

---

*This covers basic java concepts and syntax + some portions of “Advanced Java Programming” –Unit: 1 (BSc. CSIT, TU)*

---

## Exception Handling in Java

- An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time that disrupts the normal flow of instructions.  
*An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.*
- There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- Exceptions hinder normal execution of program. **Exception handling** is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the program. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we should include exception handling techniques in our code.

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

The **exception handling** in java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

### Why an exception occurs?

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an 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.

### Error vs Exception

**Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error. Errors are mostly caused by the environment in which application is running. Example: *StackOverflowError*, *OutOfMemoryError*

**Exceptions** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:

*ArithmeticException* – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

*ArrayIndexOutOfBoundsException* – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

*Recovering from Error is not possible. The only solution to errors is to terminate the execution.*

*Where as you can recover from Exception by using either try-catch blocks or throwing exception back to caller*

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. *NullPointerException* is an example of such an exception. Another branch, **Error** are used by the Java run-time system (JVM) to indicate errors having to do with the run-time environment itself (JRE). **StackOverflowError** is an example of such an error.

## Exception-Handling Fundamentals

- A 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. That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

## Exception Types

There are mainly two types of exceptions: **checked** and **unchecked**. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

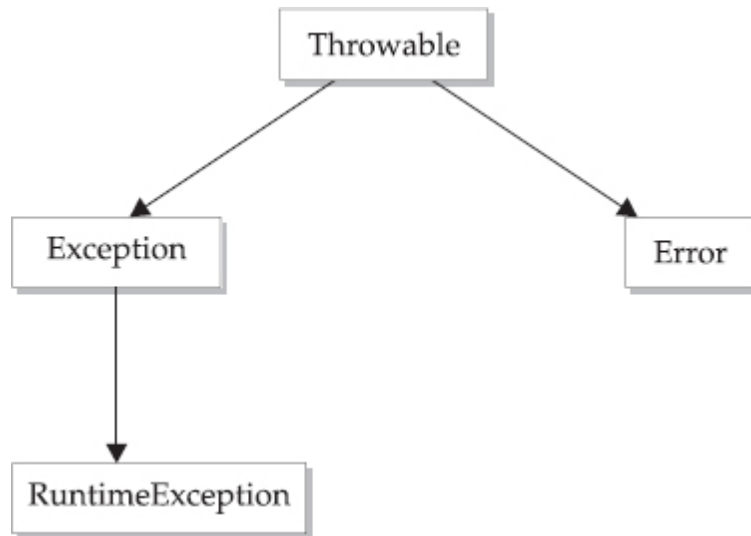
1. Checked Exception
  2. Unchecked Exception
  3. Error
- ◆ **Checked Exceptions** are those exceptions which are checked by the compiler during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, we will get compilation error. For example, **SQLException**, **IOException**, **ClassNotFoundException** etc.

The classes which directly inherit **Throwable** class except **RuntimeException** and **Error** fall under checked exceptions

- ◆ **Unchecked Exceptions** are those exceptions which are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, **ArithmeticException**, **NullPointerException**, **ArrayIndexOutOfBoundsException** etc.

The classes which inherit **RuntimeException** are unchecked exceptions e.g. Unchecked exceptions are not checked at compile-time, but they are checked at runtime Unchecked Exceptions are also known as Runtime Exceptions

- ◆ **Errors** represent serious and usually irrecoverable conditions like a library incompatibility, infinite recursion, or memory leaks. Examples: **OutOfMemoryError**, **VirtualMachineError**, **AssertionError** etc



- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment,

## Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

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.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero  
    at Exc0.main(Exc0.java:4)
```

### How JVM handle an Exception?

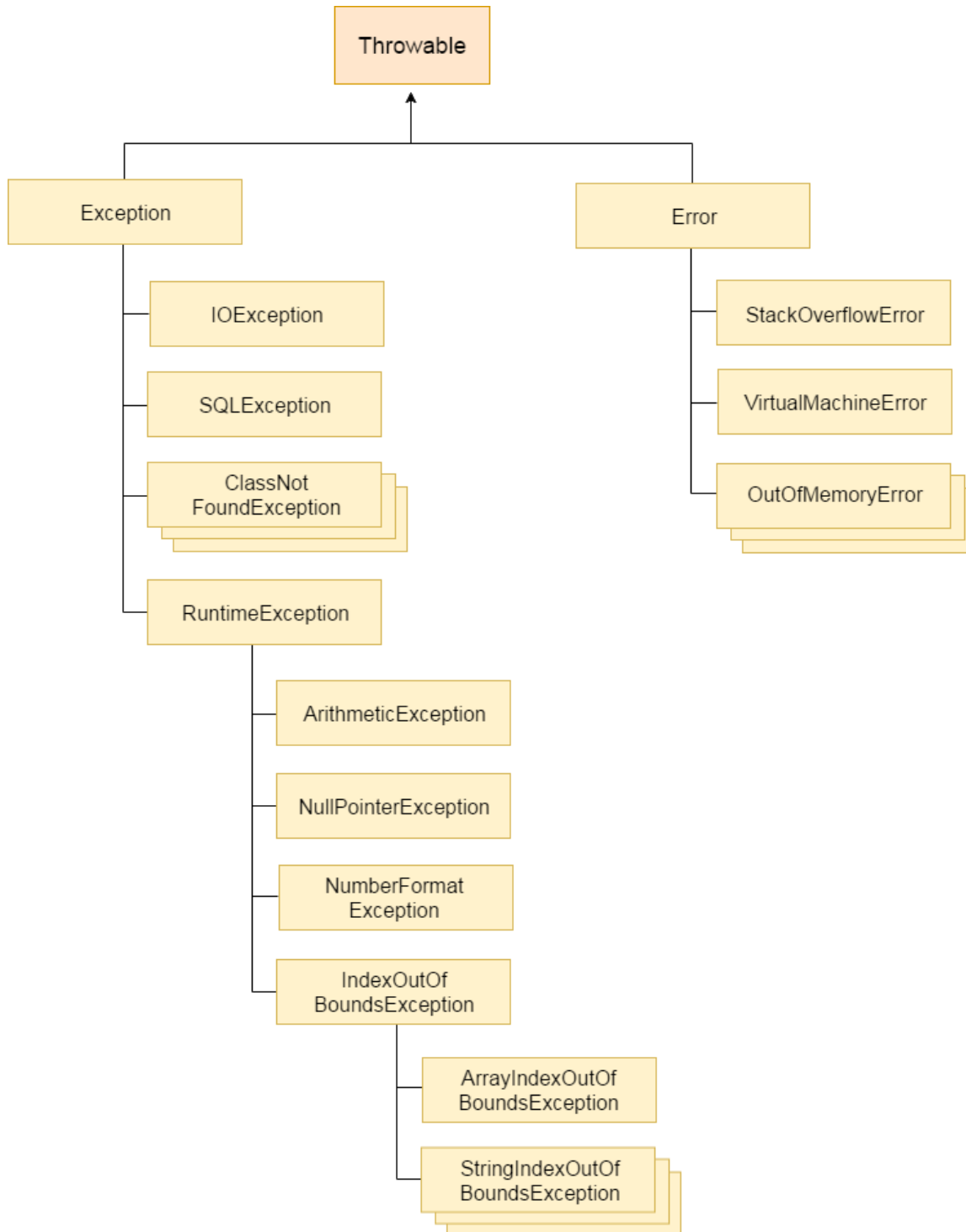
**Default Exception Handling :** Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handler**, which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

```
Exception in thread "xxx" Name of Exception : Description
... .. // Call Stack
```

### How Programmer handles an exception?

**Customized (User Defined) Exception Handling :** Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time 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.

**Hierarchy of Java Exception classes**

## Exceptions Methods

Following is the list of important methods available in the Throwable class.

Sr.No.	Method & Description
1	<code>public String getMessage()</code>  Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	<code>public Throwable getCause()</code>  Returns the cause of the exception as represented by a Throwable object.
3	<code>public String toString()</code>  Returns the name of the class concatenated with the result of <code>getMessage()</code> .
4	<code>public void printStackTrace()</code>  Prints the result of <code>toString()</code> along with the stack trace to <code>System.err</code> , the error output stream.
5	<code>public StackTraceElement [] getStackTrace()</code>  Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	<code>public Throwable fillInStackTrace()</code>  Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.



## Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

### Syntax:

```
try {
    //statements that may cause an exception
} catch (ExceptionType name) {
    //exception handling code
}
```

The following program includes a try block and a catch clause that processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }

        System.out.println("After catch statement.");
    }
}
```

## Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try / catch block.

### Syntax:

```

try {
//statements that may cause an exception
} catch (ExceptionType1 name1) {
//exception handling code for type ExceptionType1
} catch (ExceptionType2 name2) {
//exception handling code for type ExceptionType1
}

```

Here two catch blocks are shown, but you can have any number of them after a single try. If an exception occurs in the protected code (inside try), the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

The following example traps two different exception types:

```

// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}

```

## Nested try Statements

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.

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.

```
try
{
    statement1;
    statement2;
    ...
    try
    {
        statement1;
        statement2;
        . . .
    }
    catch(Exception e)
    {
        // Exception handling code
    }
}
catch(Exception e)
{
    //Exception handling code
}
```

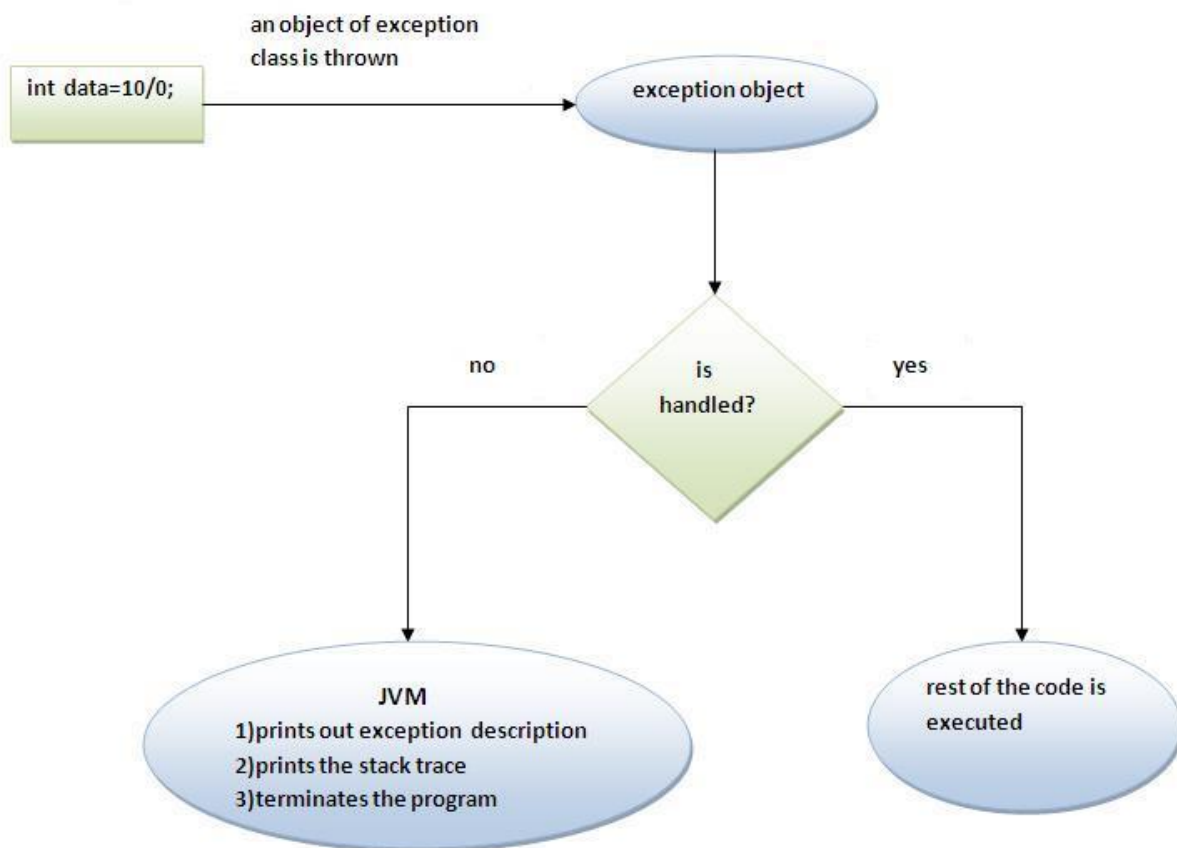
```
class NestedTryDemo{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b =39/0;
            }catch(ArithmeticException e){
                System.out.println(e);
            }
            try{
                int a[]=new int[5];
                a[5]=4;
            }catch(ArrayIndexOutOfBoundsException e){
                System.out.println(e);
            }
            System.out.println("other statement");
        }catch(Exception e){
            System.out.println("handeled");
        }
        System.out.println("normal flow..");
    }
}
```

**Internal working of java try-catch block**

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.



