UNIT – 2

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.

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.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

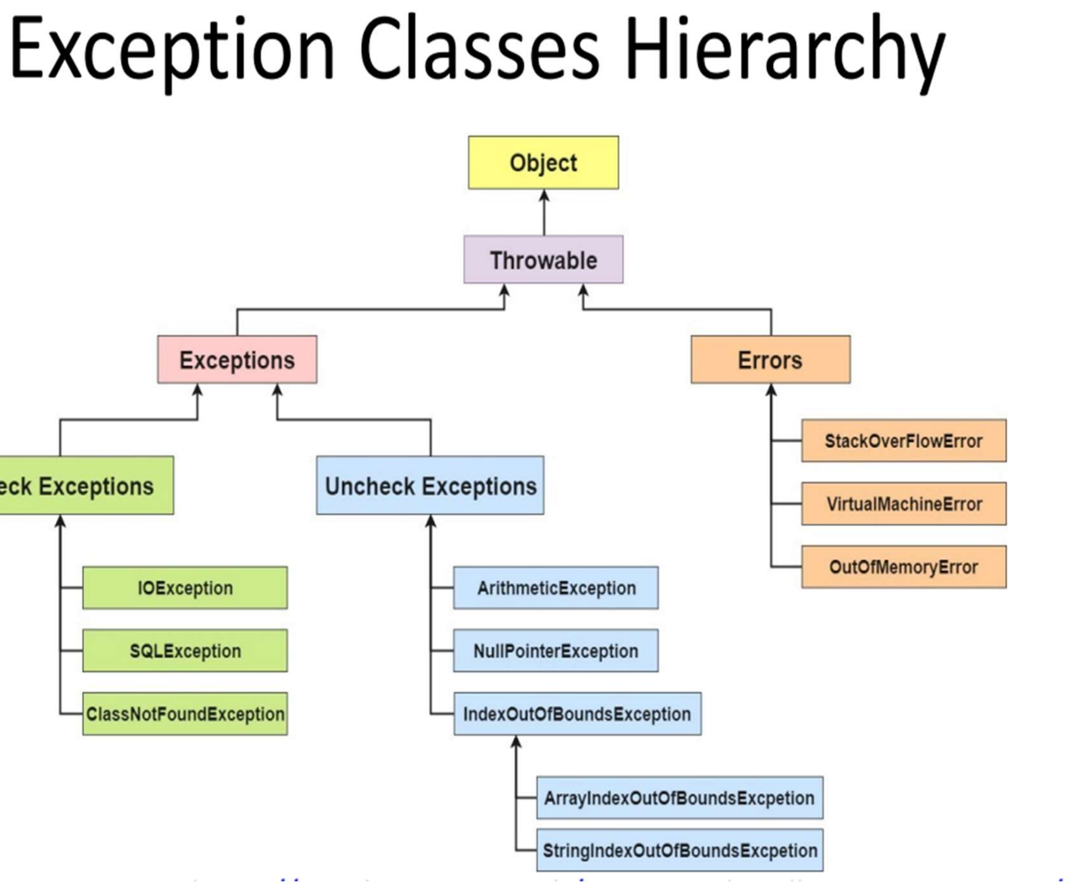
Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

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

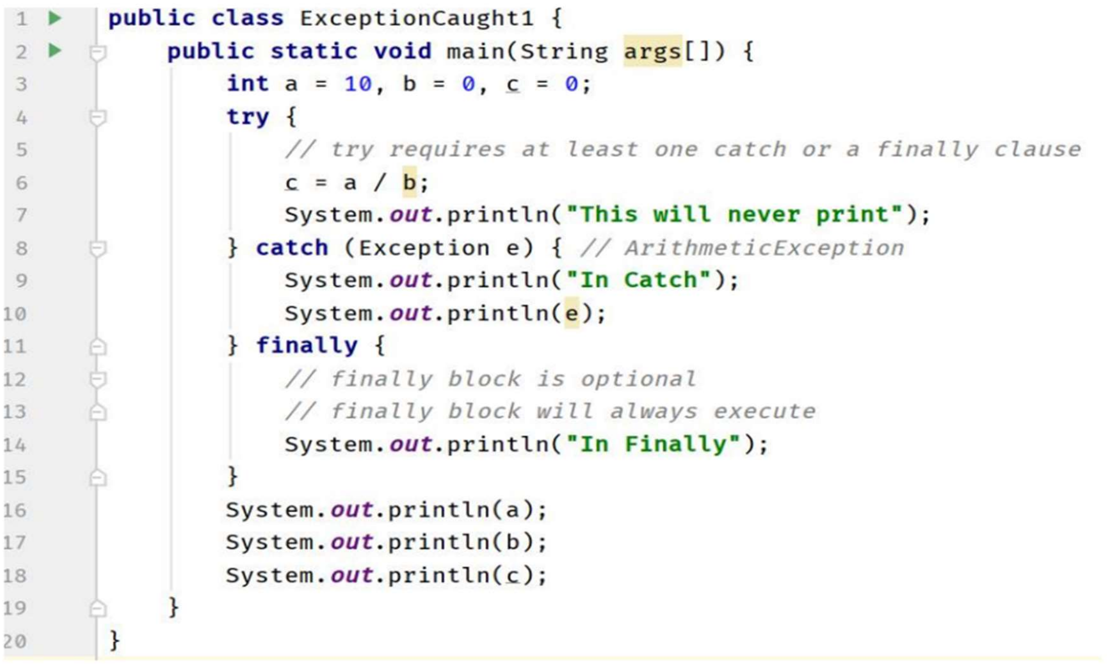
Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

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

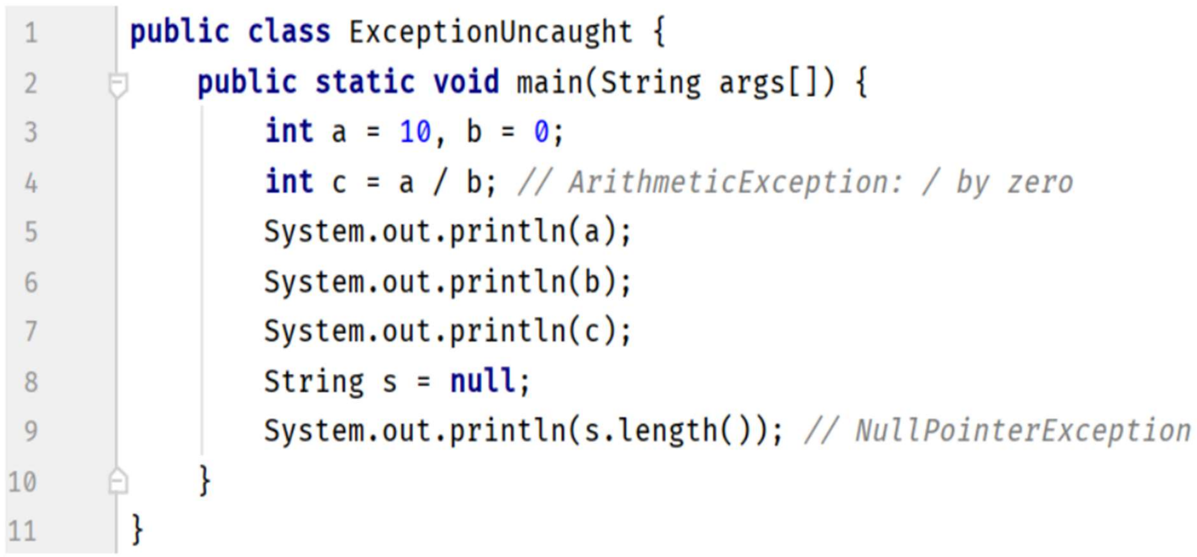
Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which 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.

