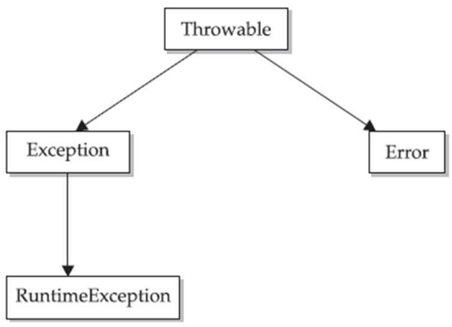
SYLLABUS

Exception Handling
Multithreaded Programming
I/O: I/O Basics, Reading Console Input, Writing Console Output, The PrintWriter class, Reading and Writing Files, Automatically Closing a File.

3.1 Exception Handling

- Exception is a runtime error.
- Exception is an event that interrupt the flow of program execution.



- Throwable is a base class for all the types of exceptions.
- Throwable has two derived classes **Exception** and **Error**.
- The Exception is used to handle the user program errors at run time like IOException.
- The RuntimeException is subclass for Exception and it automatically defined the exception in a program like divide by zero.

Exception Handling:

In java, exceptions are handled by using 5 keywords

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

1. **try:** It contains program statements that you want to monitor for exceptions. If an exception occurs within the **try** block, it is thrown.
2. **catch:** Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system.
3. **throw:** To manually throw an exception, use the keyword **throw**.
4. **throws:** Any exception that is throws out of a method must be specified as such by a **throws** clause.
5. **finally:** Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

Syntax:

```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

Example: demonstrate the **try, catch, and finally blocks**.

```
import java.util.*;
class ExceptionDemo
{
	public static void main(String ar[])
	{
			try
			{
				String lan[]={"C","C++","C#","JAVA"};
				for(int i=0;i<10;i++)
					System.out.println("Language "+(i+1)+" :"+lan[i]);
			}
			catch(Exception ie)
			{
				System.out.println("Index was not found");
			}
			finally
			{
				System.out.println("Final block is executed");
			}

	}
}
```

Example: demonstrate the **multiple catch blocks**.

```
	import java.util.*;
	class ExceptionDemo
	{
		public static void main(String ar[])
		{
				Scanner sc=new Scanner(System.in);
				try
				{
					System.out.println("Enter a value");
					int a=sc.nextInt();
					System.out.println("Enter b value");
					int b=sc.nextInt();
					int c=a/b;
					System.out.println("Division a/b is "+c);
				}
				catch(ArithmeticException ae)
				{
						System.out.println("Do not give dividend as zero");
				}
				catch(InputMismatchException ae)
				{
					System.out.println("Please enter only integer value");
				}
				finally
				{
					System.out.println("Final block is executed");
				}

		}
	}
```

throw:

- To manually throw an exception, use the keyword **throw**.
- Syntax:

  throw ***ThrowableInstance*** ;
- Example:

  throw **new ArithmeticException("Exception Message")** ;
- Here, ***ThrowableInstance*** must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways you can obtain a **Throwable** object:
  - using a parameter in a **catch** clause.
  - creating one with the **new** operator.
- **Execution process**
  - ✓ The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
  - ✓ The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.
  - ✓ If it does find a match, control is transferred to that statement.
  - ✓ If not, then the next enclosing **try** statement is inspected, and so on.
  - ✓ If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Example // Demonstrate throw and rethow

```
class ThrowDemo
{
    static void demoproc()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside demoproc.");
            throw new ArithmeticException("In valid operation") ; // rethrow the exception
        }
    }
    public static void main(String args[])
    {
        try
        {
            demoproc();
        }
        catch(ArithmeticException ae)
        {
            System.out.println("Recaught: " +ae);
        }
    }
}
```

throws:

- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception by using a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- Syntax:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Example: demonstrate the throws clause

```
class ThrowsDemo
{
    static void Display() throws IllegalAccessException
    {
        System.out.println("Inside Display.");
        throw new IllegalAccessException("Illegal access");
    }
    public static void main(String args[])
    {
        try {
            Display();
        }
        catch(IllegalAccessException ie)
        {
            System.out.println(ie);
        }
    }
}
```

finally:

- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.
- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

```
// Demonstrate finally.
class FinallyDemo
{
        // Throw an exception out of the method.
        static void procA()
        {
                try
                {
                        System.out.println("inside procA");
                        throw new RuntimeException("demo");
                }
                finally
                {
                        System.out.println("procA's finally");
                }
        }
        // Return from within a try block.
        static void procB()
        {
                try
                {
                        System.out.println("inside procB");
                        return;
                }
                finally
                {
                        System.out.println("procB's finally");
                }
        }
        // Execute a try block normally.
        static void procC()
        {
                try
                {
                        System.out.println("inside procC");
                }
                finally
                {
                        System.out.println("procC's finally");
                }
        }
        public static void main(String args[])
        {
                 try
                 {
                        procA();
                 }
                 catch (Exception e)
                 {
                        System.out.println("Exception caught");
                 }
                 procB();
                 procC();
        }
}
```

Output:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Nested try Statements:

- A **try** statement can be inside the block of another **try** is called **nested try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

Example: An example of nested try statements.

```
class NestTry
{
    public static void main(String args[])
    {
        try
        {
            int a = args.length;
            int b = 42 / a;                    // generate division by zero
            System.out.println("a = " + a);
            try {
                if(a==1)
                    a = a/(a-a);      // generate division by zero
                if(a==2)
                {
                    int c[] = { 1 };
                    c[42] = 99;       // generate an out-of-bounds exception
                }
              }
              catch(ArrayIndexOutOfBoundsException e)
              {
                System.out.println("Array index out-of-bounds: " + e);
              }
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```