**PROGRAMMING IN JAVA**

**Unit IV**

**Exception Handling**

**Introduction**

* An exception 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.

**ADVANTAGE:**

* The advantage of exception handling is to maintain the normal flow of the application.
* An exception normally disrupts the normal flow of the application that is why we use exception handling.

**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:

* Checked Exception
* Unchecked Exception
* Error

**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.

**Exception Hierarchy**

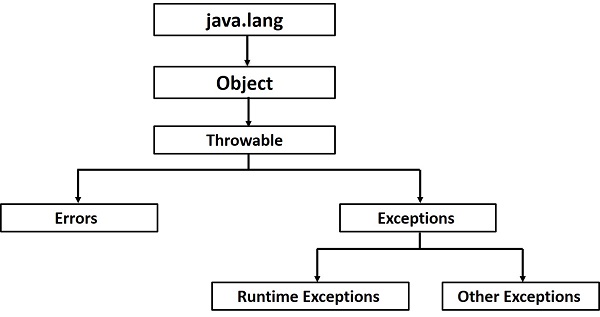
The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses:

1) Exception

2) Error.

* All exception classes are subtypes of the java.lang.Exception class.
* The exception class is a subclass of the Throwable class.
* Other than the exception class there is another subclass called Error which is derived from the Throwable class.
* Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs.
* Errors are generated to indicate errors generated by the runtime environment.
* Example: JVM is out of memory. Normally, programs cannot recover from errors.
* The Exception class has two main subclasses: IOException class and RuntimeException Class.

A hierarchy of Java Exception classes are given below



**CONSTRUCTORS AND METHODS IN THROWABLE CLASS**

**Constructors**

There are four constructors in the Throwable class:

* Throwable ()
* Throwable (String message)
* Throwable (String message, Throwable cause)
* Throwable (Throwable cause)

**Note:-** The first two allow constructing an exception object with or without a String message

encapsulated in the object.

**Methods**

There are several methods in the Throwable class. Three useful methods of the Throwable class that provide information about an exception are the following:

* getMessage ( )-Returns the message that was encapsulated when the object was initiated. It returns null, if there is no message.
* toString ()-returns a brief description about the exception of throwable object.
* printStacktrace ( )-prints the stack trace.

StackTrace is a list of the methods executed in sequence that lead to the exception and it is typically used to show the run-time errors on the screen.

The methods of the Throwable class that deal with StackTrace are the following:

* fililnStackTrace ()
* getStackTrace ()
* printStackTrace ()
* setStackTrace (Stack TraceElement [] stack Trace)

When an exceptional condition occurs within a method, the method may instantiate an exception object and hand it to the run-time system to deal with it.

This is called throwing an exception.

**Exceptions Methods**

* public String getMessage()
* public Throwable getCause()
* public String toString()
* public void printStackTrace()
* public StackTraceElement [] getStackTrace()
* public Throwable fillInStackTrace()

**UNCHECKED AND CHECKED EXCEPTIONS**

In Java programming, for every occurrence of an exception, there generates an exception object, which holds all the details of the exception.

Then the program searches for its respective exception handler. If found, the exception is handled or resolved, or else the program execution stops.

Java generates two types of exceptions. These are:

1. Checked exception
2. Unchecked exception

**Checked exceptions**

* A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.
* These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of these exceptions.
* If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.
* It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.
* 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.

**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.
* In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.
* 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.

**DIFFERENCE BETWEEN CHECKED EXCEPTION AND UNCHECKED EXCEPTION**

|  |  |
| --- | --- |
| Checked Exception | Unchecked Exception |
| Checked exceptions occur at compile time. | Unchecked exceptions occur at runtime. |
| The compiler checks a checked exception. | The compiler does not check these types of exceptions. |
| These types of exceptions can be handled at the time of compilation. | These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program. |
| They are the sub-class of the exception class. | They are runtime exceptions and hence are not a part of the Exception class. |
| Here, the JVM needs the exception to catch and handle. | Here, the JVM does not require the exception to catch and handle. |
| Examples of Checked exceptions:  File Not Found Exception  No Such Field Exception | Examples of Unchecked Exceptions:  No Such Element Exception  Undeclared Throwable Exception |

**INHERITANCE IN JAVA PROGRAMMING**

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called inheritance.

