UNIT III

Exception handling: Hierarchy, Fundamentals, Multiple catch clauses, Subclass exceptions, Nesting try blocks, Throwing an exception, Using Finally and Throws, Built-in exceptions, User-defined exceptions.
I/O: Byte streams and Classes, Character streams and Classes, Predefined streams, Using byte streams, Reading and Writing files using byte streams, Reading and writing binary data, Random-access files, File I/O using character streams, Wrappers.

What is exception?
Dictionary Meaning: Exception is an abnormal condition.
In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

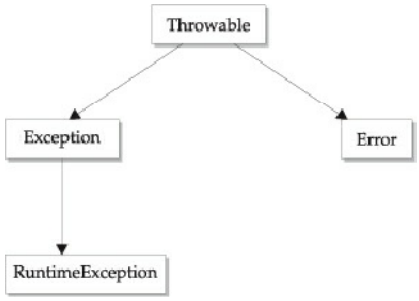
What is exception handling?
Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.
Advantage of Exception Handling
The core advantage of exception handling is to maintain the normal flow of the application. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

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

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Exception Hierarchy:

- ✓ In Java, all exceptions are represented by classes. All exception classes are derived from a class called **Throwable.**
- ✓ When an exception occurs in a program, an object of some type of exception class is generated.
- ✓ There are two direct subclasses of **Throwable**: **Exception** and **Error**.
- ✓ Exceptions of type **Error** are related to errors that are beyond our control, such as those that occur in the Java Virtual Machine itself.
- ✓ Errors that results from program activity are represented by subclasses of Exception. For example, divide-by-zero, array boundary, and I/O errors. An important subclass of **Exception** is **RuntimeException,** which is used to represent various common types of run-time errors.



Exception handling Fundamentals:

- ✓ Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- ✓ Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- ✓ Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown.
- ✓ The code can catch this exception using **catch** and handle it in some rational manner.
- ✓ System-generated exceptions are automatically thrown by the Java runtime system.
- ✓ To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- ✓ Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.
- ✓ The general form of **try/catch** exception handling blocks:

```
try{
    // block of code to monitor for errors
}
catch(ExceptionType1 exOb){
    // handle for ExceptionType1
}
catch(ExceptionType2 exOb){
    // handle for ExceptionType2
}
.
.
```

- ✓ When an exception is thrown, it is caught by its corresponding **catch** clause, which then processes the execution. When an exception is caught exOb will receive its value.
- ✓ If no exception is thrown, then a **try** block ends normally, and all of its **catch** blocks are bypassed. Execution resumes with the first statement following the last **catch.**

```
/*A simple Exception example */
// Demonstrate exception handling.
class ExcDemo1 {
  public static void main(String[] args) {
      int[] nums = new int[4];
      try {
            System.out.println("Before exception is generated.");
            // generate an index out-of-bounds exception
            nums[7] = 10;
            System.out.println("this won't be displayed");
    }
     catch (ArrayIndexOutOfBoundsException exc) {
            // catch the exception
            System.out.println("Index out-of-bounds!");
    }
    System.out.println("After catch.");
  }
}
```

O/P:

```
Before exception is generated.
Index out-of-bounds!
After catch.
```

- ✓ Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "this won't be displayed" is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try / catch** mechanism.

Uncaught Exceptions

- ✓ When an exception is thrown, it must be caught by some piece of code; Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- ✓ Problem without exception handling

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

```
public class Testtrycatch1{
  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, rest of the code is not executed (in such case, rest of the code... statement is not printed).
- ✓ Solution by exception handling
- ✓ It is important to handle exceptions by the program itself rather than rely on JVM

Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
  public static void main(String args[]){
  try{
     int data=50/0;
  }catch(ArithmeticException e){System.out.println(e);}
   System.out.println("rest of the code...");
}
}
```

- ✓ Output:

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

- ✓ Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.
- ✓ The type of exception must match the type specified in a catch. If it does not, the exception would not be caught.

```
// This won't work!
class ExcTypeMismatch {
  public static void main(String[] args) {
    int[] nums = new int[4];

    try {
      System.out.println("Before exception is generated.");

      //generate an index out-of-bounds exception
      nums[7] = 10;
      System.out.println("this won't be displayed");
    }

    /* Can't catch an array boundary error with an ArithmeticException. */
    catch (ArithmeticException exc) {
      // catch the exception
      System.out.println("Index out-of-bounds!");
    }
```

Multiple catch Clauses

- ✓ We 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.

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   catch(Exception e){System.out.println("common task completed");}
   System.out.println("rest of the code...");
  }
 }
```

Output:

```
task1 completed
rest of the code...
```

- ✓ ***Rule: At a time only one Exception is occured and at a time only one catch block is executed.***

Catching subclass Exceptions

- ✓ When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.
- ✓ ***Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception.***

```
class TestMultipleCatchBlock1{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
  }
 }
```

Output:
Compile-time error

Nesting try blocks

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

```
// Use a nested try block.
class nestTrys {
  public static void main(String[] args) {
    // Here, numer is longer than denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };
    try { // outer try
      for(int i=0; i<numer.length; i++) {
        try { // nested try
          System.out.println(numer[i] + " / " +
                             denom[i] + " is " +
                             numer[i]/denom[i]);
        }
        catch (ArithmeticException exc) {
          // catch the exception
          System.out.println("Can't divide by Zero!");
        }
      }
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // catch the exception
      System.out.println("No matching element found.");
      System.out.println("Fatal error - program terminated.");
    }
  }
}
```

OUTPUT:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error - program terminated.
```

THROWING AN EXCEPTION:

- ✓ It is possible to manually throw an exception explicitly, by using the **throw** statement.
- ✓ The general form of **throw** is shown here:

  throw *exceptionobject*;

Here, *exceptionobject* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

- ✓ The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.

```
// Manually throw an exception.
class ThrowDemo {
  public static void main(String[] args) {
    try {
      System.out.println("Before throw.");
      throw new ArithmeticException();
    }
    catch (ArithmeticException exc) {
      System.out.println("Exception caught.");
    }
    System.out.println("After try/catch block.");
  }
}
```

OUTPUT:

```
Before throw.
Exception caught.
After try/catch block.
```

Rethrowing an Exception:

- ✓ An exception caught by one catch can be rethrown so that it can be caught by an outer catch. The most likely reason for rethrowing this way is to allow multiple handlers access to the exception.
- ✓ To rethrow an exception use a **throw** statement inside a **catch** clause, throwing the exception passed as an argument.

```
// Rethrow an exception.
class Rethrow {
  public static void genException() {
    // here, numer is longer than denom
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };
    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +denom[i] + " is " + numer[i]/denom[i]);
      }
      catch (ArithmeticException exc) {
        // catch the exception
        System.out.println("Can't divide by Zero!");
      }
      catch (ArrayIndexOutOfBoundsException exc) {
        System.out.println("No matching element found.");
        throw exc; // rethrow the exception
      }
    }
  }
}
```

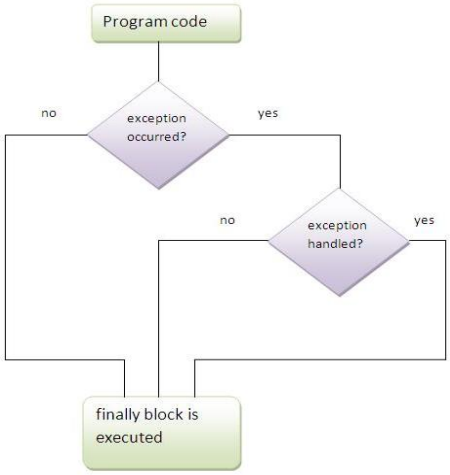
```
class RethrowDemo {
  public static void main(String[] args) {
    try {
      Rethrow.genException();
    }
    catch(ArrayIndexOutOfBoundsException exc) {
      // recatch exception
      System.out.println("Fatal error - " +
                         "program terminated.");
    }
  }
}
```

OUTPUT:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error - program terminated.
```

Using finally:

- ✓ **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- ✓ The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- ✓ Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- ✓ The **finally** clause is optional.
- ✓ **finally** block in java can be used to put "cleanup" code such as closing a file, closing connection etc
- ✓ ***Rule: For each try block there can be zero or more catch blocks, but only one finally block.***
- ✓ ***Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).***



Usage of Java finally

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

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
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:

```
5
finally block is always executed
rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{
  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:

```
finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...
```

Using Throws:

- ✓ If a method generates an exception that it does not handle, it must specify that exception in a throws clause.
- ✓ A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- ✓ All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.
- ✓ This is the general form of a method declaration that includes a **throws** clause:
- ✓ *type method-name(parameter-list)* throws *exception-list*

```
{
    // body of method
}
```

- ✓ Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.
- ✓ Which exception should be declared

  Ans) checked exception only, because:

  unchecked Exception: under your control so correct your code.

  error: beyond your control e.g. you 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

✓ Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
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...

✓ ***Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.***

There are two cases:

✓ **Case1:** You caught the exception i.e. handle the exception using try/catch.
✓ **Case2:** You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

✓ In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

Output:
exception handled
normal flow...

Case2: You declare the exception

- ✓ A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- ✓ B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:
device operation performed
normal flow...

B) Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:
Runtime Exception

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java's Built-in Exceptions

- ✓ Inside the standard package **java.lang**, Java defines several exception classes.
- ✓ The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- ✓ In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table.
- ✓ Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBound	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

- ✓ Following is the list of Java Checked Exceptions Defined in java.lang.

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

User-Defined Exceptions:

- ✓ We can create our own exceptions easily by making them a subclass of an existing exception, such as Exception, the superclass of all exceptions. In a subclass of Exception, there are only two methods you might want to override: Exception () with no arguments and Exception () with a String as an argument. In the latter, the string should be a message describing the error that has occurred.

```
import java.util.Scanner;
class InvalidAgeException extends Exception{
        InvalidAgeException(String s){
                super(s);
        }
        public String toString(){
                return "age is less than 18. not eligible to vote";
        }
 }
class Validateage {
        static void validate(int age)throws InvalidAgeException{
            if(age<18)
                 throw new InvalidAgeException("not valid");
            else
                 System.out.println("welcome to vote");
        }
}
public class UDExcepiton {
        public static void main(String args[]){
                Scanner s=new Scanner(System.in);
                System.out.println("Enter age");
                int age=s.nextInt();
                try{
                        Validateage.validate(age);
                }
                catch(Exception m){
                        System.out.println("Exception occured: "+m);
                }
        }
}
```

I/O:

- ✓ Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- ✓ All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.
- ✓ Java implements streams within class hierarchies defined in the **java.io** package.

Byte Streams and Character Streams

- ✓ Java defines two types of streams: byte and character.
- ✓ *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.
- ✓ *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.
- ✓ At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

The Byte Stream Classes

- ✓ Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in **java.io** are shown in Table

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

The Character Stream Classes

- ✓ Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams.
- ✓ Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in Table .

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Predefined Streams

- ✓ **java.lang** package defines a class called **System**, which encapsulates several aspects of the runtime environment.
- ✓ **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.
- ✓ **System.out** refers to the standard output stream. By default, this is the console.
- ✓ **System.in** refers to standard input, which is the keyboard by default.
- ✓ **System.err** refers to the standard error stream, which also is the console by default.
- ✓ **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they are typically used to read and write characters from and to the console.

Using Byte streams:

Reading Console Input

- ✓ **InputStream,** defines only one input method, **read(),** which reads bytes. There are three versions of read()
  int read() throws IOException
  int read(byte[] buffer) throws IOException
  int read(byte[] buffer, int offset, int numBytes) throws IOException
- ✓ The first version read a single character from the keyboard. It returns -1 when the end of stream is encountered.
- ✓ The second version reads bytes from the input stream and puts them into buffer until either the array is full, the end of stream is reached or an error occurs. It returns the number of bytes read or -1 when the end of stream is encountered.
- ✓ The third version reads input into buffer beginning at location specified by offset upto numBytes bytes are stored. It returns the number of bytes read or -1 when the end of stream is encountered.

```
import java.io.*;

class ReadBytes {
  public static void main(String[] args) throws IOException {
     byte[] data = new byte[10];

     System.out.println("Enter some characters.");
     int numRead = System.in.read(data);
     System.out.print("You entered: ");
     for(int i=0; i < numRead; i++)
       System.out.print((char) data[i]);
  }
}
```

Writing Console Output

- ✓ Console output is most easily accomplished with **print( )** and **println( ).** These methods are defined by the class PrintStream (which is the type of object referenced by System.out).
- ✓ Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here:

  void write(int *b*)
- ✓ This method writes the byte specified by *b*. Although *b* is declared as an integer, only the low-order eight bits are written.

```
// Demonstrate System.out.write().
class WriteDemo {
  public static void main(String[] args) {
    int b;

    b = 'X';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

READING AND WRITING FILES USING BYTE STREAMS:

- ✓ Java provides a number of classes and methods that allow you to read and write files.
- ✓ Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.
  FileInputStream(String *fileName*) throws FileNotFoundException
  FileOutputStream(String *fileName*) throws FileNotFoundException
- ✓ Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown.
- ✓ **FileNotFoundException** is a subclass of **IOException**.
- ✓ When an output file is opened, any preexisting file by the same name is destroyed.
- ✓ To read from a file, you can use a version of **read( )** that is defined within **FileInputStream**.
  int read( ) throws IOException
- ✓ Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read( )** returns –1 when the end of the file is encountered. It can throw an **IOException**.
- ✓ When you are done with a file, you must close it. This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:
  void close( ) throws IOException
- ✓ Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in "memory leaks" because of unused resources remaining allocated.

/* Display a text file.
To use this program, specify the name of the file that you want to see. For example, to see a file called TEST.TXT, use the following command line. java ShowFile TEST.TXT*/

```
import java.io.*;
class ShowFile {
  public static void main(String[] args)
  {
    int i;
    FileInputStream fin;
    // First make sure that a file has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile File");
      return;
    }
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException exc) {
      System.out.println("File Not Found");
      return;
    }
    try {
      // read bytes until EOF is encountered
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(IOException exc) {
      System.out.println("Error reading file.");
    }
    try {
      fin.close();
    } catch(IOException exc) {
      System.out.println("Error closing file.");
    }
  }
}
```

- ✓ In the above code **close( )** can be written within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file.

- ✓ Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file.

```
/* This variation wraps the code that opens and accesses the file within a single try block.
   The file is closed by the finally block. */

import java.io.*;

class ShowFile {
  public static void main(String[] args)
  {
    int i;
    FileInputStream fin = null;

    // First, confirm that a file name has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // The following code opens a file, reads characters until EOF
    // is encountered, and then closes the file via a finally block.
    try {
      fin = new FileInputStream(args[0]);

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(FileNotFoundException exc) {
      System.out.println("File Not Found.");
    } catch(IOException exc) {
      System.out.println("An I/O Error Occurred");
    } finally {
      // Close file in all cases.
      try {
        if(fin != null) fin.close();
      } catch(IOException exc) {
        System.out.println("Error Closing File");
      }
    }
  }
}
```

Writing to a file:

- ✓ To write to a file, you can use the **write( )** method defined by **FileOutputStream**.

  void write(int *b*) throws IOException

- ✓ This method writes the byte specified by *b* to the file. Although *b* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown.

/* Copy a text file. To use this program, specify the name of the source file and the destination file. For example, to copy a file called FIRST.TXT to a file called SECOND.TXT, use the following command line. java CopyFile FIRST.TXT SECOND.TXT */

```
import java.io.*;

class CopyFile {
  public static void main(String[] args)
  {
    int i;
    FileInputStream fin = null;
    FileOutputStream fout = null;

    // First, make sure that both files has been specified.
    if(args.length != 2) {
      System.out.println("Usage: CopyFile from to");
      return;
    }

    // Copy a File.
    try {
      // Attempt to open the files.
      fin = new FileInputStream(args[0]);
      fout = new FileOutputStream(args[1]);

      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);

    } catch(IOException exc) {
      System.out.println("I/O Error: " + exc);
    } finally {
      try {
        if(fin != null) fin.close();
      } catch(IOException exc) {
        System.out.println("Error Closing Input File");
      }
      try {
        if(fout != null) fout.close();
      } catch(IOException exc) {
        System.out.println("Error Closing Output File");
      }
    }
  }
}
```

AUTOMATICALLY CLOSING A FILE:

- ✓ JDK 7 adds a new feature to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short.
- ✓ Automatic resource management is based on an expanded form of the **try** statement.

Here is its general form:

```
try (resource-specification) {
        // use the resource
}
```

- ✓ Here, *resource-specification* is a statement that declares and initializes a resource, such as a file stream.
- ✓ It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released.
- ✓ The **try**-with-resources statement can be used only with those resources that implement the **AutoCloseable** interface defined by **java.lang**. This interface defines the **close( )** method.
- ✓ **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try**-with-resources can be used when working with streams, including file streams.

```
/* This version of the ShowFile program uses a try-with-resources
   statement to automatically close a file when it is no longer needed.

   Note: This code requires JDK 7 or later.
*/

import java.io.*;

class ShowFile {
  public static void main(String[] args)
  {
    int i;

    // First, make sure that a file name has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // The following code uses try-with-resources to open a file
    // and then automatically close it when the try block is left.
    try(FileInputStream fin = new FileInputStream(args[0])) {

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    }
    catch(IOException exc) {
      System.out.println("I/O Error: " + exc);
    }
  }
}
```

- ✓ We can manage more than one resource within a single **try** statement. To do so, simply separate each resource specification with a semicolon.

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
```

Reading and writing binary data

- ✓ To read and write binary values of java primitive types we can use **DataInputStream** and **DataOutputStream** .
- ✓ **DataOutputStream** implements the **DataOutput** interface. This interface defines methods that write all of Java's primitive types to a file. It is important to understand that this data is written using its internal, binary format.
- ✓ Output methods defined by DataOutputStream.

Output Method	Purpose
void writeBoolean(boolean val)	Writes the boolean specified by val
void writeByte(int val)	Writes the lower-order byte specified by val
void writeChar(int val)	Writes the value specified by val as a char
void writeDouble(double val)	Writes the double specified by val
void writeFloat(float val)	Writes the float specified by val
void writeInt(int val)	Writes the int specified by val
void writeLong(long val)	Writes the long specified by val
void writeShort(int val)	Writes the value specified by val as a short

- ✓ The constructor for DataOutputStream is :
  DataOutputStream(OutputStream outputstream)
  Here outputstream is the stream to which data is written. To write output to a file we can use the object created by **FileOutputStream.**
- ✓ **DataInputStream** implements the DataInput interface which provides methods for reading all of java's primitives

Input Method	Purpose
boolean readBoolean()	Reads a boolean
byte readByte()	Reads a byte
char readChar()	Reads a char
double ReadDouble()	Reads a double
float readFloat()	Reads a float
int readInt()	Reads an int
long readLong()	Reads a long
short readShort()	Reads a short

- ✓ The constructor for DataInputStream is:
  DataInputStream(Inputstream inputstream)
  Here inputstream is the stream that is linked to the instance of **DataInputStream** being created. To read input from a file we can use the object created by **FileInputStream**

```
// Write and then read back binary data.
// This code requires JDK 7 or later.

import java.io.*;
class RWData {
  public static void main(String[] args)
  {
    int i = 10;
    double d = 1023.56;
    boolean b = true;
    // Write some values.
    try (DataOutputStream dataOut =
            new DataOutputStream(new FileOutputStream("testdata")))
    {
      System.out.println("Writing " + i);
      dataOut.writeInt(i);

      System.out.println("Writing " + d);
      dataOut.writeDouble(d);
```

```
        System.out.println("Writing " + b);
        dataOut.
        writeBoolean(b);

        System.out.println("Writing " +12.2 * 7.4);
        dataOut.writeDouble(12.2 * 7.4);
      }
      catch(IOException exc) {
        System.out.println("Write error.");
        return;
      }
      System.out.println();
      / / Now, read them back.
      try (DataInputStream dataIn =
              new DataInputStream(new FileInputStream("testdata")))
      {
        i = dataIn.readInt();
        System.out.println("Reading " + i);
        d = dataIn.readDouble();
        System.out.println("Reading " + d);

        b = dataIn.readBoolean();
        System.out.println("Reading " + b);

        d = dataIn.readDouble();
        System.out.println("Reading " + d);
      }
      catch(IOException exc) {
        System.out.println("Read error.");
      }
    }
  }
```

OUTPUT:

```
Writing 10
Writing 1023.56
Writing true
Writing 90.28

Reading 10
Reading 1023.56
Reading true
Reading 90.28
```

RANDOM ACCESS FILES:

- ✓ To access the contents of a file in random order we can use **RandomAccessFile.**
- ✓ **RandomAccessFile** implements the interface **DataInput** and **DataOutput.**
  RandomAccessFile(String filename, String access) throws FileNotFoundException
- ✓ Here, the name of the file is passed in filename, and access determines what type of file access is permitted. "r"- the file can be read but not written;
  "rw"- the file is opened in read-write mode
- ✓ The method seek() is used to set current position of the file pointer within the file:
  void seek(long newPos) throws IOException
- ✓ Here newPos specifies the new position, in bytes, of the file pointer from the beginning of the file.

```
import java.io.*;

class Random{
  public static void main(String[] args)
  {
      int d;
     / / Open and use a random access file.
     try (RandomAccessFile raf = new RandomAccessFile("knr.txt", "rw"))
     {
              System.out.println("Every fifth characters in the file is : ");
              int i=0;
              do {
                       raf.seek(i);
                       d = raf.read();
                       if(d!=-1)
                       System.out.print((char)d + " ");
                       i=i+5;
              }while(d!=-1);
     }
     catch(IOException exc) {
              System.out.println("I/O Error: " + exc);
     }
  }
}
```

For execution of this program assume that there exists a file called knr.txt which contains alphabets a-z.
When the above program is executed we get the following output;

```
E:\java>javac Random.java

E:\java>java Random
Every fifth characters in the file is :
a f k p u z
```

FILE I/O USING CHARACTER STREAMS:

- ✓ To perform character based file I/O we can use **FileReader** and **FileWriter** classes.

Using FileWriter

- ✓ FileWriter creates a Writer that you can use to write to a file. Two of its most commonly used constructors are shown here:

  FileWriter(String *filePath*) throws IOException

  FileWriter(String *filePath*, boolean *append*) throws IOException
- ✓ FileWriter is derived from OutputStreamWriter and Writer. Thus it has access to the methods defined by those classes.
- ✓ Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object.

```
import java.io.*;
class KtoD{
    public static void main(String[] args){
            String str;
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
            System.out.println("enter test.stop to quit");
            try(FileWriter fw=new FileWriter("test.txt")){
                    do{
                            System.out.print(":");
                            str=br.readLine();
                            if (str.compareTo("stop")==0)  break;
                            str=str+"\r\n";
                            fw.write(str);
                    }while(str.compareTo("stop")!=0);
            }
            catch(IOException exe){
                    System.out.println("I/O Error: " +exe);
            }
    }
}
```

Using a FileReader:

- ✓ The **FileReader** class creates a **Reader** that you can use to read the contents of a file. A commonly used constructors is shown here:

  FileReader(String *filePath*) throws FileNotFoundEception
- ✓ FileReader is derived from InputStreamReader and Reader. Thus, it has access to the methods defined by those classes.

```
import java.io.*;
class DtoS{
    public static void main(String[] args){
            String str;
            try(BufferedReader br=new BufferedReader(new FileReader("test.txt"))){
                    while((str=br.readLine())!=null){
                            System.out.println(str);
                    }
            }
            catch(IOException exe){
                    System.out.println("I/O Error: " +exe);
            }
    }
}
```

File:

- ✓ **File** class deals directly with files and file systems. The **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.
- ✓ A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

File (String path)
File (String directory path, String filename)

Obtaining a File's properties:

- ✓ **File** defines many methods that obtain the standard properties of a **File** object.

Method	Description
boolean canRead()	Returns true if the file can be read
boolean canWrite()	Returns true if the file can be written
boolean exists()	Returns true if the file exists
String getAbsolutePath()	Returns the absolute path to the file
String getName()	Returns the file's name
String getParent()	Returns the name of the file's parent directory, or null if no parent exists
boolean isAbsolute()	Returns true if the path is absolute. It returns false if the path is relative
boolean isDirectory()	Returns true if the file is directory
boolean isFile()	Returns true if the file is a normal file. It returns false if the file is a directory or some other nonfile object.
boolean isHidden()	Returns true if the invoking file is hidden. Returns false otherwise
long length()	Returns the length of the file, in bytes.

// Obtain information about a file.

```
import java.io.*;
class FileDemo {
  public static void main(String[] args) {
    File myFile = new File("/pack/k.txt");
    System.out.println("File Name: " + myFile.getName());
    System.out.println("Path: " + myFile.getPath());
    System.out.println("Abs Path: " + myFile.getAbsolutePath());
    System.out.println("Parent: " + myFile.getParent());
    System.out.println(myFile.exists() ? "exists" : "does not exist");
    System.out.println(myFile.isHidden() ? "is hidden":  "is not hidden");
    System.out.println(myFile.canWrite() ? "is writeable": "is not writeable");
    System.out.println(myFile.canRead() ? "is readable": "is not readable");
    System.out.println("is " + (myFile.isDirectory() ? "" : "not" + " a directory"));
    System.out.println(myFile.isFile() ? "is normal file" : "might be a named pipe");
    System.out.println(myFile.isAbsolute() ? "is absolute" : "is not absolute");
    System.out.println("File size: " + myFile.length() + " Bytes");
  }
}
```

Obtaining a Directory Listing:

- ✓ A directory is a file that contains a list of other files and directories. When we create a File object that is a directory, the **isDirectory()** method will return true.
- ✓ The list of the files in the directory can be obtained by calling list() on the object.

String[] list()

```
// Using directories.
import java.io.*;
class DirList {
  public static void main(String[] args) {
    String dirname = "/java";
    File myDir = new File(dirname);
    if (myDir.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String[] s = myDir.list();
```

```
        for (int i=0; i < s.length; i++) {
          File f = new File(dirname + "/" + s[i]);
          if (f.isDirectory()) {
            System.out.println(s[i] + " is a directory");
          } else {
            System.out.println(s[i] + " is a file");
          }
        }
      } else {
        System.out.println(dirname + " is not a directory");
      }
    }
  }
```

The listFiles( )Alternative

- ✓ There is a variation to the **list( )** method, called **listFiles( )**. The signatures for **listFiles( )** are shown here:
  File[ ] listFiles( )
  File[ ] listFiles(FilenameFilter *FFObj*)
  File[ ] listFiles(FileFilter *FFObj*)
- ✓ These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FilenameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles()** work like their equivalent **list( )** methods.
  The third version of **listFiles( )** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:
  boolean accept(File *path*)
  The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument) and **false** for those that should be excluded.

Various File Utility methods:

- ✓ **File** includes several other utility methods.

Method	Description
boolean delete()	Deletes the file specified by the invoking object. returns true if the file was deleted and false if the file cannot be removed
void deleteOnExit()	removes the file associated with the invoking object when the java virtual machine terminates
long getFreeSpace()	Returns the number of free bytes of storage available on the partition associated with the invoking object.
boolean mkdir()	Creates the directory specified by the invoking object. Returns true if the directory was created and false if the directory could not be created.
boolean mkdirs()	Creates the directory and all required parent directories specified by the invoking object. Returns true if the entire path was created and false otherwise.
boolean setReadOnly()	set the file read-only
boolean setWritible(boolean how)	If how is true, the file is set to writable. If how is false, the file is set to read only. Returns true if the status of the file was modified and false if the write status cannot be changed.

WRAPPERS

- ✓ Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object.
- ✓ The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Wrapper	Conversion Method
Double	static double parseDouble(string str) throws NumberFormatException
Float	static float parseFloat(String str)throws NumberFormatException
Long	static long parseLong(String str)throws NumberFormatException
Wrapper	Conversion Method
Integer	static int parseInt(String str)throws NumberFormatException
Short	static short parseShort(String str)throws NumberFormatException
Byte	static byte parseByte(String str)throws NumberFormatException

/ / This program averages a list of numbers entered by the user.

```
import java.io.*;

class AvgNums {
  public static void main(String[] args) throws IOException {
    / / create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String str;
    int n;
    double sum = 0.0;
    double avg, t;

    System.out.print("How many numbers will you enter: ");
    str = br.readLine();
    try {
      n = Integer.parseInt(str);
    }
    catch(NumberFormatException exc) {
      System.out.println("Invalid format");
      n = 0;
    }

    System.out.println("Enter " + n + " values.");
    for(int i=0; i < n ; i++) {
      System.out.print(": ");
      str = br.readLine();
      try {
        t = Double.parseDouble(str);
      } catch(NumberFormatException exc) {
        System.out.println("Invalid format");
        t = 0.0;
      }
      sum += t;
    }
    avg = sum / n;
    System.out.println("Average is " + avg);
  }
}
```