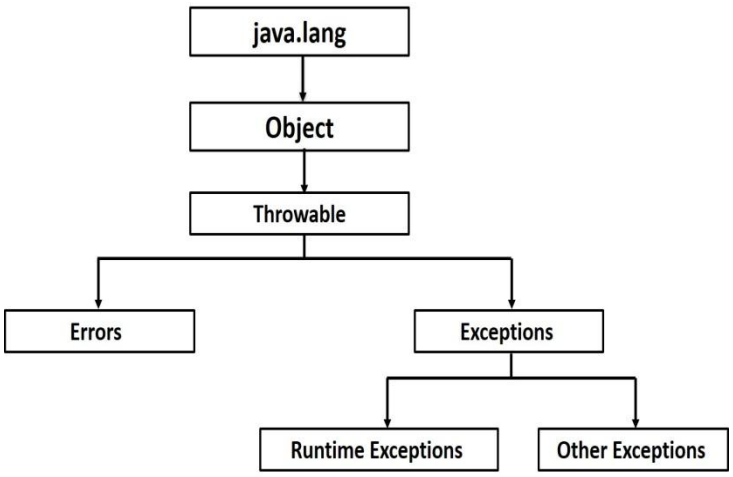
Exceptions Handling

- 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, therefore these exceptions are needs to be handled.
- A **Java Exception** is an **object** that describes the exception that occurs in a program. When an **exceptional event** occurs in java, an **exception** is said to be **thrown**.
- An exception can occur for many different reasons, some of them are as given below:
  - A user has entered **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.
- Exceptions are caused by users, programmers or when some physical resources get failed.
- The **Exception Handling** in java is one of the powerful mechanisms to handle the exception (runtime errors), so that **normal flow** of the application can be maintained.
- In Java there are **three categories** of Exceptions:
  1) **Checked exceptions:** A checked exception is an exception that occurs at **the compile time**, these are also called as **compile time exceptions**. Example, **IOException, SQLException** etc.
  2) **Runtime exceptions:** An Unchecked exception is an exception that occurs during the 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.
     - **Example :** ArithmeticException, NullPointerException, Array Index out of Bound exception.
  3) **Errors:** These are not exceptions at all, but problems that arise beyond the control of **the user or the programmer**. **Example:** OutOfMemoryError, VirtualMachineErrorException.
- 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 not normally trapped form the Java programs. These conditions normally happen in case of **severe** failures, which 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.

Exception Handling Mechanism

- Exception handling is done using five keywords:
  1) try
  2) catch
  3) finally
  4) throw
  5) throws

Using try and catch

1) try block :

- Java try block is used to enclose the code that **might throw an exception**. It must be used within the **method**.
- Java try block must be followed by **either catch or finally block**.

2) catch block:

- Java catch block is used to **handle** the Exception. It must be used **after the try block only**.
- The catch block that follows the **try is checked**, if the type of exception that occurred is listed in the **catch block** then the exception is handed over to the catch block that handles it.
- You can use **multiple catch block with a single try**.

Syntax:

```
try
{
        //Protected code
}
catch(ExceptionName1  e1)
{
        //Catch block 1
}
catch(ExceptionName2  e2)
{
        //Catch block 2
}
```

- In above syntax, there are two catch blocks. In try block, we write code that might generate exception. If the exception generated by protected code then exception thrown to the **first catch block**.
- If the data type of the exception thrown matches **ExceptionName1**, it gets caught there and **execute** the catch block.
- If not, the exception passes down to the **second catch block**.
- This continues **until** the exception either **is caught or falls through all catches**, in that case the current method stops execution.

Example:

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

```
        {
                try
                {
                        int arr[]={1,2,3};
                        arr[3]=3/0;
                }
                catch(ArithmeticException  ae)
                {
                        System.out.println("Divide by zero  :: " + ae);
                }
                catch(ArrayIndexOutOfBoundsException  e)
                {
                        System.out.println("Array index out of bound exception  :: "+e);
                }
        }
}
```

Output:

```
divide by zero  :: java.lang.ArithmeticException: / by zero
```

Nested Try-Catch Blocks

- In java, the **try block** within a **try block** is known as **nested try block**.
- Nested try block is used when a **part of a block** may cause **one error** while **entire block** may cause **another error**.
- In that case, if **inner try block** does not have a **catch** handler for a particular **exception** then the **outer try** is checked for **match**.
- This continues until one of the catch statements succeeds, or until the entire nested try statements are done in. If no one catch statements match, then the Java run-time system will handle the exception.