The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

**ChildClass:**  
The class that extends the features of another class is known as child class, sub class or derived class.

**ParentClass:**  
The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

**Syntax:**

class XYZ extends ABC

{

}

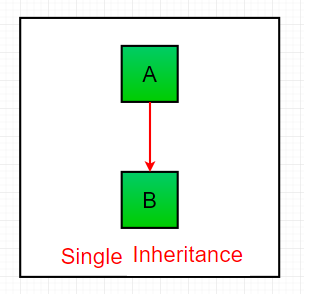
**TYPES OF INHERITANCE IN JAVA**

Below are the different types of inheritance which is supported by Java.

1. Single Inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple Inheritance
5. Hybrid inheritance

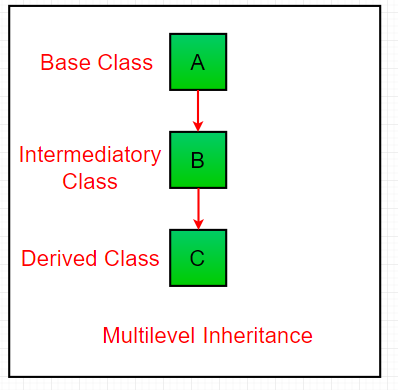
**Single Inheritance**

* It refers to a child and parent class relationship where a class extends the another class.
* In single inheritance, subclasses inherit the features of one superclass
* the class A serves as a base class for the derived class B.



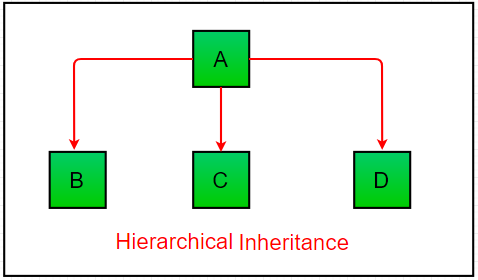
**Multilevel inheritance**

* It refers to a child and parent class relationship where a class extends the child class.
* In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
* The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent’s members.



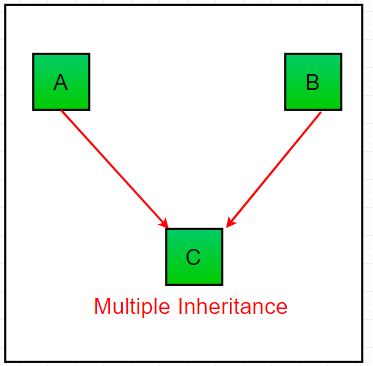
**Hierarchical inheritance**

* It refers to a child and parent class relationship where more than one classes extends the same class.
* In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.
* the class A serves as a base class for the derived class B,C and D.



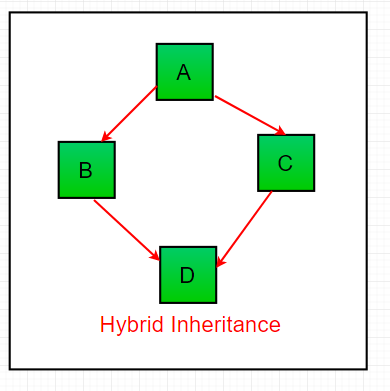
**Multiple Inheritance**

* It refers to the concept of one class extending more than one classes, which means a child class has two parent classes.
* In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritance with classes.
* In java, we can achieve multiple inheritance only through Interfaces.
* Class C is derived from interface A and B.



**Hybrid inheritance**

* It Combination of more than one types of inheritance in a single program.
* java doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes.
* In java, we can achieve hybrid inheritance only through Interfaces.



**USER DEFINED EXCEPTION IN JAVA**

* In java we have already defined, exception classes such as ArithmeticException, NullPointerException etc.
* These exceptions are already set to trigger on pre-defined conditions such as when you divide a number by zero it triggers ArithmeticException
* n java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions.

