UNIT –IV & UNIT-V

Exception Handling:

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

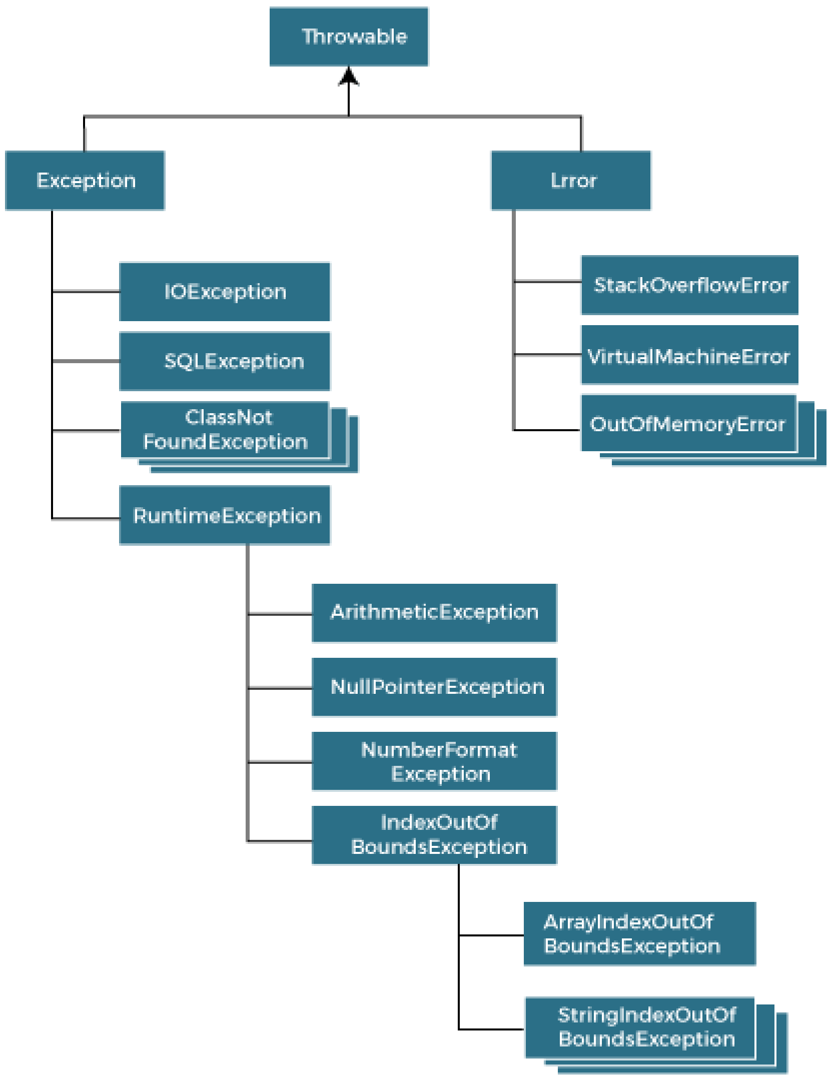
Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