*Figure : Internal working of Java Try-Catch blocks*

**throw**

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

Here, **ThrowableInstance** 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**.

There are two ways you can obtain a **Throwable** object:

- **Using a parameter in a catch clause** or
- **Creating one with the new operator**.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

- ✓ The Java **throw** keyword is used to explicitly throw an exception.
- ✓ We can throw either checked or unchecked exception in java by **throw** keyword.
- ✓ The **throw** keyword is mainly used to throw custom exception.
- ✓ Only object of **Throwable** class or its sub classes can be thrown.
- ✓ Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. Here is the output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

**throws**

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration.
- **throws** keyword is used to declare that a method may throw one or more exceptions. The caller must catch the exceptions.
- 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.

**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.**
  - In case you declare the exception, if exception does not occur, the code will be executed fine.
  - In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile



```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne()** throws **IllegalAccessException**. Second, **main()** must define a **try /catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Output:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

### Advantage of Java throws keyword

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

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...");
    }
}

```

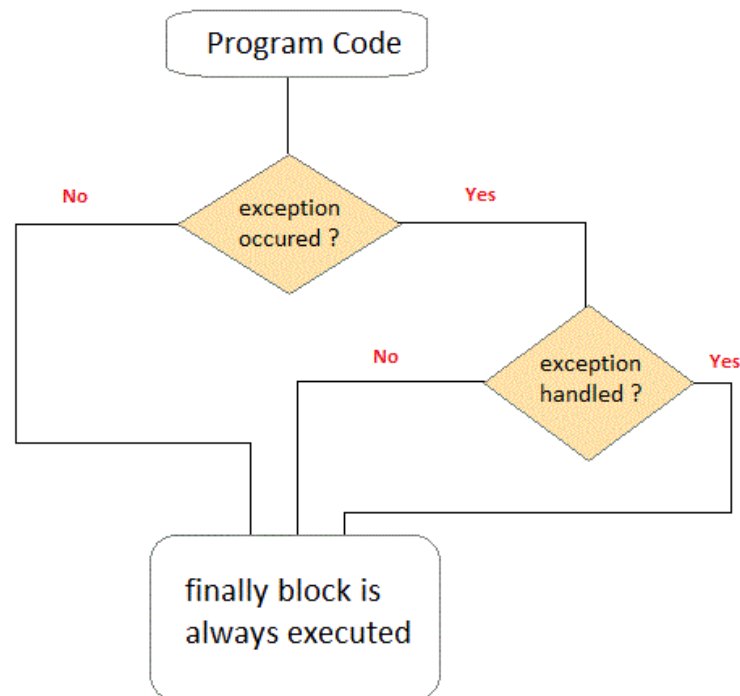
### Difference between throw and throws in Java

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

S.No.	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.

**finally**

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.
- **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.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- **finally** block in java can be used to put "cleanup" code such as closing a file, closing connection etc.
- The **finally** clause is optional. However, *each try statement requires at least one catch or a finally clause.*



- If you don't handle exception, before terminating the program, JVM executes finally block(if any)
- 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 program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).
- A finally block appears at the end of the catch blocks (or after try block if catch is not present, and has the following syntax –

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}finally {  
    // The finally block always executes.  
}
```

Example:

```
public class ExcepTest {  
    public static void main(String args[]) {  
        int a[] = new int[2];  
        try {  
            System.out.println("Access element three :" + a[3]);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception thrown  :" + e);  
        }finally {  
            a[0] = 6;  
            System.out.println("First element value: " + a[0]);  
            System.out.println("The finally statement is  
executed");  
        }  
    }  
}
```

**Difference between final, finally and finalize**

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

S.No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

**Java's Built-in Exceptions**

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's throws list. 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 **checked** exceptions defined by **java.lang** must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions.

**List of Java's built in Exceptions****Java Unchecked Exceptions**

**Java's Unchecked RuntimeException Subclasses defined in java.lang:**

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



## Java Checked Exceptions

### Java's Checked Exceptions Defined in java.lang:

Exception	Description
<b>ClassNotFoundException</b>	Class not found
<b>CloneNotSupportedException</b>	Attempt to clone an object that doesn't implement the <b>Cloneable</b> interface
<b>IllegalAccessException</b>	Access to a class is denied
<b>InstantiationException</b>	Attempt to create an object of an abstract class or interface
<b>InterruptedException</b>	One thread has been interrupted by another thread
<b>NoSuchFieldException</b>	A requested field does not exist
<b>NoSuchMethodException</b>	A requested method doesn't exist
<b>ReflectiveOperationException</b>	Superclass of reflection-related exceptions

## Creating Your Own Exception Subclasses [Custom Exceptions]

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**.
- Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.
- If you are creating your own Exception that is known as **custom exception** or **user-defined exception**. Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.
  - ✎ All exceptions must be a child of **Throwable**.
  - ✎ If you want to write a checked exception you need to extend the **Exception** class.
  - ✎ If you want to write a runtime exception, you need to extend the **RuntimeException** class.

### Example :

```
// File: InvalidAgeException.java
package examples;
public class InvalidAgeException extends Exception{
    public InvalidAgeException(String s)
    {
        // Calli constructor of parent Exception
        super(s);
    }
}
```

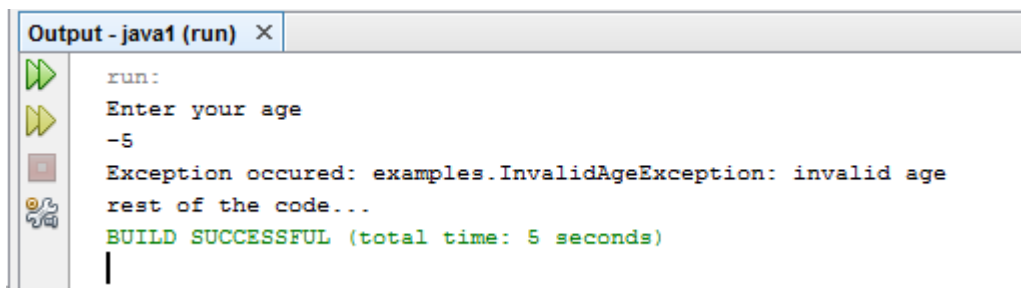
-----

**// File: TestCustomException.java**

```
package examples;
import java.util.Scanner;

/**
 *
 * @author BIPIN
 */
public class TestCustomException {
    static void validate(int age)throws
    InvalidAgeException{
        if(age<0||age>150)
            throw new InvalidAgeException("invalid age ");
        else
            System.out.println("Welcome!");
    }

    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your age");
        int age = sc.nextInt();
        try{
            validate(age);
        }catch(Exception m){
            System.out.println("Exception occurred: "+m);
        }
        System.out.println("rest of the code...");
    }
}
```



```
Output - java1 (run) X
run:
Enter your age
-5
Exception occurred: examples.InvalidAgeException: invalid age
rest of the code...
BUILD SUCCESSFUL (total time: 5 seconds)
```

## Introducing Nested and Inner Classes

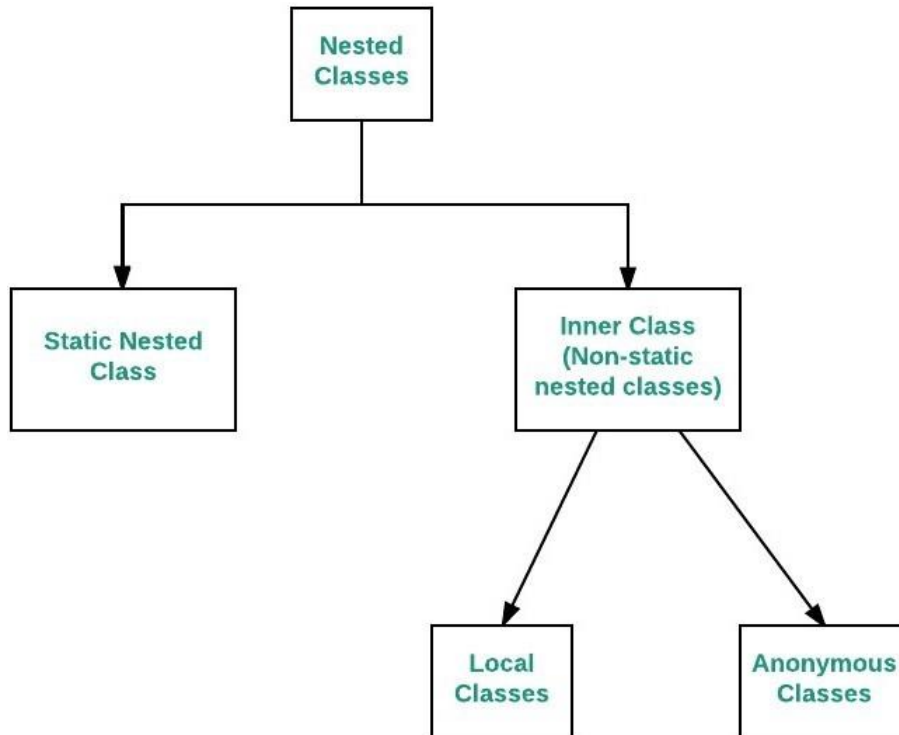
- ◆ It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class.
- ◆ Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested.
- ◆ However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- ◆ It is also possible to declare a nested class that is local to a block.

## Types of Nested classes

There are two types of nested classes: **static** and **non-static**.

A **static nested class** is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the **inner class**. An inner class is a **non-static nested class**. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.



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

**1. Non-static nested class (inner class)**

- 1.1 Member inner class
- 1.2 Anonymous inner class
- 1.3 Local inner class

**2. Static nested class**

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

- ✎ Java inner class or nested class is a class which is declared inside the class or interface.
- ✎ We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- ✎ Additionally, it can access all the members of outer class including private data members and methods.

### Advantages

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

### Difference between nested class and inner class in Java

- ♣ Inner class is a type of nested class. **Non-static nested classes** are known as **inner classes**.

**Examples of nested classes****Example 1: Static Nested class****Syntax:**

```

class Outer_Class_Name {
    static class Nested_class_Name {
    }
}

```

**Code:**

```

-----
public class Outer {
    static class Nested_Demo {
        public void my_method() {
            System.out.println("This is my nested class");
        }
    }

    public static void main(String args[]) {
        Outer.Nested_Demo nested = new Outer.Nested_Demo();
        nested.my_method();
    }
}
-----

```

**Example 2: Non Static Nested Class ( Inner Class)****Syntax:**

```

class Outer_Demo {
    class Inner_Demo {
    }
}

```

**Code:**

```
-----  
class Outer_Demo {  
    int num;  
    // inner class  
    private class Inner_Demo {  
        public void print() {  
            System.out.println("This is an inner class");  
        }  
    }  
}  
  
