UNIT -4. Multithreading:

Java Thread

- Thread can be called ***lightweight process***. Thread requires less resources to create and exists in the process, thread shares the process resources. A thread is a single sequential flow of control within a program.

A thread is a:

- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state

- **The need of a thread or why we use Threads**

1 To perform asynchronous or background processing

2 Increases the responsiveness of GUI applications

3 Take advantage of multiprocessor systems

4 Simplify program logic when there are multiple independent entities

- When a thread is invoked, there will be two paths of execution. One path will execute the thread and the other path will follow the statement after the thread invocation. There will be a separate stack and memory space for each thread.

- Thread implementation in java can be achieved in two ways:

1 Extending the java.lang.Thread class

2 Implementing the java.lang.Runnable Interface

- **1) By extending thread class**
- The class should extend Java Thread class.
- The class should override the run() method.
- The functionality that is expected by the Thread to be executed is written in the run() method.
- void start(): Creates a new thread and makes it runnable.
  void run(): The new thread begins its life inside this method.
- **Example**

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

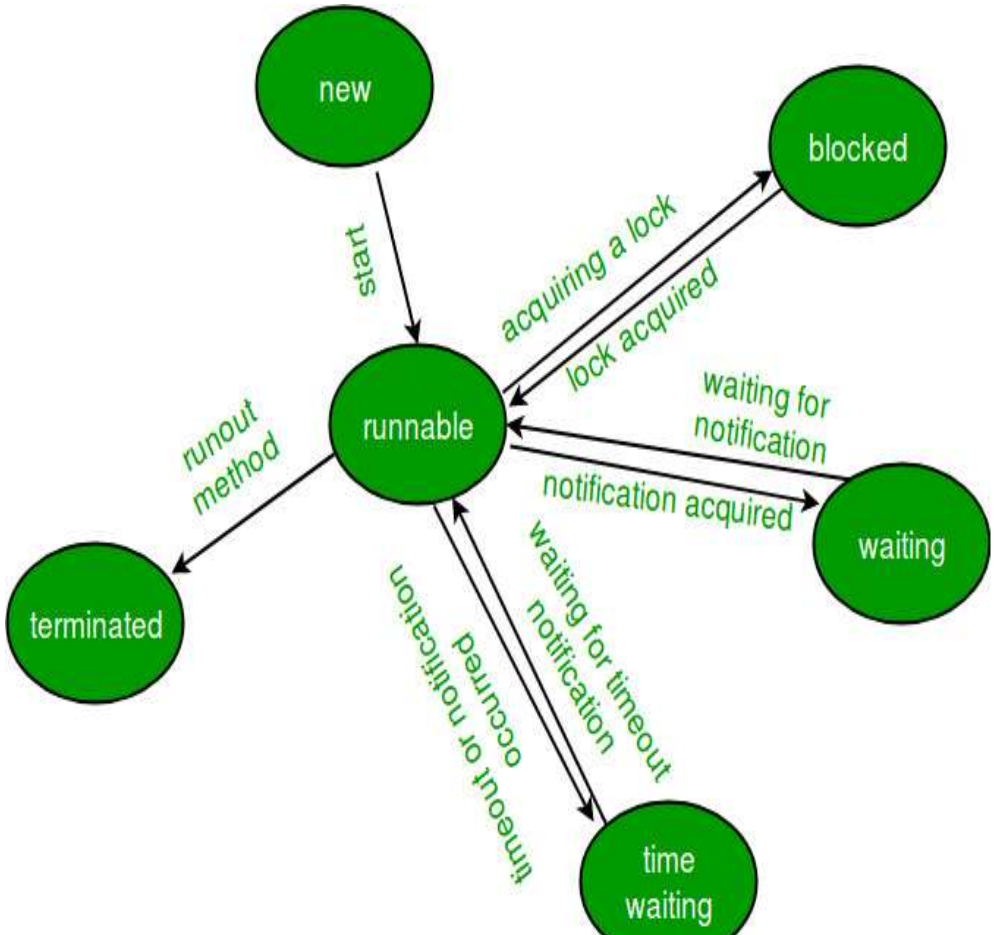
- **2) By Implementing Runnable interface**
- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method
- **Example:**

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

- **Extends Thread class vs Implements Runnable Interface.**
- Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
- If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

Lifecycle and States of a Thread

- A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:
- New
- Runnable
- Blocked
- Waiting
- Timed Waiting
- Terminated



1) **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute.

