Object Oriented Programming Through Java

Lecture Notes

Unit-III

Prepared by

Mr. P.Bhaskar

Assistant Professor (Dept of CSE)

ADITYA COLLEGE OF ENGINEERING

MADANAPALLI

Unit-III

Exception handling, Stream based I/O

Exception handling - Fundamentals, Exception types, Uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws and finally, built-in exceptions, creating own exceptionsubclasses.

Stream based I/O (java.io) – The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and Writing Files, Random access file operations,

The Console class, Serialization, Enumerations, Autoboxing , Generics.

CONTENTS

3.1 Exceptions:

- An exception is **a problem** or **an abnormal condition** that arises during the program execution (**at run time)**.
- In other words an exception is a **run time error**.
- When an Exception occurs the normal flow of the program is **interrupts the flow** and **the program terminates abnormally**, therefore these exceptions are to be handled.

An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Errors

No matter how good a program we write it may cause the following types of errors.

1. Syntax Errors or 2. Logical Errors or 3. Runtime Errors

Syntax Errors:

It is an error which occurs when the rules of language are violated .
If we are not following the rules, those cause the error. It can be easily detected and corrected by compilers.

Eg: Statement missing **semicolon;** , Missing/expected **„ }"** , Un-defined symbol **varname** etc.

Logical Error:

It is an error caused due to manual mistake in the logical expressions. These errors cannot be detected by the compiler.

Eg. Current_balance = old_balance*cash_deposit

Runtime Error:

It is an Error, which occurs after executing the program at runtime.

Eg. Divide by zero.
File not found
No disc in the drive etc.

Exception Handling

- Exception Handling is the technique to handle runtime malfunctions.
- We need to handle such exceptions to prevent unexpected termination of program.
- The term exception means exceptional condition, it is a problem that may arise during the execution of program.
- A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

Usage:

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally interrupts the normal flow of the application; that is why we need to handle exceptions.

Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;//exception occurs
4. statement 4;
5. statement 5;

Suppose there are 5 statements in a Java program and an exception occurs at statement 3; the rest of the code will not be executed, i.e., statements 4 to 5 will not be executed.

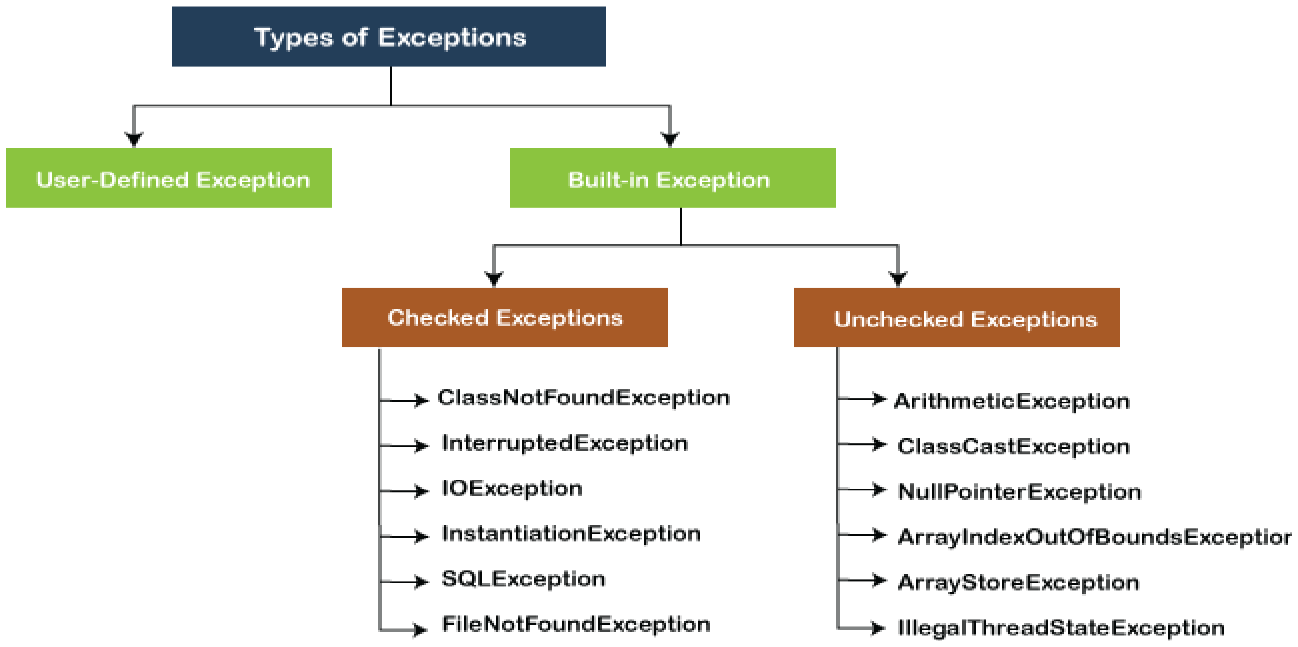
However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java

3.2 Types of Exception in Java

Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as exceptions.
In this section, we will focus on the types of exceptions in Java and the differences between the two.

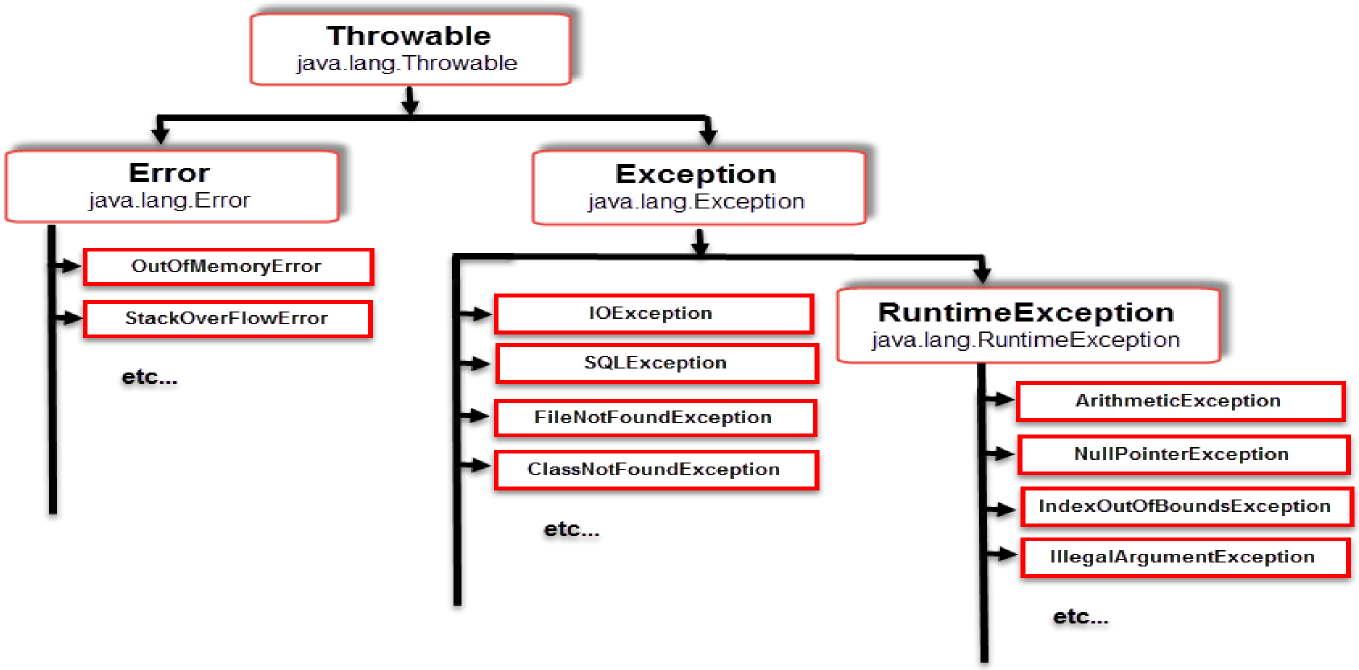
Exceptions can be categorized into two ways:

1. Built-in Exceptions
   - Checked Exception
   - Unchecked Exception
2. User-Defined Exceptions



3.8 Built-in Exceptions (checked & Unchecked exceptions)

- Exception that are already available in Java libraries are referred to as built-in exception.
- These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., checked exceptions and unchecked
- All the built-in exception classes in Java were defined a package java.lang.



The hierarchy of Java Exception classes

3.8 Checked Exceptions

- Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- The compiler ensures whether the programmer handles the exception or not.
- The programmer must to handle the exception; otherwise, the system has shown a compilation error.