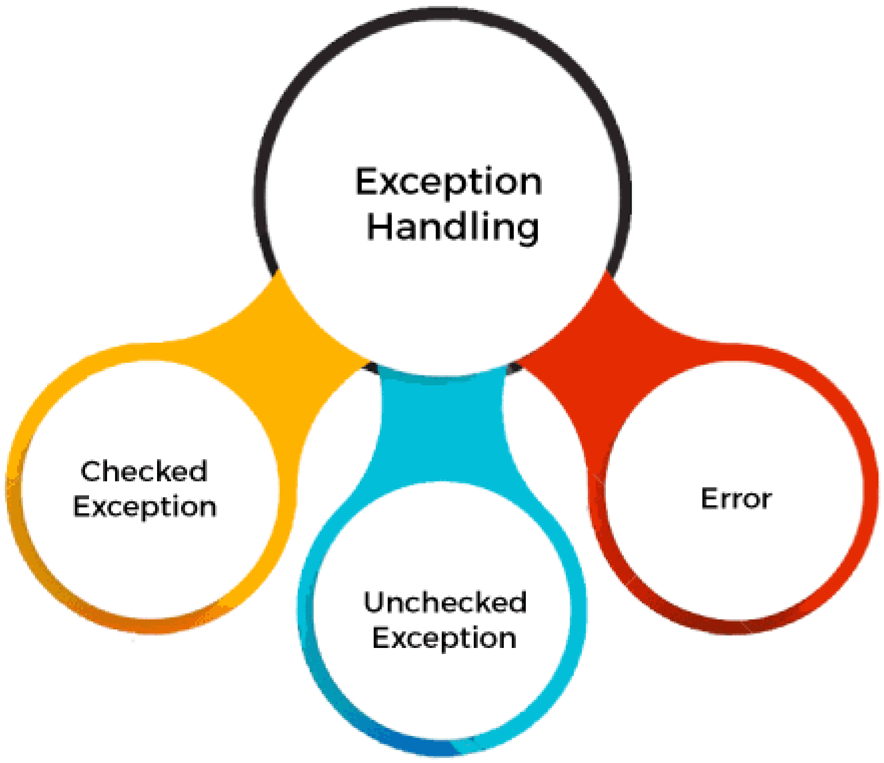


Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception

2. Unchecked Exception
3. Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
    }catch(ArithmeticException e){System.out.println(e);}
    //rest code of the program
    System.out.println("rest of the code...");
  }
}
```

Java Catch Multiple Exceptions

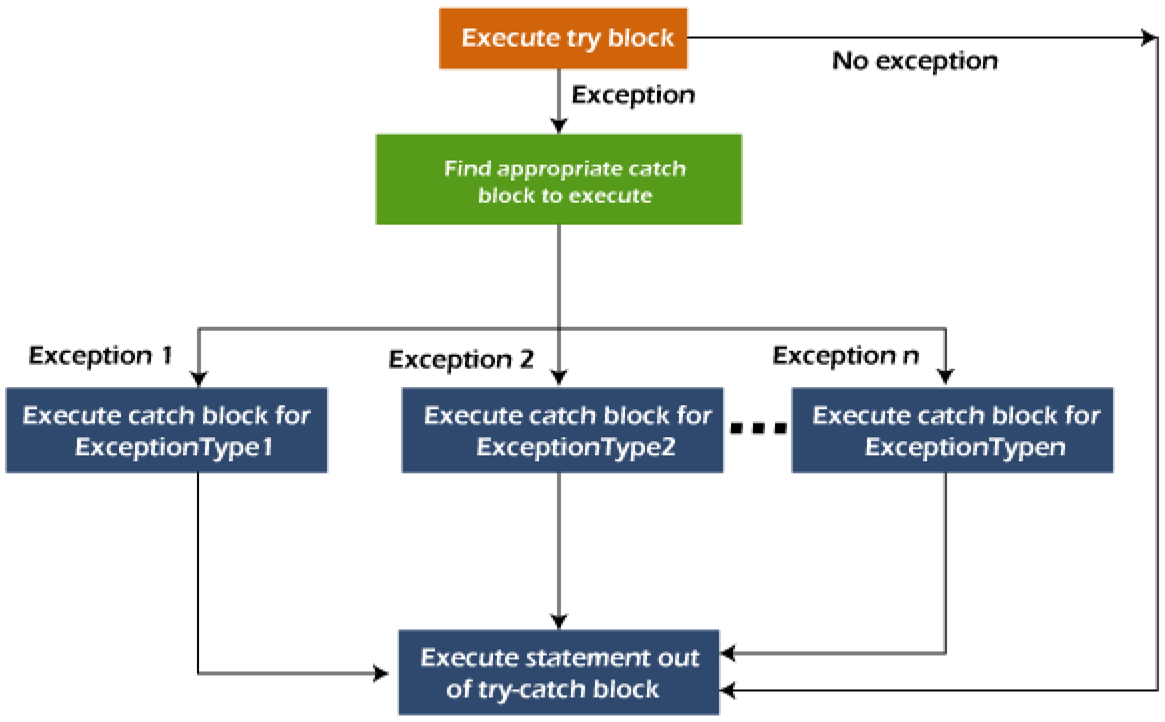
Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType1
Execute catch block for ExceptionType2
Execute catch block for ExceptionTypen
Execute statement out of try-catch block

Example 1