// Accessing the inner class from the method within  
void display_Inner() {  
    Inner_Demo inner = new Inner_Demo();  
    inner.print();  
}  
}  
  
public class My_class {  
    public static void main(String args[]) {  
        // Instantiating the outer class  
        Outer_Demo outer = new Outer_Demo();  
        // Accessing the display_Inner() method.  
        outer.display_Inner();  
    }  
}  
-----
```



- Inner classes are also used to access the private members of a class
- To instantiate the inner class, initially you have to instantiate the outer class.  
Syntax to do so is as follows :

**Outer\_Class\_Name outer = new Outer\_Class\_Name();**

**Outer\_Class\_Name.Inner\_Class\_Name inner = outer.new Inner\_Class\_Name();**

**//Program to demonstrate accessing the Private Members using inner class**

```
class Outer_Demo {
    // private variable of the outer class
    private int num = 175;
    // inner class
    public class Inner_Demo {
        public int getNum() {
            System.out.println("This is the getnum method of the
inner class");
            return num;
        }
    }
}

public class My_class2 {
    public static void main(String args[]) {
        // Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();
        // Instantiating the inner class
        Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
        System.out.println(inner.getNum());
    }
}
```

**Example 3 : Local inner class example**

- In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

```
public class Outerclass {
    // instance method of the outer class
    void my_Method() {
        int num = 23;

        // method-local inner class
        class MethodInner_Demo {
            public void print() {
                System.out.println("This is method inner class "+num);
            }
        } // end of inner class

        // Accessing the inner class
        MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }
    public static void main(String args[]) {
        Outerclass outer = new Outerclass();
        outer.my_Method();
    }
}
```

**Example 4: Anonymous Inner Class**

- An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows –

```
AnonymousInner an_inner = new AnonymousInner() {
    public void my_method() {
        .....
        .....
    }
};
```

**Example program :**

```
-----
abstract class AnonymousInner {
    public abstract void mymethod();
}

public class Outer_class {

    public static void main(String args[]) {
        AnonymousInner inner = new AnonymousInner() {
            public void mymethod() {
                System.out.println("This is an example of anonymous
inner class");
            }
        };
        inner.mymethod();
    }
}
```

### **Advantage of java inner classes**

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

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

## Java I/O (Input and Output)

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

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

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

### Stream

#### Stream

- A stream is a sequence of data. It's called a stream because it is like a stream of water that continues to flow.
- A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical detail.

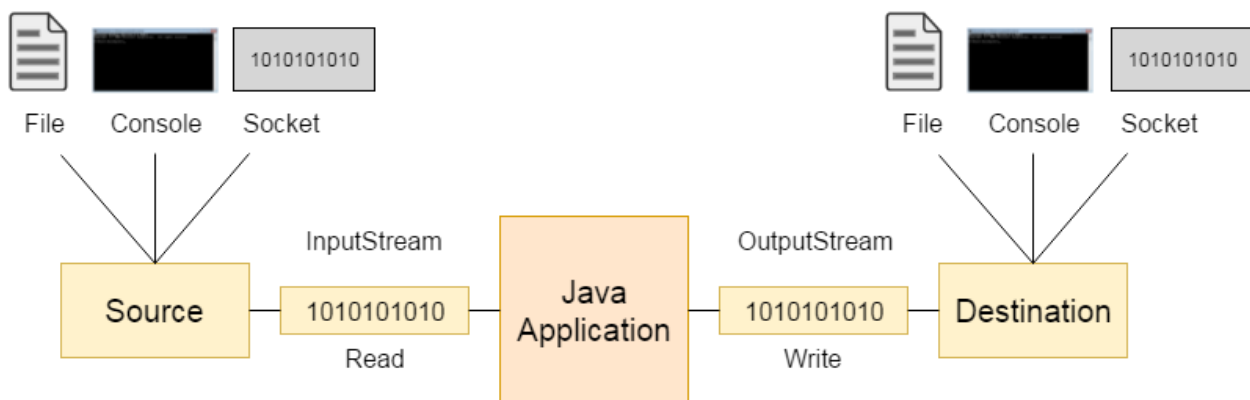
### Output stream vs Input stream

#### OutputStream

- The **Output stream** is used for writing data to a destination.
- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

#### InputStream

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



## Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. Java provides the following three standard streams –

- **Standard Input** – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

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

## File

- Although most of the classes defined by **java.io** operate on streams, the **File** class does not.
- It deals directly with files and the file system. That is, 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
- Java **File** class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc. The **File** object represents the actual file/directory on the disk
- Files are a primary source and destination for data within many programs.
- A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list()** method.
- The following constructors can be used to create **File** objects:

```
File(String directoryPath)
```

```
File(String directoryPath, String filename)
```

```
File(File dirObj, String filename)
```

```
File(URI uriObj)
```

Here, **directoryPath** is the path name of the file; **filename** is the name of the file or subdirectory; **dirObj** is a File object that specifies a directory; and **uriObj** is a URI object that describes a file.

- The following example creates three files: **f1**, **f2**, and **f3**. The first File object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to f1 and a filename; f3 refers to the same file as f2.

```
File f1 = new File("/");
File f2 = new File("/", "autoexec.bat");
File f3 = new File(f1, "autoexec.bat");
```

File defines many methods that obtain the standard properties of a File object. For example, **getName()** returns the name of the file; **getParent()** returns the name of the parent directory; and **exists()** returns true if the file exists, false if it does not. The following example demonstrates several of the **File** methods. It assumes that a directory called “JAVA-I” exists in E drive and it contains a file called “MyFile.txt”

```
package filehandling;

import java.io.File;

public class FileDemo {

    static void printThis(String s) {
        System.out.println(s);
    }

    public static void main(String[] args) {
        File f1 = new File("E:/JAVA-I/MyFile.txt");
        printThis("File Name: "+f1.getName());
        printThis("Path: "+f1.getPath());
        printThis("Parent: "+f1.getParent());
        printThis("Abs Path: "+f1.getAbsolutePath());
        printThis(f1.exists()? "exists": " does not exist");
        printThis(f1.canWrite()? "is writeable": " is not writeable");
        printThis(f1.canRead()? "is readable": " is not readable");
        printThis("is " + (f1.isDirectory()? " ": " not " + " a
        directory"));
```

```

printThis(f1.isFile()? "is normal file" : " might be named
pipe");

printThis(f1.isAbsolute()? "is absolute" : " is not absolute");

    printThis("File last modified :"+f1.lastModified());
    printThis("File size: "+f1.length()+ " Bytes");
}
}

```

```

Output - java1 (run)
run:
File Name: MyFile.txt
Path: E:\JAVA-I\MyFile.txt
Parent: E:\JAVA-I
Abs Path: E:\JAVA-I\MyFile.txt
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified :1550420024904
File size: 5 Bytes
BUILD SUCCESSFUL (total time: 0 seconds)

```

- Most of the **File** methods are self-explanatory. **isFile()** and **isAbsolute()** are not.
- **isFile()** returns true if called on a file and false if called on a directory. Also, **isFile()** returns false for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file.
- The **isAbsolute()** method returns true if the file has an absolute path and false if its path is relative.

**File** includes two useful utility methods of special interest.

- The first is **renameTo()**, shown here:

```
boolean renameTo(File newName)
```



Here, the filename specified by **newName** becomes the new name of the invoking **File** object. It will return true upon success and false if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).

- The second utility method is **delete()**, which deletes the disk file represented by the path of the invoking File object. It is shown here:

**boolean delete( )**

You can also use **delete()** to delete a directory if the directory is empty. **delete()** returns true if it deletes the file and false if the file cannot be removed.

**Here are some other File methods that you will find helpful:**

Sr.No.	Method & Description
1	<a href="#">boolean canExecute()</a> This method tests whether the application can execute the file denoted by this abstract pathname.
2	<a href="#">boolean canRead()</a> This method tests whether the application can read the file denoted by this abstract pathname.
3	<a href="#">boolean canWrite()</a> This method tests whether the application can modify the file denoted by this abstract pathname.
4	<a href="#">int compareTo(File pathname)</a> This method compares two abstract pathnames lexicographically.
5	<a href="#">boolean createNewFile()</a> This method atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
6	<a href="#">static File createTempFile(String prefix, String suffix)</a> This method creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
7	<a href="#">static File createTempFile(String prefix, String suffix, File directory)</a> This method Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
8	<a href="#">boolean delete()</a> This method deletes the file or directory denoted by this abstract pathname.
9	<a href="#">void deleteOnExit()</a> This method requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
10	<a href="#">boolean equals(Object obj)</a> This method tests this abstract pathname for equality with the given object.
11	<a href="#">boolean exists()</a> This method tests whether the file or directory denoted by this abstract pathname exists.
12	<a href="#">File getAbsolutePath()</a>

	This method returns the absolute form of this abstract pathname.
13	<a href="#"><u>String getAbsolutePath()</u></a> This method returns the absolute pathname string of this abstract pathname.
14	<a href="#"><u>File getCanonicalFile()</u></a> This method returns the canonical form of this abstract pathname.
15	<a href="#"><u>String getCanonicalPath()</u></a> This method returns the canonical pathname string of this abstract pathname.
16	<a href="#"><u>long getFreeSpace()</u></a> This method returns the number of unallocated bytes in the partition named by this abstract path name.
17	<a href="#"><u>String getName()</u></a> This method returns the name of the file or directory denoted by this abstract pathname.
18	<a href="#"><u>String getParent()</u></a> This method returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
19	<a href="#"><u>File getParentFile()</u></a> This method returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
20	<a href="#"><u>String getPath()</u></a> This method converts this abstract pathname into a pathname string.
21	<a href="#"><u>long getTotalSpace()</u></a> This method returns the size of the partition named by this abstract pathname.
22	<a href="#"><u>long getUsableSpace()</u></a> This method returns the number of bytes available to this virtual machine on the partition named by this abstract pathname.
23	<a href="#"><u>int hashCode()</u></a> This method computes a hash code for this abstract pathname.
24	<a href="#"><u>boolean isAbsolute()</u></a> This method tests whether this abstract pathname is absolute.
25	<a href="#"><u>boolean isDirectory()</u></a> This method tests whether the file denoted by this abstract pathname is a directory.
26	<a href="#"><u>boolean isFile()</u></a> This method tests whether the file denoted by this abstract pathname is a normal file.
27	<a href="#"><u>boolean isHidden()</u></a> This method tests whether the file named by this abstract pathname is a hidden file.
28	<a href="#"><u>long lastModified()</u></a> This method returns the time that the file denoted by this abstract pathname was last modified.
29	<a href="#"><u>long length()</u></a> This method returns the length of the file denoted by this abstract pathname.
30	<a href="#"><u>String[] list()</u></a> This method returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
31	<a href="#"><u>String[] list(FilenameFilter filter)</u></a> This method returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
32	<a href="#"><u>File[] listFiles()</u></a>

	This method returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
33	<a href="#">File[] listFiles(FileFilter filter)</a> This method returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
34	<a href="#">File[] listFiles(FilenameFilter filter)</a> This method returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
35	<a href="#">static File[] listRoots()</a> This method lists the available filesystem roots.
36	<a href="#">boolean mkdir()</a> This method creates the directory named by this abstract pathname.
37	<a href="#">boolean mkdirs()</a> This method creates the directory named by this abstract pathname, including any necessary but non existent parent directories.
38	<a href="#">boolean renameTo(File dest)</a> This method renames the file denoted by this abstract pathname.
39	<a href="#">boolean setExecutable(boolean executable)</a> This is a convenience method to set the owner's execute permission for this abstract pathname.
40	<a href="#">boolean setExecutable(boolean executable, boolean ownerOnly)</a> This method Sets the owner's or everybody's execute permission for this abstract pathname.
41	<a href="#">boolean setLastModified(long time)</a> This method sets the last-modified time of the file or directory named by this abstract pathname.
42	<a href="#">boolean setReadable(boolean readable)</a> This is a convenience method to set the owner's read permission for this abstract pathname.
43	<a href="#">boolean setReadable(boolean readable, boolean ownerOnly)</a> This method sets the owner's or everybody's read permission for this abstract pathname.
44	<a href="#">boolean setReadOnly()</a> This method marks the file or directory named by this abstract pathname so that only read operations are allowed.
45	<a href="#">boolean setWritable(boolean writable)</a> This is a convenience method to set the owner's write permission for this abstract pathname.
46	<a href="#">boolean setWritable(boolean writable, boolean ownerOnly)</a> This method sets the owner's or everybody's write permission for this abstract pathname.
47	<a href="#">String toString()</a> This method returns the pathname string of this abstract pathname.
48	<a href="#">URI toURI()</a> This method constructs a file : URI that represents this abstract pathname.

## Directories

- ◆ A directory is a **File** that contains a list of other files and directories. When you create a **File** object that is a directory, the **isDirectory()** method will return true. In this case, you

can call **list()** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