Some list of checked exceptions

S. No.	Exception Class with Description
1	**ClassNotFoundException** It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath
2	**FileNotFoundException** It is thrown when the Java Virtual Machine (JVM) tries to load a particular file specified file cannot be found in that location.
3	**InterruptedException** It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.
4	**NoSuchMethodException** It is thrown when some JAR file has a different version at runtime that it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

<u>CheckedExceptionExample.java</u>

```
import   java.io.*;
import   java.util.Scanner;

class CheckedDemo
{
         public static void main(String args[])
        {
                  FileInputStream   mydata   = new FileInputStream("G:/soft wares/names.txt");
                           Scanner   myReader = new Scanner(mydata);
                         while (myReader.hasNextLine())
                          {
                                   String data = myReader.nextLine();
                                   System.out.println(data);
                         }
              }
}
```

<u>Output:</u>

In the above code, we are trying to read the names.txt file and display its data or content on the screen.

```
D:\ACEM\II CSE  Bsection>javac CheckedDemo.java
CheckedDemo.java:9: error: unreported exception FileNotFoundException; must be caught or declared
to be thrown
FileInputStream   mydata   = new FileInputStream("G:/soft wares/names.txt");
                                         ^
1 error
```

<u>How to resolve the error?</u>

There are basically two ways through which we can solve these errors.

- The exceptions occur in the main method. We can get clear from these compilation errors by declaring the exception in the main method using the throws . We only declare the IOException, not FileNotFoundException, because of the child-parent relationship.

- The IOException class is the parent class of FileNotFoundException, so this exception will automatically cover by IOException. We will declare the exception in the following way:

Example:

```
import   java.io.*;
import   java.util.Scanner;

class CheckedDemo
{
          public static void main(String args[])  throws IOException
          {
                    FileInputStream   mydata   = new FileInputStream("G:/soft wares/names.txt");
                              Scanner   myReader = new Scanner(mydata);
                            while (myReader.hasNextLine())
                             {
                                      String data = myReader.nextLine();
                                      System.out.println(data);
                            }
               }
}
```

Output:

```
D:\ACEM\II CSE  Bsection>javac CheckedDemo.java
D:\ACEM\II CSE  Bsection>java CheckedDemo
raja
ram
raghavan
```

2) We can also handle these exception using try-catch However, the way which we have used above is not correct. We have to a give meaningful message for each exception type.

By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

Exception.java

```
import   java.io.*;
import   java.util.Scanner;
class CheckedDemo
{
          public static void main(String args[])
          {
                  try
                  {
                            FileInputStream   mydata   = new FileInputStream("G:/soft wares/names.txt");
                            Scanner   myReader = new Scanner(mydata);
                            while (myReader.hasNextLine())
                            {
                                      String data = myReader.nextLine();
                                      System.out.println(data);
                            }
                  }
```

```
                catch(FileNotFoundException e)
                {
                        System.out.println(" no file found ");
                }
        }
}
```

Output:

```
D:\ACEM\II CSE  Bsection>javac CheckedDemo.java

D:\ACEM\II CSE  Bsection>java CheckedDemo

raja

ram

raghavan
```

Note:

If the file is not found in the specified location it will generate the following error

File Not Found

3.3 and 3.8 Unchecked Exceptions (uncaught Exceptions)

- The unchecked exceptions are just opposite to the checked exceptions.
- The compiler will not check these exceptions at compile time.
- Usually, it occurs when the user provides bad data during the interaction with the program.
- The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

S. No.	Exception Class with Description	Example

1	**ArithmeticException**
	It handles the arithmetic exceptions like dividion by zero
		int a = 30, b = 0;
		int c = a/b;

2	**ArrayIndexOutOfBoundsException**
	It handles the situations like an array has been accessed with an illegal index like (index is either negative or greater than or equal to the size of the array).
		int a[] = new int[5];
		a[6] = 9;

3	**ArrayStoreException**
	It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects
		Double[] a = new Double[2];
		
		a[0] = new Integer(4);

4	**ClassCastException**
	It handles the situation when we try to improperly cast a class from one type to another.
		Object o = new Object();
		String s = (String)o;

S. No.	Exception Class with Description	Example

5	IndexOutOfBoundsException
	It is thrown when attempting to access an invalid index within a collection, such as an array, vector , string , and so forth.
		int ar[] = { 1, 2, 3, 4, 5 };
		System.out.println(ar[9]);

6	NegativeArraySizeException It is thrown if an applet tries to create an array with negative size.	int ar[] = new int[-9];

7	NullPointerException
	it is thrown when program attempts to use an object reference that has the null value.
		String s =null;
		s.length();

8	NumberFormatException
	It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.
		String s = "aditya ";
		Integer i = Integer.parseInt(s);

9	StringIndexOutOfBounds
	It is thrown by the methods of the String class, in order to indicate that an index is either negative, or greater than the size of the string itself.
		String s = "aditya ";
		s.charAt[8];

UncheckedExceptionExample1.java

```
class UncheckedExceptionExample1
 {
          public static void main(String args[])
          {
                   int positive = 35;
                   int zero = 0;
                   int result = positive/zero;          //Give Unchecked Exception here.
                   System.out.println(result);
          }
}
```

Output:

```
D:\sasi>javac UncheckedExceptionExample1.java
D:\sasi>java UncheckedExceptionExample1
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at UncheckedExceptionExample1.main(UncheckedExceptionExample1.java:6)
```

Difference Between Checked and Unchecked Exception(uncaught Exception)

S.No	Checked Exception	Unchecked Exception
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of **Exception**	They are the direct subclasses of the **RuntimeException** class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic.
4.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
5.	These exceptions are propagated using the throws keyword	These are automatically propagated.
6.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

3.4 try-catch block

- In java, the try and catch, both are the keywords used for exception handling.
- The keyword try is used to define a block of code that will be tests the occurrence of an exception.
- The keyword catch is used to define a block of code that handles the exception occurred in the respective try block.
- Both try and catch are used as a pair. Every try block must have one or more catch blocks.
- We can not use try without at least one catch, and catch alone can be used (catch without try is not allowed).

Syntax

```
try
{
        code to be tested
}
catch(ExceptionType  object)
{
        code for handling the exception
}
```

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1 (TryCatchExample1.java)

```
public class TryCatchExample1
{
        public static void main(String[] args)
        {
                int data=50/0;              //may throw exception
                System.out.println("rest of the code");
        }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling (the above problem by a java try-catch block)

Example 2 (TryCatchExample2.java)

```
public class TryCatchExample2
{
   public static void main(String[] args)
    {
        try
        {
                int data  =  50/0;       //may throw exception
        }
       catch(ArithmeticException e)         //handling the exception
       {
               System.out.println(e);
       }
      System.out.println("rest of the code");
   }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

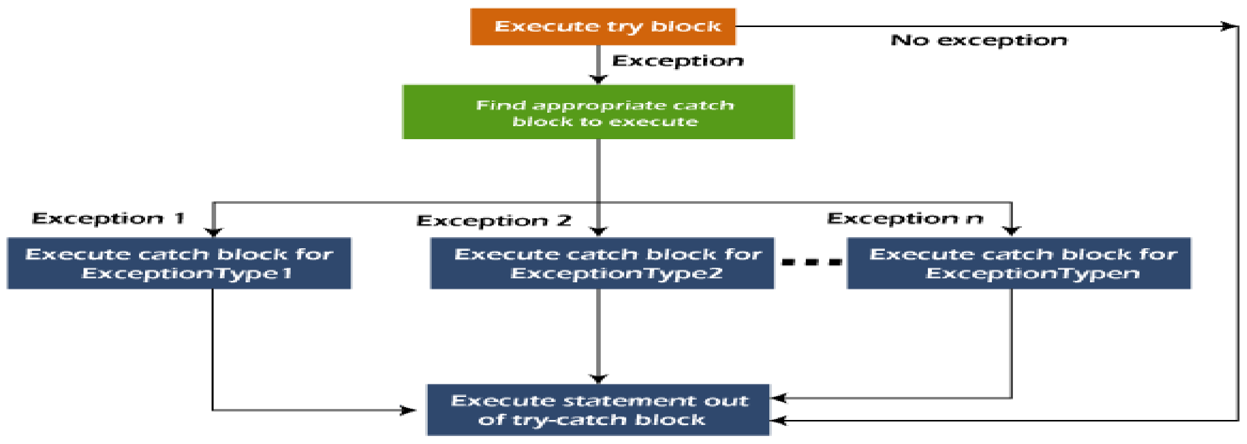
3.5 Multi-catch block

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

Points to remember

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

Flowchart of Multi-catch Block


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

Example 1 **MultipleCatchBlock1.java**

```
public class MultipleCatchBlock1
{
    public static void main(String[] args)
   {
                try
                 {
                         int    a[]   =   new   int[5];
                                a[5]  =   30/0;
                  }
            catch(ArithmeticException   e)
              {
                     System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException     e)
              {
                     System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
```

```
                    catch(Exception      e)
                      {
                                System.out.println("Parent Exception occurs");
                      }
                           System.out.println("rest of the code");
           }
        }
```

Output:

```
                Arithmetic Exception occurs
                rest of the code
```

3.6 Java Nested try block

- In Java, using a try block inside another try block is permitted. It is called as nested try block.
- Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.
- For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

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.