Caught Exceptions



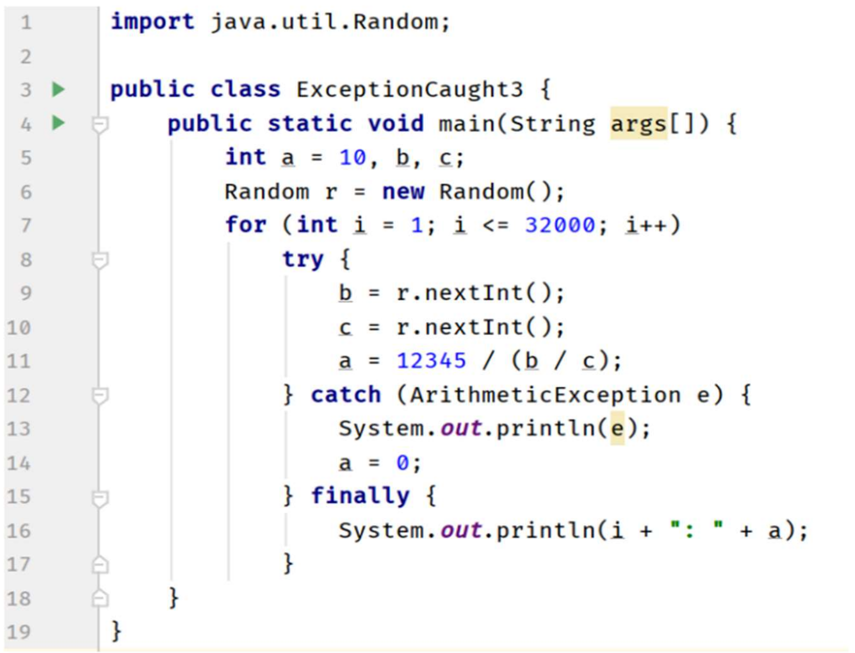
```java
public class ExceptionCaught1 {
    public static void main(String args[]) {
        int a = 10, b = 0, c = 0;
        try {
            // try requires at least one catch or a finally clause
            c = a / b;
            System.out.println("This will never print");
        } catch (Exception e) { // ArithmeticException
            System.out.println("In Catch");
            System.out.println(e);
        } finally {
            // finally block is optional
            // finally block will always execute
            System.out.println("In Finally");
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Uncaught Exceptions



```java
public class ExceptionUncaught {
    public static void main(String args[]) {
        int a = 10, b = 0;
        int c = a / b; // ArithmeticException: / by zero
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        String s = null;
        System.out.println(s.length()); // NullPointerException
    }
}
```

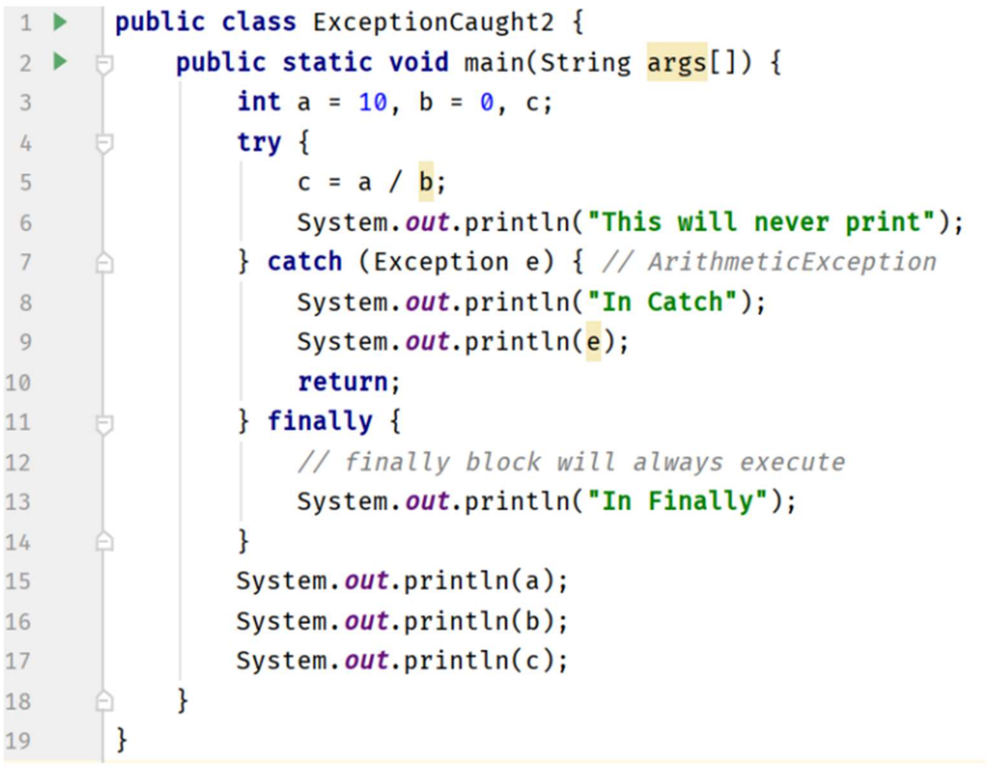
Caught Exceptions



```java
import java.util.Random;

public class ExceptionCaught3 {
    public static void main(String args[]) {
        int a = 10, b, c;
        Random r = new Random();
        for (int i = 1; i <= 32000; i++)
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b / c);
            } catch (ArithmeticException e) {
                System.out.println(e);
                a = 0;
            } finally {
                System.out.println(i + ": " + a);
            }
    }
}
```

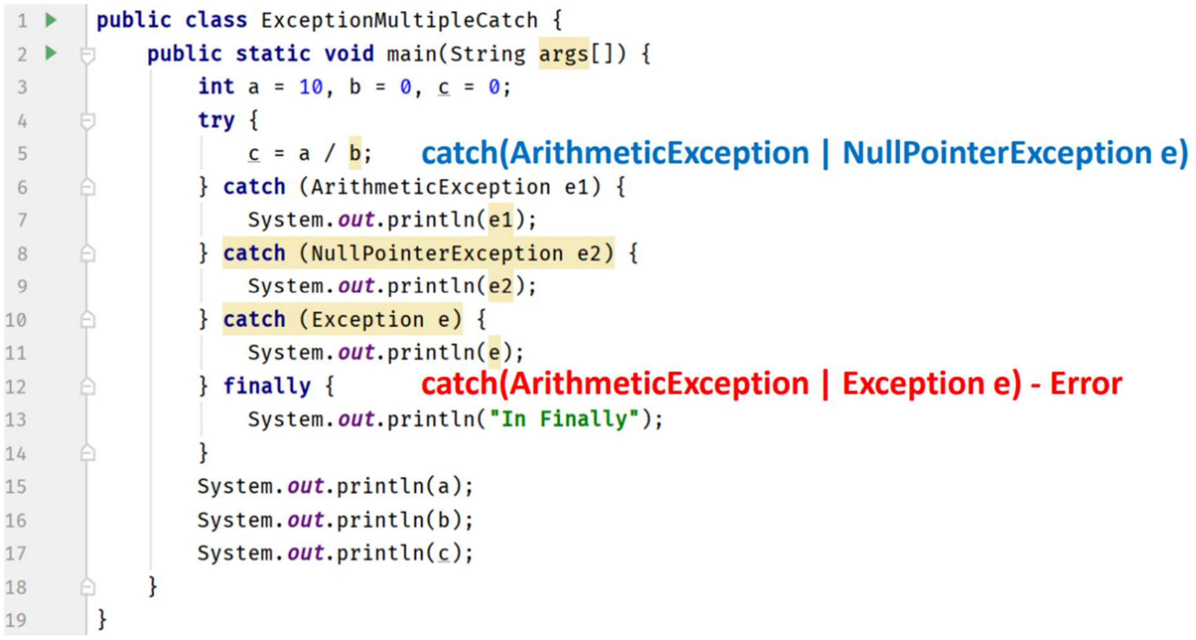
try can be nested, please refer to **ExceptionTryNested.java**

finally