Syntax:

```
try
{
        Statement 1;
        try
        {
                //Protected code
        }
        catch(ExceptionName  e1)
        {
                //Catch block1
        }
}
catch(ExceptionName1  e2)
{
        //Catch block 2
}
```

Example:

```
class demoTry1
```

```
{
        public static void main(String[]  args)
        {
                try
                {
                        int  arr[]={5,0,1,2};
                        try
                        {
                                arr[4] = arr[3]/arr[1];
                        }
                        catch(ArithmeticException e)
                        {
                                System.out.println("divide by zero :: "+e);
                        }
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("array index out of bound exception :: "+e);
                }
                catch(Exception e)
                {
                        System.out.println("Generic exception :: "+e);
                }
                System.out.println("Out of try..catch block");
        }
}
```

Output:

```
divide by zero :: java.lang.ArithmeticException: / by zero
Out of try..catch block
```

3) finally:

- A **finally** keyword is used to create a block of code that **follows a try or catch** block.
- A **finally block** of code always executes **whether or not exception has occurred**.
- Using a finally block, 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.

Syntax:

```
try
{
        //Protected code
}
catch(ExceptionType1  e1)
{
        //Catch block 1
}
catch(ExceptionType2 e2)
{
```

```
                //Catch block 2
    }
    finally
    {
                //The finally block always executes.
    }
```

Example:

```
class  demoFinally
{
        public static void main(String args[])
        {
                int  a[] = new int[2];
                try
                {
                        System.out.println("Access element three :" + a[3]);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Exception thrown  :" + e);
                }
                finally
                {
                        a[0] = 10;
                        System.out.println("First element value: " +a[0]);
                        System.out.println("The finally block is always executed");
                }
                System.out.println("Out of try...catch...finally... ");
        }
}
```

Output:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 10
The finally block is always executed
Out of try...catch...finally...
```

Key points to keep in mind:

- A **catch** clause cannot exist **without a try** statement.
- It is not compulsory to have **finally** clause for every try/catch.
- The **try** block cannot be present without either **catch** clause or **finally** clause.
- **Any code** cannot be present in between the **try, catch, finally blocks**.

4) throw :

- The **throw** keyword is used to **explicitly** throw an exception.
- We can throw either **checked** or **uncheked** exception using **throw** keyword.
- Only object of **Throwable** class or its **sub classes** can be **thrown**.
- Program **execution stops** on encountering throw statement, and the **closest catch** statement is checked for matching type of **exception**.

Syntax:

```
throw ThrowableInstance;
```

Example:

```
class demoThrow
{
        static void demo()
        {
                try
                {
                        throw new ArithmeticException("demo");
                }
                catch(ArithmeticException  e)
                {
                        System.out.println("Exception caught");
                }
        }
        public static void main(String args[])
        {
                demo();
        }
}
```

Output:

```
Exception caught
```

5) throws:

- The throws keyword is used to **declare an exception**.
- If a method does not handle a **checked** exception, the method must declare it using the **throws** keyword. The **throws** keyword appears **at the end of a method's signature**.
- You can declare **multiple exceptions**.

Syntax:

```
return_type  method_name()  throws  exception_class_name_list
{
        //method code
}
```

Example:

```
class  demoThrows
{
        static void display() throws ArithmeticException
        {
                System.out.println("Inside check function");
                throw new ArithmeticException("Demo");
        }
        public static void main(String args[])
        {
                try
                {
                        display();
                }
```

```
                catch(ArithmeticException e)
                {
                        System.out.println("caught :: " + e);
                }
        }
}
```

Output:

```
Inside check function
caught :: java.lang.ArithmeticException: Demo
```

User Defined Exception

- In java, we can create our own exception that is known as **custom exception or user-defined exception**.
- We can have our own exception and message.
- **Key points to keep in mind:**
  - All exceptions must be a child of **Throwable**.
  - If you want to write a **checked** exception that is automatically enforced by the Declare Rule, you need to extend the **Exception** class.
  - If you want to write a **runtime exception**, you need to extend the **RuntimeException** class.

Example:

```
class demoUserException extends Exception
{
        private int ex;
        demoUserException(int a)
        {
                ex=a;
        }
        public String toString()
        {
                return "MyException[" + ex +"] is less than zero";
        }
}
class demoException
{
        static void sum(int a,int b) throws demoUserException
        {
                if(a<0)
                {
                        throw new demoUserException (a);
                }
                else
                {
                        System.out.println(a+b);
                }
        }
        public static void main(String[] args)
        {
```

```
                try
                {
                        sum(-10, 10);
                }
                catch(demoUserException e)
                {
                        System.out.println(e);
                }
        }
}
```

Output:

MyException[-10] is less than zero

List of Java Exception (Built-In Exception)

- Java defines several built-in exception classes inside the standard package **java.lang.**
- **Checked Exception:**

Exception	Description
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program
IllegalAccessException	This Exception occurs when you create an object of an abstract class and interface
NotSuchMethodException	This Exception occurs when the method you call does not exist in class.
NoSuchFieldException	A requested field does not exist.

- **Unchecked Exception:**

Exception	Description
ArithmeticException	This Exception occurs, when you divide a number by zero causes an Arithmetic Exception.
ArrayIndexOutOfBounds Exception	This Exception occurs, when you assign an array which is not compatible with the data type of that array.
NumberFormatException	This Exception occurs, when you try to convert a string variable in an incorrect format to integer (numeric format) that is not compatible with each other.
ClassCastException	Invalid cast.
NullPointerException	Invalid use of a null reference.

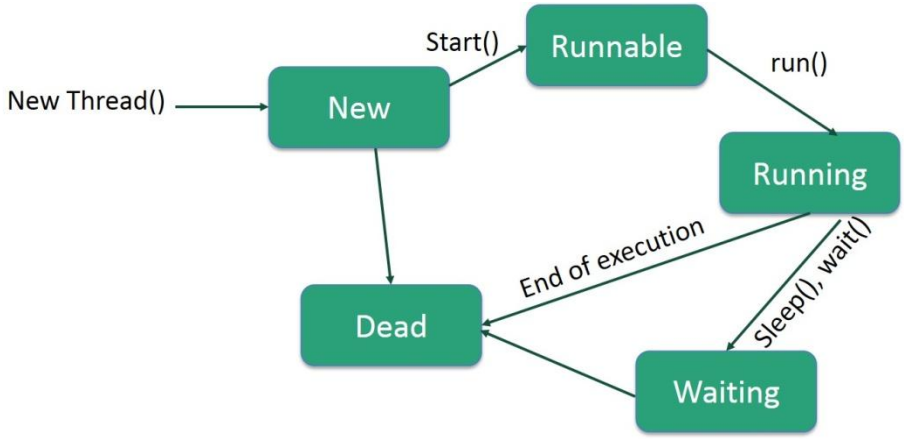
Multithreading

- Java is a **multithreaded** programming language which means we can develop multi threaded program using Java.
- **Multithreaded** programs contain **two or more threads** that can run concurrently. This means that a single program can perform two or more tasks simultaneously.
- Thread is basically a **lightweight sub-process**, a smallest unit of processing.
- Multiprocessing and multithreading, both are used to **achieve multitasking**.

- But we use **multithreading** than **multiprocessing** because threads share a **common memory area**. They don't allocate separate memory area which saves memory, and context-switching between the threads **takes less time than process**.
- Threads are **independent.** So, it doesn't affect other threads if **exception occurs** in a single thread.
- Java Multithreading is mostly used in games, animation etc.
- For example, **one thread** is writing content on a file at the same time **another thread** is performing spelling check.
- In **Multiprocessing**, Each process has its own address in memory. So, each process allocates separate memory area.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc. So that cost of communication between **the processes is high**.
- **Disadvantage:** If you create too many threads, you can actually **degrade the performance** of your program rather than **enhance** it.
- Remember, some overhead is associated with **context switching**. If you create too many threads, more **CPU time** will be spent changing contexts than executing your program.

Life Cycle of Thread

- A thread goes through various stages in its life cycle. **For example**, a thread is **born**, **started**, **runs**, and then **dies**.
- **Diagram:**



- **New:**
  - A new thread begins its life cycle in the **New** state. It remains in this state until the program **starts the thread** by invoking **Start()** method. It is also referred to as a **born** thread.
- **Runnable:**
  - The thread is in **runnable** state after invocation of **start()** method, but the thread **scheduler** has not selected it to be **the running thread**.
  - The thread is in **running** state if the **thread scheduler** has selected it.
- **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 transition 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 (**run() method exits**) completes its task.

Thread Priorities

- Every Java thread has a **priority** that helps the operating system determine the **order in which threads are scheduled**.
- Java priorities are in the range between **MIN_PRIORITY** (a constant of 1) and **MAX_PRIORITY** (a constant of 10).
- 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.
- The thread scheduler mainly uses **preemptive** or **time slicing** scheduling to schedule the threads.

Creating a Thread

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

1) Constructor of Thread class:

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

2) Methods of Thread Class:

Method	Description
public void **run**()	Entry point for a thread
public void **start**()	start a thread by calling run() method
public String **getName**()	return thread's name
public void **setName**(String **name**)	to give thread a name
public int **getPriority**()	return thread's priority
public int **setPriority**(int **priority**)	Sets the priority of this Thread object. The possible values are between 1 and 10.
public final boolean **isAlive**()	checks whether thread is still running or not
public static void **sleep**(long **millisec**)	suspend thread for a specified time
public final void **join**(long **millisec**)	Wait for a thread to end

- Java defines two ways by which a thread can be created.
  - By implementing the **Runnable** interface.
  - By extending the **Thread** class.

1) By implementing the Runnable interface:

- The easiest way to create a thread is to create a **class that implements the runnable interface**.
- After implementing runnable interface , the class needs to implement the **run()** method, which has following form:

public void run( )

- This method provides **entry point** for the thread and you will put you complete **business logic** inside this method.
- After that, you will instantiate a **Thread object** using the following constructor:

Thread (Runnable threadObj**, String** threadName**) ;**

- Where, **threadObj** is an instance of a class that implements the **Runnable** interface and **threadName** is the **name** given to the **new thread**.
- Once **Thread object** is created, you can start it by calling **start( )** method, which executes a call to **run( )** method.

void start ( );

Example:

```
class demoThread3 implements Runnable
{
        public void run()
        {
                System.out.println("Thread is running...");
        }
        public static void main(String args[])
        {
                demoThread3 d1=new demoThread3 ();
                Thread t1 =new Thread(d1);
                t1.start();
        }
}
```

Output:

```
Thread is running...
```

Example:

```
class RunnableDemo implements Runnable
{
        Thread t;
        String threadName;
        RunnableDemo( String name)
        {
                threadName = name;
                System.out.println("Creating " +  threadName );
        }
        public void run()
        {
```

```
                System.out.println("Running " +  threadName );
                try
                {
                        for(int i = 2; i >= 0; i--)
                        {
                                System.out.println("Thread: " + threadName + ", " + i);
                                // Let the thread sleep for a while.
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Thread " +  threadName + " interrupted.");
                }
                System.out.println("Thread " +  threadName + " exiting.");
        }
        public void start ()
        {
                System.out.println("Starting " +  threadName );
                if (t == null)
                {
                        t = new Thread (this, threadName);
                        t.start ();
                }
        }

}
public class ThreadDemo
{
        public static void main(String args[])
        {
                RunnableDemo R1 = new RunnableDemo( "Thread-1");
                R1.start();
        }
}
```

Output:

```
Creating Thread-1
Starting Thread-1
Running Thread-1
Thread: Thread-1, 2
Thread: Thread-1, 1
Thread: Thread-1, 0
Thread Thread-1 exiting.
```

2) By extending the Thread class.

- Second way to create a thread is to **create a new class** that extends **Thread** class and then create an instance of that class.

- The extending class must override the **run( )** method, which is the entry point for the new thread.
- Once **Thread object** is created, you can start it by calling **start( )** method, which executes a call to **run( )** method.

Example:

```
class demoThread2 extends Thread
{
        public void run()
        {
                System.out.println("Thread is running...");
        }
        public static void main(String args[])
        {
                demoThread2  t1 = new demoThread2();
                t1.start();
        }
}
```

Output:

```
Thread is running...
```

Example:

```
class ThreadDemo extends Thread
{
        Thread t;
        String threadName;
        ThreadDemo( String name)
        {
                threadName = name;
                System.out.println("Creating " +  threadName );
        }
        public void run()
        {
                System.out.println("Running " +  threadName );
                try
                {
                        for(int i = 2; i >= 0; i--)
                        {
                                System.out.println("Thread: " + threadName + ", " + i);
                                // Let the thread sleep for a while.
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e)
```