Syntax:

```
try
{
            statement 1;
            statement 2;
                        try      //try catch block within another try block
                         {
                                  statement 3;
                                   statement 4;
                         }
                        catch(Exception e1)
                        {
                             //exception message
                        }
}
catch(Exception e2)        //catch block of parent (outer) try block
{
           //exception message
}
.
```

Java Nested try Example   (NestedTryBlock.java)

```
public class NestedTryBlock
{
    public static void main(String args[])
    {
            try     //outer try block
            {
                   try          //inner try block 1
                   {
                           System.out.println("going to divide by 0");
                           int b   =  39/0;
                   }
                  catch(ArithmeticException e)      //catch block of inner try block 1
                   {
                        System.out.println(e);
                   }
                   try    //inner try block 2
                   {
                       int a[]   =  new   int[5];
                          a[5] = 4;
                   }
                 catch(ArrayIndexOutOfBoundsException  e)     //catch block of inner try block 2
                   {
                            System.out.println(e);
                   }
                        System.out.println("other statement");
             }
            catch(Exception e)       //catch block of outer try block
             {
                 System.out.println("handled the exception (outer catch)");
             }
              System.out.println("normal flow..");
       }
}
```

Output:

```
D:\ACEM\II CSE  Bsection>javac NestedTryBlock.java
D:\ACEM\II CSE  Bsection>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
other statement
normal flow..
```

Explanation:

- When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.
- If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

3.7 throw , throws and finally keywords

Java throw keyword

- The Java **throw** keyword is used to **throw an exception explicitly**.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom(user defined) exception.
- We can also define our own set of conditions and throw an exception explicitly using throw keyword.

syntax

```
throw new exception_class("error message");
```

Example

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

- Where the Instance(object) must be of type Throwable or subclass of Throwable.
- For example, Exception is the sub class of Throwable and the user-defined exceptions usually extends the Exception class.

Example 1: Throwing Unchecked Exception

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

```
        public static void main(String args[])
        {
                validate(13);         //calling the function
                 System.out.println("rest of the code...");
        }
    }
```

Output:

```
D:\sasi>javac TestThrow1.java
D:\sasi>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to vote
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Java throws keyword

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

Syntax:

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

Which exception should be declared?

Ans: Checked exception only, because **unchecked exception is** under our control so we can correct our code.

error is beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Java throws Example

```
import java.io.IOException;
class Testthrows1
{
        void   m()   throws   IOException
        {
             throw     new  IOException("device error");    //checked exception
        }
        void   n()      throws  IOException
        {
              m();
        }
        void p()
        {
             try
             {
                   n();
             }
             catch(Exception   e)
             {
                  System.out.println("exception handled");
             }
        }
        public static void main(String args[])
        {
             Testthrows1   obj   =   new   Testthrows1();
              obj.p();
              System.out.println("normal flow...");
        }
}
```

Output:

```
exception handled
normal flow...
```

Rules:

If we are calling a method that declares an exception, we must either caught or declare the exception.

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Difference between throw and throws in Java

S.no	Differences	Throw	Throws
1.	Definition	throw keyword is used throw an exception explicitly from inside the function .	Java throws keyword is used in the method signature.
2.	Type of exception	Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.

Java throw and throws Example **(TestThrowAndThrows.java)**

```
public class TestThrowAndThrows
{
   // defining a user-defined method   which throws ArithmeticException
      static void method() throws ArithmeticException
      {
           System.out.println("  Inside the method()  ");
           throw   new      ArithmeticException("   throwing ArithmeticException   ");
      }
    public static void main(String args[])
    {
          try
          {
               method();
          }
           catch(ArithmeticException e)
          {
               System.out.println("caught in main() method");
          }
    }
}
```

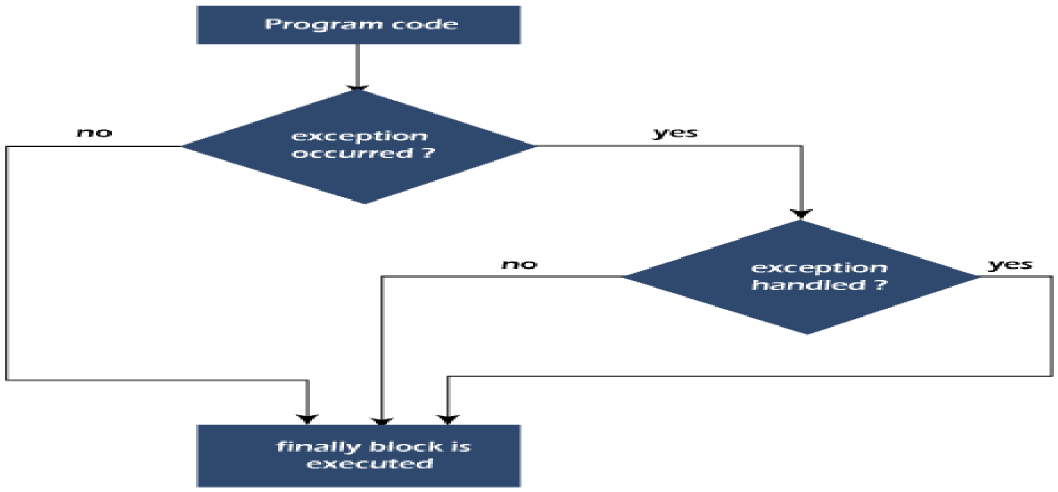
Output:

```
D:\sasi>javac TestThrowAndThrows.java
D:\sasi>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Java finally block

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

Flowchart of finally block



Note:

If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

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

Uses:

Case 1: When an exception does not occur
Case 2: When an exception occurs but not handled by the catch block
Case 3: When an exception occurs and is handled by the catch block

Case 1: When an exception does not occur (**TestFinallyBlock.java)**

```
class TestFinallyBlock
 {
     public static void main(String args[])
    {
              try
             {
                     int  data  =  25/5;
                     System.out.println(data);
             }
             catch(NullPointerException   e)
            {
                    System.out.println(e);
            }
             finally
            {
                    System.out.println("finally block is always executed");
            }
                    System.out.println("rest of the code...");
     }
 }
```

Output:

```
D:\sasi>javac TestFinallyBlock.java
D:\sasi>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occurs but not handled by the catch block

The code throws an exception however the catch block cannot handle it.
Even though, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```
class TestFinallyBlock
 {
        public static void main(String args[])
        {
              try
              {
                     int   data  =  25/0;
                     System.out.println(data);
              }
               catch(NullPointerException   e)
              {
                     System.out.println(e);
              }
              finally
              {
                     System.out.println("finally block is always executed");
              }
                     System.out.println("rest of the code...");
       }
 }
```

Output:

```
D:\ACEM\II CSE  Bsection>javac TestFinallyBlock.java
D:\ACEM\II CSE  Bsection>java TestFinallyBlock
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Case 3:   When an exception occurs and is handled by the catch block

Example where the Java code throws an exception and the catch block handles the exception.
Later the finally block is executed after the try-catch block.
Further, the rest of the code is also executed normally.

```
class TestFinallyBlock
 {
       public static void main(String args[])
      {
              try
             {
                     int   data  =  25/0;
                    System.out.println(data);
             }
              catch(ArithmeticException   e)
             {
                    System.out.println(" Exception handled ");
             }
             finally
             {
                    System.out.println("finally block is always executed");
             }
                    System.out.println("rest of the code...");
      }
 }
```

Output:

```
D:\ACEM\II CSE  Bsection>javac TestFinallyBlock.java
D:\ACEM\II CSE  Bsection>java  TestFinallyBlock

 Exception handled
finally block is always executed
rest of the code...
```

Note:

For each try block there can be zero or more catch blocks, but only one finally block.
The finally block will not be executed if the program exits (by calling System.exit())

Difference between final, finally and finalize

Sr. no.	Key	Final	Finally	Finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed..	finalize method is executed just before the object is destroyed.

3.9 Java Custom Exception or Creating our own Exceptions or User-defined Exception

- In Java, we can create our own exceptions that are derived classes of the Exception class.
- User Defined Exception or custom exception is creating your own exception class and throws that exception using 'throw' keyword. This can be done by extending the class Exception class that belongs to java.lang package.
- Basically, Java custom exceptions are used to customize the exception according to user need.
- Using the custom exception, we can have your own exception and message.

Example 1:

- In the following code, constructor of InvalidAgeException takes a string as an argument.
- This string is passed to constructor of parent class Exception using the super() method.
- Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

TestCustomException1.java

```
   class InvalidAgeException  extends Exception      // class representing custom exception
   {
      public InvalidAgeException (String str)
      {
            super(str);           // calling the constructor of parent Exception
      }
   }
public class TestCustomException1      // class that uses custom exception InvalidAgeException
{
    static void validate (int age) throws InvalidAgeException         // method to check the age
   {
      if(age < 18)
      {
              // throw an object of user defined exception
            throw   new    InvalidAgeException("age is not valid to vote");
      }
     else
      {
              System.out.println("welcome to vote");
       }
   }
   public static void main(String args[])
  {
     try
     {
             validate(13);       // calling the method
     }
    catch (InvalidAgeException     ex)
    {
        System.out.println("Caught the exception");
        System.out.println("Exception occured: " + ex);
    }
    System.out.println("rest of the code...");
  }
}
```

Output:

```
D:\ACEM\II CSE  Bsection>javac TestCustomException1.java
D:\ACEM\II CSE  Bsection>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Throwable class

- The **Throwable** class is the super class of every error and exception in the Java language.
- Only objects that are one of the subclasses this class are thrown by any “Java Virtual Machine” or may be thrown by the Java throw statement..

Example:

```
import java.io.IOException;
class Testthrows2
{
      public static void main(String args[])
     {
          try
          {
               int   a  =  10/0;
          }
           catch(Throwable   t)
           {
           System.out.println(t.toString());  //This method returns a short description of current throwable.
           System.out.println(t.getMessage());  //Returns the detail message string of current throwable.
           t.printStackTrace();  //Prints the current throwable and its backtrace to the standard error stream.
         }
     }
}
```

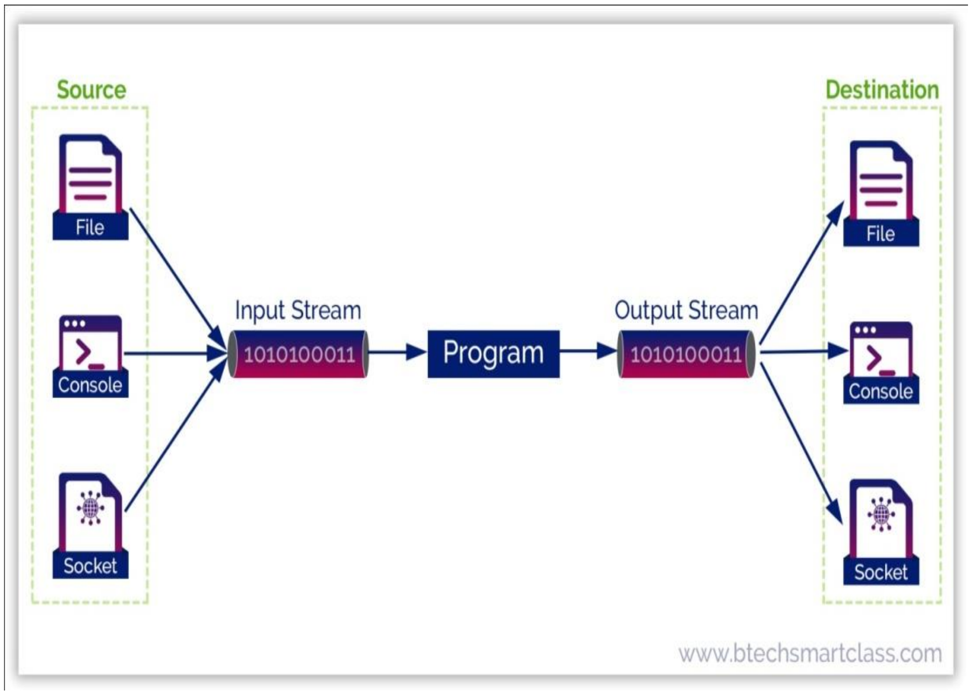
Output:

```
D:\ACEM\II  AI   DS>java Testthrows2
java.lang.ArithmeticException: / by zero
/ by zero
java.lang.ArithmeticException: / by zero
   at Testthrows2.main(Testthrows2.java:8)
```

3.10 Stream in java

- In java, the IO operations are performed using the concept of streams.
- Generally, a stream means a continuous flow of data.
- In java, a stream is a logical container of data that allows us to read from and write to it.
- The stream-based IO operations are faster than normal IO operations.
- The Stream is defined in the java.io package.

To understand the functionality of java streams, look at the following picture.



- The stream-based IO operations are performed using two separate streams input stream and output stream.
- The input stream is used for input operations, and the output stream is used for output operations.
- The java stream is composed of bytes.

In Java, every program creates 3 streams automatically, and these streams are attached to the console.

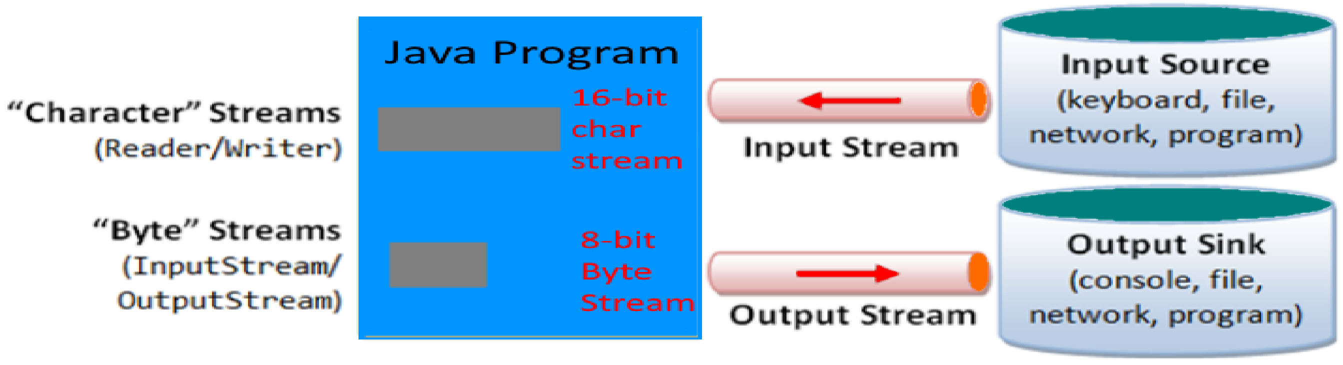
- **System.out**: standard output stream for console output operations.
- **System.in:** standard input stream for console input operations.
- **System.err:** standard error stream for console error output operations.

The Java streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

Java provides two types of streams, and they are as follows.

- Byte Stream
- Character Stream

The below picture shows how streams are categorized, and various built-in classes used by the java IO system.

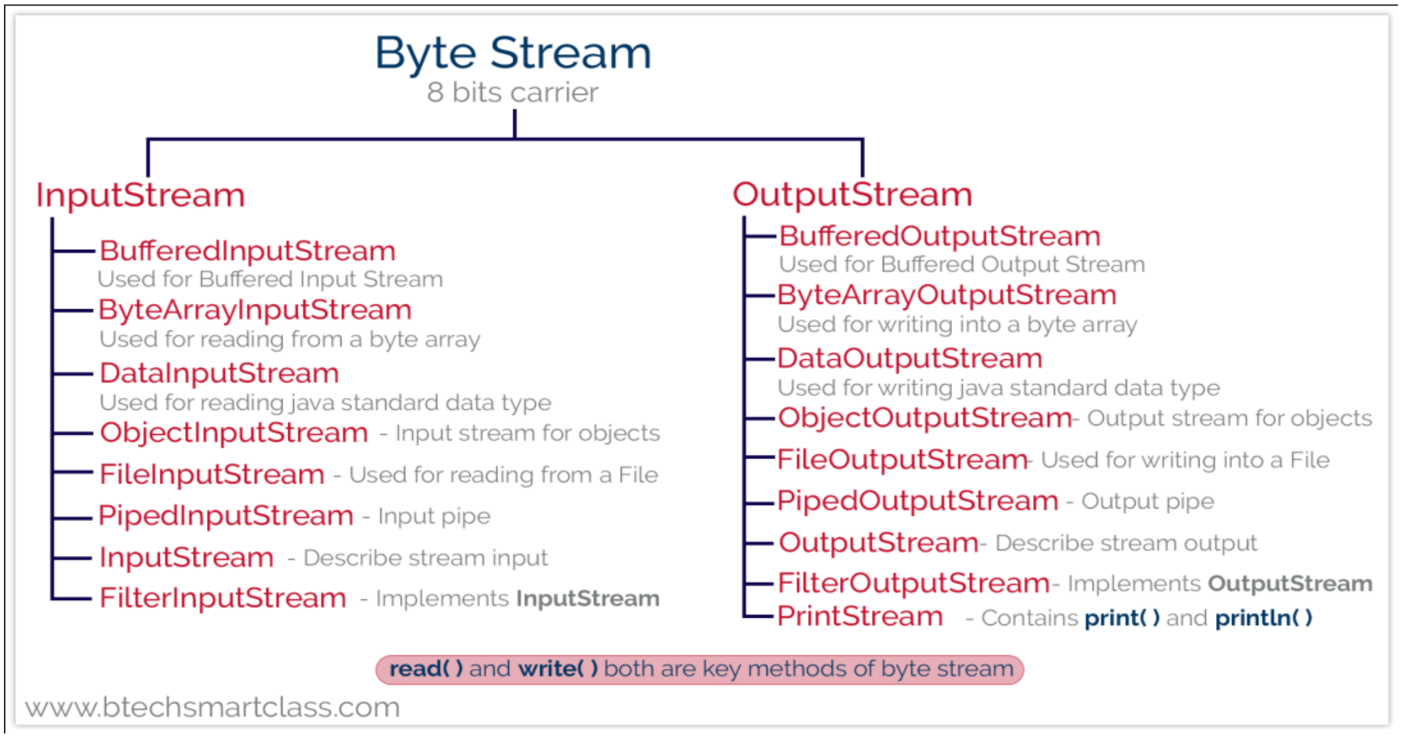


Both character and byte streams essentially provides a convenient and efficient way to handle data streams

3.11 Byte Stream in java

- In java, the byte stream is an 8 bits carrier. The byte stream in java allows us to transmit 8 bits of data.
- In Java 1.0 version all IO operations were byte oriented, there was no other stream (character stream).
- The java byte stream is defined by two abstract classes, InputStream and OutputStream.
- The InputStream class used for byte stream based input operations, and the OutputStream class used for byte stream based output operations.
- The InputStream and OutputStream classes have several concrete classes to perform various IO operations based on the byte stream.

The following picture shows the classes used for byte stream operations.


Byte Stream
8 bits carrier
InputStream
BufferedInputStream
Used for Buffered Input Stream
ByteArrayInputStream
Used for reading from a byte array
DataInputStream
Used for reading java standard data type
ObjectInputStream - Input stream for objects
FileInputStream - Used for reading from a File
PipedInputStream - Input pipe
InputStream - Describe stream input
FilterInputStream - Implements InputStream
OutputStream
BufferedOutputStream
Used for Buffered Output Stream
ByteArrayOutputStream
Used for writing into a byte array
DataOutputStream
Used for writing java standard data type
ObjectOutputStream- Output stream for objects
FileOutputStream- Used for writing into a File
PipedOutputStream - Output pipe
OutputStream- Describe stream output
FilterOutputStream- Implements OutputStream
PrintStream - Contains print( ) and println( )
read( ) and write( ) both are key methods of byte stream
www.btechsmartclass.com

InputStream class

The InputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**int available()** It returns the number of bytes that can be read from the input stream.
2	**int read()** It reads the next byte from the input stream.
3	**int read(byte[] b)** It reads a chunk of bytes from the input stream and store them in its byte array, b.
4	**void close()** It closes the input stream and also frees any resources connected with this input stream.

OutputStream class

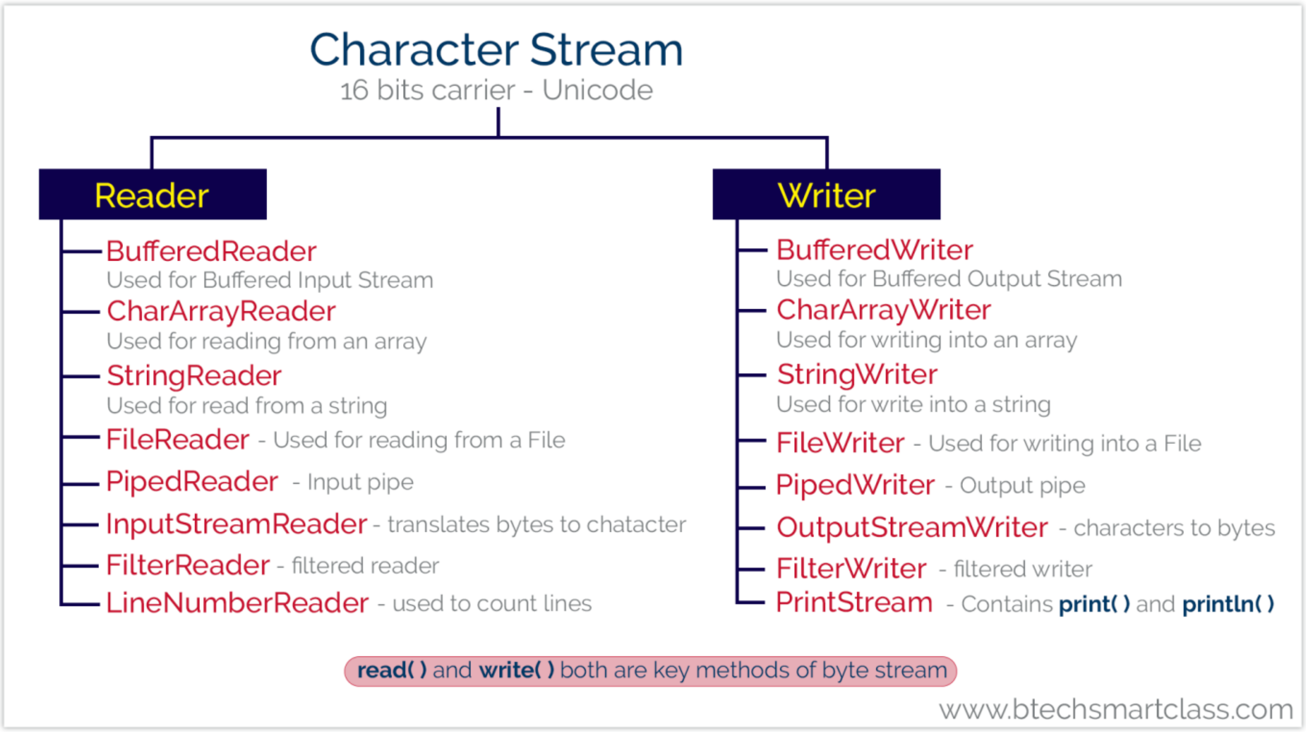
The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**void write(int n)** It writes byte(contained in an int) to the output stream.
2	**void write(byte[] b)** It writes a whole byte array(b) to the output stream.
3	**void flush()** It flushes the output steam by forcing out buffered bytes to be written out.
4	**void close()** It closes the output stream and also frees any resources connected with this output stream.

Character Stream in java

- In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream.
- The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.
- The character stream is a 16 bits carrier. The character stream allows us to transmit 16 bits of data.
- The character stream was introduced in Java 1.1 version. The character stream
- The java character stream is defined by two abstract classes, Reader and Writer.
- The Reader class used for character stream based input operations, and the Writer class used for character stream based output operations.
- The Reader and Writer classes have several concrete classes to perform various IO operations based on the character stream.

The following picture shows the classes used for character stream operations.


Character Stream
16 bits carrier - Unicode
Reader
BufferedReader
Used for Buffered Input Stream
CharArrayReader
Used for reading from an array
StringReader
Used for read from a string
FileReader - Used for reading from a File
PipedReader - Input pipe
InputStreamReader - translates bytes to chatacter
FilterReader - filtered reader
LineNumberReader - used to count lines
Writer
BufferedWriter
Used for Buffered Output Stream
CharArrayWriter
Used for writing into an array
StringWriter
Used for write into a string
FileWriter - Used for writing into a File
PipedWriter - Output pipe
OutputStreamWriter - characters to bytes
FilterWriter - filtered writer
PrintStream - Contains print( ) and println( )
read( ) and write( ) both are key methods of byte stream
www.btechsmartclass.com

Reader class

The Reader class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	**Method with Description**
1	**int read()** It reads the next character from the input stream.
2	**int read(char[] cbuffer)** It reads a chunk of characters from the input stream and store them in its byte array, cbuffer.
3	**int read(char[] cbuf, int off, int len)** It reads characters into a portion of an array.
4	**int read(CharBuffer target)** It reads characters into into the specified character buffer.
5	**String readLine()** It reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.
	boolean ready() It tells whether the stream is ready to be read.
7	**void close()** It closes the input stream and also frees any resources connected with this input stream.

Writer class

The Writer class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**void flush()** It flushes the output steam by forcing out buffered bytes to be written out.
2	**void write(char[] cbuf)** It writes a whole array(cbuf) to the output stream.
3	**void write(int c)** It writes single character.
4	**void write(String str)** It writes a string.
5	**void write(String str, int off, int len)** It writes a portion of a string.
6	**Writer append(char c)** It appends the specified character to the writer.
7	**Writer append(CharSequence csq)** It appends the specified character sequence to the writer
8	**Writer append(CharSequence csq, int start, int end)** It appends a subsequence of the specified character sequence to the writer.
9	**void close()** It closes the output stream and also frees any resources connected with this output stream.

3.12 Console IO Operations in Java

Reading console input in java

There are three ways to read console input. Using these 3 ways we can read input data from the console.

- Using BufferedReader class
- Using Scanner class

1. Reading console input using BufferedReader class in java

- Reading input data using the BufferedReader class is the traditional technique.
- This way of the reading method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.
- The BufferedReader class has defined in the java.io package.

Example

```
        import java.io.BufferedReader;
        import java.io.IOException;
        import java.io.InputStreamReader;

public class FirstCode
{
        public static void main(String args[]) throws IOException
        {
                BufferedReader   reader   =  new   BufferedReader(new   InputStreamReader(System.in));
                System.out.println("  Enter your favorite subject  ");
                String   word   =   reader.readLine();
                System.out.println(word);
        }
}
```

Input and Output

```
D:\ACEM\II  AI   DS>javac FirstCode.java
D:\ACEM\II  AI   DS>java FirstCode
Enter your favorite subject
Maths
Maths
```

2. Reading console input using Scanner class in java

- Reading input data using the Scanner class is the most commonly used method.
- This way of the reading method is used by wrapping the System.in (standard input stream) which is wrapped in a Scanner, we can read input from the console.
- The Scanner class has defined in the java.util package.

Example

```
import java.util.Scanner;
public class ReadingDemo
{
        public static void main(String[] args)
        {
                Scanner   in   =   new   Scanner(System.in);

                String    name   = "";

                System.out.print(" Please enter your name : ");

                name = in.next();

                System.out.println("Hello, " + name);
        }
}
```

Output:

```
D:\ACEM\II  AI   DS>javac ReadingDemo.java
D:\ACEM\II  AI   DS>java ReadingDemo
Please enter your name : Aditya
Hello, Aditya
```

Writing console output in java

In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

- Using print() and println() methods
- Using write() method

1. Writing console output using print() and println() methods

- There are two ways to write at the console. Both ways belong to class **PrintStream(Character-based class).**
- **print and println methods:** Here, the methods are defined by class **Printstream** and is referenced by System.out. These methods are widely used.

```
public class    WritingToConsole
{
        public static void main(String args[])
        {
                 System.out.println("   *****  I am System.out.print. I print the passed string   *****   ");
                 for(int   i=1;i<=5;i++)
                {
                             System.out.print(i);
                 }
                System.out.println();
               System.out.println("I am System.out.println. I print the passed string and end with a new line");
                for(int   i=1;i<=3;i++)
                {
                          System.out.println(i);
                }
    }
}
```

Output

```
D:\ACEM\II  AI   DS>javac WritingToConsole.java
D:\ACEM\II  AI   DS>java WritingToConsole
*****   I am System.out.print. I print the passed string   *****
12345
I am System.out.println. It print the passed string and end with a new line
1
2
3
```

3.13 File class in Java

- The **File** is a built-in class in Java.
- In java, the File class has been defined in the **java.io** package.
- The File class represents a reference to a file or directory.
- The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory.

The File class in java has the following constructors.

S.No.	Constructor with Description
1	**File(String pathname)** It creates a new File instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.
2	**File(String parent, String child)** It Creates a new File instance from a parent abstract pathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
3	**File(File parent, String child)** It creates a new File instance from a parent abstract pathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
4	**File(URI uri)** It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class in java has the following methods.

S.No.	Methods with Description
1	**String getName()** It returns the name of the file or directory that referenced by the current File object.
2	**String getParent()** It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.
3	**String getPath()** It returns the path of curent File.
4	**File getParentFile()** It returns the path of the current file's parent; or null if it does not exist.
5	**String getAbsolutePath()** It returns the current file or directory path from the root.
6	**boolean isAbsolute()** It returns true if the current file is absolute, false otherwise.
7	**boolean isDirectory()** It returns true, if the current file is a directory; otherwise returns false.

S.No.	Methods with Description
8	**boolean isFile()** It returns true, if the current file is a file; otherwise returns false.
9	**boolean exists()** It returns true if the current file or directory exist; otherwise returns false.
10	**boolean canRead()** It returns true if and only if the file specified exists and can be read by the application; false otherwise.
11	**boolean canWrite()** It returns true if and only if the file specified exists and the application is allowed to write to the file; false otherwise.
12	**long length()** It returns the length of the current file.

Example:

```
import java.io.File;
public class FileClassTest
{
        public static void main(String args[])
        {
                File    f    =    new File("D:\\ACEM\\II  AI   DS\\datFile.txt");
                System.out.println("Executable File : " + f.canExecute());
                System.out.println("Name of the file : " + f.getName());
                System.out.println("Path of the file : " + f.getAbsolutePath());
                System.out.println("Parent name :  " + f.getParent());
                System.out.println("Write mode :  " + f.canWrite());
                System.out.println("Read mode :  " + f.canRead());
                System.out.println("Existance :  " + f.exists());
                System.out.println("Last Modified :  " + f.lastModified());
                System.out.println("Length :  " + f.length());
        }
}
```

Input & Output

```
C:\Raja\>javac FileClassTest.java

C:\Raja\>java FileClassTest

Executable File : true
Name of the file : textfile.txt
Path of the file : C:\Raja\textfile.txt
Parent name :  C:\Raja\
Write mode :  true
Read mode :  true
Existance :  true
Last Modified :  1641629987284
Length :  35
```

3.14 File Reading & Writing in Java

In java, there are multiple ways to read data from a file and to write data to a file.

The most commonly used ways are as follows.

Using Byte Stream (FileInputStream and FileOutputStream)

Using Character Stream (FileReader and FileWriter)

File Handling using Byte Stream

In java, we can use a byte stream to handle files. The byte stream has the following built-in classes to perform various operations on a file.

- FileInputStream - It is a built-in class in java that allows reading data from a file. This class has implemented based on the byte stream. The FileInputStream class provides a method read() to read data from a file byte by byte.
- FileOutputStream - It is a built-in class in java that allows writing data to a file. This class has implemented based on the byte stream. The FileOutputStream class provides a method write() to write data to a file byte by byte.

Example program that reads data from a file and writes the same to another file using FileInoutStream and FileOutputStream classes.

Example

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class FileReadingTest
{
        public static void main(String args[]) throws IOException
        {
                FileInputStream in = null;
                FileOutputStream out = null;
               try
               {
                        in   =   new   FileInputStream("D:\\ACEM\\II  AI   DS\\Input-File.txt");
                        out  =   new   FileOutputStream("D:\\ACEM\\II  AI   DS\\Output-File.txt");
                        int c;

                       while ((c = in.read()) != -1)
                       {
                               out.write(c);
                       }
                       System.out.println(" Reading and Writing has been success ");
               }
```

```
                catch(Exception e)
                {
                        System.out.println(e);
                 }
                finally
                {
                          if (in != null)
                        {
                                  in.close();
                        }
                          if (out != null)
                        {
                                 out.close();
                        }
                 }
        }
}
```

Output:

```
                D:\ACEM\II  AI   DS>javac FileReadingTest.java
                D:\ACEM\II  AI   DS>java FileReadingTest

                Reading and Writing has been success
```

Note: The above program will read data from one file and writes data to another file by creating new file

File Handling using Character Stream

In java, we can use a character stream to handle files. The character stream has the following built-in classes to perform various operations on a file.

- FileReader - It is a built-in class in java that allows reading data from a file. This class has implemented based on the character stream. The FileReader class provides a method read() to read data from a file character by character.
- FileWriter - It is a built-in class in java that allows writing data to a file. This class has implemented based on the character stream. The FileWriter class provides a method write() to write data to a file character by character.

Example program that reads data from a file and writes the same to another file using FIleReader and FileWriter classes.

Example

```
 import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public   class   FileReadingTest1
{
              public static void main(String args[]) throws IOException
             {
                          FileReader  in = null;
                          FileWriter  out = null;

                         try
                        {
                                      in    =    new   FileReader("D:\\ACEM\\II  AI   DS\\Input-File1.txt");
                                      out   =    new   FileWriter("D:\\ACEM\\II  AI   DS\\Output-File1.txt");
                                      int c;

                                     while ((c = in.read()) != -1)
                                  {
                                             out.write(c);
                                   }
                                  System.out.println("Reading and Writing has been success ");
                        }
                        catch(Exception e)
                        {
                                   System.out.println(e);
                         }
                        finally
                        {
                                    if (in != null)
                                   {
                                              in.close();
                                   }
                                    if (out != null)
                                   {
                                             out.close();
                                   }
                          }
             }
}
```

Output:

```
D:\ACEM\II  AI   DS>javac  FileReadingTest1.java
D:\ACEM\II  AI   DS>java FileReadingTest1
Reading and Writing has been success
```

3.15 RandomAccessFile in Java

- The java.io package has a built-in class RandomAccessFile that enables a file to be accessed randomly.
- The RandomAccessFile class has several methods used to move the cursor position in a file.
- A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The RandomAccessFile class in java has the following constructors.

S.No.	Constructor with Description
1	**RandomAccessFile(File fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
2	**RandomAccessFile(String fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, a file with the specified fileName.

Access Modes

Using the RandomAccessFile, a file may created in th following modes.

- r - Creates the file with read mode; Calling write methods will result in an IOException.
- rw - Creates the file with read and write mode.
- rwd - Creates the file with read and write mode - synchronously. All updates to file content is written to the disk synchronously.
- rws - Creates the file with read and write mode - synchronously. All updates to file content or meta data is written to the disk synchronously.

RandomAccessFile methods

S.No.	Methods with Description
1	**int read()** It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2	**int read(byte[] b)** It reads byte of data from file upto b.length, -1 if end of file is reached.
3	**int read(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
4	**boolean readBoolean()** It reads a boolean value from from the file.
5	**byte readByte()** It reads signed eight-bit value from file.
6	**char readChar()** It reads a character value from file.
7	**double readDouble()** It reads a double value from file.
8	**float readFloat()** It reads a float value from file.

Example

```
import java.io.*;
public class RandomAccessFileDemo
{
            public static void main(String[] args)
           {
              try
              {
                     double   d = 1.5;
                     float       f = 14.56f;
                     RandomAccessFile   ref = new RandomAccessFile("D:\\ACEM\\II  AI   DS\\random.txt", "rw");
                      ref.writeUTF("Hello, Good Morning!");

                     ref.seek(0);         // File Pointer at index position - 0
                     System.out.println("Use of read() method : " +ref.read());

                    ref.seek(0);
                    byte[]    b = {1, 2, 3};
                    System.out.println("Use of .read(byte[]   b) : " +ref.read(b));   // Use of .read(byte[] b) method :

                    System.out.println("Use of readBoolean() : " +ref.readBoolean());      // readBoolean() method :
                    System.out.println("Use of readByte() : " +ref.readByte());          // readByte() method :
                    System.out.println("Use of readFloat() : " +ref.readFloat());
                    System.out.println("Use of readDouble() : " +ref.readDouble());
       }
       catch (IOException ex)
       {
          System.out.println("Something went Wrong");
          ex.printStackTrace();
       }
    }
}
```

Output

```
D:\ACEM\II  AI   DS>javac  RandomAccessFileDemo.java
D:\ACEM\II  AI   DS>java  RandomAccessFileDemo

Use of read() method : 0
Use of .read(byte[]   b) : 3
Use of readBoolean() : true
Use of readByte() : 108
Use of readFloat() : 1.1565666E27
Use of readDouble() : 1.3057638631517625E36
```

3.16 Console class in Java

- In java, the java.io package has a built-in class Console used to read from and write to the console, if one exists.
- This class was added to the Java SE 6.
- The Console class implements the Flushable interface.
- In java, most the input functionalities of Console class available through System.in, and the output functionalities available through System.out.

Console class Constructors

The Console class does not have any constructor.
We can obtain the Console class object by calling System.console()

Console class methods

S.No.	Methods with Description
1	**void flush( )** It causes buffered output to be written physically to the console.
2	**String readLine( )** It reads a string value from the keyboard, the input is terminated on pressing enter key.
3	**String readLine(String promptingString, Object...args)** It displays the given prompting String, and reads a string fron the keyboard; input is terminated on pressing Enter key.
4	**char[ ] readPassword( )** It reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
5	**char[ ] readPassword(String promptingString, Object...args)** It displays the given promptingString, and reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
6	**Console printf(String str, Object....args)** It writes the given string to the console.
7	**Console format(String str, Object....args)** It writes the given string to the console.
8	**Reader reader( )** It returns a reference to a Reader connected to the console.
9	**PrintWriter writer( )** It returns a reference to a Writer connected to the console.

Example

```
import java.io.Console;
public class ReadingDemo1
 {
	public static void main(String[] args)
	{
		 String name;
		Console   con  =   System.console();
		 if(con != null)
		 {
			name  =  con.readLine("Please enter your name : ");
			System.out.println("Hello, " + name);
		 }
		 else
		  {
			System.out.println("Console not available.");
		 }
	}
}
```

Output:

```
D:\ACEM\II  AI   DS>javac ReadingDemo1.java
D:\ACEM\II  AI   DS>java ReadingDemo1

Please enter your name : raju
Hello,raju
```

Example

```
import java.io.*;
public class WritingDemo
{
	public static void main(String[] args)
	{
		int[]    list    =    new    int[5];
		for(int i = 0; i < 5; i++)
		{
			list[i] = i + 65;
		}
		for(int    i :   list)
		{
			System.out.write(i);
			System.out.write('\n');
		}
	}
}
```

Output:

```
D:\ACEM\II  AI   DS>javac WritingDemo.java
D:\ACEM\II  AI   DS>java WritingDemo
A
B
C
D
E
```

3.17 Serialization and Deserialization

- In java, the Serialization is the process of converting an object into a byte stream so that it can be stored on to a file, or memory, or a database for future access.
- The deserialization is reverse of serialization.
- The deserialization is the process of reconstructing the object from the serialized state.
- Using serialization and deserialization, we can transfer the Object Code from one Java Virtual machine to another.

Serialization in Java

- In a java programming language, the Serialization is achieved with the help of interface Serializable.
- The class whose object needs to be serialized must implement the Serializable interface.
- We use the ObjectOutputStream class to write a serialized object to write to a destination.
- The ObjectOutputStream class provides a method writeObject() to serializing an object.

steps to serialize an object

Step 1 - Define the class whose object needs to be serialized; it must implement Serializable interface.

Step 2 - Create a file reference with file path using FileOutputStream class.

Step 3 - Create reference to ObjectOutputStream object with file reference.

Step 4 - Use writeObject(object) method by passing the object that wants to be serialized.

Step 5 - Close the FileOutputStream and ObjectOutputStream.

Example

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
class Student implements Serializable
{
	int id;
	String name;

	public Student(int id, String name)
	 {
		this.id = id;
		this.name = name;
	 }
}
```

```
public class SerializationExample
{
        public static void main(String[] args)
        {
                  try
                 {
                        Student    s1    =   new    Student(211,"ravi");
                        FileOutputStream   fout   =  new FileOutputStream("f.txt"); //Creating stream and writing the obj
                        ObjectOutputStream    out  =  new   ObjectOutputStream(fout);
                        out.writeObject(s1);
                        out.flush();
                        out.close();
                        System.out.println("success");
                 }
                 catch(Exception e)
                {
                                    System.out.println(e);
                 }
            }
}
```

Output

```
D:\ACEM\II  AI   DS>javac  SerializationExample.java
D:\ACEM\II  AI   DS>java  SerializationExample
success
```

Deserialization in Java

- In a java programming language, the Deserialization is achieved with the help of class ObjectInputStream.
- This class provides a method readObject() to deserializing an object.

Steps to serialize an object.

- **Step 1** - Create a file reference with file path in which serialized object is available using FileInputStream class.
- **Step 2** - Create reference to ObjectInputStream object with file reference.
- **Step 3** - Use readObject() method to access serialized object, and typecaste it to destination type.
- **Step 4** - Close the FileInputStream and ObjectInputStream.

<u>Example</u> (deserializing an object)

```
import java.io.*;
public class DeserializationExample
{
        public static void main(String[] args) throws Exception
        {
                try
                {
                        FileInputStream    fis    =    new    FileInputStream("f.txt");
                        ObjectInputStream  ois  =   new ObjectInputStream(fis);
                        Student    stud2    =    (Student) ois.readObject();
                        System.out.println(" The object has been deserialized. ");
                        fis.close();
                        ois.close();
                        System.out.println("Name = " + stud2.id);
                        System.out.println("Department = " + stud2.name);
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

<u>Output:</u>

```
D:\ACEM\II  AI   DS>javac  DeserializationExample.java
D:\ACEM\II  AI   DS>java  DeserializationExample
The object has been deserialized.
Name = 211
Department = ravi
```

<u>3.18 Enum in Java</u>

- An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).
- This concept enables the java enum to have constructors, methods, and instance variables.
- All the constants of an enum are ***public***, ***static***, and ***final***.
- As they are static, we can access directly using enum name.
- The main objective of enum is to define our own data types in Java, and they are said to be enumeration data types.

<u>Creating enum in Java</u>

- In java, an enum can be defined outside a class, inside a class, but not inside a method.
- To create an enum, use the enum keyword , and separate the constants with a comma.
- Note that they should be in uppercase letters:

- Access enum constants with the dot syntax
- Enum is short for "enumerations", which means "specifically listed".
- Every constant of an enum is defined as an object and enum does not allow to create an object of it.
- As an enum represents a class, it can have methods, constructors. It also gets a few extra methods from the Enum class, and one of them is the values() method.
- In java, enum can not extend another enum and a class but enum can implement interfaces.
- In java, every enum extends a built-in class Enum by default.

Enum using values() method

The enum type has a values() method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum:

Example

```
enum  WeekDay
{
        MONDAY, TUESDAY, WEDNESSDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
public class EnumerationExample
{
        public static void main(String[] args)
        {
                WeekDay  day  =   WeekDay.FRIDAY;
                System.out.println(" Today is " + day);
                System.out.println("\n  All WeekDays: ");
                for(WeekDay  d : WeekDay.values())
                {
                        System.out.println(d);
                }
        }
}
```

Output:

```
Today is FRIDAY

All WeekDays:
MONDAY
TUESDAY
WEDNESSDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
```

Java Enum ordinal() Method

The ordinal() method of Enum class returns the ordinal of this enumeration constant.

This method is designed to be used by sophisticated enum-based data structures, like EnumSet and EnumMap.

Syntax

```
public final int ordinal()
```

Return Value

The ordinal() method returns this enumeration constant's ordinal.

Example

```
public class Enum_ordinalMethodExample1
{
        enum Colours
        {
                Red,Green,Brown,Yellow;
         }
        public static void main(String[] args)
        {
                Colours    Red   = Colours.Red;
                Colours   Green  = Colours.Green;
                Colours  Brown  = Colours.Brown;
                Colours  Yellow  = Colours.Yellow;
                System.out.println(" Red ordinal = "+Red.ordinal());
                System.out.println(" Green Ordinal = "+Green.ordinal());
                System.out.println(" Brown Ordinal = "+Brown.ordinal());
                System.out.println(" Yellow Ordinal = "+Yellow.ordinal());
        }
}
```

Output:

```
Red ordinal = 0
Green Ordinal = 1
 Brown Ordinal = 2
Yellow Ordinal = 3
```

3.19 Autoboxing and Unboxing in Java

- All the primitive data types have defined using the class concept, these classes known as **wrapper classes**.
- In java, every primitive type has its corresponding wrapper class.
- All the wrapper classes in Java were defined in the **java.lang** package.
- The following table shows the primitive type and its corresponding wrapper class.

S.No.	**Primitive Type**	**Wrapper class**
1	byte	**Byte**
2	short	**Short**
3	int	**Interger**
4	long	**Long**
5	float	**Float**
6	double	**Double**
7	char	**Character**
8	boolean	**Boolean**

The Java 1.5 version introduced a concept that converts primitive type to corresponding wrapper type and reverses of it.

Autoboxing in Java

- In java, the process of converting a primitive type value into its corresponding wrapper class object is called autoboxing or simply boxing.
- For example, converting an int value to an Integer class object.
- The compiler automatically performs the autoboxing when a primitive type value has assigned to an object of the corresponding wrapper class.
- We can also perform autoboxing manually using the method **valueOf( )**, which is provided by every wrapper class.

Example - Autoboxing

```
import java.lang.*;
public class AutoBoxingExample
{
        public static void main(String[] args)
        {
                int   num   =  100;    // Auto boxing : primitive to Wrapper
                Integer   i  =  num;
                Integer   j  =  Integer.valueOf(num);
                System.out.println("num = " + num + ", i = " + i + ", j = " + j);
        }
}
```

Output:

```
        D:\SASI\sasijava>javac AutoBoxingExample.java
        D:\SASI\sasijava>java AutoBoxingExample
        num = 100, i = 100, j = 100
```

Auto un-boxing in Java

- In java, the process of converting an object of a wrapper class type to a primitive type value is called auto un-boxing or simply unboxing.
- For example, converting an Integer object to an int value.
- The compiler automatically performs the auto un-boxing when a wrapper class object has assigned to a primitive type.
- We can also perform auto un-boxing manually using the method **intValue( )**, which is provided by Integer wrapper class. Similarly every wrapper class has a method for auto un-boxing.

Example - Auto unboxing

```
import java.lang.*;
public class AutoUnboxingExample
{
        public static void main(String[] args)
        {
                Integer   num   =  200;    // Auto un-boxing : Wrapper to primitive
                int  i   =   num;
                int  j = num.intValue();
                System.out.println("num = " + num + ", i = " + i + ", j = " + j);
        }
}
```

Output:

```
                D:\SASI\sasijava>javac AutoUnboxingExample.java
                D:\SASI\sasijava>java AutoUnboxingExample
                num = 200, i = 200, j = 200
```

3.20 Generics in Java

- The java generics is a language feature that allows creating methods and class which can handle any type of data values.
- Most of the collection framework classes are generic classes.
- The java generics allows only non-primitive type, it does not support primitive types like int, float, etc.
- Generics provide us compile time safety which helps the programmer to catch the invalid type at compile time.
- There are **two reasons why we need generics**.

  Type Safety

  Type Casting

The java generics feature was introduced in Java **1.5 version**. In java, generics used angular brackets "< >".

In java, the generics feature implemented using the following.

Generic Method or Generic Class

Generic methods

- The java generics allows creating generic methods which can work with a different type of data values.
- Using a generic method, we can create a single method that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Example - Generic method

```
public class GenericMethodTest
{
        public  static < E >   void   printArray( E[] inputArray)
        {
                for(E   element :   inputArray)
                {
                        System.out.printf("%s",element);
                }
                System.out.println();
        }
        public static void main(String args[])
        {
                Integer[]   intArray   =   {1,2,3,4,5};
                Character[]   charArray   =   {'h','e','l','l','o'};
                System.out.println(" Array inter contains ");
                printArray(intArray);
                System.out.println(" Array character contains ");
                printArray(charArray);
        }
}
```

<u>Output:</u>

```
                D:\SASI\sasijava>javac GenericMethodTest.java
                D:\SASI\sasijava>java GenericMethodTest
                Array inter contains
                12345
                Array character contains
                Hello
```

In the above example code, the method printArray ( ) is a generic method that allows a different type of parameter values for every function call.

<u>Generic Class</u>

In java, a class can be defined as a generic class that allows creating a class that can work with different types. A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

Example - Generic class

```
public class GenericClass <T1,T2>
{
        public void display(T1 var1,T2 var2)
        {
                System.out.println("Name:"+var1+", ID:"+var2);
        }
```

```
	public static void main(String args[])
	{
		GenericClass<String,Integer>    obj1  =  new   GenericClass<String,Integer>();
		GenericClass<Integer,Integer>   obj2  =  new   GenericClass<Integer,Integer>();
		obj1.display("Mounica",101);
		obj2.display(100,101);
	}
}
```

Output:

```
		D:\SASI\sasijava>javac GenericClass.java
		D:\SASI\sasijava>java GenericClass
		Name:Mounica, ID:101
		Name:100, ID:101
```

Unit-III (Assignement Questions)

1) Explain exception handling using try-catch block with a suitable program.
2) Explain the different types of exception in java by giving examples of each(built in exceptions).
3) Explain the difference between throw, throws in exception handling and Write a program to demonstrate the multiple catch.
4) Explain random access file in java. Explain the methods for writing and reading byte in random access file class.
5) Explain reading console Input and writing console output in Java with programs.
6) Explain Byte streams and Character streams in detail.
7) Explain the various methods defined in File Reader Class.
8) Demonstrate the use of FileInputstream and Explain serialization in Java with a simple program.