```java
public class ExceptionCaught2 {
    public static void main(String args[]) {
        int a = 10, b = 0, c;
        try {
            c = a / b;
            System.out.println("This will never print");
        } catch (Exception e) { // ArithmeticException
            System.out.println("In Catch");
            System.out.println(e);
            return;
        } finally {
            // finally block will always execute
            System.out.println("In Finally");
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Multiple catch clauses

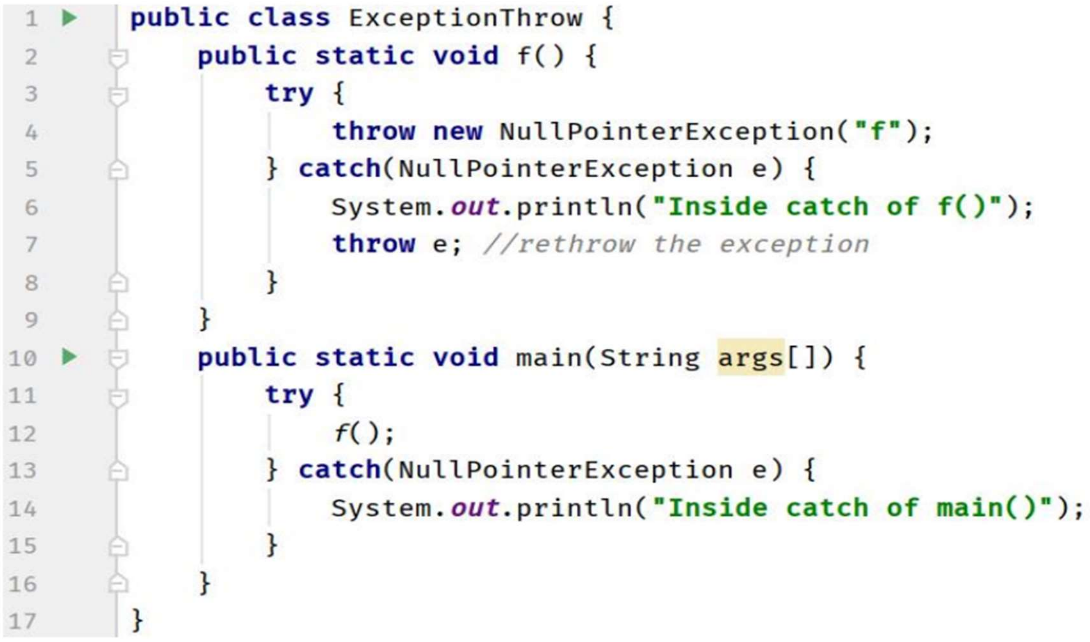


```java
public class ExceptionMultipleCatch {
    public static void main(String args[]) {
        int a = 10, b = 0, c = 0;
        try {
            c = a / b;
        } catch (ArithmeticException e1) {
            System.out.println(e1);
        } catch (NullPointerException e2) {
            System.out.println(e2);
        } catch (Exception e) {
            System.out.println(e);
        } finally {
            System.out.println("In Finally");
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

catch(ArithmeticException | NullPointerException e)

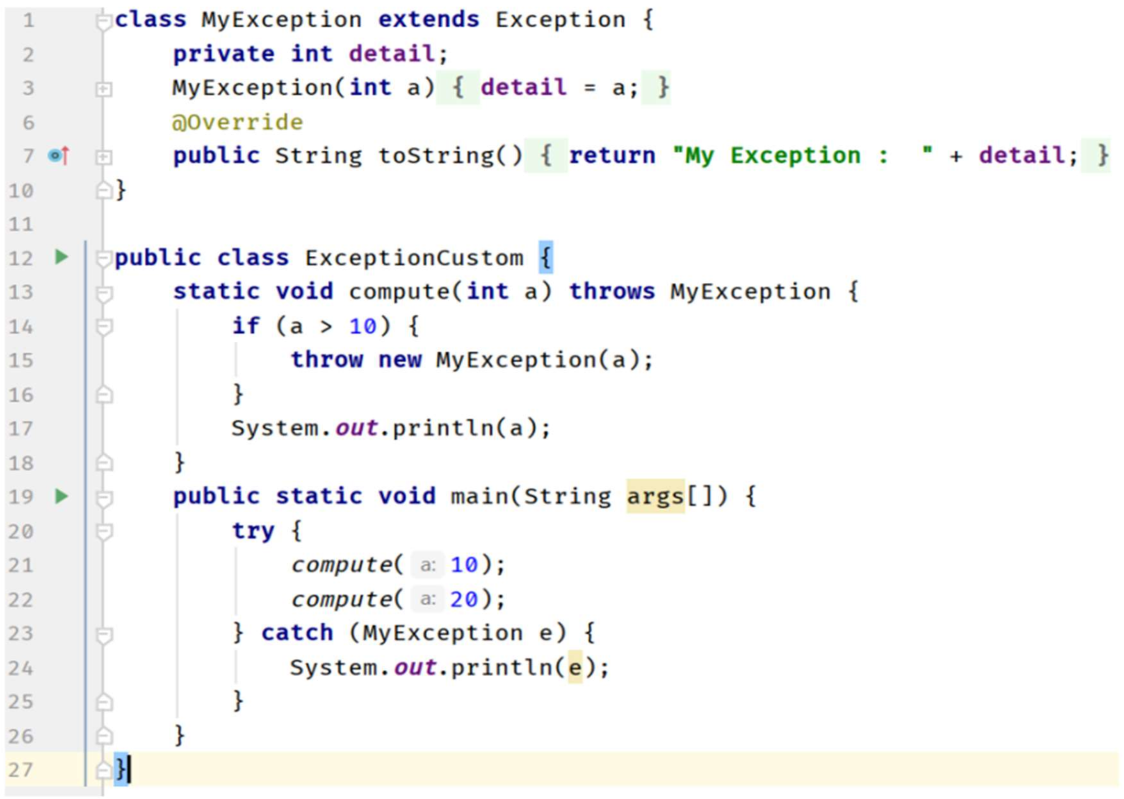
catch(ArithmeticException | Exception e) - Error

throw



```java
public class ExceptionThrow {
    public static void f() {
        try {
            throw new NullPointerException("f");
        } catch(NullPointerException e) {
            System.out.println("Inside catch of f()");
            throw e; //rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            f();
        } catch(NullPointerException e) {
            System.out.println("Inside catch of main()");
        }
    }
}
```

Custom Exceptions



```java
class MyException extends Exception {
    private int detail;
    MyException(int a) { detail = a; }
    @Override
    public String toString() { return "My Exception :  " + detail; }
}

public class ExceptionCustom {
    static void compute(int a) throws MyException {
        if (a > 10) {
            throw new MyException(a);
        }
        System.out.println(a);
    }
    public static void main(String args[]) {
        try {
            compute( a: 10);
            compute( a: 20);
        } catch (MyException e) {
            System.out.println(e);
        }
    }
}
```

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.

```
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...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Exception Class and Hierarchy

The exception class identifies the kind of error that occurred. A NumberFormatException, for example, gets thrown when a String had the wrong format and couldn't be converted into a number.

As every Java class, the exception class is part of an inheritance hierarchy. It has to extend java.lang.Exception or one of its subclasses.

The hierarchy is also used to group similar kinds of errors. An example for that is the IllegalArgumentException. It indicates that a provided method argument is invalid and it's the superclass of the NumberFormatException.

You can also implement your own exception classes by extending the Exception class or any of its subclasses. The following code snippet shows a simple example of a custom exception.

```
public class MyBusinessException extends Exception {

        private static final long serialVersionUID = 7718828512143293558L;

        public MyBusinessException() {
                super();
        }

        public MyBusinessException(String message, Throwable cause, boolean enableSuppression,
boolean writableStackTrace) {
                super(message, cause, enableSuppression, writableStackTrace);
        }

        public MyBusinessException(String message, Throwable cause) {
                super(message, cause);
        }
```

```
    public MyBusinessException(String message) {
        super(message);
    }

    public MyBusinessException(Throwable cause) {
        super(cause);
    }
}
```

Exception Object

An exception object is an instance of an exception class. It gets created and handed to the Java runtime when an exceptional event occurred that disrupted the normal flow of the application. This is called “to throw an exception” because in Java you use the keyword “throw” to hand the exception to the runtime.

JAVA REFLECTION

Java Reflection is a process of examining or modifying the run time behavior of a class at run time.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

Where it is used

The Reflection API is mainly used in:

- IDE (Integrated Development Environment) e.g., Eclipse, MyEclipse, NetBeans etc.
- Debugger
- Test Tools etc.

java.lang.Class class

The java.lang.Class class performs mainly two tasks:

- provides methods to get the metadata of a class at run time.
- provides methods to examine and change the run time behavior of a class.

Commonly used methods of Class class:

Method	**Description**
1) public String getName()	returns the class name
2) public static Class forName(String className)throws ClassNotFoundException	loads the class and returns the reference of Class class.

3) public Object newInstance()throws InstantiationException,IllegalAccessException	creates new instance.
4) public boolean isInterface()	checks if it is interface.
5) public boolean isArray()	checks if it is array.
6) public boolean isPrimitive()	checks if it is primitive.
7) public Class getSuperclass()	returns the superclass class reference.
8) public Field[] getDeclaredFields()throws SecurityException	returns the total number of fields of this class.
9) public Method[] getDeclaredMethods()throws SecurityException	returns the total number of methods of this class.
10) public Constructor[] getDeclaredConstructors()throws SecurityException	returns the total number of constructors of this class.
11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException	returns the method class instance.

How to get the object of Class class?

There are 3 ways to get the instance of Class class. They are as follows:

- forName() method of Class class
- getClass() method of Object class
- the .class syntax

1) forName() method of Class class

- is used to load the class dynamically.
- returns the instance of Class class.
- It should be used if you know the fully qualified name of class.This cannot be used for primitive types.

Let's see the simple example of forName() method.

FileName: Test.java

```
class Simple{}


public class Test{
 public static void main(String args[]) throws Exception {
  Class c=Class.forName("Simple");
  System.out.println(c.getName());
 }
}
```

Output:

```
Simple
```

2) getClass() method of Object class

It returns the instance of Class class. It should be used if you know the type. Moreover, it can be used with primitives.

FileName: Test.java

```
class Simple{}

class Test{
  void printName(Object obj){
  Class c=obj.getClass();
  System.out.println(c.getName());
  }
  public static void main(String args[]){
   Simple s=new Simple();

   Test t=new Test();
   t.printName(s);
 }
}
```

Output:

```
Simple
```

3) The .class syntax

If a type is available, but there is no instance, then it is possible to obtain a Class by appending ".class" to the name of the type. It can be used for primitive data types also.

FileName: Test.java

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

```
  Class c = boolean.class;
  System.out.println(c.getName());

  Class c2 = Test.class;
  System.out.println(c2.getName());
 }
}
```

Output:

```
boolean
Test
```

Determining the class object

The following methods of Class class are used to determine the class object:

1) public boolean isInterface(): determines if the specified Class object represents an interface type.

2) public boolean isArray(): determines if this Class object represents an array class.

3) public boolean isPrimitive(): determines if the specified Class object represents a primitive type.

Let's see the simple example of reflection API to determine the object type.

FileName: Test.java

```
class Simple{}
interface My{}

class Test{
 public static void main(String args[]){
  try{
   Class c=Class.forName("Simple");
   System.out.println(c.isInterface());

   Class c2=Class.forName("My");
   System.out.println(c2.isInterface());

  }catch(Exception e){System.out.println(e);}
```

```
        }
      }
```

Output:

```
false
true
```

Pros and Cons of Reflection

Java reflection should always be used with caution. While the reflection provides a lot of advantages, it has some disadvantages too. Let's discuss the advantages first.

Pros: Inspection of interfaces, classes, methods, and fields during runtime is possible using reflection, even without using their names during the compile time. It is also possible to call methods, instantiate a clear or to set the value of fields using reflection. It helps in the creation of Visual Development Environments and class browsers which provides aid to the developers to write the correct code.

Cons: Using reflection, one can break the principles of encapsulation. It is possible to access the private methods and fields of a class using reflection. Thus, reflection may leak important data to the outside world, which is dangerous. For example, if one access the private members of a class and sets null value to it, then the other user of the same class can get the NullReferenceException, and this behaviour is not expected.

Another demerit is the overhead in performance. Since the types in reflection are resolved dynamically, JVM (Java Virtual Machine) optimization cannot take place. Therefore, the operations performed by reflections are usually slow.

What are Interfaces?

An interface is like a class, except that it has the interface keyword, and it only declares method names. If we go back to our Vehicle analogy, any vehicle should have methods like drive(), stop(), and reverse(). But default implementation should not be added to the base class as these methods are specific to their respective classes. Such methods could be moved to an interface Driveable as follows:

```
public interface Driveable {


   void drive();

   void stop();

   void reverse();


}
```

Then a Car class would implement this interface and provide its own implementation:

```
public class Car implements Driveable {
    @Override
    public void drive() {
        System.out.println("Car is in drive mode");
    }

    @Override
    public void stop() {
        System.out.println("Car is in stop mode");
    }

    @Override
    public void reverse() {
        System.out.println("Car is in reverse mode");
    }
}
```

OBJECT CLONING IN JAVA

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

protected Object clone() **throws** CloneNotSupportedException

Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.

- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

Disadvantage of Object cloning

Following is a list of some disadvantages of clone() method:

- To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example of clone() method (Object cloning)

Let's see the simple example of object cloning

```
class Student18 implements Cloneable{
int rollno;
String name;

Student18(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws CloneNotSupportedException{
return super.clone();
}
public static void main(String args[]){
try{
Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}
}
```

Output:

```
101 amit
101 amit
```

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

JAVA INNER CLASSES (NESTED CLASSES)

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
2. Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
3. Code Optimization: It requires less code to write.

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    - Member inner class
    - Anonymous inner class
    - Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

PROXIES

A proxy class is present in java.lang package. A proxy class has certain methods which are used for creating dynamic proxy classes and instances, and all the classes created by those methods act as subclasses for this proxy class.

Class declaration:

```
public class Proxy
    extends Object
        implements Serializable
```

Fields:

protected InvocationHandler h

It handles the invocation for this proxy instance.

Constructor:

protected Proxy(InvocationHandler h)

Constructs a Proxy instance from a subclass which is typically a dynamic proxy class. The instance is automatically created with the required value of this invocation handler h.

Creating Invocation Handler:

```
// Invocation handler implementation
import java.lang.reflect.InvocationHandler;


class demoInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable
    {
        return null;
    }
}


public class GFG {
    public static void main(String[] args)
    {
        InvocationHandler h = new demoInvocationHandler();
    }
}
```

Methods

Method	**Description**
getInvocationHandler(Object proxy)	This method returns the invocation handler for the specified proxy instance.
getProxyClass(ClassLoader loader, Class<?>… interfaces)	This method returns the java.lang.Class object for a proxy class given a class loader and an array of interfaces.
isProxyClass(Class<?> cl)	This method returns true if and only if the specified class was dynamically generated to be a proxy class using the getProxyClass method or the newProxyInstance method.
newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)	This method returns an instance of a proxy class for the specified interfaces that dispatches method invocations to the specified invocation handler.

1. static InvocationHandler getInvocationHandler(Object proxy):

Returns the invocation handler for this proxy instance.

```
// getInvocationHandler() Method implementation


//Import required libraries
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.List;

// demoInvocationHandler class
class demoInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable
    {
        return null;
    }
}

// demoInterface
interface demoInterface {
}

// Driver code
public class GFG {

    public static void main(String[] args)
    {
        // Object created
        InvocationHandler h = new demoInvocationHandler();

        // ProxyClass objects stored in list
        List proxyClass = (List)Proxy.newProxyInstance(
            GFG.class.getClassLoader(),
            new Class[] { List.class },
          new demoInvocationHandler());
        System.out.println(
          Proxy.getInvocationHandler(proxyClass));
    }
}
```

Output

```
demoInvocationHandler@378fd1ac
```

2. static Class<?> getProxyClass(ClassLoader loader, Class<?>… interfaces):

Returns the java.lang.Class object for a proxy class. The proxy class are going to be defined by the required class loader and can implement all the supplied interfaces.

```
// getProxyClass() Method implementation

//Import required libraries
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.List;

// demoInvocationHandler class
class demoInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable
    {
        return null;
    }
}
```

```
// demoInterface
interface demoInterface {
}

// demo class
class demo {
}

// driver code
public class GFG {

    public static void main(String[] args)
    {

        // Object created
        InvocationHandler h = new demoInvocationHandler();

        @SuppressWarnings("deprecation")
        Class<?> proxyClass = (Class<?>)Proxy.getProxyClass(
            demo.class.getClassLoader(),
            new Class[] { demoInterface.class });
        System.out.println("Program executed successfully");
    }
}
```

Output

```
Program executed successfully
```

3. static boolean isProxyClass(Class<?> cl):

Returns true if this class was dynamically generated to be a proxy class using the getProxyClass method or the newProxyInstance method else returns false.

```
// isProxyClass() Method implementation

// Import required libraries
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.List;

// demoInvocationHandler class
class demoInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable
    {
        return null;
    }
}

// demoInterface
interface demoInterface {
}

// demo class
class demo {
}

// Driver code
public class GFG {

    public static void main(String[] args)
    {

        // Object created
        InvocationHandler h = new demoInvocationHandler();

        @SuppressWarnings("deprecation")
        Class<?> proxyClass = (Class<?>)Proxy.getProxyClass(
            demo.class.getClassLoader(),
            new Class[] { demoInterface.class });
        System.out.println(Proxy.isProxyClass(proxyClass));
    }
}
```

Output

```
true
```

4. static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h):

Returns a proxy object of a proxy class which is defined by the class loader. It implements all the required interfaces.

```
// newProxyInstance() Method implementation

// Import required libraries
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.List;

// demoInvocationHandler class
class demoInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable
    {
        return null;
    }
}

// demoInterface
interface demoInterface {
}

// driver code
public class GFG {

    public static void main(String[] args)
    {

        // Object created
        InvocationHandler h = new demoInvocationHandler();
        List proxyClass = (List)Proxy.newProxyInstance(
            GFG.class.getClassLoader(),
            new Class[] { List.class },
            new demoInvocationHandler());
        System.out.println(
            "Proxy object returned successfully");
    }
}
```

Output

```
Proxy object returned successfully
```

I/O STREAMS

Java I/O (Input and Output) is used to process the input and produce the output.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform file handling in Java by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print output and an error message to the console.

```
System.out.println("simple message");

System.err.println("error message");
```

Let's see the code to get input from console.

```
int i=System.in.read();//returns ASCII code of 1st character

System.out.println((char)i);//will print the character
```

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

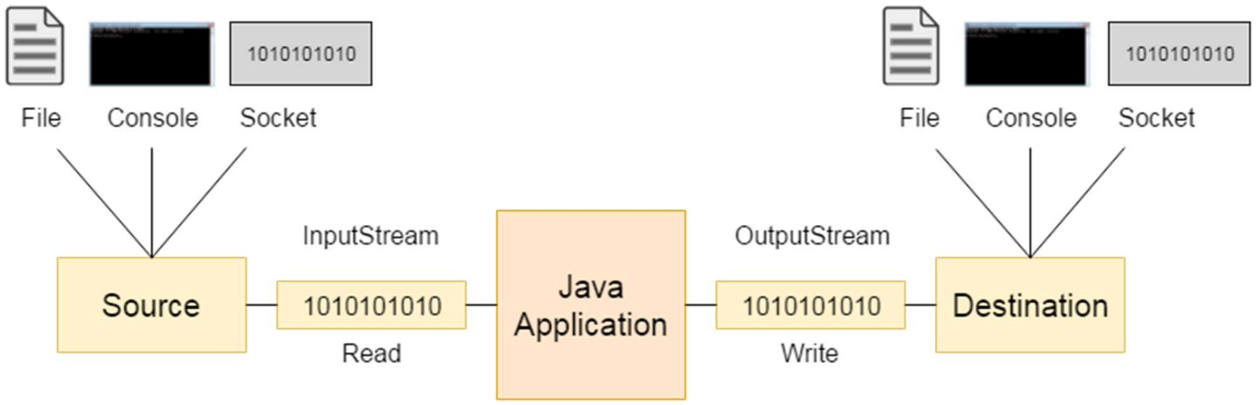
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



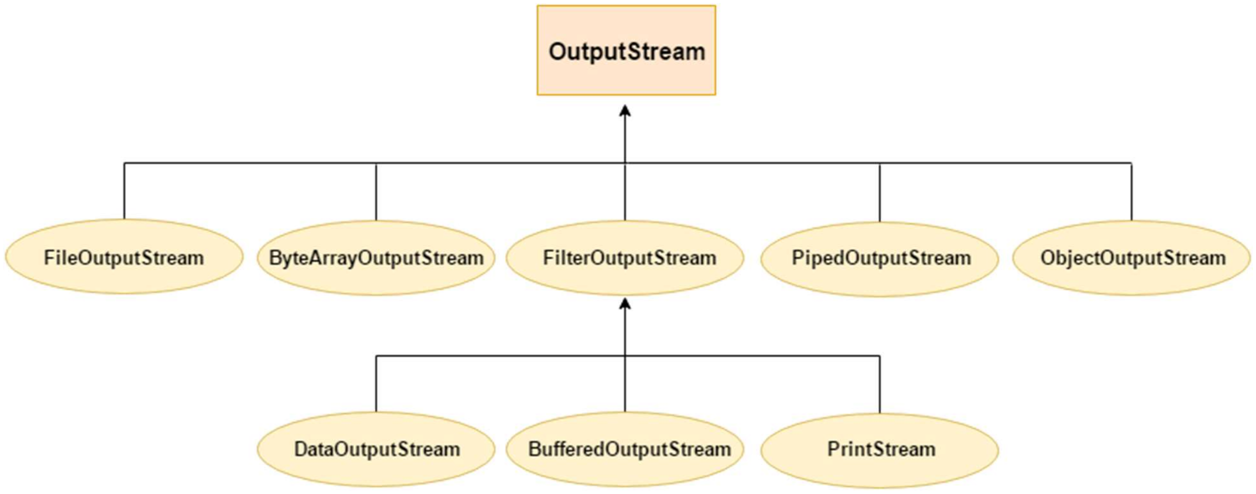
OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	**Description**
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



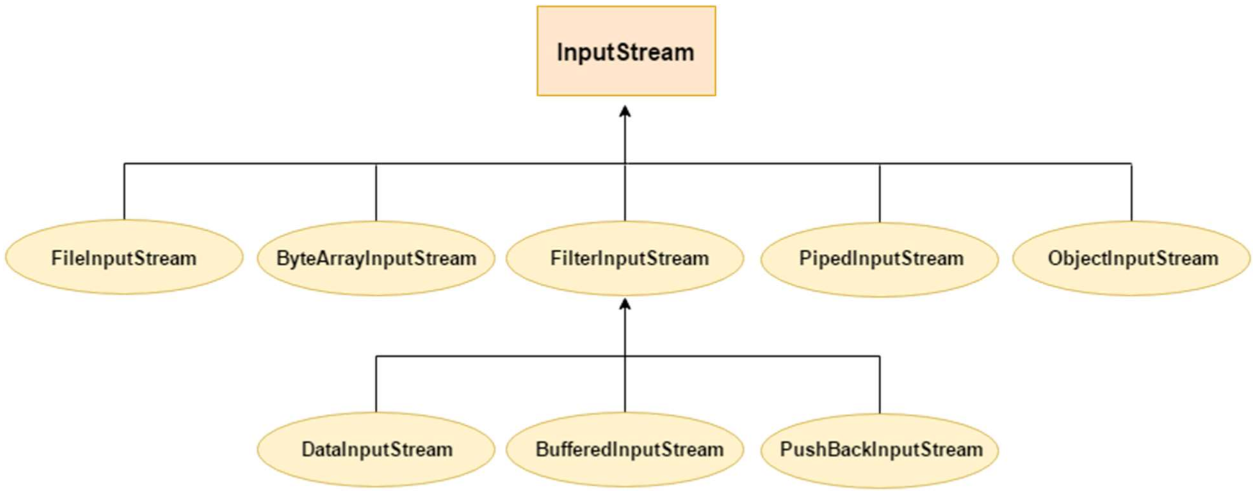
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	**Description**
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



GRAPHICS PROGRAMMING

Displaying Graphics in Applet

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.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;

public class GraphicsDemo extends Applet{

public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);

g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);

}
}
```

myapplet.html

```
<html>

<body>

<applet code="GraphicsDemo.class" width="300" height="300">

</applet>

</body>

</html>
```

JAVA JFRAME

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

Nested Class

Modifier and Type	**Class**	**Description**
protected class	JFrame.AccessibleJFrame	This class implements accessibility support for the JFrame class.

Fields

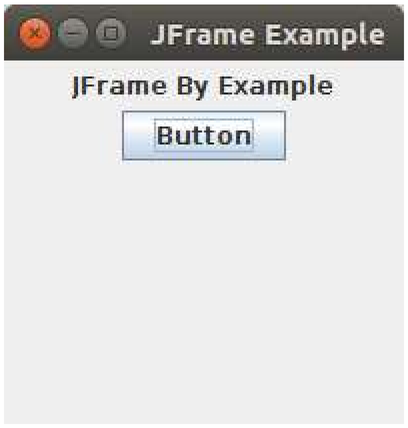
Modifier and Type	**Field**	**Description**
protected AccessibleContext	accessibleContext	The accessible context property.
static int	EXIT_ON_CLOSE	The exit application default window close operation.
protected JRootPane	rootPane	The JRootPane instance that manages the contentPane and optional menuBar for this frame, as well as the glassPane.
protected boolean	rootPaneCheckingEnabled	If true then calls to add and setLayout will be forwarded to the contentPane.

Constructors

Constructor	**Description**
JFrame()	It constructs a new frame that is initially invisible.
JFrame(GraphicsConfiguration gc)	It creates a Frame in the specified GraphicsConfiguration of a screen device and a blank title.
JFrame(String title)	It creates a new, initially invisible Frame with the specified title.
JFrame(String title, GraphicsConfiguration gc)	It creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen device.

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JFrameExample {
    public static void main(String s[]) {
        JFrame frame = new JFrame("JFrame Example");
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());
        JLabel label = new JLabel("JFrame By Example");
        JButton button = new JButton();
        button.setText("Button");
        panel.add(label);
        panel.add(button);
        frame.add(panel);
        frame.setSize(200, 300);
        frame.setLocationRelativeTo(null);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Output



Class Component

- java.lang.Object
-
    - java.awt.Component
- **All Implemented Interfaces:**

ImageObserver, MenuContainer, Serializable

Direct Known Subclasses:

Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

---

public abstract class **Component**

extends Object

implements ImageObserver, MenuContainer, Serializable

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

The Component class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components. Class Component can also be extended directly to create a lightweight component. A lightweight component is a component that is not associated with a native window. On the contrary, a heavyweight component is associated with a native window. The isLightweight() method may be used to distinguish between the two kinds of the components.

Lightweight and heavyweight components may be mixed in a single component hierarchy. However, for correct operating of such a mixed hierarchy of components, the whole hierarchy must be valid. When the hierarchy gets invalidated, like after changing the bounds of components, or adding/removing components to/from containers, the whole hierarchy must be validated afterwards by means of the Container.validate() method invoked on the top-most invalid container of the hierarchy.

Serialization

It is important to note that only AWT listeners which conform to the Serializable protocol will be saved when the object is stored. If an AWT object has listeners that aren't marked serializable, they will be dropped at writeObject time. Developers will need, as always, to consider the implications of making an object serializable. One situation to watch out for is this:

```
import java.awt.*;

import java.awt.event.*;

import java.io.Serializable;


class MyApp implements ActionListener, Serializable

{

  BigObjectThatShouldNotBeSerializedWithAButton bigOne;

  Button aButton = new Button();


  MyApp()

  {

    // Oops, now aButton has a listener with a reference

    // to bigOne!

    aButton.addActionListener(this);

  }


  public void actionPerformed(ActionEvent e)

  {

    System.out.println("Hello There");

  }

}
```

In this example, serializing aButton by itself will cause MyApp and everything it refers to to be serialized as well. The problem is that the listener is serializable by coincidence, not by design. To separate the decisions about MyApp and the ActionListener being serializable one can use a nested class, as in the following example:

```
import java.awt.*;

import java.awt.event.*;

import java.io.Serializable;


class MyApp implements java.io.Serializable

{

    BigObjectThatShouldNotBeSerializedWithAButton bigOne;

    Button aButton = new Button();


    static class MyActionListener implements ActionListener

    {

        public void actionPerformed(ActionEvent e)

        {

            System.out.println("Hello There");

        }

    }


    MyApp()

    {

        aButton.addActionListener(new MyActionListener());

    }

}
```

Note: For more information on the paint mechanisms utilitized by AWT and Swing, including information on how to write the most efficient painting code, see Painting in AWT and Swing.

JAVAFX 2D SHAPES

In some of the applications, we need to show two dimensional shapes to the user. However, JavaFX provides the flexibility to create our own 2D shapes on the screen .

There are various classes which can be used to implement 2D shapes in our application. All these classes resides in **javafx.scene.shape** package.

This package contains the classes which represents different types of 2D shapes. There are several methods in the classes which deals with the coordinates regarding 2D shape creation.

What are 2D shapes?

In general, a two dimensional shape can be defined as the geometrical figure that can be drawn on the coordinate system consist of X and Y planes. However, this is different from 3D shapes in the sense that each point of the 2D shape always consists of two coordinates (X,Y).

Using JavaFX, we can create 2D shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Cubic Curve, quad curve, Arc, etc. The class **javafx.scene.shape.Shape** is the base class for all the shape classes.

How to create 2D shapes?

As we have mentioned earlier that every shape is represented by a specific class of the package **javafx.scene.shape**. For creating a two dimensional shape, the following instructions need to be followed.

1. Instantiate the respective class : for example, **Rectangle rect = new Rectangle()**

2. Set the required properties for the class using instance setter methods: for example,

```
rect.setX(10);
   rect.setY(20);
   rect.setWidth(100);
   rect.setHeight(100);
```

3. Add class object to the Group layout: for example,

```
Group root = new Group();
   root.getChildren().add(rect);
```

The following table consists of the JavaFX shape classes along with their descriptions.

Shape	Description
Line	In general, Line is the geometrical figure which joins two (X,Y) points on 2D coordinate system. In JavaFX, **javafx.scene.shape.Line** class needs to be instantiated in order to create lines.
Rectangle	In general, Rectangle is the geometrical figure with two pairs of two equal sides and four right angles at their joint. In JavaFX, **javafx.scene.shape.Rectangle** class needs to be instantiated in order to create Rectangles.
Ellipse	In general, ellipse can be defined as a curve with two focal points. The sum of the distances to the focal points are constant from each point of the ellipse. In JavaFX. **javafx.scene.shape.Ellipse** class needs to be instantiated in order to create Ellipse.
Arc	Arc can be defined as the part of the circumference of the circle of ellipse. In JavaFX, **javafx.scene.shape.Arc** class needs to be instantiated in order to create Arcs.

Circle	A circle is the special type of Ellipse having both the focal points at the same location. In JavaFX, Circle can be created by instantiating **javafx.scene.shape.Circle** class.
Polygon	Polygon is a geometrical figure that can be created by joining the multiple Co-planner line segments. In JavaFX, **javafx.scene.shape**. Pollygon class needs to be instantiated in order to create polygon.
Cubic Curve	A Cubic curve is a curve of degree 3 in the XY plane. In Javafx, **javafx.scene.shape.CubicCurve** class needs to be instantiated in order to create Cubic Curves.
Quad Curve	A Quad Curve is a curve of degree 2 in the XY plane. In JavaFX, **javafx.scene.shape.QuadCurve** class needs to be instantiated in order to create QuadCurve.

JAVA APPLET

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

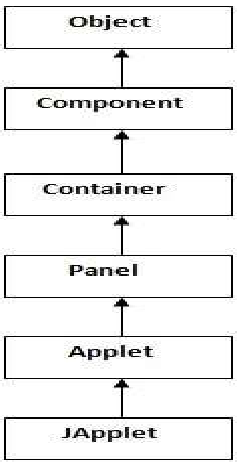
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

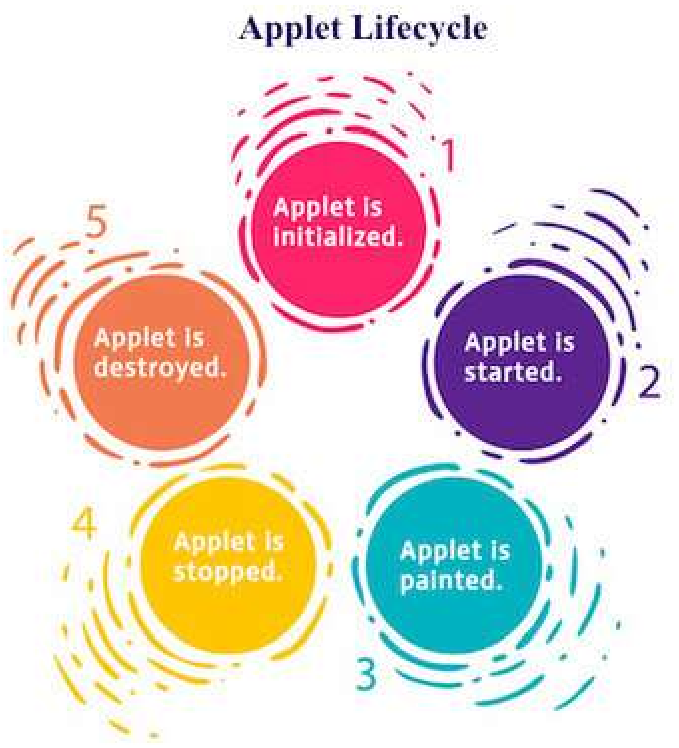
Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.



Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.

3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Who is responsible to manage the life cycle of an applet?

Java Plug-in software.

How to run an Applet?

There are two ways to run an applet

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

Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome",150,150);
}

}
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}

}
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

Java Applet Basics

Note:
java.applet package package has been deprecated in Java 9 and later versions,as applets are no longer widely used on the web.

Let's understand first how many Package does GUI support:

1. AWT(Abstract Window Toolkit)
2. Swing

Throwback of making GUI application:

Java was launched on 23-Jan-1996(JDK 1.0) and at that time it only supported CUI(Character User Interface) application. But in 1996 VB(Visual Basic) of Microsoft was preferred for GUI programming. So the Java developers in hurry(i.e within 7 days) have given the support for GUI from Operating System(OS). Now, the components like button, etc. were platform-dependent(i.e in each platform there will be different size, shape button). But they did the intersection of such components from all platforms and gave a small library which contains these intersections and it is available in AWT(Abstract Window Toolkit) technology but it doesn't have advanced features like dialogue box, etc.

Now to run Applet, java needs a browser and at that time only "Internet Explorer" was there of Microsoft but Microsoft believes in monopoly. So "SUN Micro-System"(the company which developed Java) contracted with other company known as "Netscape"(which developed Java Script) and now the "Netscape" company is also known as "Mozilla Firefox" which we all know is a browser. Now, these two companies have developed a technology called "SWING" and the benefit is that the SWING components are produced by Java itself. Therefore now it is platform-independent as well as some additional features have also been added which were not in AWT technology. So we can say that SWING is much more advanced as compared to AWT technology.

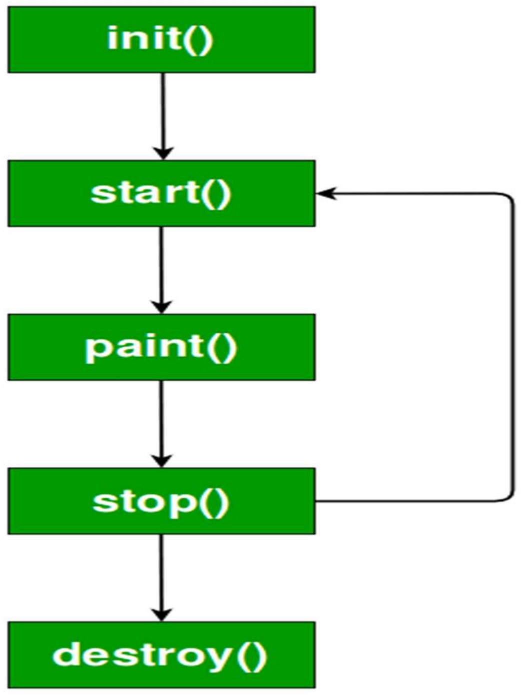
What is Applet?

An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.
Applets are used to make the website more dynamic and entertaining.

Important points :

1. All applets are sub-classes (either directly or indirectly) of *java.applet.Applet* class.
2. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
3. In general, execution of an applet does not begin at main() method.
4. Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

Life cycle of an applet :



It is important to understand the order in which the various methods shown in the above image are called. When an applet begins, the following methods are called, in this sequence:

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

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

Let's look more closely at these methods.

1. init( ) : The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.

2. start( ) : The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )** is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

3. paint( ) : The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.
paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must

redraw its output, **paint( )** is called.
The **paint( )** method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
Note: This is the only method among all the method mention above, which is parameterized. It's prototype is
public void paint(Graphics g)
where g is an object reference of class Graphic.

Now the **Question Arises:**

Q. In the prototype of paint() method, we have created an object reference without creating its object. But how is it possible to create object reference without creating its object?

Ans. Whenever we pass object reference in arguments then the object will be provided by its caller itself. In this case the caller of paint() method is browser, so it will provide an object. The same thing happens when we create a very basic program in normal Java programs. For Example:

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

Here we have created an object reference without creating its object but it still runs because it's caller,i.e JVM will provide it with an object.

4. stop( ) : The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

5. destroy( ) : The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Creating Hello World applet :
Let's begin with the HelloWorld applet :

```
// A Hello World Applet

// Save file as HelloWorld.java


import java.applet.Applet;

import java.awt.Graphics;


// HelloWorld class extends Applet

public class HelloWorld extends Applet

{
    // Overriding paint() method
    @Override
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 20, 20);
    }


}
```

Explanation:

1. The above java program begins with two import statements. The first import statement imports the Applet class from applet package. Every AWT-based(Abstract Window Toolkit) applet that you create must be a subclass (either directly or indirectly) of Applet class. The second statement import the Graphics class from AWT package.
2. The next line in the program declares the class HelloWorld. This class must be declared as public because it will be accessed by code that is outside the program. Inside HelloWorld, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet.
3. Inside **paint( )** is a call to *drawString( )*, which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

```
void drawString(String message, int x, int y)
```

Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to *drawString( )* in the applet causes the message "Hello World" to be displayed beginning at location 20,20.

Notice that the applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

Running the HelloWorld Applet :

After you enter the source code for HelloWorld.java, compile in the same way that you have been compiling java programs(using *javac* command). However, running HelloWorld with the *java* command will generate an error because it is not an application.

```
java HelloWorld


Error: Main method not found in class HelloWorld,
please define the main method as:
  public static void main(String[] args)
```

There are **two** standard ways in which you can run an applet :

1. Executing the applet within a Java-compatible web browser.
2. Using an applet viewer, such as the standard tool, applet-viewer. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Each of these methods is described next.

1. Using java enabled web browser : To execute an applet in a web browser we have to write a short HTML text file that contains a tag that loads the applet. We can use APPLET or OBJECT tag for this purpose. Using APPLET, here is the HTML file that executes HelloWorld :

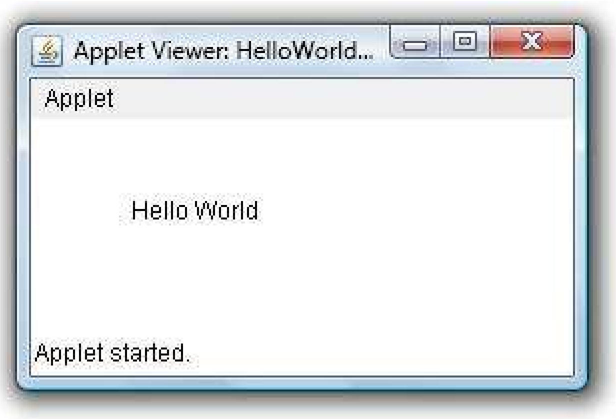
```
<applet code="HelloWorld" width=200 height=60>

</applet>
```

The width and height statements specify the dimensions of the display area used by the applet. The APPLET tag contains several other options. After you create this html file, you can use it to execute the applet.

NOTE : Chrome and Firefox no longer supports NPAPI (technology required for Java applets). Refer here

2. Using appletviewer : This is the easiest way to run an applet. To execute HelloWorld with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is saved with RunHelloWorld.html, then the following command line will run HelloWorld :

```
appletviewer RunHelloWorld.html
```



3. appletviewer with java source file : If you include a comment at the head of your Java source code file that contains the APPLET tag then your code is documented with a prototype of the necessary HTML statements, and you can run your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the HelloWorld source file looks like this :

- Java

```java
// A Hello World Applet
// Save file as HelloWorld.java


import java.applet.Applet;
import java.awt.Graphics;


/*
<applet code="HelloWorld" width=200 height=60>
</applet>
*/


// HelloWorld class extends Applet
public class HelloWorld extends Applet
{
    // Overriding paint() method
    @Override
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 20, 20);
    }


}
```

With this approach, first compile HelloWorld.java file and then simply run the below command to run applet :

```
appletviewer HelloWorld
```

To prove above mentioned point,i.e paint is called again and again.

To prove this, let's first study what is "Status Bar" in Applet:
"Status Bar" is available in the left bottom window of an applet. To use the status bar and write something in it, we use method showStatus() whose prototype is
public void showStatus(String)
By default status bar shows "Applet Started"
By default background color is white.

To prove paint() method is called again and again, here is the code:

Note: This code is with respect to Netbeans IDE.

```
//code to illustrate paint
//method gets called again
//and again


import java.applet.*;// used
//to access showStatus()
import java.awt.*;//Graphic
//class is available in this package
import java.util.Date;// used
//to access Date object
public class GFG extends Applet
{
public void paint(Graphics g)
{
Date dt = new Date();
super.showStatus("Today is" + dt);
//in this line, super keyword is
// avoidable too.
}
}
```

Note:- Here we can see that if the screen is maximized or minimized we will get an updated time. This shows that paint() is called again and again.

Features of Applets over HTML

- Displaying dynamic web pages of a web application.
- Playing sound files.
- Displaying documents
- Playing animations

Restrictions imposed on Java applets

Due to security reasons, the following restrictions are imposed on Java applets:

1. An applet cannot load libraries or define native methods.
2. An applet cannot ordinarily read or write files on the execution host.

3. An applet cannot read certain system properties.
4. An applet cannot make network connections except to the host that it came from.
5. An applet cannot start any program on the host that’s executing it.