```
public class Multicatch
{

	public static void main(String[] args)
	{
		try
		{
		String s1=null;
		System.out.println(s1.length());
		}
		catch(ArithmeticException e)
		{
			System.out.println(e);
		}
		catch(NumberFormatException e)
		{
			System.out.println(e);
		}
		catch(NullPointerException e)
		{
			System.out.println(e);
		}
	}

}
```

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
public class MultipleCatchBlock1 {

    public static void main(String[] args) {

            try{
                int a[]=new int[5];
                a[5]=30/0;
               }
               catch(ArithmeticException e)
                 {
                  System.out.println("Arithmetic Exception occurs");
                 }
               catch(ArrayIndexOutOfBoundsException e)
                 {
                  System.out.println("ArrayIndexOutOfBounds Exception occurs");
                 }
               catch(Exception e)
                 {
                  System.out.println("Parent Exception occurs");
                 }
               System.out.println("rest of the code");
    }
}
```

Example 2

MultipleCatchBlock2.java

```
public class MultipleCatchBlock2 {

    public static void main(String[] args) {
        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
import java.io.DataInputStream;
import java.io.IOException;

/* compile time exceptions:
 * 1. ClassNotFoundException
 * 2. FileNotFoundException abc.txt    ----->xyz.txt
 * 3. SQLException----------------------------------->
 * 4.IllegalThreadStateException.----------------------------->  t1.start();
t1.start();
 *
 *
 *
 *
 */
public class Sample_throws
{
         public static void main(String[] args) throws IOException
   {
         DataInputStream ds=new DataInputStream(System.in);
         String uname,pwd;
         System.out.println("Enter username:");
         uname=ds.~~readLine~~();
         System.out.println("Enter password:");
         pwd=ds.~~readLine~~();
         System.out.println("the user name is:"+uname);
         System.out.println("The password is:"+pwd);
    }
}
```

```
Enter username:
abc
Enter password:
abc+23
the user name is:abc
The password is:abc+23
```

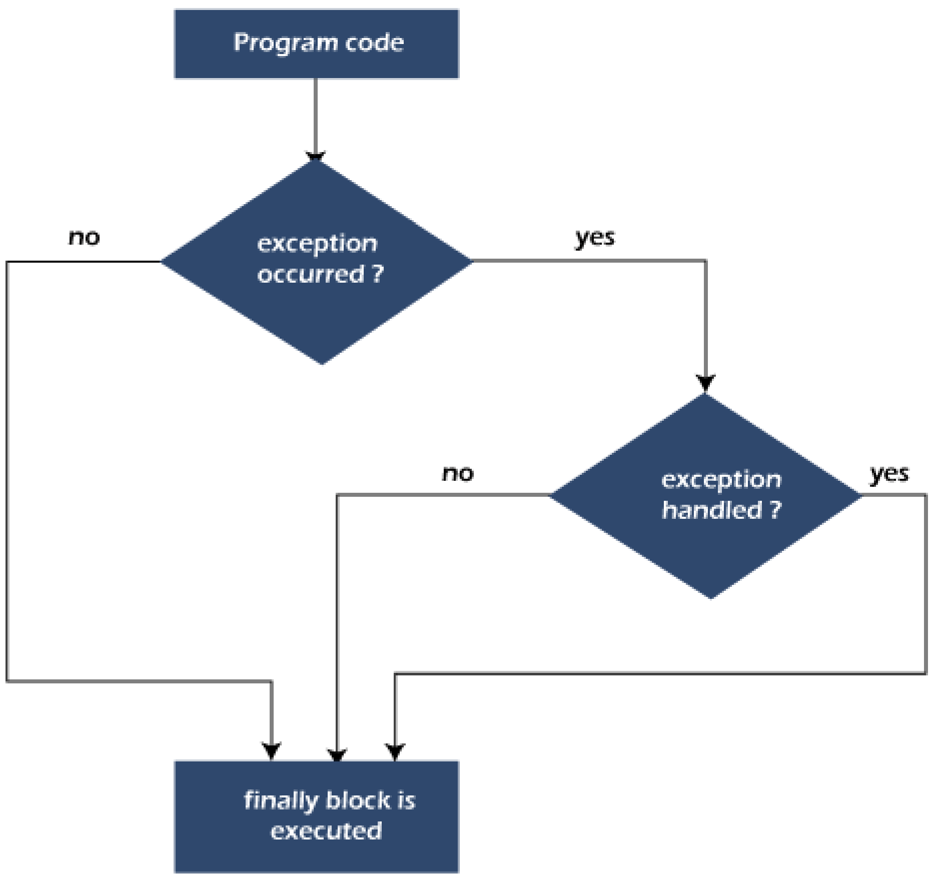
Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

TestFinallyBlock.java

```
public class Finally_Samp1le
{
  public static void main(String[] args)
  {
        try
        {
               int i=10;
               System.out.println(i/0);
        }
        finally
        {
               System.out.println("Done !!!");
        }
}

}
```

```
Output:

Done !!!
Exception in thread "main" java.lang.ArithmeticException: / by zero
       at Finally_Samp1le.main(Finally_Samp1le.java:9)
```

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

```
throw new exception_class("error message");
throw new IOException("sorry device error");
```

Java throw keyword Example

Example 1: Throwing Unchecked Exception

In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
```

```
        System.out.println("rest of the code...");
  }
}
```

```
import java.io.*;

public class TestThrow2 {

    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {

        FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);


        throw new FileNotFoundException();

    }
    //main method
    public static void main(String args[]){
        try
        {
            method();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
  }
}
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

```
public class FinalExampleTest {
   //declaring final variable
   final int age = 18;
   void display() {

   // reassigning value to age variable
   // gives compile time error
   age = 55;
   }

   public static void main(String[] args) {

   FinalExampleTest obj = new FinalExampleTest();
   // gives compile time error
   obj.display();
   }
}
```

Java finally Example

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```
public class FinallyExample {
     public static void main(String args[]){
     try {
       System.out.println("Inside try block");
```

```
  // below code throws divide by zero exception
   int data=25/0;
   System.out.println(data);
  }
  // handles the Arithmetic Exception / Divide by zero exception
  catch (ArithmeticException e){
    System.out.println("Exception handled");
    System.out.println(e);
  }
  // executes regardless of exception occurred or not
  finally {
    System.out.println("finally block is always executed");
  }
  System.out.println("rest of the code...");
  }
}
```

Java finalize Example

FinalizeExample.java

```
public class FinalizeExample {
     public static void main(String[] args)
    {
        FinalizeExample obj = new FinalizeExample();
        // printing the hashcode
        System.out.println("Hashcode is: " + obj.hashCode());
        obj = null;
        // calling the garbage collector using gc()
        System.gc();
        System.out.println("End of the garbage collection");
    }
   // defining the finalize method
    protected void finalize()
    {
            System.out.println("Called the finalize() method");
        }
    }
```

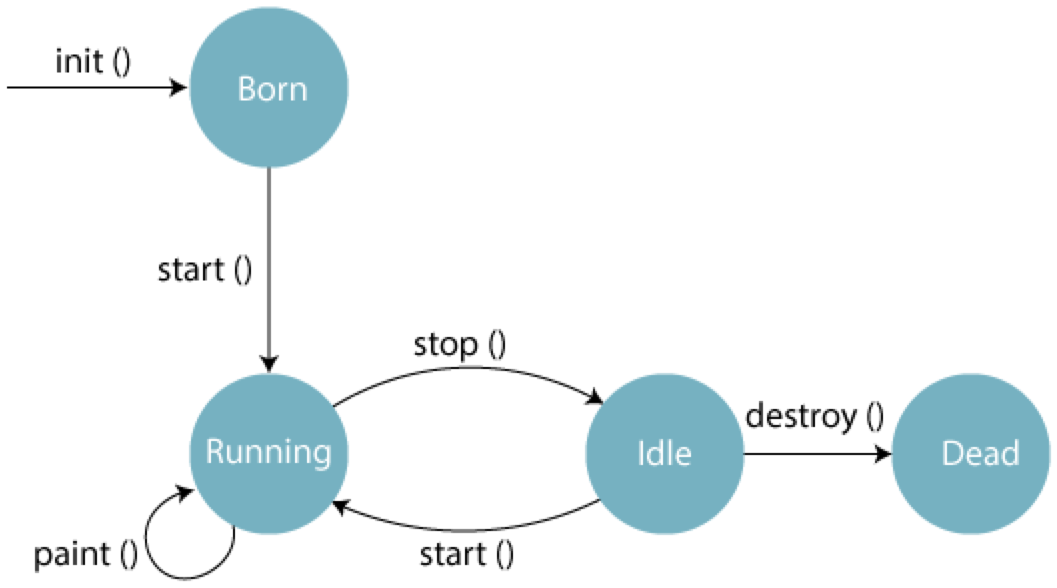
Applet Life Cycle in Java

In Java, an applet is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

Along with the browser, the applet also works on the client side, thus having less processing time.

Methods of Applet Life Cycle



- **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.

- **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.
- **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
- **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
- **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

Sequence of method execution when an applet is executed:

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

Sequence of method execution when an applet is executed:

1. stop()
2. destroy()

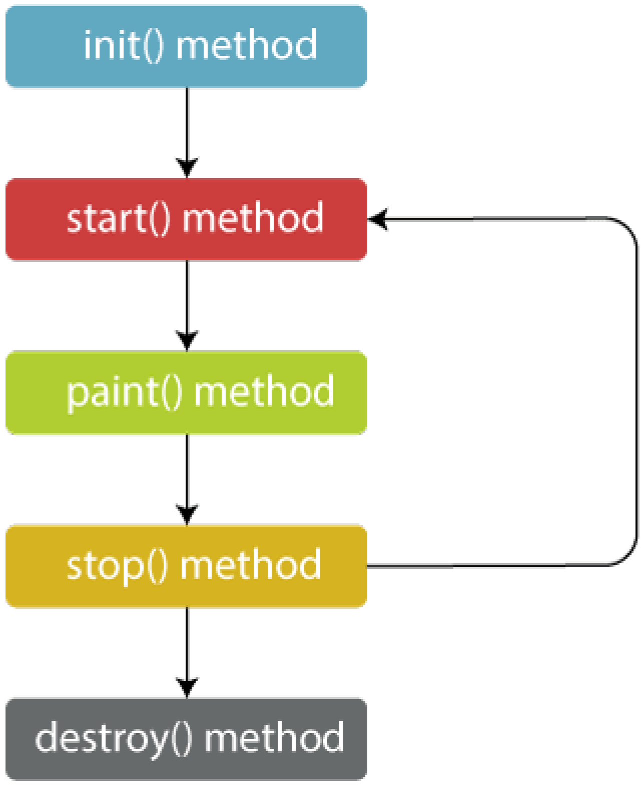
Applet Life Cycle Working

- The Java plug-in software is responsible for managing the life cycle of an applet.

- An applet is a Java application executed in any web browser and works on the client-side. It doesn't have the main() method because it runs in the browser. It is thus created to be placed on an HTML page.
- The init(), start(), stop() and destroy() methods belongs to the **applet.Applet** class.
- The paint() method belongs to the **awt.Component** class.
- In Java, if we want to make a class an Applet class, we need to extend the **Applet**
- Whenever we create an applet, we are creating the instance of the existing Applet class. And thus, we can use all the methods of that class.

Flow of Applet Life Cycle:

These methods are invoked by the browser automatically. There is no need to call them explicitly.



Syntax of entire Applet Life Cycle in Java

```
class TestAppletLifeCycle extends Applet {
public void init() {
// initialized objects
}
public void start() {
// code to start the applet
}
public void paint(Graphics graphics) {
// draw the shapes
}
public void stop() {
// code to stop the applet
}
public void destroy() {
// code to destroy the applet
}
}
```

Graphics class methods in java

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.

6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

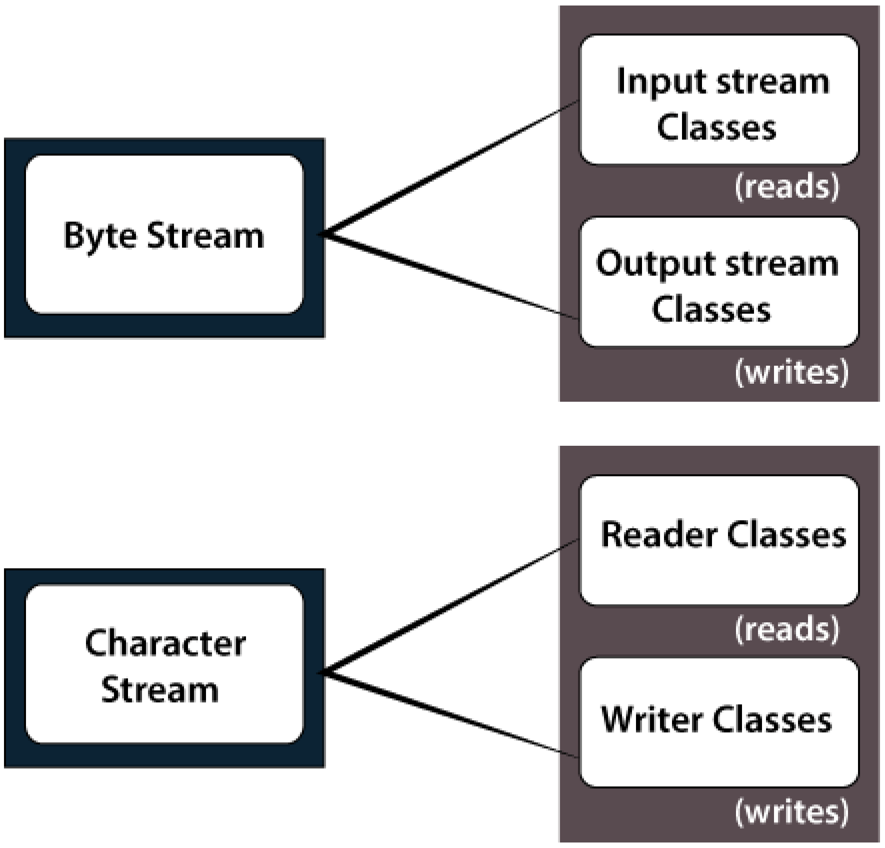
File Operations in Java

In Java, a **File** is an abstract data type. A named location used to store related information is known as a **File**. There are several **File Operations** like **creating a new File, getting information about File, writing into a File, reading from a File** and **deleting a File**.

Before understanding the File operations, it is required that we should have knowledge of **Stream** and **File methods**. If you have knowledge about both of them, you can skip it.

Stream

A series of data is referred to as **a stream**. In Java, **Stream** is classified into two types, i.e., **Byte Stream** and **Character Stream**.



Brief classification of I/O streams

Byte Stream

Byte Stream is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

Character Stream

Character Stream is mainly involved with character data. A file handling process with a character stream is a process in which an input is provided and executed with the character data.

Java File Class Methods

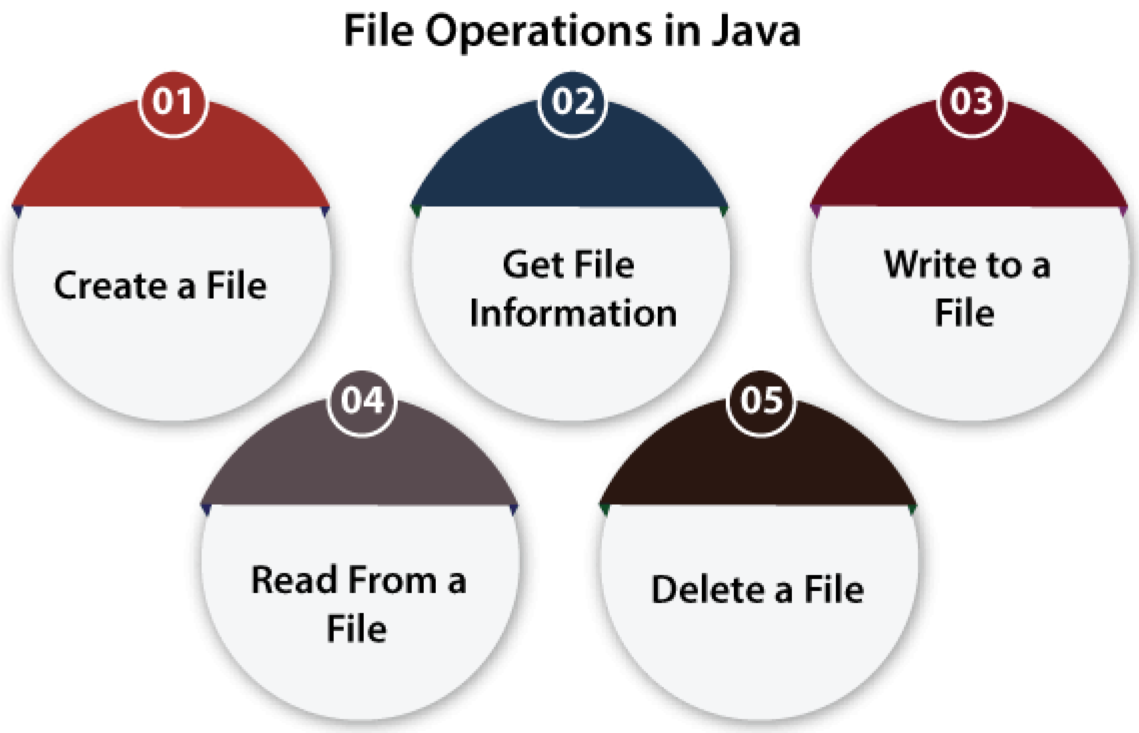
S.No.	Method	Return Type	Description
1.	canRead()	Boolean	The **canRead()** method is used to check whether we can read the data of the file or not.

2.	createNewFile()	Boolean	The **createNewFile()** method is used to create a new empty file.
3.	canWrite()	Boolean	The **canWrite()** method is used to check whether we can write the data into the file or not.
4.	exists()	Boolean	The **exists()** method is used to check whether the specified file is present or not.
5.	delete()	Boolean	The **delete()** method is used to delete a file.
6.	getName()	String	The **getName()** method is used to find the file name.
7.	getAbsolutePath()	String	The **getAbsolutePath()** method is used to get the absolute pathname of the file.
8.	length()	Long	The **length()** method is used to get the size of the file in bytes.
9.	list()	String[]	The **list()** method is used to get an array of the files available in the directory.
10.	mkdir()	Boolean	The **mkdir()** method is used for creating a new directory.

File Operations

We can perform the following operation on a file:

- Create a File
- Get File Information
- Write to a File
- Read from a File
- Delete a File



Create a File

Create a File operation is performed to create a new file. We use the **createNewFile()** method of file. The **createNewFile()** method returns true when it successfully creates a new file and returns false when the file already exists.

Let's take an example of creating a file to understand how we can use the **createNewFile()** method to perform this operation.

CreateFile.java

```
// Importing File class
import java.io.File;
// Importing the IOException class for handling errors
import java.io.IOException;
 class CreateFile {
            public static void main(String args[]) {
            try {
                  // Creating an object of a file
                  File f0 = new File("D:FileOperationExample.txt");
```

```
                    if (f0.createNewFile()) {
                            System.out.println("File " + f0.getName() + " is created successfull
y.");
                    } else {
                            System.out.println("File is already exist in the directory.");
                    }
                  } catch (IOException exception) {
                         System.out.println("An unexpected error is occurred.");
                         exception.printStackTrace();
                }
        }
}
```

Write to a File

The next operation which we can perform on a file is **"writing into a file"**. In order to write data into a file, we will use the **FileWriter** class and its **write()** method together. We need to close the stream using the **close()** method to retrieve the allocated resources.

Let's take an example to understand how we can write data into a file.

WriteToFile.java

```
// Importing the FileWriter class
import java.io.FileWriter;

// Importing the IOException class for handling errors
import java.io.IOException;

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

   try {
      FileWriter fwrite = new FileWriter("D:FileOperationExample.txt");
      // writing the content into the FileOperationExample.txt file
```

```
        fwrite.write("A named location used to store related information is referred to as a F
ile.");

        // Closing the stream
        fwrite.close();
        System.out.println("Content is successfully wrote to the file.");
    } catch (IOException e) {
        System.out.println("Unexpected error occurred");
        e.printStackTrace();
        }
    }
}
```

Read from a File

The next operation which we can perform on a file is **"read from a file"**. In order to write data into a file, we will use the **Scanner** class. Here, we need to close the stream using the **close()** method. We will create an instance of the Scanner class and use the **hasNextLine()** method **nextLine()** method to get data from the file.

Let's take an example to understand how we can read data from a file.

ReadFromFile.java

```
// Importing the File class
import java.io.File;
// Importing FileNotFoundException class for handling errors
import java.io.FileNotFoundException;
// Importing the Scanner class for reading text files
import java.util.Scanner;

class ReadFromFile {
    public static void main(String[] args) {
        try {
            // Create f1 object of the file to read data
            File f1 = new File("D:FileOperationExample.txt");
            Scanner dataReader = new Scanner(f1);
            while (dataReader.hasNextLine()) {
                String fileData = dataReader.nextLine();
                System.out.println(fileData);
            }
            dataReader.close();
        } catch (FileNotFoundException exception) {
            System.out.println("Unexcpected error occurred!");
            exception.printStackTrace();
        }
    }
}
```

Delete a File

The next operation which we can perform on a file is **"deleting a file"**. In order to delete a file, we will use the **delete()** method of the file. We don't need to close the stream using the **close()** method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

Let's take an example to understand how we can write data into a file.

DeleteFile.java

```
// Importing the File class
import java.io.File;
class DeleteFile {
  public static void main(String[] args) {
    File f0 = new File("D:FileOperationExample.txt");
    if (f0.delete()) {
      System.out.println(f0.getName()+ " file is deleted successfully.");
    } else {
      System.out.println("Unexpected error found in deletion of the file.");
    }
  }
}
```