```
String[ ] list( )
```

The list of files is returned in an array of String objects. The program shown here illustrates how to use **list()** to examine the contents of a directory:

### Using FilenameFilter

You will often want to limit the number of files returned by the **list()** method to include only those files that match a certain filename pattern, or filter. To do this, you must use a second form of **list()**, shown here:

```
String[ ] list(FilenameFilter FFObj)
```

In this form, **FFObj** is an object of a class that implements the **FilenameFilter** interface.

**FilenameFilter** 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 directory, String filename)
```

The **accept()** method returns true for files in the directory specified by directory that should be included in the list (that is, those that match the filename argument) and returns false for those files that should be excluded.

### The listFiles() Alternative

There is a variation to the **list()** method, called **listFiles()**, which you might find useful. The signatures for **listFiles()** are shown here:

```
File[ ] listFiles( )
```

```
File[ ] listFiles(FilenameFilter FFObj)
```

```
File[ ] listFiles(FileFilter FObj)
```

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.

## Creating Directories

Another two useful **File** utility methods are **mkdir()** and **makedirs()**.

- ✓ The **mkdir()** method creates a directory, returning true on success and false on failure. Failure can occur for various reasons, such as the path specified in the **File** object already exists, or the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the **makedirs()** method. It creates both a directory and all the parents of the directory.

// Example

```
package filehandling;

import java.io.File;
import java.io.FilenameFilter;

public class DirDemo {
    public static void main(String[] args) {
        //Use the mkdir() method to create a single directory
        File d1 = new File("F:/CoreJAVA");
        if (d1.mkdir()) {
            System.out.println("Directory created");
        } else {
            System.out.println("Directory not created");
        }

        //You can also use the mkdirs()method to create nested
        //directories
        File d2 = new
File("F:/Books/ComputerScience/Programming/Java");
        if (d2.mkdirs()) {
            System.out.println("Directories created");
        } else {
            System.out.println("Directories not created");
        }
    }
}
```

```

    //Use the list() method on a Fileobject to list the content
    //of a directory.

    File d3 = new File("E:/JAVA-I");
    String[] content = d3.list();
    for (String s : content) {
        System.out.println(s);
    }
}

//You can supply a FilenameFilter to another overload of the
//list()

//method to list only files and directories that satisfy a
//condition.

File d5 = new File("E:/JAVA-I");
FilenameFilter filter = new FilenameFilter() {
    public boolean accept(File file, String name) {
        return name.endsWith(".txt");
    }
};

String[] cont = d5.list(filter);
for (String entry : cont) {
    System.out.println(entry);
}

//Similarly, the methods listFiles() and
//listFiles(FilenameFilter) do the same, but return an
//array of File objects

File d6 = new File("E:/Deep Learning");
File allfiles[] = d6.listFiles();
for(File f:allfiles) {
    System.out.println(f);
}

```

```

}
}

```

### The **AutoCloseable**, **Closeable**, and **Flushable** Interfaces

- ✓ There are three interfaces that are quite important to the stream classes.
- ✓ They are **AutoCloseable**, **Closeable**, and **Flushable**
- ✓ **Closeable** and **Flushable** are defined in **java.io** and were added by JDK 5. The third, **AutoCloseable**, was added by JDK 7. It is packaged in **java.lang**.

#### The **AutoCloseable** interface

- **AutoCloseable** provides support for the **try-with-resources** statement, which automates the process of closing a resource. Only objects of classes that implement **AutoCloseable** can be managed by **try-with-resources**.
- The **AutoCloseable** interface defines only the **close( )** method:

**void close( ) throws Exception**

This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a **try-with-resources** statement, thus eliminating the need to explicitly call **close( )**. Because this interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a **try-with-resources** statement.

- Automatically closing a stream ensures that it is properly closed when it is no longer needed, thus preventing memory leaks and other problems.

#### The **Closeable** interface

- ❖ The **Closeable** interface also defines the **close( )** method. Objects of a class that implement **Closeable** can be closed. Beginning with JDK 7, **Closeable** extends **AutoCloseable**. Therefore, any class that implements **Closeable** also implements **AutoCloseable**.

#### **Flushable** Interface

- ❖ Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached. It defines the **flush( )** method, shown here:

**void flush( ) throws IOException**

- ❖ Flushing a stream typically causes buffered output to be physically written to the underlying device. This interface is implemented by all of the I/O classes that write to a stream.

## I/O Exceptions

- ❖ Two exceptions play an important role in I/O handling.
  - The first is **IOException**. As it relates to most of the I/O classes described in this chapter, if an I/O error occurs, an **IOException** is thrown.
  - In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single catch that catches **IOException**. However, you might find it useful to catch each exception separately.
- ❖ Another exception class that is sometimes important when performing I/O is **SecurityException**. In situations in which a security manager is present, several of the file classes will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via java (dos command to run java program) do not use a security manager.

## Two Ways to Close a Stream

- In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation.
- Beginning with JDK 7, there are two basic ways in which you can close a stream.
  - The first is to explicitly call **close()** on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, **close()** is typically called within a finally block. Thus, a simplified skeleton for the traditional approach is shown here:

```
try {
    // open and access file
} catch( I/O-exception) {
    // ...
} finally {
    // close the file
}
```

- The second approach to closing a stream is to automate the process by using the **try-with-resources** statement that was added by JDK 7.
- The **try-with-resources** statement is an enhanced form of try that has the following form:

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

Here, **resource-specification** is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. 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. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close( )** explicitly.

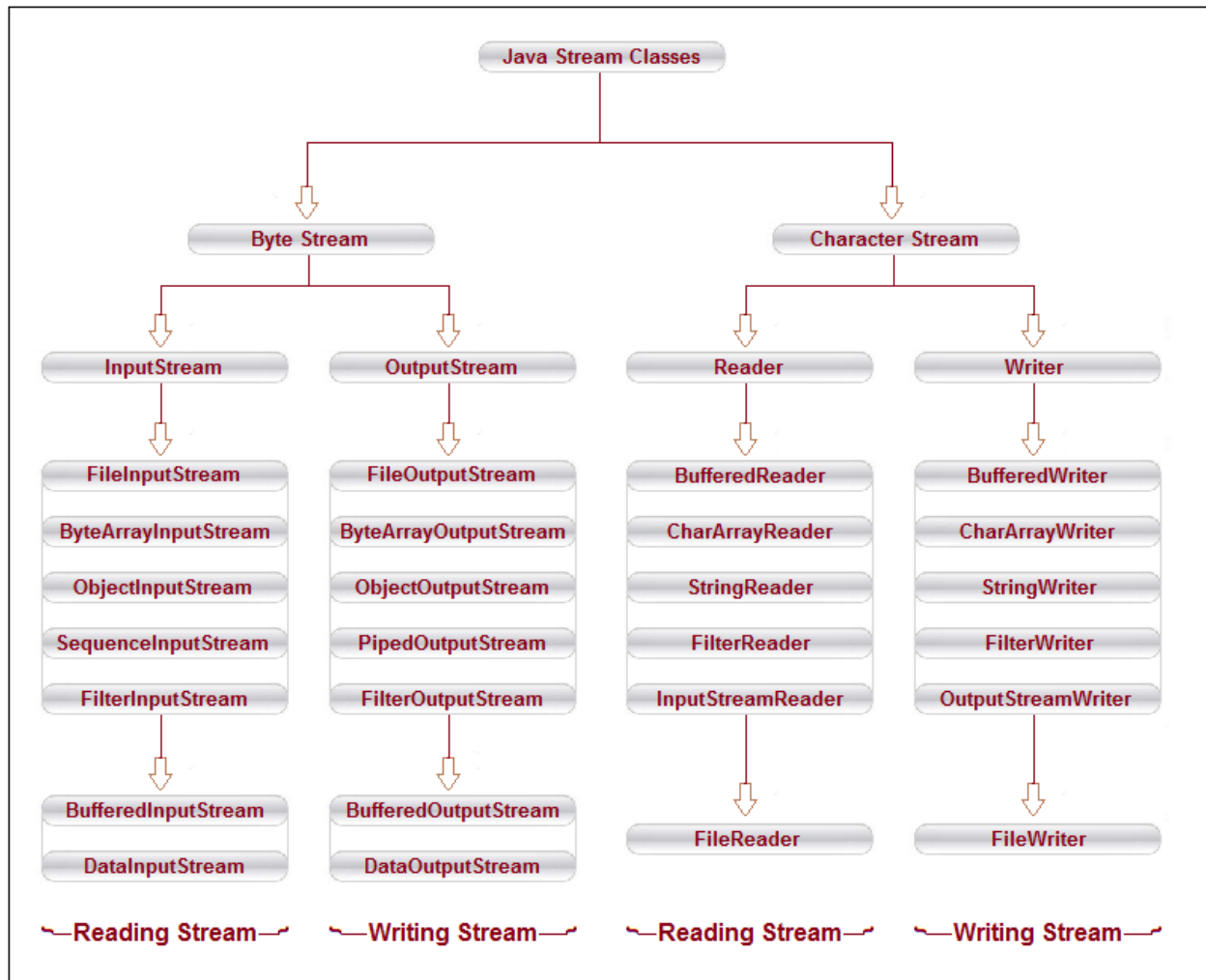
Here are three key points about the **try-with-resources** statement:

- ✎ Resources managed by **try-with-resources** must be objects of classes that implement **AutoCloseable**.
- ✎ The resource declared in the **try** is implicitly **final**.
- ✎ You can manage more than one resource by separating each declaration by a semicolon.

Also, remember that the scope of the declared resource is limited to the **try-with-resources** statement. The principal advantage of **try-with-resources** is that the resource (in this case, a stream) is closed automatically when the **try** block ends. Thus, it is not possible to forget to close the stream, for example. The **try-with-resources** approach also typically results in shorter, clearer, easier-to-maintain source code.

## The Stream Classes

- Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**.
- They are used to create several concrete stream subclasses.
- Although our programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.
- **InputStream** and **OutputStream** are designed for **byte streams**.
- **Reader** and **Writer** are designed for **character streams**.
- The byte stream classes and the character stream classes form separate hierarchies.
- In general, we should use the character stream classes when working with characters or strings and we should use the byte stream classes when working with bytes or other binary objects.



## The Byte Streams

- ✓ The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs.
- ✓ The byte stream classes are topped by **InputStream** and **OutputStream**.

### InputStream (an abstract class)

- ◆ **InputStream** is an abstract class that defines Java's model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs. (The exceptions are **mark()** and **markSupported()** methods.)
- ◆ List of methods is shown in following table

Sr.No.	Method & Description
1	<a href="#"><u>int available()</u></a> This method returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
2	<a href="#"><u>void close()</u></a> This method closes this input stream and releases any system resources associated with the stream.
3	<a href="#"><u>void mark(int readlimit)</u></a> This method marks the current position in this input stream.
4	<a href="#"><u>boolean markSupported()</u></a> This method tests if this input stream supports the mark and reset methods.
5	<a href="#"><u>abstract int read()</u></a> This method reads the next byte of data from the input stream.
6	<a href="#"><u>int read(byte[] b)</u></a> This method reads some number of bytes from the input stream and stores them into the buffer array b.
7	<a href="#"><u>int read(byte[] b, int off, int len) &lt;</u></a> This method reads up to len bytes of data from the input stream into an array of bytes.
8	<a href="#"><u>void reset()</u></a> This method repositions this stream to the position at the time the mark method was last called on this input stream.
9	<a href="#"><u>long skip(long n)</u></a> This method skips over and discards n bytes of data from this input stream.

**OutputStream ( an abstract class)**

- ◆ **OutputStream** is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors.
- ◆ List of methods is shown in following table

Sr.No.	Method & Description
1	<a href="#">void close()</a> This method closes this output stream and releases any system resources associated with this stream.
2	<a href="#">void flush()</a> This method flushes this output stream and forces any buffered output bytes to be written out.
3	<a href="#">void write(byte[] b)</a> This method writes b.length bytes from the specified byte array to this output stream.
4	<a href="#">void write(byte[] b, int off, int len)</a> This method writes len bytes from the specified byte array starting at offset off to this output stream.
5	<a href="#">abstract void write(int b)</a> This method writes the specified byte to this output stream.

**FileInputStream**

- ◆ The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Two commonly used constructors are shown here:

```
FileInputStream(String filePath)
```

```
FileInputStream(File fileObj)
```

Either can throw a **FileNotFoundException**. Here, **filePath** is the full path name of a file, and **fileObj** is a **File** object that describes the file.

**Steps to follow to read data from file using `FileInputStream`**

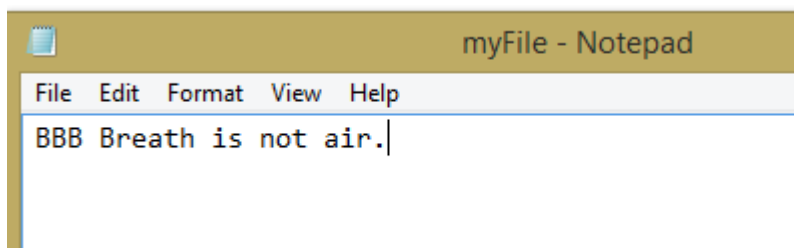
- ✓ Add **`import java.io.*;`** statement
- ✓ Handle **`FileNotFoundException`** and **`IOException`**
- ✓ Create **`FileInputStream`** class object ( let's say, **`fis`**)
- ✓ Invoke **`fis.read()`** ; method and assign it to **`int`** type variable
- ✓ Display or read the byte data return from the file
- ✓ Close stream after usage by using **`fis.close()`** ; [We can also use try-with resource where closing happens automatically]

Notes:

- **`fis.read()`** returns the given byte available in the file, if there is no byte available, it returns -1
- Only one byte at a time can be read. To read multiple bytes (complete file) we have to use while loop.
- The value is read in integer format so casting is needed.
- We can also use other overloaded versions of **`read()`** method. In such case, we should proceed accordingly.

**Example:**

**File: myFile.txt**



**File: FISExample.java**

```
package filehandling;

import java.io.*;

public class FISExample {

    public static void main(String args[]){

        final String FILEPATH ="E:/JAVA-I/myFile.txt";

        try{

            FileInputStream fin=new FileInputStream(FILEPATH);

            //this reads only a byte from the file

            int v = fin.read();

            //printing v by casting to char

            System.out.println((char)v);

            //to read multiple bytes

            int i=0;

            while((i=fin.read())!=-1){

                System.out.print((char)i);

            }

            fin.close();

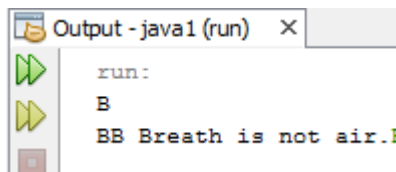
        }catch(Exception e){

            System.out.println("Proble occured : "+e);

        }

    }

}
```



**Useful Methods of java.io.FileInputStream class**

Sr.No.	Method & Description
1	<a href="#">int available()</a> This method returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
2	<a href="#">void close()</a> This method closes this file input stream and releases any system resources associated with the stream.
3	<a href="#">protected void finalize()</a> This method ensures that the close method of this file input stream is called when there are no more references to it.
4	<a href="#">FileChannel getChannel()</a> This method returns the unique FileChannel object associated with this file input stream.
5	<a href="#">FileDescriptor getFD()</a> This method returns the FileDescriptor object that represents the connection to the actual file in the file system being used by this FileInputStream.
6	<a href="#">int read()</a> This method reads a byte of data from this input stream.
7	<a href="#">int read(byte[] b)</a> This method reads up to <i>b.length</i> bytes of data from this input stream into an array of bytes.
8	<a href="#">int read(byte[] b, int off, int len)</a> This method reads up to <i>len</i> bytes of data from this input stream into an array of bytes.
9	<a href="#">long skip(long n)</a> This method skips over and discards <i>n</i> bytes of data from the input stream.

**FileOutputStream**

- ◆ **FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Four of its constructors are shown here:

```
FileOutputStream(String filePath)
```

```
FileOutputStream(File fileObj)
```

```
FileOutputStream(String filePath, boolean append)
```

```
FileOutputStream(File fileObj, boolean append)
```

They can throw a **FileNotFoundException**. Here, **filePath** is the full path name of a file, and **fileObj** is a **File** object that describes the file. If **append** is true, the file is opened in append mode.

- ◆ Creation of a **FileOutputStream** is not dependent on the file already existing. **FileOutputStream** will create the file before opening it for output when you create the object.
- ◆ In the case where you attempt to open a read-only file, an exception will be thrown.

Steps to follow to write data to a file using **FileOutputStream**

- ✓ Add **import java.io.\*;** statement
- ✓ Handle **FileNotFoundException** and **IOException**
- ✓ Create **FileOutputStream** class object ( let's say, **fos**)
- ✓ Invoke **fos.write(data) ;**
- ✓ Close stream after usage by using **fos.close() ;** [We can also use try-with resource where closing happens automatically]

Notes:

- **fos.write()** writes one byte at a time.
- We can also use other overloaded versions of **write()** method



Example:

```
package filehandling;

import java.io.*;

public class FOSExample{

    public static void main(String args[]) throws IOException{

        FileOutputStream fout=new FileOutputStream("E:/abc.txt");

        //these statements write single byte

        fout.write(66);

        fout.write('T');

        fout.write(' ');

        //to write string

        String s="Who am I?";

        byte b[]=s.getBytes();//converting string into byte array

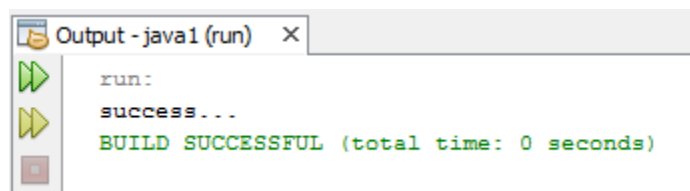
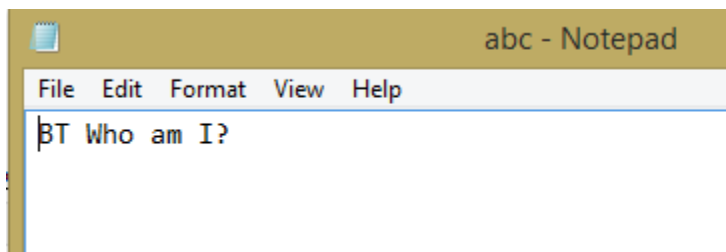
        fout.write(b);

        fout.close();

        System.out.println("success...");

    }

}
```



### Useful methods of java.io.FileOutputStream class

Sr.No.	Method & Description
1	<a href="#">void close()</a> This method closes this file output stream and releases any system resources associated with this stream.
2	<a href="#">protected void finalize()</a> This method cleans up the connection to the file, and ensures that the close method of this file output stream is called when there are no more references to this stream.
3	<a href="#">FileChannel getChannel()</a> This method returns the unique FileChannel object associated with this file output stream.
4	<a href="#">FileDescriptor getFD()</a> This method returns the file descriptor associated with this stream.
5	<a href="#">void write(byte[] b)</a> This method writes b.length bytes from the specified byte array to this file output stream.
6	<a href="#">void write(byte[] b, int off, int len)</a> This method writes len bytes from the specified byte array starting at offset off to this file output stream.
7	<a href="#">void write(int b)</a> This method writes the specified byte to this file output stream

### RandomAccessFile

This class is used for reading and writing to random access file. A random access file behaves like a large array of bytes. There is a cursor implied to the array called **file pointer**, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

- ✎ **RandomAccessFile** encapsulates a random-access file. It is not derived from **InputStream** or **OutputStream**. Instead, it implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods. It also implements the **AutoCloseable** and **Closeable** interfaces.
- ✎ **RandomAccessFile** is special because it supports positioning requests—that is, you can position the file pointer within the file.
- ✎ It has these two constructors:

```
RandomAccessFile(File fileObj, String access)
    throws FileNotFoundException
```

```
RandomAccessFile(String filename, String access)
    throws FileNotFoundException
```

In the first form, **fileObj** specifies the file to open as a **File** object. In the second form, the name of the file is passed in filename. In both cases, access determines what type of file access is permitted.

- If it is "**r**", then the file can be read, but not written.
- If it is "**rw**", then the file is opened in read-write mode.
- If it is "**rws**", the file is opened for read-write operations and every change to the file's data or metadata will be immediately written to the physical device.
- If it is "**rwd**", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

- ✎ The method **seek()**, shown here, is used to set the 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. After a call to **seek()**, the next read or write operation will occur at the new file position.

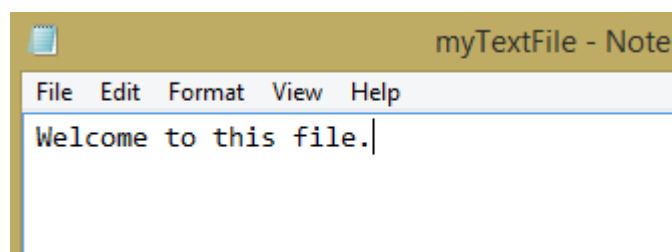
- ✎ **RandomAccessFile** implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is **setLength()**. It has this signature:

```
void setLength(long len) throws IOException
```

This method sets the length of the invoking file to that specified by **len**. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

Example:

Content of myTextFile.txt before executing the program



```
package filehandling;
```

```
import java.io.*;
```

```

public class RandomAccessFileExample {
    static final String FILEPATH = "E:/JAVA-I/myTextFile.txt";

    //creating method that can read randomly
    private static byte[] readFromFile(RandomAccessFile f, long position, int size)
        throws IOException{
        f.seek(position);
        byte[] bytes = new byte[size];
        f.read(bytes);
        return bytes;
    }

    //creating method that can write randomly
    private static void writeToFile(RandomAccessFile f, String data, long position)
        throws IOException {
        f.seek(position);
        f.write(data.getBytes());
    }

    public static void main(String[] args) {

        String reading=null;
        String writing ="Love all";
        try {
            long fp=0;

            //creating random access file object in read write mode
            RandomAccessFile file = new RandomAccessFile(FILEPATH, "rw");
            fp= file.getFilePointer();
            System.out.println("Current position of file pointer = "+ fp);
        }
    }
}

```

```

file.seek(6);

fp= file.getFilePointer();

System.out.println("After seek(6), position of file pointer = "+ fp);

System.out.println("Lets read file from this position");

//this reads 20 bytes from current position

byte a[] = readFromFile(file,fp,20);

reading = new String(a);

System.out.println(reading);

fp= file.getFilePointer();

System.out.println("Now position of file pointer = "+ fp);

System.out.println("Lets write in the file");

writeToFile(file, writing,fp);

fp= file.getFilePointer();

System.out.println("Now position of file pointer = "+ fp);

file.close();

} catch (IOException e) {

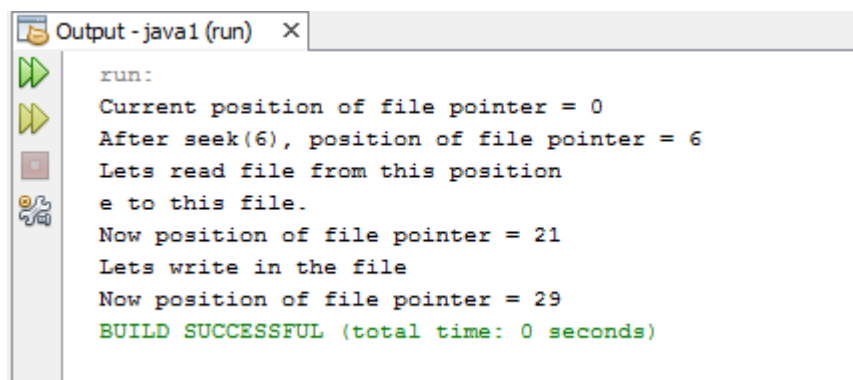
    e.printStackTrace();

}

}

}

```

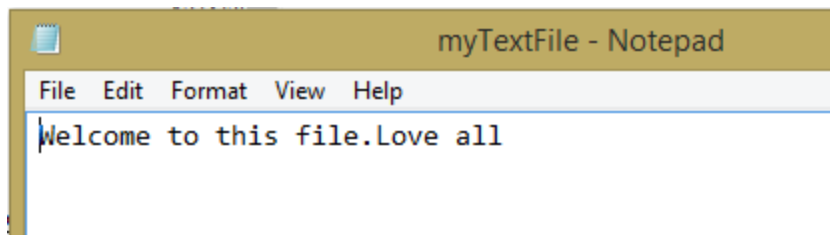


```

Output - java1 (run) ×
run:
Current position of file pointer = 0
After seek(6), position of file pointer = 6
Lets read file from this position
e to this file.
Now position of file pointer = 21
Lets write in the file
Now position of file pointer = 29
BUILD SUCCESSFUL (total time: 0 seconds)

```

Content of myTextFile.txt after executing the program



**Useful methods of java.io.RandomAccessFile class**

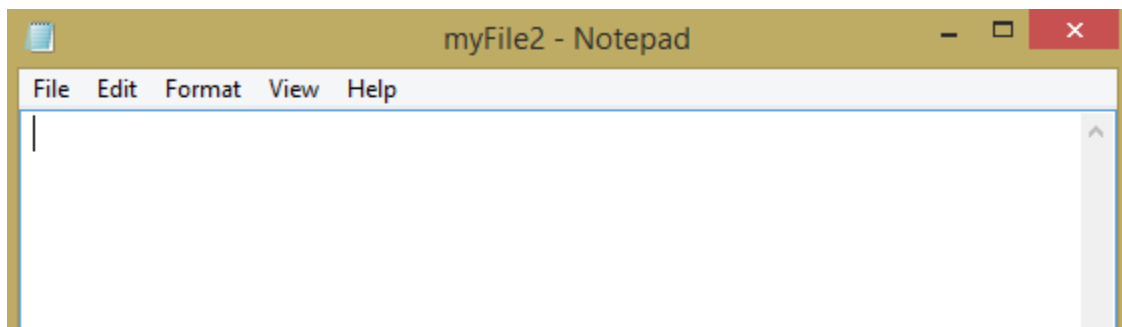
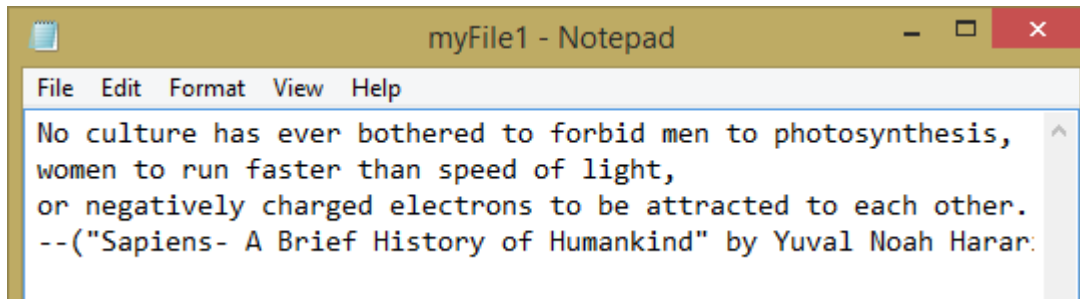
Sr.No.	Method & Description
1	<a href="#"><u>void close()</u></a> This method Closes this random access file stream and releases any system resources associated with the stream.
2	<a href="#"><u>FileChannel getChannel()</u></a> This method returns the unique FileChannel object associated with this file.
3	<a href="#"><u>FileDescriptor getFD()</u></a> This method returns the opaque file descriptor object associated with this stream.
4	<a href="#"><u>long getFilePointer()</u></a> This method returns the current offset in this file.
5	<a href="#"><u>long length()</u></a> This method returns the length of this file.
6	<a href="#"><u>int read()</u></a> This method reads a byte of data from this file.
7	<a href="#"><u>int read(byte[] b)</u></a> This method reads up to b.length bytes of data from this file into an array of bytes.
8	<a href="#"><u>int read(byte[] b, int off, int len)</u></a> This method reads up to len bytes of data from this file into an array of bytes.
9	<a href="#"><u>boolean readBoolean()</u></a> This method reads a boolean from this file.
10	<a href="#"><u>byte readByte()</u></a> This method reads a signed eight-bit value from this file.
11	<a href="#"><u>char readChar()</u></a> This method reads a character from this file.
12	<a href="#"><u>double readDouble()</u></a> This method reads a double from this file.
13	<a href="#"><u>float readFloat()</u></a> This method reads a float from this file.
14	<a href="#"><u>void readFully(byte[] b)</u></a> This method reads b.length bytes from this file into the byte array, starting at the current file pointer.
15	<a href="#"><u>void readFully(byte[] b, int off, int len)</u></a> This method reads exactly len bytes from this file into the byte array, starting at the current file pointer.
16	<a href="#"><u>int readInt()</u></a> This method reads a signed 32-bit integer from this file.
17	<a href="#"><u>String readLine()</u></a> This method reads the next line of text from this file.
18	<a href="#"><u>long readLong()</u></a> This method reads a signed 64-bit integer from this file.
19	<a href="#"><u>short readShort()</u></a> This method reads a signed 16-bit number from this file.
20	<a href="#"><u>int readUnsignedByte()</u></a> This method reads an unsigned eight-bit number from this file.

21	<a href="#"><code>int readUnsignedShort()</code></a> This method reads an unsigned 16-bit number from this file.
22	<a href="#"><code>String readUTF()</code></a> This method reads a string from this file.
23	<a href="#"><code>void seek(long pos)</code></a> This method sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
24	<a href="#"><code>void setLength(long newLength)</code></a> This method sets the length of this file.
25	<a href="#"><code>int skipBytes(int n)</code></a> This method attempts to skip over n bytes of input discarding the skipped bytes.
26	<a href="#"><code>void write(byte[] b)</code></a> This method writes b.length bytes from the specified byte array to this file, starting at the current file pointer.
27	<a href="#"><code>void write(byte[] b, int off, int len)</code></a> This method writes len bytes from the specified byte array starting at offset off to this file.
28	<a href="#"><code>void write(int b)</code></a> This method writes the specified byte to this file.
29	<a href="#"><code>void writeBoolean(boolean v)</code></a> This method writes a boolean to the file as a one-byte value.
30	<a href="#"><code>void writeByte(int v)</code></a> This method writes a byte to the file as a one-byte value.
31	<a href="#"><code>void writeBytes(String s)</code></a> This method writes the string to the file as a sequence of bytes.
32	<a href="#"><code>void writeChar(int v)</code></a> This method writes a char to the file as a two-byte value, high byte first.
33	<a href="#"><code>void writeChars(String s)</code></a> This method writes a string to the file as a sequence of characters.
34	<a href="#"><code>void writeDouble(double v)</code></a> This method converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
35	<a href="#"><code>void writeFloat(float v)</code></a> This method converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
36	<a href="#"><code>void writeInt(int v)</code></a> This method writes an int to the file as four bytes, high byte first.
37	<a href="#"><code>void writeLong(long v)</code></a> This method writes a long to the file as eight bytes, high byte first.
38	<a href="#"><code>void writeShort(int v)</code></a> This method writes a short to the file as two bytes, high byte first.
39	<a href="#"><code>void writeUTF(String str)</code></a> This method writes a string to the file using modified UTF-8 encoding in a machine-independent manner.



## Write a program to read from a file and write to another file

Before executing the program



```
//Program to read from a file and write to another file
//File copying
package filehandling;
import java.io.*;

public class FileCopyingDemo {
    public static void main(String[] args) {
        //creating two File objects
        File f1 = new File("E:/myFile1.txt");
        File f2 = new File("E:/myFile2.txt");
        try{
            //creating FileInputStream object for reading
            FileInputStream fis = new FileInputStream(f1);
            //creating FileOutputStream object for writing
```

```

        FileOutputStream fos = new FileOutputStream(f2);

        int i=0;

        //to read from one file and write to another
        while((i=fis.read())!=-1){

            char c = (char)i;

            fos.write(c);

        }

        fis.close();

        fos.close();

        System.out.println("contents of one file is copied to
        another file");

        }catch(IOException ioe){

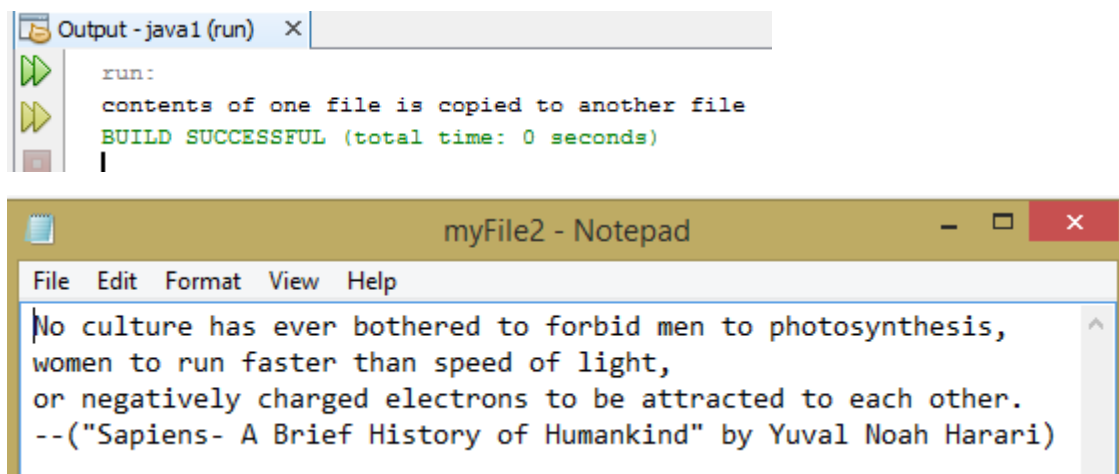
            System.out.println("I/O Error: "+ioe);

        }

    }}

```

### After executing the program



## The Character Streams

- ❖ While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters.
- ❖ At the top of the character stream hierarchy there are the **Reader** and **Writer** abstract classes.

### Reader

- ✓ **Reader** is an abstract class that defines Java's model of streaming character input. It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces.
- ✓ All of the methods in this class (except for **markSupported()**) will throw an **IOException** on error conditions. **Writer**

#### Useful methods of java.io.Reader Class

Sr.No.	Method & Description
1	<a href="#"><u>abstract void close()</u></a> This method closes the stream and releases any system resources associated with it.
2	<a href="#"><u>void mark(int readAheadLimit)</u></a> This method marks the present position in the stream.
3	<a href="#"><u>boolean markSupported()</u></a> This method tells whether this stream supports the mark() operation.
4	<a href="#"><u>int read()</u></a> This method reads a single character.
5	<a href="#"><u>int read(char[] cbuf)</u></a> This method reads characters into an array.
6	<a href="#"><u>abstract int read(char[] cbuf, int off, int len)</u></a> This method reads characters into a portion of an array.
7	<a href="#"><u>int read(CharBuffer target)</u></a> This method attempts to read characters into the specified character buffer.
8	<a href="#"><u>boolean ready()</u></a> This method tells whether this stream is ready to be read.
9	<a href="#"><u>void reset()</u></a> This method resets the stream.
10	<a href="#"><u>long skip(long n)</u></a> This method skips characters.

### Writer

- ✓ **Writer** is an abstract class that defines streaming character output. It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces.
- ✓ All of the methods in this class throw an **IOException** in the case of errors.

#### Useful methods of java.io.Reader Class

Sr.No.	Method & Description
1	<a href="#"><u>Writer append(char c)</u></a> This method appends the specified character to this writer.
2	<a href="#"><u>Writer append(CharSequence csq)</u></a> This method appends the specified character sequence to this writer.
3	<a href="#"><u>Writer append(CharSequence csq, int start, int end)</u></a> This method appends a subsequence of the specified character sequence to this writer.
4	<a href="#"><u>abstract void close()</u></a> This method loses the stream, flushing it first.
5	<a href="#"><u>abstract void flush()</u></a> This method flushes the stream.
6	<a href="#"><u>void write(char[] cbuf)</u></a> This method writes an array of characters.
7	<a href="#"><u>abstract void write(char[] cbuf, int off, int len)</u></a> This method writes a portion of an array of characters.
8	<a href="#"><u>void write(int c)</u></a> This method writes a single character.
9	<a href="#"><u>void write(String str)</u></a> This method writes a string.
10	<a href="#"><u>void write(String str, int off, int len)</u></a> This method writes a portion of a string.

## FileReader

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

**FileReader(String filePath)**

**FileReader(File fileObj)**

Either can throw a **FileNotFoundException**. Here, **filePath** is the full path name of a file, and **fileObj** is a File object that describes the file.

## Useful methods of java.io.FileReader class

Methods	Description
<b>int read()</b>	It is used to return a character in ASCII form. It returns -1 at the end of file.
<b>void close()</b>	It is used to close the <b>FileReader</b> class.

- ✓ The following example shows how to read lines from a file and display them on the standard output device

```
//to demonstrate FileReader
//try-with resource is used for exception handling
package filehandling;
import java.io.*;
public class FRExample {
    public static void main(String args[]){
        try(FileReader fr=new FileReader("E:/abc.txt")){
            int i;
            while((i=fr.read())!=-1)
                System.out.print((char)i);
            System.out.println("\n File reading completed ");
        }catch(IOException ioe){
            System.out.println("I/O Error : "+ ioe);
        }
    }
}
```

## FileWriter

- ✓ **FileWriter** creates a **Writer** that you can use to write to a file. Four commonly used constructors are shown here:

**FileWriter(String filePath)**

**FileWriter(String filePath, boolean append)**

**FileWriter(File fileObj)**

**FileWriter(File fileObj, boolean append)**

They can all throw an **IOException**. Here, **filePath** is the full path name of a file, and **fileObj** is a **File** object that describes the file. If **append** is true, then output is appended to the end of the file.

- ✓ 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. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

## Useful methods of java.io.FileWriter class

Methods	Description
<b>void write(String text)</b>	It is used to write the string into FileWriter.
<b>void write(char c)</b>	It is used to write the char into FileWriter.
<b>void write(char[] c)</b>	It is used to write char array into FileWriter.
<b>void flush()</b>	It is used to flush the data of FileWriter.
<b>void close()</b>	It is used to close the FileWriter.

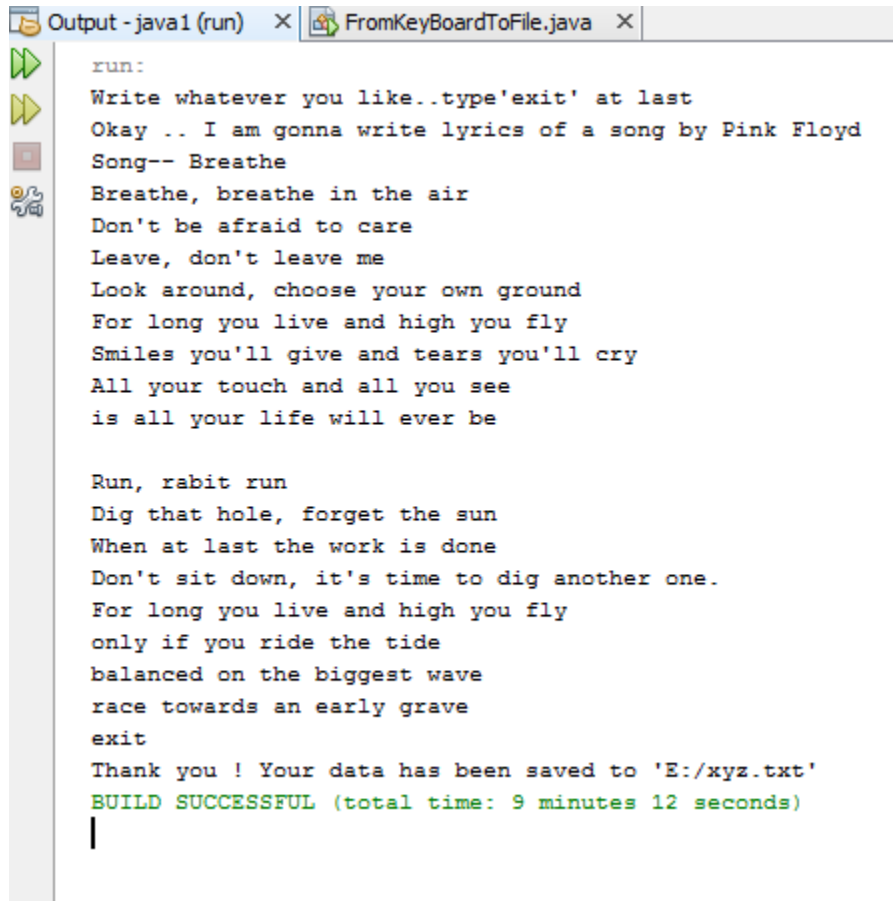
```
//to demonstrate FileWriter
package filehandling;
import java.io.*;
public class FWExample {
    public static void main(String args[]) throws Exception {
        FileWriter fw=new FileWriter("E:/abc.txt");
        fw.write("Do good.Be good.");
        fw.close();
        System.out.println("writing completed!");
    }
}
```

### **Write a program to read the text from keyboard and write to a file.**

```
//To read from keyboard and write to a file
package filehandling;
import java.io.*;
import java.util.Scanner;
public class FromKeyBoardToFile {
    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        String inputData = "";
        System.out.println("Write whatever you like..type'exit' at last");
```

```
File myfile = new File("E:/xyz.txt");

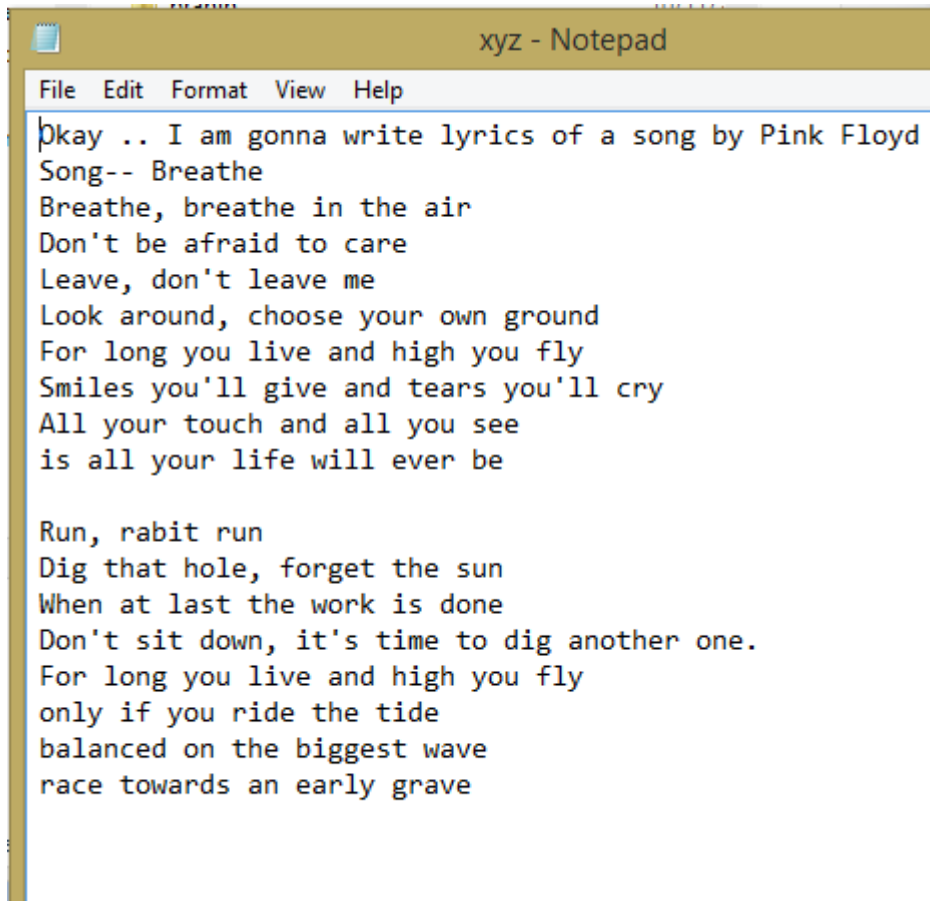
try(FileWriter fw = new FileWriter(myfile)) {
    while(true){
        //read line by line
        inputData= sc.nextLine();
        //if "Exit" is typed,come out from the loop
        if(inputData.equalsIgnoreCase("exit"))
            break;
        //write the line
        fw.write(inputData);
        //write newline character
        fw.write("\r\n");
    }
    System.out.println("Thank you ! Your data has been saved to 'E:/xyz.txt'");
} catch(IOException ioe){
    System.out.println("I/O proble: "+ ioe);
}
}
```



```
run:
Write whatever you like..type 'exit' at last
Okay .. I am gonna write lyrics of a song by Pink Floyd
Song-- Breathe
Breathe, breathe in the air
Don't be afraid to care
Leave, don't leave me
Look around, choose your own ground
For long you live and high you fly
Smiles you'll give and tears you'll cry
All your touch and all you see
is all your life will ever be

Run, rabbit run
Dig that hole, forget the sun
When at last the work is done
Don't sit down, it's time to dig another one.
For long you live and high you fly
only if you ride the tide
balanced on the biggest wave
race towards an early grave
exit
Thank you ! Your data has been saved to 'E:/xyz.txt'
BUILD SUCCESSFUL (total time: 9 minutes 12 seconds)
|
```





```
xyz - Notepad
File Edit Format View Help
Okay .. I am gonna write lyrics of a song by Pink Floyd
Song-- Breathe
Breathe, breathe in the air
Don't be afraid to care
Leave, don't leave me
Look around, choose your own ground
For long you live and high you fly
Smiles you'll give and tears you'll cry
All your touch and all you see
is all your life will ever be

Run, rabbit run
Dig that hole, forget the sun
When at last the work is done
Don't sit down, it's time to dig another one.
For long you live and high you fly
only if you ride the tide
balanced on the biggest wave
race towards an early grave
```