2) **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

3) **Blocked/Waiting state:**When a thread is temporarily inactive, then it’s in one of the following states:
   - Blocked
   - Waiting

- For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.
- A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.
- If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

4) **Timed Waiting:** A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

5) **Terminated State:** A thread terminates because of either of the following reasons:
   - Because it exists normally. This happens when the code of thread has entirely executed by the program.
   - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

- A thread that lies in a terminated state does no longer consumes any cycles of CPU.

Interrupting a Thread:

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.
- If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behavior and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.
- The 3 methods provided by the Thread class for interrupting a thread
- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

- In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked.

```
class TestInterruptingThread1 extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
throw new RuntimeException("Thread interrupted..."+e);
}

}

public static void main(String args[]){
TestInterruptingThread1 t1=new TestInterruptingThread1();
t1.start();
try{
t1.interrupt();
}catch(Exception e){System.out.println("Exception handled "+e);}

}
}
```

```
Output:Exception in thread-0
 java.lang.RuntimeException: Thread interrupted...
java.lang.InterruptedException: sleep interrupted
at A.run(A.java:7)
```

- Example of interrupting thread that behaves normally
- If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
class TestInterruptingThread3 extends Thread{

public void run(){
for(int i=1;i<=5;i++)
System.out.println(i);
}

public static void main(String args[]){
TestInterruptingThread3 t1=new TestInterruptingThread3();
t1.start();

t1.interrupt();

}
}
```

Output:

```
1
2
3
4
5
```

Thread Properties

- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). 1 is the min priority and 10 is the max priority. But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- Thread Priorities determines how a thread should be treated with respect to others.
- Several threads executes concurrently. Every thread has some priority.
- Which thread will get a chance first to execute it is decided by thread scheduler based on thread priority.

3 constants defined in Thread class:

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

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

- **Syntax:**
- Thread.MIN_PRIORITY Thread.NORM_PRIORITY Thread.MAX_PRIORITY
- Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

- Output:running thread name is:Thread-0
- running thread priority is:10
- running thread name is:Thread-1
- running thread priority is:1

Synchronization

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
- So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.
- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword.
- A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time.

- Types of Synchronization
- There are two types of synchronization

1) Process Synchronization
2) Thread Synchronization

Thread Synchronization

- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
  - Synchronized method.
  - Synchronized block.
  - static synchronization.
- Cooperation (Inter-thread communication in java)

Mutual Exclusive

- Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:
- by synchronized method
- by synchronized block
- by static synchronization

- In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table{
void printTable(int n){//method not synchronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

- **class** TestSynchronization1{
- **public static void** main(String args[]){
- Table obj = **new** Table();//only one object
- MyThread1 t1=**new** MyThread1(obj);
- MyThread2 t2=**new** MyThread2(obj);
- t1.start();
- t2.start();
- }
- }
- Output: 5 100 10 200 15 300 20 400 25 500

- Java synchronized method
- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

- **public class** TestSynchronization2{
- **public static void** main(String args[]){
- Table obj = **new** Table();//only one object
- MyThread1 t1=**new** MyThread1(obj);
- MyThread2 t2=**new** MyThread2(obj);
- t1.start();
- t2.start();
- }
- }
- Output: 5 10 15 20 25 100 200 300 400 500

Thread-Safe Collections

- The Collections class of java.util package methods that exclusively work on collections these methods provide various additional operations which involves polymorphic algorithms.
- This class provides different variants of the synchronized Collection() method as shown below –

Sr. No.	Method & Descriptions
1	**static <T> Collection<T> synchronized Collection(Collection<T> c)** This method accepts any collection object and, returns a synchronized (thread-safe) collection backed by the specified collection.
2	**static <T> List<T> synchronized List(List<T> list)** This method accepts an object of the List interface returns a synchronized (thread-safe) list backed by the specified list.
3	**static <K,V> Map<K,V> synchronized Map(Map<K,V> m)** This method accepts an object of the Map interface and, returns a synchronized (thread-safe) map backed by the specified map.
4	**static <T> Set<T> synchronized Set(Set<T> s)** This method accepts an object of Set interface and, returns a synchronized (thread-safe) set backed by the specified set.
5	**static <K,V> SortedMap<K,V> synchronized SortedMap(SortedMap<K,V> m)** This method accepts an object of the Map interface and, returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
6	**static <T> SortedSet<T> synchronized SortedSet(SortedSet<T> s)** This method accepts an object of the synchronized SortedSet interface and, returns a synchronized (thread-safe) sorted set backed by the specified sorted set.

- **Example**

```
package thrdsafe;
import java.util.* ;
public class Thrdsafe
{
   public static void main(String[] args)
   {
     Vector<String> vector1 = new Vector<String>();
      vector1.add ("C");
      vector1.add ("C++");
      vector1.add ("Java");
      vector1.add (".net");
      System.out.println (vector1);

      Collection<String>  vector2 = Collections.synchronizedCollection(vector1);
      System.out.println (vector2);
   }
   }
```

Thread Class Methods

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.

ThreadGroup

ThreadGroup in Java

- Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.
- Note: Now suspend(), resume() and stop() methods are deprecated.
- Java thread group is implemented by *java.lang.ThreadGroup* class.
- A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.
- Constructors of ThreadGroup class-There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with a given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods is given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.
11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.
19)	String	toString()	This method returns a string representation of the Thread group.

UNIT -4. Exceptions, Assertions:

Dealing with Errors

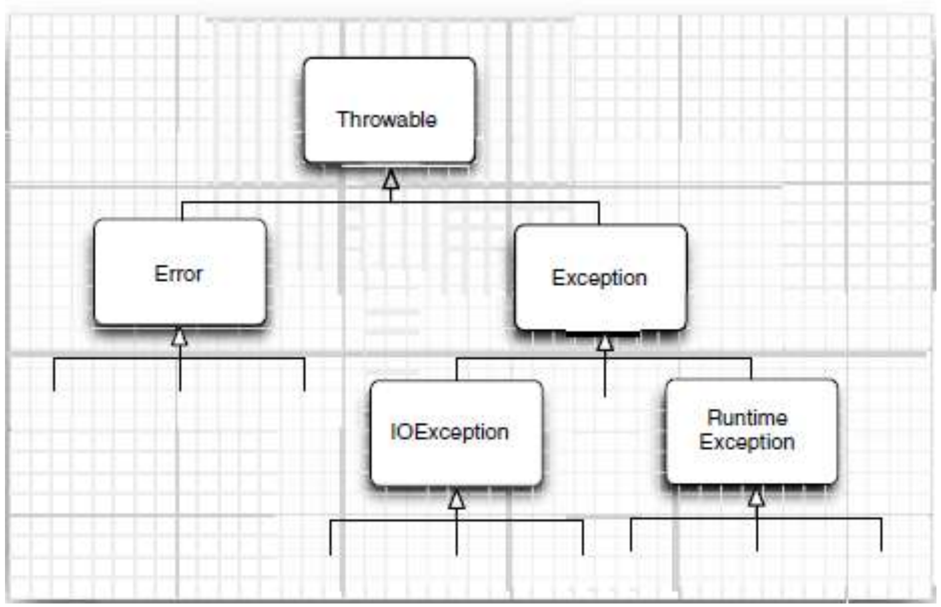
- Suppose an error occurs while a Java program is running. The error might be caused by afile containing wrong information, a flaky network connection, or (we hate to mention it)use of an invalid array index or an attempt to use an object reference that hasn’t yet been assigned to an object. Users expect that programs will act sensibly when errors happen. If an operation cannot be completed because of an error, the program ought to either.

  •Return to a safe state and enable the user to execute other commands; or
  • Allow the user to save all work and terminate the program gracefully.

- The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation. To handle exceptional situations in your program, you must take into account the errors and problems that may occur. What sorts of problems do you need to consider?
- *User input errors.* In addition to the inevitable typos, some users like to blaze their own trail instead of following directions. Suppose, for example, that a user asks to connect to a URL that is syntactically wrong. Your code should check the syntax, but suppose it does not. Then the network layer will complain.
- *Device errors.* Hardware does not always do what you want it to. The printer may be turned off. A web page may be temporarily unavailable. Devices will often fail in the middle of a task. For example, a printer may run out of paper during printing.
- *Physical limitations.* Disks can fill up; you can run out of available memory.
- *Code errors.* A method may not perform correctly. For example, it could deliver wrong answers or use other methods incorrectly. Computing an invalid array index, trying to find nonexistent entry in a hash table, and trying to pop an empty stack are all examples of a code error.
- The traditional reaction to an error in a method is to return a special error code that the calling method analyzes. For example, methods that read information back from files often return a –1 end-of-file value marker rather than a standard character. This can bean efficient method for dealing with many exceptional conditions

The Classification of Exceptions

- In the Java programming language, an exception object is always an instance of a class derived from Throwable. As you will soon see, you can create your own exception classes if the ones built into Java do not suit your needs.
- Figure below is a simplified diagram of the exception hierarchy in Java.



- When doing Java programming, you focus on the Exception hierarchy. The Exception hierarchy also splits into two branches: exceptions that derive from Runtime Exception and those that do not.
- The general rule is this: A RuntimeException happens because you made a programming error. Any other exception occurs because a bad thing, such as an I/O error, happened to your otherwise good program.
- Exceptions that inherit from RuntimeException include such problems as
  - A bad cast
  - An out-of-bounds array access
  - A null pointer access
- Exceptions that do not inherit from RuntimeException include
  - Trying to read past the end of a file
  - Trying to open a malformed URL
  - Trying to find a Class object for a string that does not denote an existing class.

- The rule “If it is a RuntimeException, it was your fault” works pretty well. You could have avoided that Array IndexOut OfBounds Exception by testing the array index against the array bounds. The NullPointer Exception would not have happened had you checked whether the variable was null before using it.
- The Java Language Specification calls any exception that derives from the class Error or the class Runtime Exception an ***unchecked*** **exception**. All other exceptions are called ***checked*** **exceptions**.
- 1) Checked Exception
- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.
- 2) Unchecked Exception
- The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime

- ***How to Throw an Exception***
- Let us suppose something terrible has happened in your code. You have a method, readData, that is reading in a file whose header promised Content-length: 1024 but you get an end of file after 733 characters. You decide this situation is so abnormal that you want to throw an exception. You need to decide what exception type to throw. Some kind of IOException would be a good choice. Perusing the Java API documentation, you find an EOFException with the description “Signals that an EOF has been reached unexpectedly during input.” Perfect.
- Here is how you throw it:

- throw new EOFException();
- or, if you prefer,
- EOFException e = new EOFException();
- throw e;
- Here is how it all fits together:
- String readData(Scanner in) throws EOFException
- {
- . . .
- while (. . .)
- {
- if (!in.hasNext()) // EOF encountered
- {
- if (n < len) throw new EOFException();
- }
- . . .
- }
- return s;
- }
- The EOFException has a second constructor that takes a string argument. You can put this to good use by describing the exceptional condition more carefully.
- String gripe = "Content-length: " + len + ", Received: " + n;
- throw new EOFException(gripe);
- As you can see, throwing an exception is easy if one of the existing exception classes works for you. In this case:
- 1. Find an appropriate exception class.
  2. Make an object of that class.
  3. Throw it.

- ***Creating Exception Classes***
- Your code may run into a problem that is not adequately described by any of the standard exception classes. In this case, it is easy enough to create your own exception class. Just derive it from Exception or from a child class of Exception such as IOException. It is customary to give both a default constructor and a constructor that contains a detailed message. (The toString method of the Throwable superclass prints that detailed message, which is handy for debugging.)

- class FileFormatException extends IOException
- {
- public FileFormatException() {}
- public FileFormatException(String gripe)
- {
- super(gripe);
- }
- }
- Now you are ready to throw your very own exception type.
- String readData(BufferedReader in) throws FileFormatException
- {
- . . .
- while (. . .)
- {
- if (ch == -1) // EOF encountered
- {
- if (n < len) throw new FileFormatException();
- }
- . . .
- }
- return s;
- }
- **java.lang.Throwable 1.0**
- Throwable()
  constructs a new Throwable object with no detailed message.
- Throwable(String message)
  constructs a new Throwable object with the specified detailed message. By convention, all derived exception classes support both a default constructor and a constructor with a detailed message.
- String getMessage()
  gets the detailed message of the Throwable object.

Java Exceptions

- When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
- When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).
- In Java, we use the exception handler components try, catch and finally blocks to handle exceptions.
- To catch and handle an exception, we place the try...catch...finally block around the code that might generate an exception. The finally block is optional.

- **The syntax for try...catch...finally is:**
- try
- { // code
- }
- catch (ExceptionType e)
- { // catch block }
- Finally
- { // finally block }
- **Java try...catch block**
- The code that might generate an exception is placed in the try block.
- Every try block should be immediately followed by the catch or finally block. When an exception occurs, it is caught by the catch block that immediately follows it.
- catch blocks cannot be used alone and must always be preceded by a try block.

• Example 1: try...catch blocks

```
class Main {
public static void main(String[] args)
 {
      try
      {
               int divideByZero = 5 / 0;            System.out.println("Rest of code in try block");
       }
      catch (ArithmeticException e)
      {
      System.out.println("ArithmeticException => " + e.getMessage());
       }
 }
 }
```

- **Output**
- ArithmeticException => / by zero
- In the example,
- We have divided a number by 0 inside the try block. This produces an ArithmeticException.
- When the exception occurs, the program skips the rest of the code in the try block.
- Here, we have created a catch block to handle ArithmeticException. Hence, the statements inside the catch block are executed.
- If none of the statements in the try block generates an exception, the catch block is skipped.

- **Multiple Catch blocks:-** For each try block, there can be zero or more catch blocks.
- The argument type of each catch block indicates the type of exception that can be handled by it. Multiple catch blocks allow us to handle each exception differently.
- **Example 2: Multiple catch blocks**

```
class ListOfNumbers
{
 public int[] arrayOfNumbers = new int[10];
public void writeList()
 {
try
 {
arrayOfNumbers[10] = 11;
}
 catch (NumberFormatException e1)
{
System.out.println("NumberFormatException => " + e1.getMessage());
 }
catch (IndexOutOfBoundsException e2)
 {
System.out.println("IndexOutOfBoundsException => " + e2.getMessage());
 } } }
 class Main
{
public static void main(String[] args)
 {
 ListOfNumbers list = new ListOfNumbers();
list.writeList();
 } }
```

- **Output:-** IndexOutOfBoundsException => Index 10 out of bounds for length 10

- In this example, we have declared an array of integers arrayOfNumbers of size 10.
- We know that an array index always starts from 0. So, when we try to assign a value to index 10, an IndexOutOfBoundsException occurs because the array bounds for arrayOfNumbers is 0 to 9.
- When an exception occurs in the try block,
- The exception is thrown to the first catch block. The first catch block does not handle an IndexOutOfBoundsException, so it is passed to the next catch block.
- The second catch block in the above example is the appropriate exception handler because it handles an IndexOutOfBoundsException. Hence, it is executed.
-

- **Java finally block**
- For each try block, there can be only one finally block.
- The finally block is optional. However, if defined, it is always executed (even if the exception doesn't occur).
- If an exception occurs, it is executed after the try...catch block. If no exception occurs, it is executed after the try block.
- The basic syntax of finally block is:
- try
- {
- //code
- }
- catch (ExceptionType1 e1)
- {
- // catch bloc
- }
- catch (ExceptionType1 e2)
- {
- // catch block
- }
- Finally
- {
- // finally block always executes
- }

- **Example 3: finally block**
- class Main
- {
- public static void main(String[] args)
- {
- try
- {
- int divideByZero = 5 / 0;
- }
- catch (ArithmeticException e)
- {
- System.out.println("ArithmeticException => " + e.getMessage());
- }
- finally {
- System.out.println("Finally block is always executed");
- } } }
- **Output**
- ArithmeticException => / by zero Finally block is always executed

➢ In this example, we have divided a number by 0. This throws an ArithmeticException which is caught by the catch block. The finally block always executes.

- Having a finally block is considered a good practice. It is because it includes important cleanup code such as:
- code that might have been accidentally skipped by return, continue or break statements
- closing a file or connection

➢ We have mentioned that finally always executes and that is usually the case. However, there are some cases when a finally block does not execute:

- Use of System.exit() method
- An exception occurs in the finally block
- The death of a thread

throw and throws in Java

- **1) throw:-**
- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions. **Syntax:**
- **throw *Instance*** Example: **throw new ArithmeticException("/ by zero");**
- But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class. Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.
- The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

```
•   // Java program that demonstrates the use of throw
•   class ThrowExcep
•   {
•           static void fun()
•           {
•                   try
•                   {
•                           throw new NullPointerException("demo");
•                   }
•                   catch(NullPointerException e)
•                   {
•                           System.out.println("Caught inside fun().");
•                           throw e; // rethrowing the exception
•                   }
•           }

•           public static void main(String args[])
•           {
•                   try
•                   {
•                           fun();
•                   }
•                   catch(NullPointerException e)
•                   {
•                           System.out.println("Caught in main.");
•                   }
•           }
•   }
•   Output:
•   Caught inside fun(). Caught in main.
```

- **throws**
- 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(parameters) throws exception_list**
- exception_list is a comma separated list of all the exceptions which a method might throw.
- In a program, if there is a chance of rising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying **unreported exception XXX must be caught or declared to be thrown**. To prevent this compile time error we can handle the exception in two ways:
- By using try catch
- By using **throws** keyword
- We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

```
•  // Java program to illustrate error in case
•  // of unhandled exception
•  class tst
•  {
•          public static void main(String[] args)
•          {
•                  Thread.sleep(10000);
•                  System.out.println("Hello Geeks");
•          }
•  }
```

- Output:
- error: unreported exception InterruptedException; must be caught or declared to be thrown
- **Explanation :** In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute main() method which will cause InterruptedException.

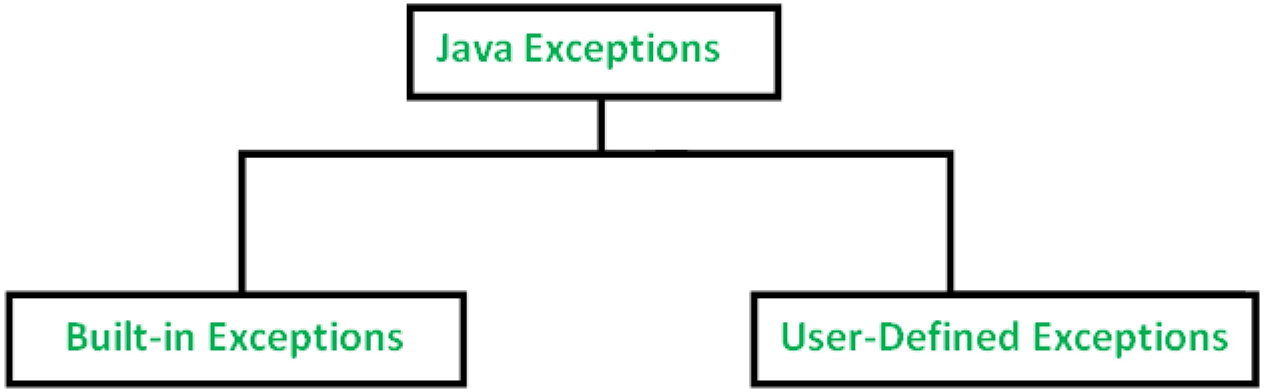
- // Java program to illustrate throws
- class tst
- {
- public static void main(String[] args)throws InterruptedException
- {
- Thread.sleep(10000);
- System.out.println("Hello Geeks");
- }
- }
- Output:
- Hello Geeks
- **Explanation :**
- In the above program, by using throws keyword we handled the InterruptedException and we will get the output as **Hello Geeks**

```
•   // Java program to demonstrate working of throws
•   class ThrowsExecp
•   {
•           static void fun() throws IllegalAccessException
•           {
•                   System.out.println("Inside fun(). ");
•                   throw new IllegalAccessException("demo");
•           }
•           public static void main(String args[])
•           {
•                   try
•                   {
•                           fun();
•                   }
•                   catch(IllegalAccessException e)
•                   {
•                           System.out.println("caught in main.");
•                   }
•           }
•   }
•   Output:
•   Inside fun().
•    caught in main.
```

- **Important points to remember about throws keyword:**
- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
- By the help of throws keyword we can provide information to the caller of the method about the exception.

Types of Exception in Java with Examples

- Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. 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. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
    This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**
    This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException**
    It is thrown by String class methods to indicate that an index is either negative than the size of the string

- **Examples of Built-in Exception:**

1) **Arithmetic exception**

```
// Java program to demonstrate 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
```

2) **NullPointer Exception**

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
        public static void main(String args[])
        {
                try {
                        String a = null; //null value
                        System.out.println(a.charAt(0));
                } catch(NullPointerException e) {
                        System.out.println("NullPointerException..");
                }
        }
}
```

Output: NullPointerException..

3) ArrayIndexOutOfBounds Exception

```
// Java program to demonstrate ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo
{       public static void main(String args[])
        {
                try{
                                int a[] = new int[5];
                                a[6] = 9; // accessing 7th element in an array of // size 5
                }
                catch(ArrayIndexOutOfBoundsException e){
                                System.out.println ("Array Index is Out Of Bounds");
                }
        }
}
```

- **Output:** Array Index is Out Of Bounds

4) StringIndexOutOfBound Exception

```
// Java program to demonstrate StringIndexOutOfBoundsException
class StringIndexOutOfBound_Demo
{
          public static void main(String args[])
          {
                try {
                                String a = "This is like chipping "; // length is 22
                                char c = a.charAt(24); // accessing 25th element
                                System.out.println(c);
                }
                catch(StringIndexOutOfBoundsException e) {
                                System.out.println("StringIndexOutOfBoundsException");
                }
          } }
```

- **Output:** StringIndexOutOfBoundsException

6) FileNotFound Exception

```
//Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo {

	public static void main(String args[]) {
		try {

			// Following file does not exist
			File file = new File("E://file.txt");

			FileReader fr = new FileReader(file);
		} catch (FileNotFoundException e) {
		System.out.println("File does not exist");
		}
	}
}
```

- **Output:**
- File does not exist

7) **NumberFormat Exception**

```
// Java program to demonstrate NumberFormatException
class NumberFormat_Demo
{
	public static void main(String args[])
	{
		try {
			// "akki" is not a number
			int num = Integer.parseInt ("akki") ;

			System.out.println(num);
		} catch(NumberFormatException e) {
			System.out.println("Number format exception");
		}
	}
```

- } **Output:** Number format exception

User-Defined Exceptions

- Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’.
  Following steps are followed for the creation of user-defined Exception.
- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:
- **class MyException extends Exception**
- We can write a default constructor in his own exception class.
- **MyException(){}**
- We can also create a parameterized constructor with a string as a parameter.
  We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.
- MyException(String str)
- {
  super(str);
  }
- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:
- MyException me = new MyException(“Exception details”);
- throw me;

- The following program illustrates how to create own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be ept in the account.
- If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

```
// Java program to demonstrate user defined exception

// This program throws an exception whenever balance
// amount is below Rs 1000
class MyException extends Exception
{
        //store account information
        private static int accno[] = {1001, 1002, 1003, 1004};

        private static String name[] =
                                {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

        private static double bal[] =
                {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

        // default constructor
        MyException() { }

        // parametrized constructor
        MyException(String str) { super(str); }

        // write main()

public static void main(String[] args)
   {
      try  {
         // display the heading for the table
         System.out.println("ACCNO" + "\t" + "CUSTOMER" +
                                         "\t" + "BALANCE");

         // display the actual account information
         for (int i = 0; i < 5 ; i++)
         {
            System.out.println(accno[i] + "\t" + name[i] +
                                         "\t" + bal[i]);

            // display own exception if balance < 1000
            if (bal[i] < 1000)
            {
               MyException me =
                 new MyException("Balance is less than 1000");
               throw me;
            }
         }
      } //end of try

      catch (MyException e) {
         e.printStackTrace();
      }
   }
}
```

- RunTime Error
- MyException: Balance is less than 1000 at MyException.main(fileProperty.java:36)

- **Output:**

-
  ```
ACCNO   CUSTOMER          BALANCE
1001          Nish        10000.0
1002          Shubh       12000.0
1003           Sush       5600.0
 1004         Abhi        999.0
```

Assertions

- An assertion is a statement in Java which ensures the correctness of any assumptions which have been done in the program. When an assertion is executed, it is assumed to be true. If the assertion is false, the JVM will throw an Assertion error. It finds it application primarily in the testing purposes. Assertion statements are used along with boolean expressions.
- Assertions in Java can be done with the help of the assert keyword. There are two ways in which an assert statement can be used.

First Way −

```
assert expression;
```

Second Way −

```
assert expression1 : expression2
```

By default, assertions are disabled in Java. In order to enable them we use the following command −

```
java -ea Example (or)
java –enable assertions Example
```

where Example is the name of the Java file. Let us see an example for generation of an assertion error by the JVM −

Example:-

```
Public class Example {
  public static void main(String args[])
 {
     int value = 15;
     assert value >= 20 : " Underweight";
     System.out.println("value is " + value);
   }  }
```

Output

```
 value is 15
```

After enabling assertions:

Output

```
Exception in thread "main" java.lang.AssertionError: Underweight
```