In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between 1 to 10 and there are three constant variables which are static and used to fetch priority of a Thread. They are as following:

1. public static int MIN_PRIORITY

It holds the minimum priority that can be given to a thread. The value for this is 1.

2. public static int NORM_PRIORITY

It is the default priority that is given to a thread if it is not defined. The value for this is 0.

3. public static int MAX_PRIORITY

It is the maximum priority that can be given to a thread. The value for this is 10.

Get and Set methods in Thread priority

1. public final intgetPriority()

In Java, getPriority() method is in java.lang.Thread package. it is used to get the priority of a thread.

2. public final void setPriority(intnewPriority)

In Java setPriority(intnewPriority) method is in java.lang.Thread package. It is used to set the priority of a thread. The setPriority() method throws IllegalArgumentException if the value of new priority is above minimum and maximum limit.

Example:

```
class MyThread extends Thread
{
   public void run()
   {
            System.out.println("Thread Running...");
    }

    public static void main(String[]args)
    {
        MyThread p1 = new MyThread();
        MyThread p2 = new MyThread();
        MyThread p3 = new MyThread();
        p1.start();
        System.out.println("P1 thread priority : " +
                p1.getPriority());
        System.out.println("P2 thread priority : " +
                p2.getPriority());
         System.out.println("P3 thread priority : " +
                p3.getPriority());

   }
}
```

Output:

```
P1 thread priority : 5
Thread Running...
 P2 thread priority : 5
P3 thread priority : 5
```

6. PACKAGES IN JAVA:

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create our own packages)

1. Built-in Packages:

The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment. These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) **java.io:** Contains classed for supporting input / output operations.

3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) **java.applet:** Contains classes for creating Applets.

5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) **java.net:** Contain classes for supporting networking operations.

The library is divided into **packages** and **classes**. Meaning we can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, we need to use the **import** keyword:

Syntax:-

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class

If we find a class we want to use, for example, the Scanner class, **which is used to get user input**, write the following code**:**

Example

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

In the example above, java.util is a package, while **Scanner** is a class of the java.util package.

To use the **Scanner** class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the **nextLine()** method, which is used to read a complete line:

Example:

Using the Scanner class to get user input:

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

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

```
  String userName;

  // Enter username and press Enter
  System.out.println("Enter username");
  userName = myObj.nextLine();

  System.out.println("Username is: " + userName);
 }
}
```

- **Import a Package:**

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the java.util package:

Example

```
import java.util.*;
```

2.User-defined Packages:

To create our own package, we need to understand that Java uses a file system directory to store them. Just like folders on our computer:

```
Example
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the **package** keyword:

MyPackageClass.java

```
package mypack;

class MyPackageClass {
 public static void main(String[] args) {
  System.out.println("This is my package!");
 }
}
```

Save the file as **MyPackageClass.java**, and compile it:

C:\Users\Your Name>javac MyPackageClass.java

Then compile the package:

C:\Users\Your Name>javac -d . MyPackageClass.java

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. we can use any directory name, like c:/user (windows), or, if we want to keep the package within the same directory, we can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users\Your Name>java mypack.MyPackageClass

The output will be:

```
This is my package!
```

❖ **Advantages 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.

4) Packages provide reusability of code.

5) To bundle classes and interfaces.

6) We can create our own Package or extend already available Package.

3. 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.

Example of package that import the packagename.*

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B{
6.   public static void main(String args[]){
7.    A obj = new A();
```

```
    obj.msg();
   }
}
```

```
Output:Hello
```

2) Using packagename.classname

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

Example of package by import package.classname

```
//save by A.java

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

```
package mypack;  //save by B.java

import pack.A;

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

```
Output:Hello
```

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{
```

```
4.    public void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  class B{
4.    public static void main(String args[]){
5.     pack.A obj = new pack.A();//using fully qualified name
6.     obj.msg();
7.    }
8.  }
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

❖ Subpackage in java:

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Syntax: ***The standard of defining package is domain.company.package***

e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1.  package com.javatpoint.core;
2.  class Simple{
3.    public static void main(String args[]){
4.     System.out.println("Hello subpackage");
5.    }
6.  }
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

❖ ACCESSING A ACESS SPECIFIERS IN PACKAGES:

Access Specifier: Specifies the scope of the data members, class and methods.

- private members of the class are available with in the class only. The scope of private members of the class is “CLASS SCOPE”.
- public members of the class are available anywhere .The scope of public members of the class is "GLOBAL SCOPE".
- default members of the class are available with in the class, outside the class and in its sub class of same package. It is not available outside the package. So the scope of default members of the class is "PACKAGE SCOPE".
- protected members of the class are available with in the class, outside the class and in its sub class of same package and also available to subclasses in different package also.

Class Member Access	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package sub class	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

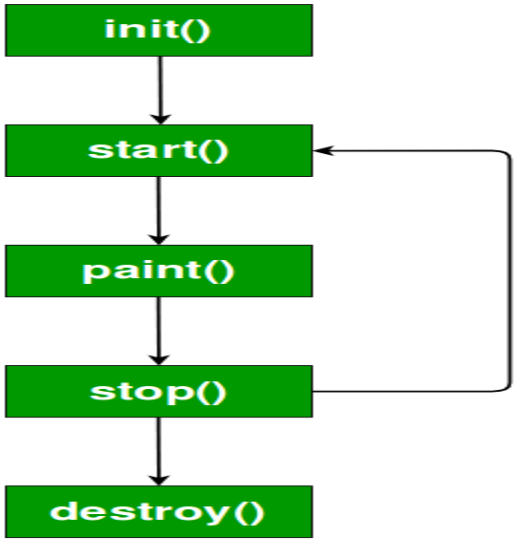
7. APPLETS:

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

❖ Life Cycle of an Applet:



The methods in the Applet class gives you the framework on which you build any serious applet

When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

Let's look more closely at these methods.

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
   public void paint (Graphics g) {
      g.drawString ("Hello World", 25, 50);
   }
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

❖ The Applet Class:

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.
These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

❖ Invoking an Applet: or Applet tag or <applet> tag

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet −

```
<html>
   <title>The Hello, World Applet</title>
   <hr>
   <applet code = "HelloWorldApplet.class" width = "320" height = "120">
      If your browser was Java-enabled, a "Hello, World"
      message would appear here.
   </applet>
   <hr>
</html>
```

Note − You can refer to HTML Applet Tag to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.

Example program:

```
import java.applet.*;// used
//to access showStatus()
import java.awt.*;//Graphic
//class is available in this package
import java.util.Date;// used
//to access Date object
public class GFG extends Applet
{
public void paint(Graphics g)
{
Date dt = new Date();
super.showStatus("Today is" + dt);
//in this line, super keyword is
// avoidable too.
}
}
```

Save as: GFG.java

Save as: GFG.HTML

```
<html>
  <applet code="GFG123" height="100" width="100">
  </applet>
</html>
```

Run as: Javac GFG.java

appletviewer GFG.html

Output: