UNIT-6 Exception Handling, I/O, abstract classes and interfaces (Chapter12,13)

Introduction to Exception

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time that disrupts the normal flow of the program's instructions.

Error: An Error indicates serious problem that a reasonable application should not try to catch.
Exception: Exception indicates conditions that a reasonable application might try to catch.

Discuss Exception and Error.

Exception:

- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an **object**.
- This object is called the **exception object.**
- It contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.
- Exceptions can be Categorized into 2 Ways:
    - Built-in Exceptions
        - Checked Exception: checked at compile-time by the compiler.
        - Unchecked Exception: checked at run-time by the compiler
    - User-Defined Exceptions: Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions which are called 'user-defined Exceptions'.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Error:

- Errors represent irrecoverable conditions such as:
    - Java virtual machine (JVM) running out of memory

    - memory leaks
    - stack overflow errors
    - library incompatibility
    - Infinite recursion, etc.
- Errors are usually beyond the control of the programmer and we should not try to handle errors.

Importance of 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.*
- An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out.
- For example:
    - If you access an array using an index that is out of bounds, you will get a runtime error with an *ArrayIndexOutOfBoundsException.*
    - If you enter a double value when your program expects an integer, you will get a runtime error with an *InputMismatchException.*

Exception handling

- *Exceptions are thrown from a method. The caller of the method can catch and handle the exception.*
- To understand exception handling, including how an exception object is created and thrown, let's begin with the example, which reads in two integers and displays their quotient.

Example-: Without Exception Handling

```
import java.util.Scanner;
public class Quotient {
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   // Prompt the user to enter two integers
   System.out.print("Enter two integers: ");
   int number1 = input.nextInt();
   int number2 = input.nextInt();
   System.out.println(number1 + " / " + number2 + " is " +
                     (number1 / number2));
    }
  }
```

```
Enter two integers: 5 2 [Enter]
5 / 2 is 2
```

```
Enter two integers: 3 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

- If you entered 0 for the second number, a runtime error would occur, because you cannot divide an integer by 0.
- (Note that a floating-point number divided by 0 does not raise an exception.)

Java enables a method to throw an exception that can be caught and handled by the caller.

Example: With Exception Handling (Try-Catch)

```
   import java.util.Scanner;
   public class QuotientWithException {
```

```
public static void main(String[] args) {
Scanner input = new Scanner(System.in);


// Prompt the user to enter two integers
System.out.print("Enter two integers: ");
int number1 = input.nextInt();
int number2 = input.nextInt();

try {
int result = (number1/number2);
System.out.println(number1 + " / " + number2 + " is "
+ result);
}
catch (ArithmeticException ex) {
System.out.println("Exception: an integer " + "cannot be divided by zero ");
}
System.out.println("Rest of the code after catch...");
}
}
```

```
Enter two integers: 6 3
6 / 3 is 2
Rest of the code after try-catch..
```

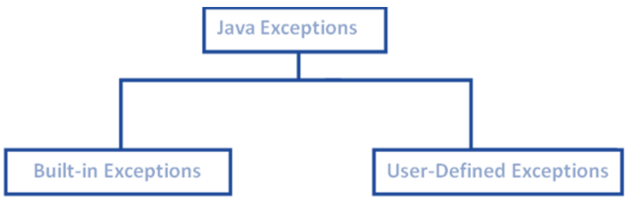
```
Enter two integers: 2 0
Exception: an integer cannot be divided by zero
Rest of the code after try-catch...
```

- The try block contains the code that is executed in normal circumstances. The exception is caught by the catch block.
- The code in the catch block is executed to handle the exception. Afterward, the statement after the catch block is executed.
- If number2 is 0, the JVM throws an exception object throw new ArithmeticException(); that is caught in **ex** object in catch block.

6.1 Exception Types

Exceptions can be Categorized into 2 Ways:

- **Built-in Exceptions**
  - Checked Exception: checked at compile-time by the compiler.
  - Unchecked Exception: checked at run-time by the compiler
- **User-Defined Exceptions:** Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions which are called 'user-defined Exceptions'.



Hierarchy of Java Exception classes

Exception class is a subclass of the built-in **Throwable** class. There is another subclass which is derived from the Throwable class i.e. **Error** as illustrated in below Figure. The error can be defined as an abnormal condition that indicates something has gone wrong with the execution of the program. These are not handled by Java programs.

There are mainly two types of exceptions in Java as follows:

- Checked exception
- Unchecked exception

1) Checked Exception:

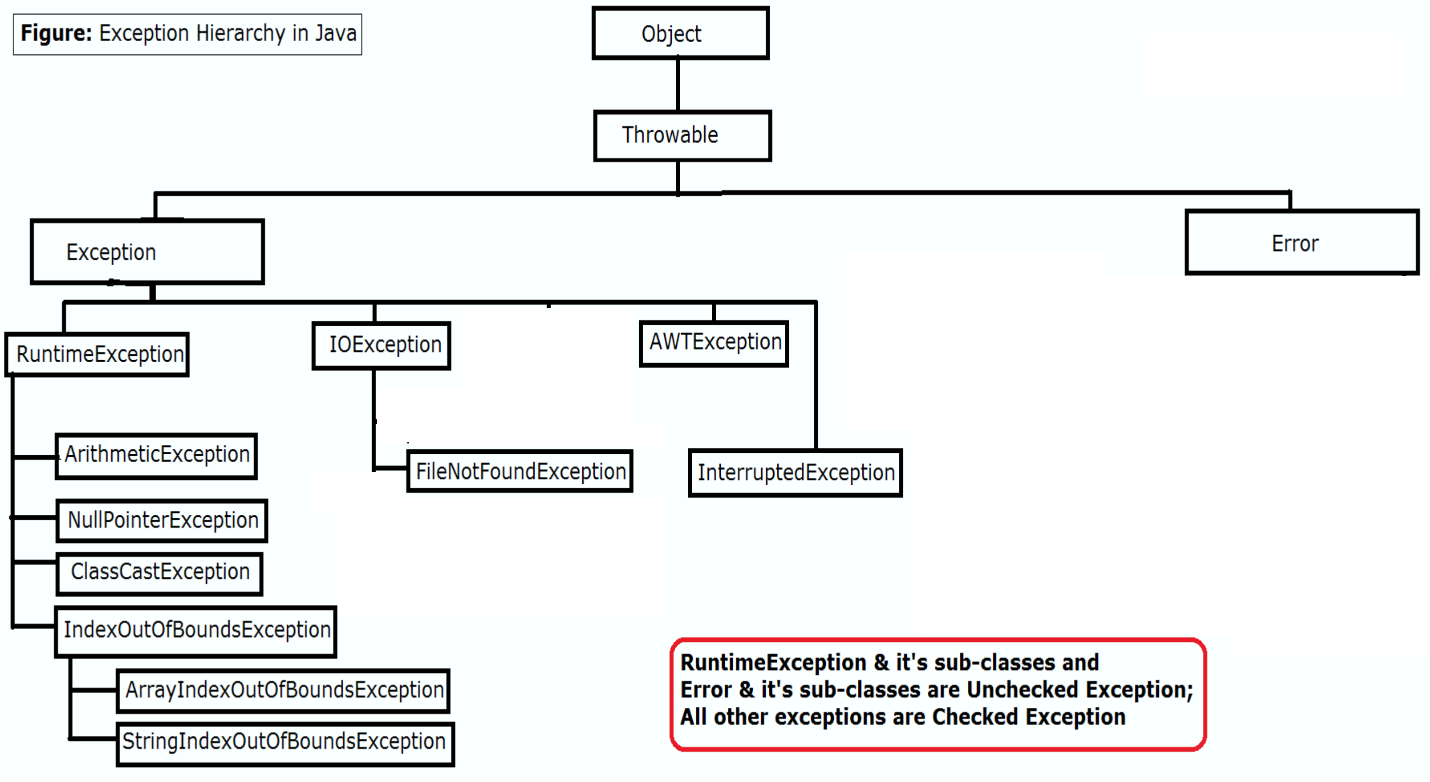
- The classes that directly inherit the **Throwable** class (**except** RuntimeException and Error) are known as checked exceptions.
- For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

- The classes that inherit the **RuntimeException** class 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.

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

Figure: Exception Hierarchy in Java


Object
Throwable
Exception
Error
RuntimeException
IOException
AWTException
ArithmeticException
FileNotFoundException
InterruptedException
NullPointerException
ClassCastException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
RuntimeException & it's sub-classes and
Error & it's sub-classes are Unchecked Exception;
All other exceptions are Checked Exception

- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.
- 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
```

Examples of Built-in Checked and Unchecked Exceptions

Runtime exceptions are generally thrown by the JVM.

Examples of Unchecked exceptions - Subclasses of RuntimeException are listed in Table

Class	Reasons for Exception
ArithmeticException	Dividing an integer by zero.
NullPointerException	Attempt to access an object through a null reference variable.
ArrayIndexOutOfBoundsException	Index to an array is out of range.
StringIndexOutOfBoundsException	Index to a character in String is out of range.
IllegalArgumentException	A method is passed an argument that is illegal or inappropriate.
NumberFormatException	Converting a character string into digit using ParseInt()
InputMismatchException	Expecting an integer in input, but a character is entered

- **ArithmeticException**

```
class NewClass
{
      public static void main(String args[])
      {
            try {
                  int n1 = 50, n2 = 0;
                  int n3 = n1/n2; // cannot divide by zero
                  System.out.println ("Result = " + n3);
            }
            catch(ArithmeticException e) {
                  System.out.println ("Can't divide a number by 0");
                  System.out.println (e);
            }
        }
}
```

```
Output:
Can't divide a number by 0
java.lang.ArithmeticException: / by zero
```

- **NullPointerException**

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

```
Output:
NullPointerException..
```

- **ArrayIndexOutOfBoundsException**

```
public static void main(String args[])
    {
        int num[] ={10,20,30,40,50};
        try{
             System.out.println(num[6]);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println(e);
           //System.out.println(Array Index is pointing to nonexistent
location);

        }
     }
```

```
Output:
java.lang.ArrayIndexOutOfBoundsException: Index 6 out of bounds for length 5
```

- **StringIndexOutOfBoundException**

```
public static void main(String args[])
    {
         try {
            String str = "This is new LJ Engineering ";
            System.out.println(str.length());
            char c = str.charAt(29); // accessing 29th element
            System.out.println(c);
        }
        catch(StringIndexOutOfBoundsException e) {
            System.out.println("StringIndexOutOfBoundsException has ocuured ");
            System.out.println(e);
        }
    }
```

Output:
```
27
StringIndexOutOfBoundsException has ocuured
java.lang.StringIndexOutOfBoundsException: String index out of range: 29
```

- **NumberFormatException**

```
public static void main(String args[])
      {
      try {
            int number = Integer.parseInt ("newLJ") ;
            System.out.println(number);
         }
         catch(NumberFormatException e) {
            System.out.println("Number format exception has ocuurred");
            System.out.println(e);
         }
      }
```

Output:
```
Number format exception has ocuurred
java.lang.NumberFormatException: For input string: "newLJ"
```

- **InputMismatchException**

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class InputMismatchExceptionDemo {

 public static void main(String[] args) {
        // create scanner class object
        Scanner sc = new Scanner(System.in);
        // use try-catch block for taking input from the user and handling exception
        try {
            System.out.print("Enter value of a to get its square value:");
            Integer a = sc.nextInt(); // we give any character value as input
            System.out.println((a*a));
        }
        catch (InputMismatchException ex) {
            System.out.println(ex);
         }
}
}
```

Output:
```
Enter value of a to get its square value: a
java.util.InputMismatchException
```

Example: IllegalArgumentException

```
public class TestCircleWithException {
 public static void main(String[] args) throws IllegalArgumentException{
    try {
    CircleWithException c1 = new CircleWithException(5);
    CircleWithException c2 = new CircleWithException(-5);
    CircleWithException c3 = new CircleWithException(0);
    }
     catch (IllegalArgumentException ex) {
     System.out.println(ex);
    }
}
}

class CircleWithException {
    private double radius;
    public CircleWithException(double newRadius) {
       setRadius(newRadius);
    }
    void setRadius(double newRadius) throws IllegalArgumentException
    {
    if (newRadius >= 0)
    {
       radius = newRadius;
    }
    else
       throw new IllegalArgumentException( "Radius cannot be negative");
    }
}
```

Output:
```
java.lang.IllegalArgumentException: Radius cannot be negative
```

Examples of Checked exceptions - subclasses of Exception are listed in Table

Class	Reasons for Exception
ClassNotFoundException	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the java command
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of IOException are InterruptedIOException, EOFException (EOF is short for End of File), and FileNotFoundException.

- **FileNotFound Exception**

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

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

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

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

```
                }
        }
}
```

```
Output:
File does not exist
java.io.FileNotFoundException: E:\MyFile.txt (The system cannot find the file specified)
```

6.2 The finally clause

Java Exception Keywords : try,catch,finally,throw,throws

try

- Java try block is used to enclose the code that might throw an exception.
- It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute.
- So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

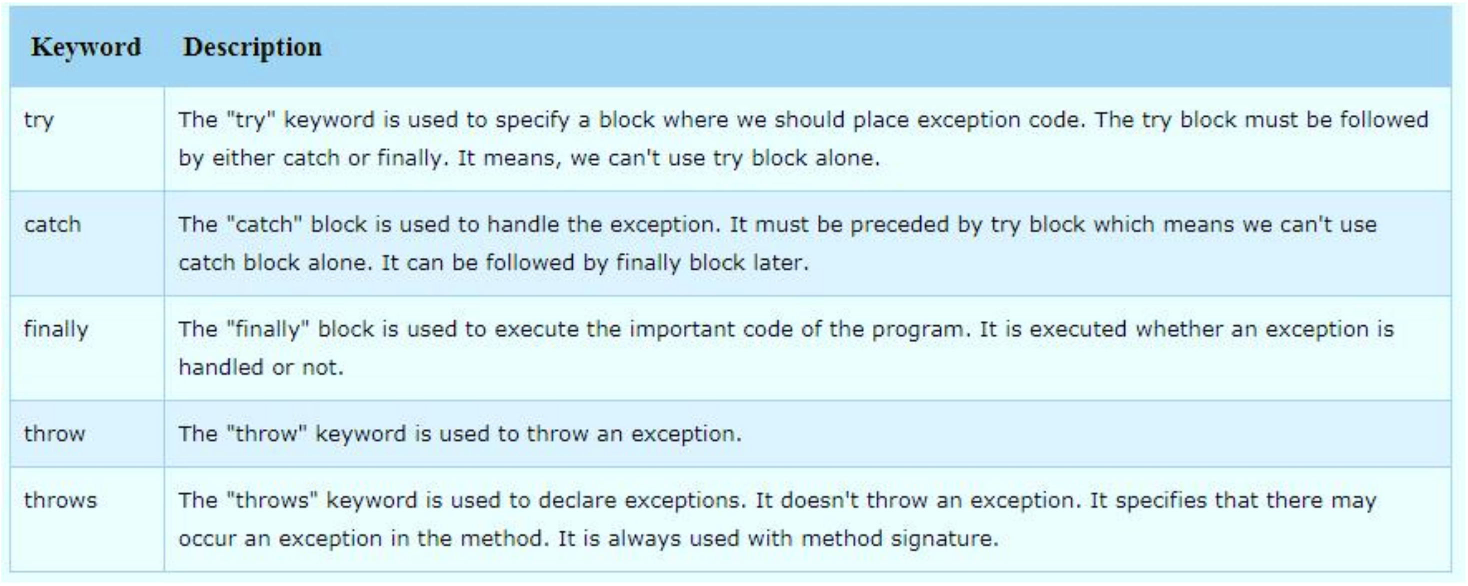
catch

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., Exception) or the generated exception type.
- However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only.
- You can use multiple catch block with a single try block.

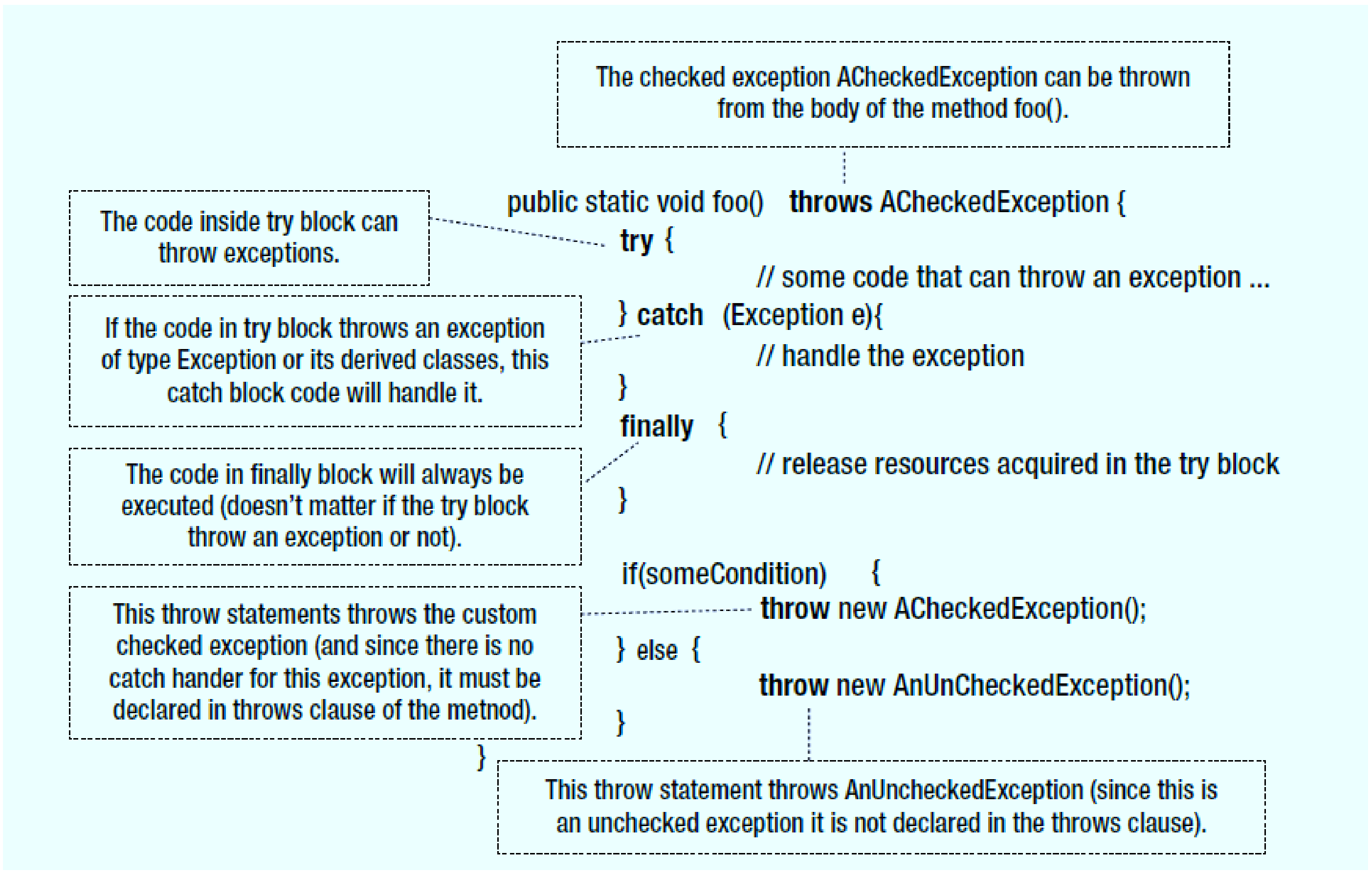
Working of Java try-catch block

- The JVM firstly checks whether the exception is handled or not.
- If exception is not handled, JVM provides a default exception ha
- ndler that performs the following tasks:
    - Prints out exception description.
    - Prints the stack trace (Hierarchy of methods where the exception occurred).
    - Causes the program to terminate.
- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

There are 5 keywords which are used in handling exceptions in Java.



Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.



```
public static void foo()   throws ACheckedException {
    try {
                // some code that can throw an exception ...
    } catch   (Exception e){
                // handle the exception
    }
    finally   {
                // release resources acquired in the try block
    }

    if(someCondition)    {
                throw new ACheckedException();
    } else {
                throw new AnUnCheckedException();
    }
}
```

The checked exception ACheckedException can be thrown from the body of the method foo().

The code inside try block can throw exceptions.

If the code in try block throws an exception of type Exception or its derived classes, this catch block code will handle it.

The code in finally block will always be executed (doesn't matter if the try block throw an exception or not).

This throw statements throws the custom checked exception (and since there is no catch hander for this exception, it must be declared in throws clause of the metnod).

This throw statement throws AnUncheckedException (since this is an unchecked exception it is not declared in the throws clause).

- *The finally clause is always executed regardless whether an exception occurred or not.*
- The **catch** block may be omitted when the **finally** clause is used.
- If the exceptions are **not** handled, statements outside try-catch-finally are **not** at all executed

<u>Example:</u> Using finally clause

```
public static void main(String[] args) {
     try {
       int[] myNumbers = {1, 2, 3};
       System.out.println(myNumbers[10]);
     } catch (Exception e) {
       System.out.println("Something went wrong.");
     } finally {
       System.out.println("The 'try catch' is finished.");
     }
   }
```

Output:
Something went wrong.
The 'try catch' is finished.

Extra Points to remember:

- In a method, there can be more than one statement that might throw an exception, so put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
- There can be more than one exception handlers. Each **catch** block is an exception handler that handles the exception of the type indicated by its argument.
- The argument, Exception Type declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.
- For each try block there can be zero (if finally block present) or more catch blocks, but **only one** final block.
- The **finally** block is optional.
- It always gets executed whether an exception occurred in try block or not.
- If an exception occurs, then it will be executed after **try and catch blocks.**
- If an exception does not occur then it will be executed after the **try** block.
- The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Example: **Using thow and throws keyword [Refer Custom Exception topic]**

Multiple catch block

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Example:

```
public class TestMulticatch {
 public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            System.out.println(1/0);
        }
        catch (ArithmeticException ex)
        {
            System.out.println("Arithmetic " + ex);
        }
        catch (NumberFormatException ex)
        {
            System.out.println("Number Format Exception " + ex);
        }
    }
}
```

```
Output:
newLJ
Number Format Exception java.lang.NumberFormatException: For input string: "newLJ"
```

```
Output:
12
Arithmetic java.lang.ArithmeticException: / by zero
```

Catching More Than One Type of Exception with One Exception Handler

- In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.
- In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException|SQLException ex) {
     throw ex;
}
```

- Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions cannot be caught in one statement.

Nested try block

- The try block within a try block is known as nested try block in java.

Why use nested try block?

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Example:

```
public class TestRethrowing {
 public static void main(String args[])
    {
        // Main try block
        try {
            int a[] = { 1, 2, 3};
            System.out.println(a[4]);

            // try-block2 inside another try block
            try {

                // performing division by zero
                int x = a[1] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
        }
}
```

System.out.println(a[5]); causes following output

```
ArrayIndexOutOfBoundsException
Element at such index does not exists
```

Commenting " System.out.println(a[5])" causes following output [caused by int x = a[2] / 0]

```
division by zero is not possible
```

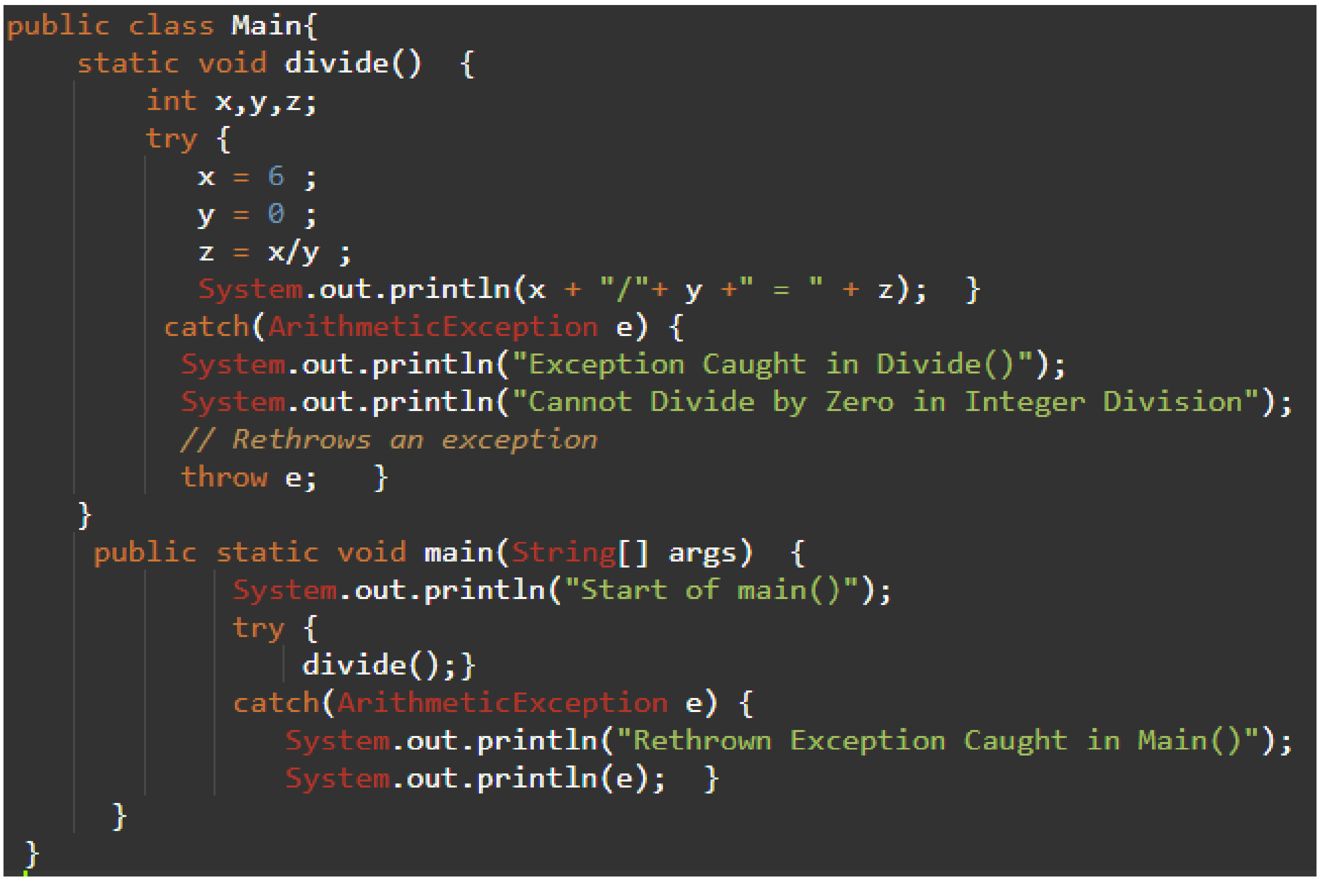
6.3 Rethrowing Exceptions

Java allows an exception handler (catch block) to rethrow the exception using **throw** keyword.

1) *if the handler cannot handle the exception or*
2) *if the handler simply wants to notify its caller about the handled exception.*

The rethrow expression causes the **originally thrown object to be rethrown.**

Example : Write program demonstrates how an exception is rethrown using the throw statement.



```
public class Main{
    static void divide()  {
        int x,y,z;
        try {
           x = 6 ;
           y = 0 ;
           z = x/y ;
           System.out.println(x + "/"+ y +" = " + z);   }
         catch(ArithmeticException e) {
          System.out.println("Exception Caught in Divide()");
          System.out.println("Cannot Divide by Zero in Integer Division");
          // Rethrows an exception
          throw e;    }
    }
     public static void main(String[] args)  {
             System.out.println("Start of main()");
             try {
                 divide();}
             catch(ArithmeticException e) {
                System.out.println("Rethrown Exception Caught in Main()");
                System.out.println(e);   }
      }
 }
```

Output:

```
Start of main()
Exception Caught in Divide()
Cannot Divide by Zero in Integer Division
Rethrown Exception Caught in Main()
```

6.4 Chained Exceptions

- Throwing an exception along with another exception forms a chained exception.
- Chained Exceptions allows to relate one exception with another exception
- In effect, the first exception causes the second exception.
- It can be very helpful to know when one exception causes another.
- You may need to throw a new exception (with additional information) along with the original exception. This is when *chained exceptions* are used.

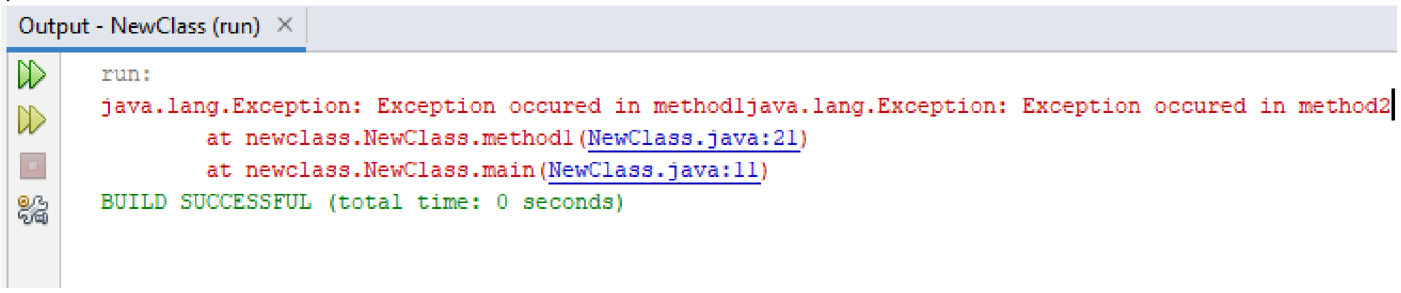
Example: Write program demonstrates Chained exceptions

```
public class ChainedException {

    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            throw new Exception("Exception occured in method1" + e);
        }
    }

    public static void method2() throws Exception {
        throw new Exception("Exception occured in method2");
    }
}
```


Output - NewClass (run)

```
run:
java.lang.Exception: Exception occured in method1java.lang.Exception: Exception occured in method2
        at newclass.NewClass.method1(NewClass.java:21)
        at newclass.NewClass.main(NewClass.java:11)
BUILD SUCCESSFUL (total time: 0 seconds)
```

6.5 Defining Custom Exception Classes

Explain user defined exceptions

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.

Example: Program to demonstrate custom exceptions

```
public class CustomException extends Exception {

     static String name[] ={"Parth", "Maulik", "Rutvi", "Dhara", "Akash"};
     static double sal[] = {10000, 12000, 5600, 900, 1100};
```

```
    // parameterized constructor
    CustomException(String str) {
        super(str);
    }
    public static void main(String[] args)
    {
        try  {
            // display the heading for the table
            System.out.println("Employee" +"\t" + "SALARY");
            // display the employee information
            for (int i = 0; i < 5 ; i++)
            {
                System.out.println(name[i] +"\t\t" + sal[i]);
                // display own exception if balance < 1000
                if (sal[i] < 10000)
                {
                    CustomException me = new CustomException("Salary is less
                                                         than 10000");
                    throw me;
                }
            }
        }
        catch (CustomException e) {
            System.out.println(e);
        }
    }
}
```

```
Employee        SALARY
Parth           10000.0
Maulik          12000.0
Rutvi           5600.0
CustomException: Salary is less than 10000
```

Explain use of throw in exception handling with example. [User Defined Exception example]

- 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.
- 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.

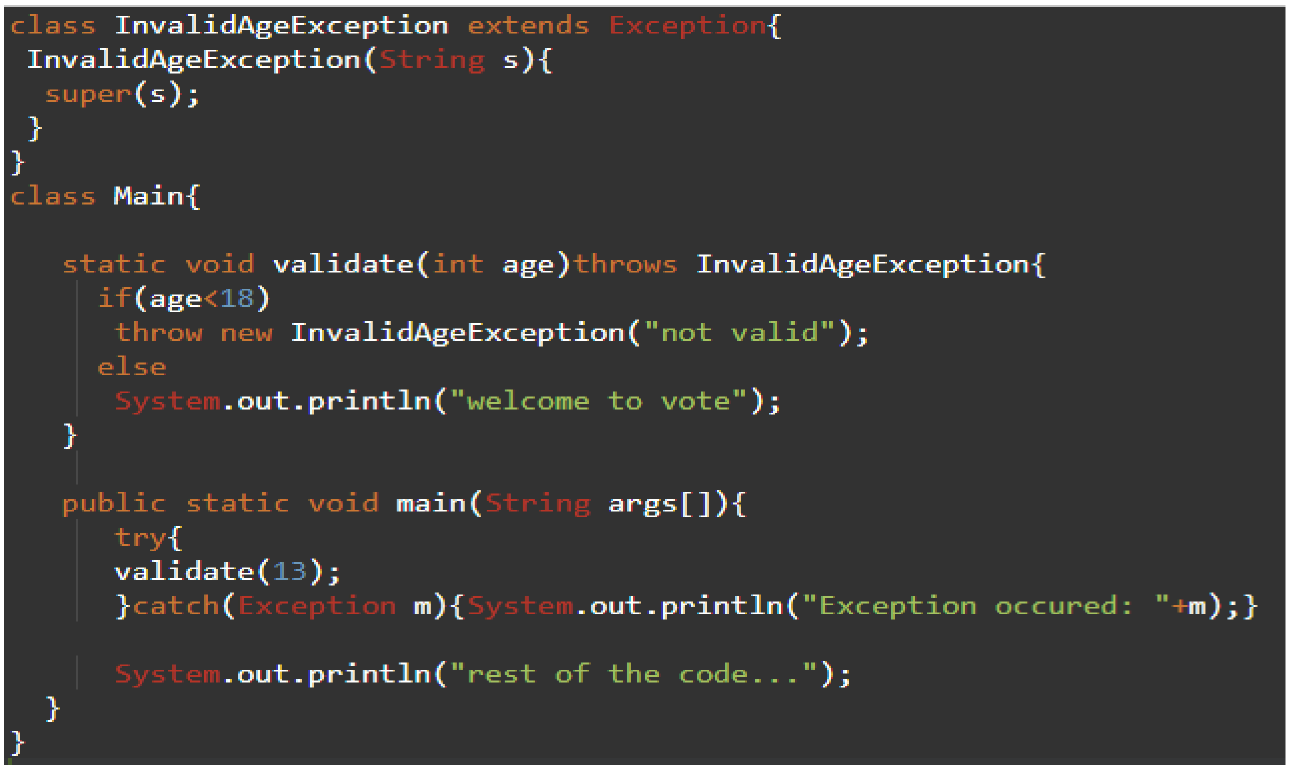
The **syntax** of the Java throw keyword is given below.
throw Instance i.e.,
```
throw new exception_class("error message");
```

Let's see the example of throw IOException.
```
throw new IOException("sorry device error");
```

- Where the **Instance** must be of type **Throwable** or **subclass** of Throwable.
- For example, Exception is the subclass of Throwable and the user-defined exceptions usually extend the Exception class.

Example: Program to demonstrate custom exceptions using **throw keyword**
- 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**.



```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
class Main{

   static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }

   public static void main(String args[]){
      try{
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}

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

```
Exception occured: InvalidAgeException: not valid
rest of the code...
```

NOTE:
Can you define a custom exception class by extending RuntimeException? Yes, but it is not a good way to go, because it makes your custom exception unchecked. It is better to make a custom exception checked, so that the compiler can force these exceptions to be caught in your program.

6.6 The File Class

- The **File** class contains the methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory.
- The File class is an abstract representation of file and directory pathname.
- A pathname can be either absolute or relative.
- An *absolute file name* (or *full name*) contains a file name with its complete path and drive letter.
- For example, **c:\book\ Welcome.java** is the absolute file name for the file **Welcome.java**
- A *relative file name* is in relation to the current working directory. The complete directory path for a relative file name is omitted.
- For example, **Welcome.java** is a relative file name.

The File class from the java.io package, allows us to work with files.
To use the File class, create an object of the class, and specify the filename or directory name:

Example

```
import java.io.File;  // Import the File class
File myObj = new File("filename.txt"); // Specify the filename
```

Constructor	Description

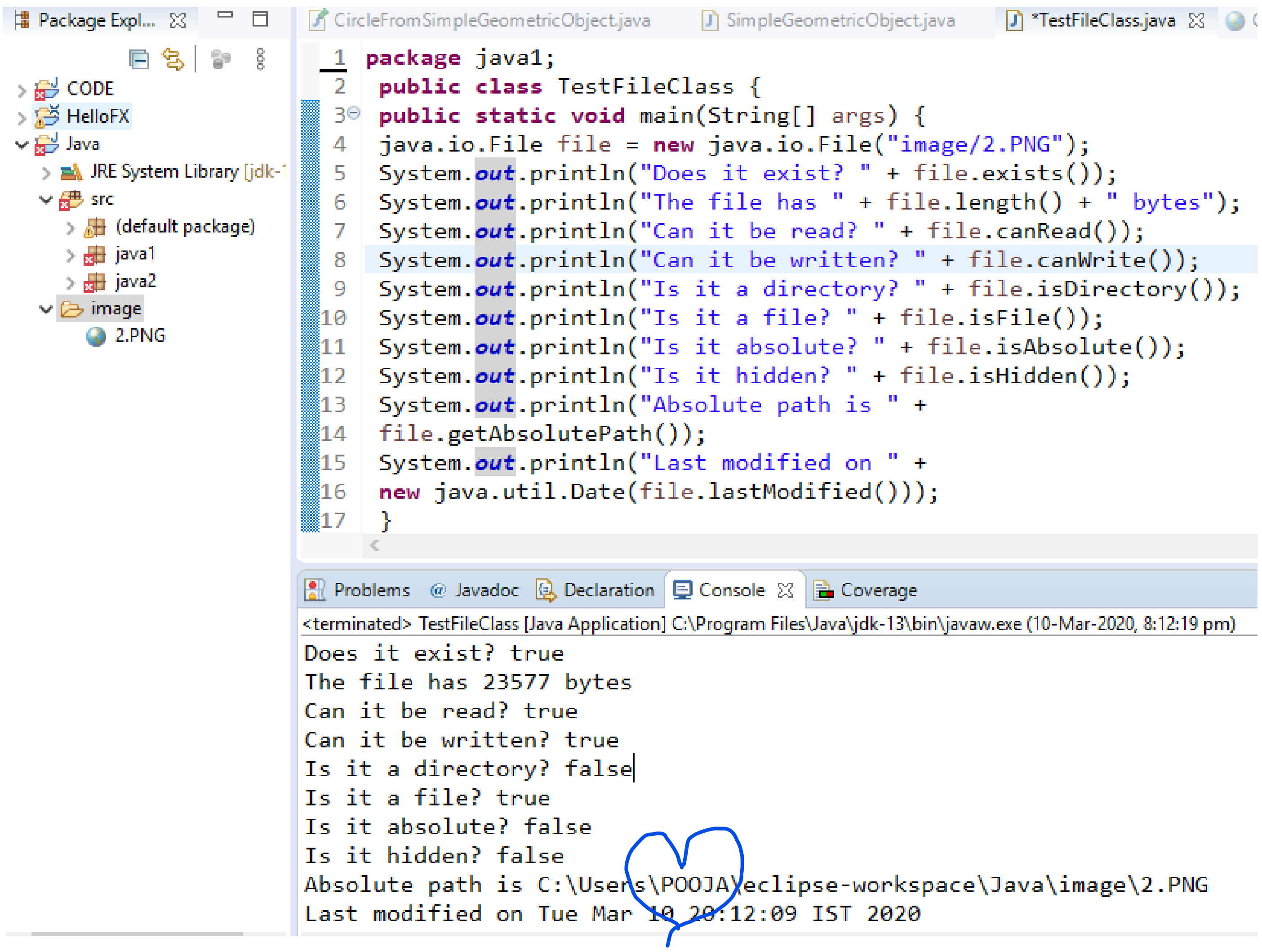
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class has many useful methods for creating and getting information about files. For example:

Method	**Type**	**Description**
createNewFile()	boolean	Creates an empty file
getName()	String	Returns the name of the file
length()	long	Returns the size of the file in bytes
getPath()	String	Returns the complete directory , file name represented by the File object
getAbsolutePath()	String	Returns the absolute pathname of the file
getParent()	String	Returns the complete parent directory of the current directory
canRead()	boolean	Tests whether the file is readable or not
canWrite()	boolean	Tests whether the file is writable or not
delete()	boolean	Deletes a file
exists()	boolean	Tests whether the file exists
mkdir()	boolean	Creates a directory
isDirectory()	boolean	Returns true if the File object represents a directory
isFile()	boolean	Returns true if the File object represents a file.
isAbsolute()	boolean	Returns true if the File object is created using an absolute path name.
isHidden()	boolean	Returns true if the file represented in the File object is hidden.
list()	String[]	Returns an array of the files in the directory
lastModified()	long	Returns the time that the file was last modified.

- The directory separator for Windows is a backslash (\). The backslash is a special character in Java and should be written as \\ in a string literal
- Constructing a File instance does not create a file on the machine. You can create a File instance for any file name regardless whether it exists or not. You can invoke the exists() method on a File instance to check whether the file exists.
- Do not use absolute file names in your program. If you use a file name such as c:\\book\\Welcome.java

Example: Methods of **File** class



```java
package java1;
public class TestFileClass {
public static void main(String[] args) {
java.io.File file = new java.io.File("image/2.PNG");
System.out.println("Does it exist? " + file.exists());
System.out.println("The file has " + file.length() + " bytes");
System.out.println("Can it be read? " + file.canRead());
System.out.println("Can it be written? " + file.canWrite());
System.out.println("Is it a directory? " + file.isDirectory());
System.out.println("Is it a file? " + file.isFile());
System.out.println("Is it absolute? " + file.isAbsolute());
System.out.println("Is it hidden? " + file.isHidden());
System.out.println("Absolute path is " +
file.getAbsolutePath());
System.out.println("Last modified on " +
new java.util.Date(file.lastModified()));
}
```

```
<terminated> TestFileClass [Java Application] C:\Program Files\Java\jdk-13\bin\javaw.exe (10-Mar-2020, 8:12:19 pm)
Does it exist? true
The file has 23577 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is C:\Users\POOJA\eclipse-workspace\Java\image\2.PNG
Last modified on Tue Mar 10 20:12:09 IST 2020
```

Example: Write program to create a file and display its path using **File** class

```java
import java.io.File;
import java.io.IOException;
public class CreateFile {
    public static void main(String[] args) {
        try {
            File file = new File("javaTestFile.txt");
            if (file.createNewFile()) {
                System.out.println("New File is created!");
                System.out.println(file.getAbsolutePath());
            } else {
                System.out.println("File already exists.");
            }
        }
  catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

```
New File is created!
E:\Netbeans Projects\SuperClasses\javaTestFile.txt
```

Example: Write program to list files of a directory using **File** class

```java
import java.io.*;
public class FileExample {
```

```
public static void main(String[] args) {
    File f=new File("/Users/Drashti/Documents");
    String filenames[]=f.list();
    for(String filename:filenames){
        System.out.println(filename);
    }
}
}
```

```
Image1.PNG
Workbook-Instructors-Version-1.pdf
Adobe
Audacity
Book1.xlsx
```

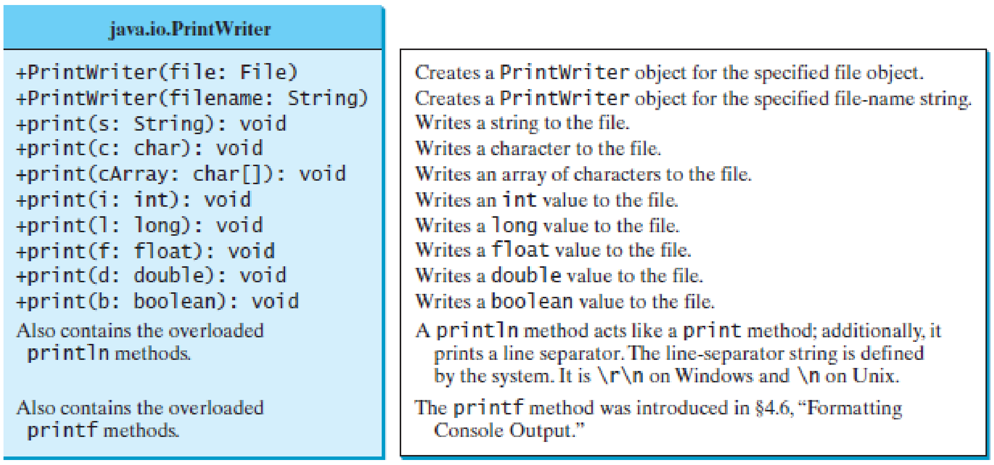
6.6 The File input and output

Use the **Scanner** *class for reading text data from a file and the* **PrintWriter** *class for writing text data to a file.*

❖ Writing Data Using PrintWriter

- The **java.io.PrintWriter** class can be used to create a file and write data to a text file.
- First, you have to create a **PrintWriter** object for a text file as follows:

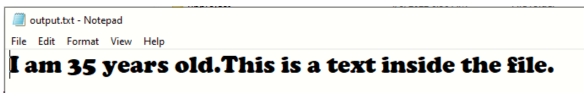
```
PrintWriter output = new PrintWriter(filename);
```



java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter object for the specified file object.
+PrintWriter(filename: String)	Creates a PrintWriter object for the specified file-name string.
+print(s: String): void	Writes a string to the file.
+print(c: char): void	Writes a character to the file.
+print(cArray: char[]): void	Writes an array of characters to the file.
+print(i: int): void	Writes an int value to the file.
+print(l: long): void	Writes a long value to the file.
+print(f: float): void	Writes a float value to the file.
+print(d: double): void	Writes a double value to the file.
+print(b: boolean): void	Writes a boolean value to the file.
Also contains the overloaded println methods.	A println method acts like a print method; additionally, it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output."

Example: Write data to file using **PrintWriter**

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class CreateFile {
    public static void main(String[] args) throws FileNotFoundException {
    String data = "This is the text inside this file.";
    try {
      PrintWriter output = new PrintWriter("output.txt");
      int age = 35;
      output.printf("I am %d years old.", age);
      output.print(data);
      output.close();
    }
    catch(FileNotFoundException e) {
      e.getStackTrace();
    }
    }
}
```



I am 35 years old.This is a text inside the file.

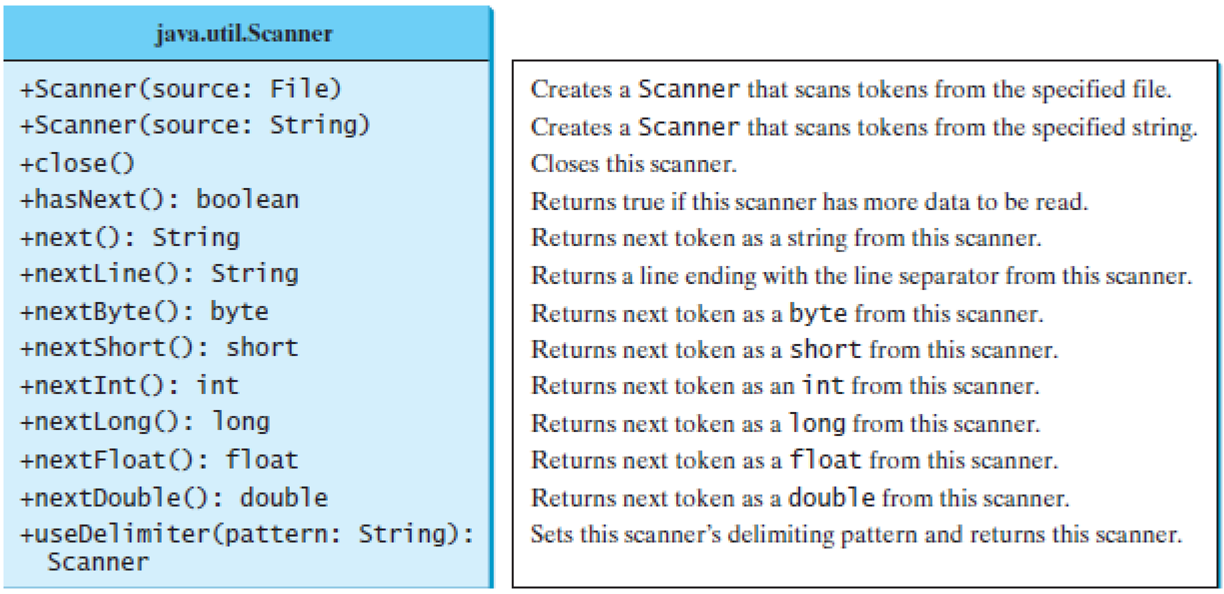
❖ Reading Data Using Scanner

The **java.util.Scanner** class was used to read strings and primitive values from the console in , Reading Input from the Console. A **Scanner** breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a **Scanner** for **System.in**, as follows:

```
Scanner input = new Scanner(System.in);
```

To read from a file, create a **Scanner** for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```



java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that scans tokens from the specified file.
+Scanner(source: String)	Creates a Scanner that scans tokens from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

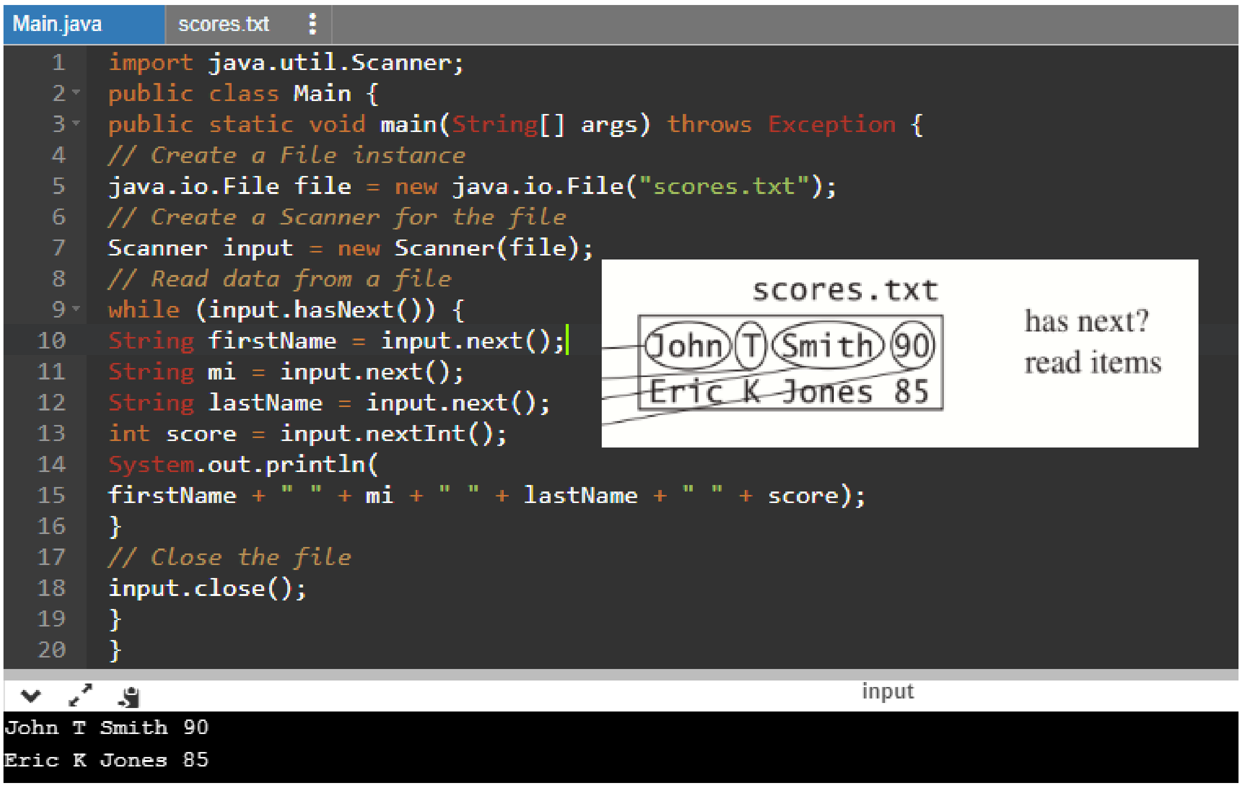
Example: Program to create an instance of **Scanner** that reads data in form the file output.txt.

```
import java.io.File;
import java.util.Scanner;
public class ReadFromFileUsingScanner
{
  public static void main(String[] args) throws Exception
  {
    // pass the path to the file as a parameter
    File file = new File("E:\\Netbeans Projects\\SuperClasses\\output.txt");
    Scanner sc = new Scanner(file);

    while (sc.hasNextLine())
      System.out.println(sc.nextLine());
  }
}
```

```
I am 35 years old.This is a text inside this file.
```

Example: Program to create an instance of **Scanner** and reads data in form of tokens from the file scores.txt.

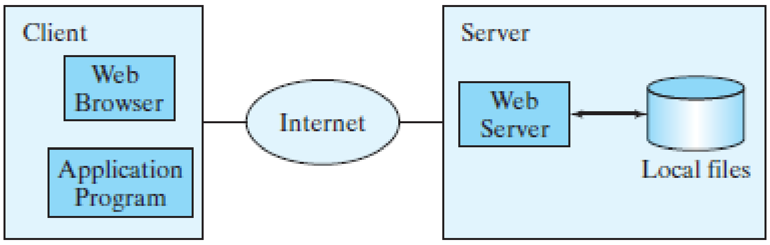


```
import java.util.Scanner;
public class Main {
public static void main(String[] args) throws Exception {
// Create a File instance
java.io.File file = new java.io.File("scores.txt");
// Create a Scanner for the file
Scanner input = new Scanner(file);
// Read data from a file
while (input.hasNext()) {
String firstName = input.next();
String mi = input.next();
String lastName = input.next();
int score = input.nextInt();
System.out.println(
firstName + " " + mi + " " + lastName + " " + score);
}
// Close the file
input.close();
}
}
```

```
John T Smith 90
Eric K Jones 85
```

6.7 Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.



The client retrieves files from a Web server.

For an application program to read data from a URL, you first need to create a **URL** object using the **java.net.URL** class with this constructor:
A **MalformedURLException** is thrown if the URL string has a syntax error.

```
public URL(String spec) throws MalformedURLException
```

Example: Write program to read a file data from web using a URL

```
import java.util.Scanner;
public class ReadFromFileUsingScanner
{
  public static void main(String[] args) {
  System.out.print("Enter a URL: ");
  String URLString = new Scanner(System.in).next();

  try {
        java.net.URL url = new java.net.URL(URLString);
        int count = 0;
        Scanner input = new Scanner(url.openStream());
        while (input.hasNext()) {
        String line = input.nextLine();
        count += line.length()
        }

       System.out.println("The file size is " + count + " characters");
 }
 catch (java.net.MalformedURLException ex) {
 System.out.println("Invalid URL");
 }
 catch (java.io.IOException ex) {
 System.out.println("I/O Errors: no such file");
 }
 }
}
```

```
Enter a URL: https://www.ljku.edu.in/
The file size is 69314 characters
```

6.8 Abstract Classes

An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in concrete subclasses.

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Ways to achieve Abstraction
There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

The abstract keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have **both** abstract and regular methods:
in Java, a separate keyword ***abstract*** is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
     int color;

     // An abstract function (like a pure virtual function in C++)
     abstract void draw();
}
```

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an **abstract** keyword.
- It **cannot** be instantiated.
- It can have **abstract** and non-abstract(**concrete**) methods.
- **Abstract** methods are **nonstatic** and **must** be **overridden** by subclasses.
- It **can** have **constructors** and **static** methods also.
- It **can** have **final** methods which will force the subclass not to change the body of the method.
- Keywords **final/static cannot** be used with an **abstract** method. It is an **illegal** combination.

Example of abstract class
abstract class A{}

Abstract Method in Java
A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method
abstract void printStatus();//no method body

1) In Java, an **instance** of an abstract class **cannot** be created, we can have **references of abstract class** type though.

```
abstract class Base {
      abstract void method();
}
class Derived extends Base {
      void method() { System.out.println("Derived class method called"); }
}
class Main {
      public static void main(String args[]) {

            // Uncommenting the following line will cause compiler error as the
            // line tries to create an instance of abstract class.
            // Base b = new Base();

            // We can have reference variable of Base type.
            Base b = new Derived();
            b.method();
      }
}
```

```
Derived class method called
```

NOTE:
Suppose that you want to manipulate an account, but you're not sure what kind of account. Maybe it's a SavingsAccount, or maybe it's a CheckingAccount, a MoneyMarketAccount, a TreasuryAccount, etc. Having a reference of type Account means that you can have a single variable that stores a reference to an object of any of those types, which makes it possible to write code that works on "accounts in general" without duplicating it for each of these types of accounts.

2) An abstract class **can** contain **constructors** in Java. And a constructor of abstract class is called when an instance of an inherited class is created. For example, the following is a valid Java program.

```
// An abstract class with constructor
abstract class Base {
      Base() { System.out.println("Base Constructor Called"); }
      abstract void fun();
}
```

```
class Derived extends Base {
      Derived() { System.out.println("Derived Constructor Called"); }
      void fun() { System.out.println("Derived fun() called"); }
}
class Main {
      public static void main(String args[]) {
      Derived d = new Derived();
      }
}
```

```
Base Constructor Called
Derived Constructor Called
```

3) In Java, we can have an abstract class **without any abstract method**. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
      void method () { System.out.println("Base class concrete method called"); }
}

class Derived extends Base { }

class Main {
      public static void main(String args[]) {
            Derived d = new Derived();
            d. method ();
      }
}
```

```
Base class concrete method called
```

4) Abstract classes can also have **final methods** (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
abstract class Base {
      final void fun() { System.out.println("Base fun() called"); }
}
class Derived extends Base {
    // method fun() cannot be defined here as it is final in base class
}
class Main {
      public static void main(String args[]) {
      Base b = new Derived();
      b.fun();
      }
}
```

```
Base fun() called
```

Example: Write program to use abstract **draw**() method of abstract class Shape to draw a circle and a rectangle.

```
abstract class Shape{
abstract void draw();
}

class Rectangle extends Shape{
@Override
void draw(){System.out.println("drawing rectangle");}
}
```

```
class Circle extends Shape{
@Override
void draw(){System.out.println("drawing circle");}
}

class Main{
public static void main(String args[]){
Shape s1=new Circle();
Shape s2= new Rectangle();
s1.draw();
s2.draw();
}
}
```

```
drawing circle
drawing rectangle
```

Example: Write program to demonstrate, that an **abstract** class can have data fields, abstract method, final concrete method and a constructor.

```
abstract class Bike{
   double speed=80.5;
   Bike(){System.out.println("bike is created");}
   abstract void run();
   final void changeGear(){System.out.println("gear changed");}
 }

//Creating a Child class which inherits Abstract class
 class Honda extends Bike{
 void run(){System.out.println("running a Honda bike..with speed " + speed);}
 }

//Creating a Child class which inherits Abstract class
class Hero extends Bike{
 void run(){System.out.println("running a Hero bike..with speed " + speed);}
 }

//Creating a Test class which calls abstract and non-abstract methods
 class Main{
 public static void main(String args[]){
  Bike bike1 = new Honda();
  bike1.run();
  bike1.changeGear();
  Bike bike2 = new Hero();
  bike2.run();
  bike1.changeGear();
 }
}
```

```
bike is created
running a Honda bike..with speed 80.5
gear changed
bike is created
running a Hero bike..with speed 80.5
gear changed
```

Interesting Points about Abstract Classes

The following points about abstract classes are worth noting:

- An abstract method cannot be contained in a nonabstract class.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract.
- Abstract methods are nonstatic because static prevents overriding.
- An abstract class cannot be instantiated using the **new** operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.
- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods.
- A subclass can be abstract even if its superclass is concrete. For example, the **Object** class is concrete, but its subclasses, such as **GeometricObject**, may be abstract.
- You cannot create an instance from an abstract class using the **new** operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the **GeometricObject** type, is correct.

  GeometricObject[] objects = **new** GeometricObject[**10**];

  You can then create an instance of **GeometricObject** and assign its reference to the array like this:

  objects[**0**] = **new** Circle();

6.9 Interface

An interface is a class-like construct that contains only ***constants*** *and* ***abstract methods****.*

An **interface in Java** is a blueprint of a class. It has **static constants** and **abstract methods**.

- **The interface in Java is *a mechanism to achieve total abstraction*.**
- There can be only abstract methods in the Java interface, not method body.
- **It is used to achieve abstraction and multiple inheritance in Java.**
- We can't create an instance of the interface (interface can't be instantiated) but we can make the reference of it that refers to the Object of its implementing class.
- A class can **implement** more than one interface.
- An interface can extend to another interface or interface (more than one interface).
- A class that implements the interface must implement all the methods in the interface.
- Interface **cannot** have a **constructor** in Java.
- All the methods are public and abstract and all the fields are public, static, and final by default.

❖ How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

To distinguish an interface from a class, Java uses the following syntax to define an interface:

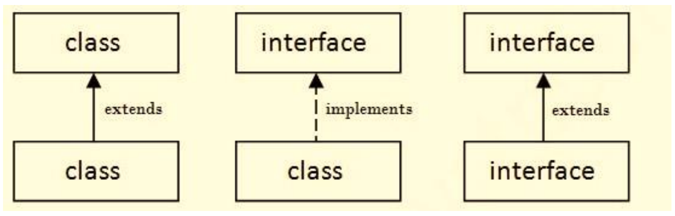
```
modifier interface InterfaceName {
/** Constant declarations */
/** Abstract method signatures */
}
```

Here is an example of an interface:

```
public interface Edible {
public static final value; //constant
public abstract String howToEat(); //abstract method
}
```

❖ ***The relationship between classes and interfaces***

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



The major differences between a class and an interface are:

S. No.	Class	Interface
1.	In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
2.	Class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
3.	The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	1) Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	2) Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	3) Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	4) Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	5) The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	6) An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	7) An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	8) Members of a Java interface are public by default.

9)Example:
public abstract class Shape{
public abstract void draw();
}
	9) Example:
	public interface Drawable{
	void draw();
	}

Note: Nested interfaces and classes can have all access modifiers.
Note: We cannot declare class/interface with private or protected access modifiers.
Java Interface also **represents the IS-A relationship**.

Since Java 8, we can have **default and static methods** in an interface.
default: Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation,

we will give a **default body for the newly added functions**. Then the old codes will still work.we can now define **static** methods in interfaces that can be called independently without an object. Note: these methods are **not inherited**.

Since Java 9, we can have **private methods** in an interface.

Example: Simple Interface Example

```
// create an interface
interface Language {
  void getName(String name);
}
// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}
class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

```
Programming Language: Java
```

Example: Write program to demonstrate **interface** Bank to get rateOfInterest() of different banks.

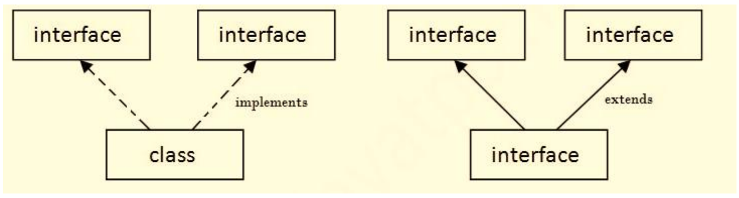
```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}

class Main{
public static void main(String[] args){
Bank b1=new SBI();
System.out.println("SBI - Rate Of Interest: "+b1.rateOfInterest());
Bank b2=new PNB();
System.out.println("PNB - Rate Of Interest: "+b2.rateOfInterest());
}}
```

```
SBI - Rate of Interest: 9.15
PNB - Rate of Interest: 9.7
```

❖ Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Example:

```
interface PersonB
{
    //any number of methods can be declared in the interface
    public void walk();
    public void run();
}
//multiple inheritance achieved
class Person implements PersonA,PersonB
{
    //overidden methods
    public void walk()
    {
        System.out.println("person is walking ");
    }
         public void run()
         {
             System.out.println("Person is running ");
         }
}
class MainClass
{
     public static void main (String args[])
     {
       Person object = new Person();

       object.walk();
       object.run();
     }
}
```

```
person is walking
Person is running
```

❖ **Interface inheritance – Interface Extending another Interface**

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Example:

```
// Java program to demonstrate inheritance in interfaces.
interface intf1
{
   void studentName();
}
interface intf2 extends intf1
{
   void studentInstitute();
}
// class implements both interfaces and provides implementation to the method.
class sample implements intf2
{
   @Override
   public void studentName()
   {
     System.out.println("My Name");
   }
   @Override
   public void studentInstitute()
   {
     System.out.println("NLJIET");
   }
}
class MainClass
{
    public static void main (String[] args)
   {
     sample ob1 = new sample();
     ob1.studentName();
     ob1.studentInstitute();
   }
}
```

```
My Name
NLJIET
```

❖ **Extending Multiple Interfaces - An interface can extend multiple interfaces**

```
interface A {
   ...
}
interface B {
   ...
}

interface C extends A, B {
   ...
}
```

Example:

Write a program that illustrates interface inheritance. Interface P is extended by P1and P2. Interface P12 inherits from both P1 and P2.Each interface declares one constant and one method. class Q implements P12.Instantiate Q and invoke each of its methods. Each method displays one of the constants

```
interface P
{
    public static final int p=100;
    void methodP();
}
interface P1 extends P
{
    public static final int p1=10;
    void methodP1();
}
interface P2 extends P
{
    public static final int p2=20;
    void methodP2();
}
interface P12 extends P1,P2
{
    public static final int p12=30;
    void methodP12();
}
class Q implements P12
{
    public void methodP12()
    {
        System.out.println("P12 Class Method Constant : "+p12);
    }
    public void methodP1()
    {
        System.out.println("P1 Class Method Constant : "+p1);
    }
    public void methodP2()
    {
        System.out.println("P2 Class Method Constant : "+p2);
    }
    public void methodP()
    {
        System.out.println("P Class Method Constant : "+p);
    }
}


class MainClass
{
    public static void main(String args[])
    {
        Q obj=new Q();
        obj.methodP12();
        obj.methodP1();
        obj.methodP2();
        obj.methodP();
    }
}
```

```
P12 Class Method Constant : 30
P1 Class Method Constant : 10
P2 Class Method Constant : 20
P Class Method Constant : 100
```

6.9 Cloneable Interface

Marker interface in Java: It is an empty interface (no field or methods). Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

```
public interface Cloneable

{

  // nothing here

}
```

Cloneable interface :

- Cloneable interface is present in java.lang package.
- There is a method clone() in Object class.
- A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class.
- Invoking Object's clone method on an instance of the class that does not implement the Cloneable interface results in an exception CloneNotSupportedException being thrown.
- By convention, classes that implement this interface should override Object.clone() method.

Example:

```
import java.lang.Cloneable;
// By implementing Cloneable interface we make sure that instances of class Clones
// can be cloned.
class Clones implements Cloneable
{
	String s;
	// A class constructor
	public Clones(String s)
	{
		this.s = s;
	}
	// Overriding clone() method by simply calling Object class clone() method.
	@Override
	protected Object clone() throws CloneNotSupportedException
	{
		return super.clone();
	}
}
public class Main
{
	public static void main(String[] args) throws CloneNotSupportedException
	{
		Clones a = new Clones("GTU NLJIET");
		// cloning 'a' and holding new cloned object reference in b

		// down-casting as clone() return type is Object
		Clones b = (Clones)a.clone();
		System.out.println(b.s);
	}
}
```

6.10 Comparable Interface

Java provides two interfaces to sort objects using data members of the class:

1. Comparable
2. Comparator (Covered in Collections and Generics unit)

Both of these are **generic interfaces**

Comparable Interface

Java Comparable interface is used to sort the objects of the user-defined class.
public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

if you want to sort the objects of custom class then you need to implement the Comparable interface in our custom class.

This interface has only one method which is:

```
public abstract int compareTo(T obj)
```

Since this method is abstract, you must implement this method in your class if you implement the Comparable interface.

- A comparable object is capable of comparing itself with another object.
- The class itself must implements the java.lang.Comparable interface to compare its instances.
- Consider a Person class that has members like, name, age.
- Suppose we wish to sort a list of Person based on year of release.
- We can implement the Comparable interface with the Person class, and we override the method compareTo() of Comparable interface.

Example:

```
import java.util.ArrayList;
import java.util.Collections;

class Person implements Comparable<Person>{
String name;
int age;
Person(String name,int age){
this.name=name;
this.age=age;
}

@Override
public int compareTo(Person st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}

public class Main{
public static void main(String args[]){
ArrayList<Person> al=new ArrayList<>();
al.add(new Person("Jay",23));
al.add(new Person("Ajay",27));
al.add(new Person("Vijay",21));

Collections.sort(al);
for(Person p:al){
System.out.println(p.name+" "+p.age);
}
}
}
```

```
Vijay 21
Jay 23
Ajay 27
```