**Explanation:**  
You can see that while throwing custom exception I gave a string in parenthesis ( throw new MyException("This is My error Message");). That’s why we have a parameterized constructor (with a String parameter) in my custom exception class.

**Notes:**  
1. User-defined exception must extend Exception class.  
2. The exception is thrown using throw keyword.

**RE-DIRECTING AND RE-THROWING EXCEPTIONS**

**Re-throwing Exceptions**

* An exception that is caught in the try block can be thrown once again and can be handled.
* The try block just above the rethrow statement will catch the rethrown object.
* If there is no try block just above the rethrow statement then the method containing the rethrow statement handles it.
* To propagate an exception, the catch block can choose to rethrow the exception by using the throw statement.
* Note that there is no special syntax for rethrowing.
* Program illustrates how an exception can be rethrown .
* Throwable is a class. Though the Throwable class is derived from the java.lang.Object class in the Java class library, Throwable is the super-class of all classes that handle exceptions.
* The keyword throw is a statement that throws an exception. Note that an exception can be thrown either by the throw statement or when an error occurs during the execution of any other statement.
* The keyword throws is a clause specified in the method definition which indicates that the method throws the exceptions mentioned after the keyword throws, which are handled in the called methods.

**Re-directing Exceptions**

* Recall that the code capable of throwing an exception is kept in the try block and the exceptions are caught in the catch block.
* When there is no appropriate catch block to handle the exception that was thrown by an object, the compiler does not compile the program.
* To overcome this, Java allows the programmer to redirect exceptions that have been raised up the call stack, by using the keyword throws.
* Thus, an exception thrown by a method can be handled either in the method itself or passed to a different method in the call stack.
* To pass exceptions up to the call stack, the method must be declared with a throws clause.
* All the exceptions thrown by a method can be declared with a single throws clause; the clause consists of the keyword throws followed by a comma-separated list of all the exceptions

**MULTITHREADING IN JAVA**

**Introduction**

* Java is a multi-threaded programming language which means we can develop multi-threaded program using Java.
* A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
* Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.
* Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.
* Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

**Threads can be created by using two mechanisms :**

1. Extending the Thread class
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**

* It create a class that extends the java.lang.Thread class.
* This class overrides the run() method available in the Thread class.
* A thread begins its life inside run() method.

**Thread creation by implementing the Runnable Interface**

* It create a new class which implements java.lang.Runnable interface and override run() method.
* Then we instantiate a Thread object and call start() method on this object.

**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.

**Thread Life-cycle**

A thread goes through various stages in its life cycle.

For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

**Following are the stages of the life cycle −**

New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

**Thread Priorities**

* Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
* Java thread priorities are in the range between MIN\_PRIORITY and MAX\_PRIORITY
* By default, every thread is given priority NORM\_PRIORITY (a constant of 5).
* Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.
* Thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

**Thread Synchronization**

* Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
* When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
* To implement the synchronous behavior java has synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block.
* When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a synchronized block in following manner.

**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.

**NOTES**

* 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.

**Methods for Java Daemon thread**

* public void setDaemon(boolean status)
* public boolean isDaemon()

**UNIT-V**

**Files and I/O Streams**

**Java I/O**

* The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.
* All these streams represent an input source and an output destination.
* The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

**Stream**

* A stream can be defined as a sequence of data.

There are two kinds of Streams −

* InPutStream − The InputStream is used to read data from a source.
* OutPutStream − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O.

**Reading and Writing Files**

* The two important streams are FileInputStream and FileOutputStream, which would be discussed in this tutorial.a stream can be defined as a sequence of data.
* The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination

**FileInputStream**

* This stream is used for reading data from the files.
* Objects can be created using the keyword new and there are several types of constructors available.
* These streams are used to read data that must be taken as an input from a source array or file or any peripheral device.
* eg., FileInputStream, BufferedInputStream, ByteArrayInputStream etc.

**Syntax:**

InputStream f = new FileInputStream("C:/java/hello");

There are other important input streams available, for more detail you can refer to the following links

1. ByteArrayInputStream
2. DataInputStream

**FileOutputStream**

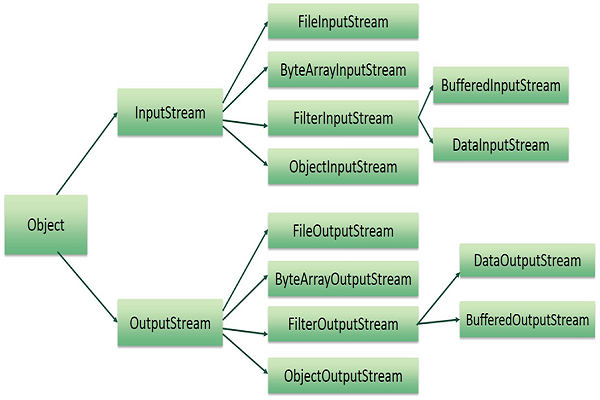
* FileOutputStream is used to create a file and write data into it.
* The stream would create a file, if it doesn't already exist, before opening it for output.
* Two constructors which can be used to create a FileOutputStream object.
* These streams are used to write data as outputs into an array or file or any output peripheral device.
* eg., FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc.

**Syntax**

OutputStream f = new FileOutputStream("C:/java/hello")

There are other important output streams available, for more detail you can refer to the following links

1. ByteArrayOutputStream
2. DataOutputStream



**RandomAccess File**

**Introduction**

* The Java.io.RandomAccessFile class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.
* This class is used for reading and writing to random access file.
* A random access file behaves like a large array of bytes.
* There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations.
* If end-of-file is reached before the desired number of byte has been read than EOFException is thrown.
* It is a type of IOException.

**Declaration**

public class RandomAccessFile

extends Object

implements DataOutput, DataInput, Closeable

**Serialization**

* Serialization is a mechanism of converting the state of an object into a byte stream.
* To make a Java object serializable we implement the java.io.Serializable interface.  
  The ObjectOutputStream class contains writeObject() method for serializing an Object.
* Only the objects of those classes can be serialized which are implementing java.io.serialization interface.

**Advantages of Serialization**

1. To save/persist state of an object.  
2. To travel an object across a network.

**APPLETS**

**Introduction**

* 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.
* 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.

**Applet-Life Cycle**

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

* 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.

**RUN AN APPLET**

There are two ways to run an applet

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

**JAVA APPLICATIONS VERSUS JAVA APPLETS**

|  |  |
| --- | --- |
| Application | Applet |
| Applications are stand-alone programs that can be run independently without having to use a web browser. | Applets are small Java programs that are designed to be included in  a HTML web document.  They require a Java-enabled browser for  execution. |
| Java applications have full access to local file system and network. | Applets have no disk and network access. |
| It requires a main method() for its execution. | It does not require a main method() for its execution. |
| Applications can run programs from the local system. | Applets cannot run programs from the local machine. |
| An application program is used to perform some task directly for the user. | An applet program is used to perform small tasks or part of it. |
| It can access all kinds of resources available on the system. | It can only access the browser specific services. |

**HTML APPLET Tag**

* The <applet> tag in HTML was used to embed Java applets into any HTML document.
* The <applet> tag was deprecated in HTML 4.01, and it’s support has been completely discontinued starting from HTML 5.

**Syntax**

<applet attribute1 attribute2....>

<param parameter1>

<param parameter2>

</applet>

**Attributes**

* The <applet> tag takes a number of attributes, with one of the most important being the code attribute.
* This code attribute is used to link a Java applet to the concerned HTML document. It specifies the file name of the Java applet.

**Parameters**

* Parameters are quite similar to command-line arguments in the sense that they provide a way to pass information to the applet after it has started.
* All the information available to the applet before it starts is said to be hard-coded i.e. embedded within it.
* Parameters make it possible to generate and use data during run-time of the applet.

**Syntax:**

<param name=parameter\_name value=parameter\_value>