```
                {
                        System.out.println("Thread " +  threadName + " interrupted.");
                }
                System.out.println("Thread " +  threadName + " exiting.");
        }
        public void start ()
        {
                System.out.println("Starting " +  threadName );
                if (t == null)
                {
                        t = new Thread (this, threadName);
                        t.start ();
                }
        }
}
public class ThreadDemo2
{
        public static void main(String args[])
        {
                ThreadDemo T1 = new ThreadDemo( "Thread-1");
                T1.start();
        }
}
```

Output:

```
Creating Thread-1
Starting Thread-1
Running Thread-1
Thread: Thread-1, 2
Thread: Thread-1, 1
Thread: Thread-1, 0
Thread Thread-1 exiting.
```

Thread Synchronization

- When two or more threads need access to a **shared** resource, they need some way to ensure that the resource will be used by **only one thread at a time**.
- The process by which this synchronization is achieved is called **thread synchronization**.
- The **synchronized** keyword in Java creates a block of code referred to as **a critical section**.
- Every Java object with a critical section of code gets a **lock associated with the object**.
- To enter **a critical section**, a thread needs to obtain the corresponding object's lock.

Syntax:

```
synchronized(object)
{
```

```
        // statements to be synchronized
    }
```

- Here, **object** is a reference to the object being **synchronized**.
- A synchronized block ensures that a call to a method that is a member of **object** occurs only after the current thread has successfully entered object's critical section.

Example:

```
class PrintDemo
{
        public void printCount()
        {
                try
                {
                        for(int i = 3; i > 0; i--)
                        {
                                System.out.println("Counter   ---   "  + i );
                        }
                }
                catch (Exception e)
                {
                        System.out.println("Thread  interrupted.");
                }
        }
}
class ThreadDemo extends Thread
{
        Thread t;
        String threadName;
        PrintDemo  PD;

        ThreadDemo( String name,  PrintDemo pd)
        {
                threadName = name;
                PD = pd;
        }
        public void run()
        {
                synchronized(PD)
                {
                        PD.printCount();
                }
                System.out.println("Thread " +  threadName + " exiting.");
        }
        public void start ()
        {
                System.out.println("Starting " +  threadName );
                if (t == null)
                {
                        t = new Thread (this, threadName);
```

```
                        t.start ();
                    }
                }
            }
            public class ThreadSchro
            {
                public static void main(String args[])
                {
                    PrintDemo PD = new PrintDemo();
                    ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
                    ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

                    T1.start();
                    T2.start();
                }
            }
```

Output:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Inter-thread Communication

- **Inter-thread communication or Co-operation** is all about allowing **synchronized threads** to communicate with each other.
- 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.
- To avoid **polling**(It is usually implemented by loop), **Inter-thread communication** is implemented by following methods of **Object class**:
  - **wait( ):** This method tells the calling thread to give up the **critical section** and go to **sleep** until some other thread enters the same **critical section** and calls notify( ).
  - **notify( ):** This method **wakes up** the first thread that called **wait**( ) on the **same object**.
  - **notifyAll( ):** This method **wakes up** all the threads that called **wait**( ) on the **same object**. The highest priority thread will run first.
- Above all methods are implemented as **final** in Object class.
- All three methods can be called only from **within a synchronized context**.

Example:

```
class Customer
{
```

```
        int amount=10000;
        synchronized void withdraw(int amount)
        {
                System.out.println("going to withdraw...");

                if(this.amount<amount)
                {
                        System.out.println("Less balance; waiting for deposit...");
                        try
                        {
                                wait();
                        }
                        catch(Exception e){}
                }
                this.amount-=amount;
                System.out.println("withdraw completed...");
        }
        synchronized void deposit(int amount)
        {
                System.out.println("going to deposit...");
                this.amount+=amount;
                System.out.println("deposit completed... ");
                notify();
        }
}
class InterThreadDemo
{
        public static void main(String args[])
        {
                final Customer c = new Customer();
                new Thread()
                {
                        public void run(){c.withdraw(15000); }
                }.start();
                new Thread()
                {
                        public void run(){c.deposit(10000);}
                }.start();
        }
}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed...
```