## Character Stream Vs Byte Stream in Java

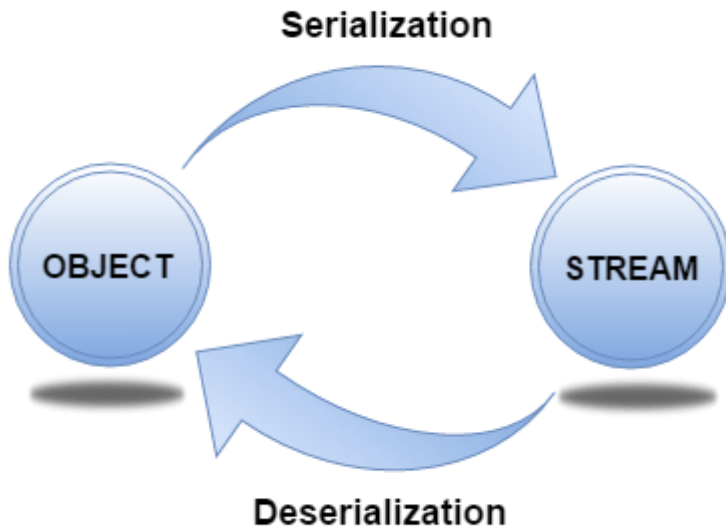
A stream is a way of sequentially accessing a file. In Streams you can process the data one at a time as bulk operations are unavailable with them. But, streams supports a huge range of source and destinations including disk file, arrays, other devices, other programs etc. In Java, a byte is not the same thing as a char. Therefore a byte stream is different from a character stream. So, Java defines two types of streams: Byte Streams and Character Streams.

A **byte stream** access the file byte by byte. Java programs use byte streams to perform input and output of 8-bit bytes. It is suitable for any kind of file, however not quite appropriate for text files. For example, if the file is using a Unicode encoding and a character is represented with two bytes, the byte stream will treat these separately and you will need to do the conversion yourself. Byte oriented streams do not use any encoding scheme while Character oriented streams use character encoding scheme (UNICODE). All byte stream classes are descended from `InputStream` and `OutputStream`.

A **character stream** will read a file character by character. Character Stream is a higher level concept than Byte Stream. A Character Stream is, effectively, a Byte Stream that has been wrapped with logic that allows it to output characters from a specific encoding. That means, a character stream needs to be given the file's encoding in order to work properly. Character stream can support all types of character sets ASCII, Unicode, UTF-8, UTF-16 etc. All character stream classes are descended from `Reader` and `Writer`.



## Serialization



- **Serialization** is the process of writing the state of an object to a byte stream.
- This is useful when you want to save the state of your program to a persistent storage area, such as a file.
- At a later time, you may restore these objects by using the process of deserialization.
- Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.
- Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios.
- If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

### Serializable

- ✓ Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

- ✓ Variables that are declared as transient are not saved by the serialization facilities. Also, static variables are not saved.

### Externalizable

- ✓ The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.
- ✓ The **Externalizable** interface defines these two methods:

```
void readExternal(ObjectInput inStream)
    throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
    throws IOException
```

In these methods, **inStream** is the byte stream from which the object is to be read, and **outStream** is the byte stream to which the object is to be written.

### ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization. It defines different methods. It defines the methods shown in table below. Note especially the **writeObject( )** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

Method	Description
<code>void close( )</code>	Closes the invoking stream. Further write attempts will generate an <b>IOException</b> .
<code>void flush( )</code>	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
<code>void write(byte buffer[ ])</code>	Writes an array of bytes to the invoking stream.
<code>void write(byte buffer[ ], int offset, int numBytes)</code>	Writes a subrange of <i>numBytes</i> bytes from the array <i>buffer</i> , beginning at <i>buffer[offset]</i> .
<code>void write(int b)</code>	Writes a single byte to the invoking stream. The byte written is the low-order byte of <i>b</i> .
<code>void writeObject(Object obj)</code>	Writes object <i>obj</i> to the invoking stream.

## ObjectOutputStream

- ✓ The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

**ObjectOutputStream(OutputStream outStream) throws IOException**

The argument **outStream** is the output stream to which serialized objects will be written.

- ✓ Closing an **ObjectOutputStream** automatically closes the underlying stream specified by **outStream**.

## Useful methods of java.io.ObjectOutputStream

Sr.No.	Method & Description
1	<a href="#"><u>protected void annotateClass(Class &lt;?&gt; cl)</u></a> Subclasses may implement this method to allow class data to be stored in the stream.
2	<a href="#"><u>protected void annotateProxyClass(Class&lt;?&gt; cl)</u></a> Subclasses may implement this method to store custom data in the stream along with descriptors for dynamic proxy classes.
3	<a href="#"><u>void close()</u></a> This method closes the stream.
4	<a href="#"><u>void defaultWriteObject()</u></a> This method writes the non-static and non-transient fields of the current class to this stream.
5	<a href="#"><u>protected void drain()</u></a> This method drain any buffered data in ObjectOutputStream.
6	<a href="#"><u>protected boolean enableReplaceObject(boolean enable)</u></a> This method enable the stream to do replacement of objects in the stream.
7	<a href="#"><u>void flush()</u></a> This method flushes the stream.
8	<a href="#"><u>ObjectOutputStream.PutField putFields()</u></a> This method retrieves the object used to buffer persistent fields to be written to the stream.
9	<a href="#"><u>protected Object replaceObject(Object obj)</u></a> This method will allow trusted subclasses of ObjectOutputStream to substitute one object for another during serialization.
10	<a href="#"><u>void reset()</u></a> This method reset will disregard the state of any objects already written to the stream.
11	<a href="#"><u>void useProtocolVersion(int version)</u></a> This method specify stream protocol version to use when writing the stream.
12	<a href="#"><u>void write(byte[] buf)</u></a> This method writes an array of bytes.
13	<a href="#"><u>void write(byte[] buf, int off, int len)</u></a> This method writes a sub array of bytes.
14	<a href="#"><u>void write(int val)</u></a>

	This method writes a byte.
15	<a href="#"><u>void writeBoolean(boolean val)</u></a> This method writes a boolean.
16	<a href="#"><u>void writeByte(int val)</u></a> This method writes an 8 bit byte.
17	<a href="#"><u>void writeBytes(String str)</u></a> This method writes a String as a sequence of bytes.
18	<a href="#"><u>void writeChar(int val)</u></a> This method writes a 16 bit char.
19	<a href="#"><u>void writeChars(String str)</u></a> This method writes a String as a sequence of chars.
20	<a href="#"><u>protected void writeClassDescriptor(ObjectStreamClass desc)</u></a> This method writes the specified class descriptor to the ObjectOutputStream.
21	<a href="#"><u>void writeDouble(double val)</u></a> This method writes a 64 bit double.
22	<a href="#"><u>void writeFields()</u></a> This method writes the buffered fields to the stream.
23	<a href="#"><u>void writeFloat(float val)</u></a> This method writes a 32 bit float.
24	<a href="#"><u>void writeInt(int val)</u></a> This method writes a 32 bit int.
25	<a href="#"><u>void writeLong(long val)</u></a> This method writes a 64 bit long.
26	<a href="#"><u>void writeObject(Object obj)</u></a> This method writes the specified object to the ObjectOutputStream.
27	<a href="#"><u>protected void writeObjectOverride(Object obj)</u></a> This method is used by subclasses to override the default writeObject method.
28	<a href="#"><u>void writeShort(int val)</u></a> This method writes a 16 bit short.
29	<a href="#"><u>protected void writeStreamHeader()</u></a> This method is provided so subclasses can append or prepend their own header to the stream.
30	<a href="#"><u>void writeUnshared(Object obj)</u></a> This method writes an "unshared" object to the ObjectOutputStream.

### ObjectInput

- ✓ The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods shown in table below.
- ✓ It supports object serialization. Note especially the **readObject()** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject()** method can also throw **ClassNotFoundException**.

Method	Description
<code>int available( )</code>	Returns the number of bytes that are now available in the input buffer.
<code>void close( )</code>	Closes the invoking stream. Further read attempts will generate an <b>IOException</b> .
<code>int read( )</code>	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
<code>int read(byte <i>buffer</i>[ ])</code>	Attempts to read up to <i>buffer.length</i> bytes into <i>buffer</i> , returning the number of bytes that were successfully read. -1 is returned when the end of the file is encountered.
<code>int read(byte <i>buffer</i>[ ], int <i>offset</i>, int <i>numBytes</i>)</code>	Attempts to read up to <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of bytes that were successfully read. -1 is returned when the end of the file is encountered.
<code>Object readObject( )</code>	Reads an object from the invoking stream.
<code>long skip(long <i>numBytes</i>)</code>	Ignores (that is, skips) <i>numBytes</i> bytes in the invoking stream, returning the number of bytes actually ignored.

### ObjectInputStream

- ✓ The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream.
- ✓ One constructor of this class is shown here:

#### **ObjectInputStream(InputStream inStream) throws IOException**

The argument **inStream** is the input stream from which serialized objects should be read.

- ✓ Closing an **ObjectInputStream** automatically closes the underlying stream specified by **inStream**.
- ✓ Several commonly used methods in this class are shown in following table. They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

### Useful methods of java.io.ObjectInputStream

Sr.No.	Method & Description
1	<a href="#"><u>int available()</u></a> This method returns the number of bytes that can be read without blocking.
2	<a href="#"><u>void close()</u></a> This method closes the input stream.
3	<a href="#"><u>void defaultReadObject()</u></a>



	This method reads the non-static and non-transient fields of the current class from this stream.
4	<a href="#"><u>protected boolean enableResolveObject(boolean enable)</u></a> This method enables the stream to allow objects read from the stream to be replaced.
5	<a href="#"><u>int read()</u></a> This method reads a byte of data.
6	<a href="#"><u>int read(byte[] buf, int off, int len)</u></a> This method reads into an array of bytes.
7	<a href="#"><u>boolean readBoolean()</u></a> This method reads in a boolean.
8	<a href="#"><u>byte readByte()</u></a> This method reads an 8 bit byte.
9	<a href="#"><u>char readChar()</u></a> This method reads a 16 bit char.
10	<a href="#"><u>protected ObjectStreamClass readClassDescriptor()</u></a> This method reads a class descriptor from the serialization stream.
11	<a href="#"><u>double readDouble()</u></a> This method reads a 64 bit double.
12	<a href="#"><u>ObjectInputStream.GetField readFields()</u></a> This method reads the persistent fields from the stream and makes them available by name.
13	<a href="#"><u>float readFloat()</u></a> This method reads a 32 bit float.
14	<a href="#"><u>void readFully(byte[] buf)</u></a> This method reads bytes, blocking until all bytes are read.
15	<a href="#"><u>void readFully(byte[] buf, int off, int len)</u></a> This method reads bytes, blocking until all bytes are read.
16	<a href="#"><u>int readInt()</u></a> This method reads a 32 bit int.
17	<a href="#"><u>long readLong()</u></a> This method reads a 64 bit long.
18	<a href="#"><u>Object readObject()</u></a> This method reads an object from the ObjectInputStream.
19	<a href="#"><u>protected Object readObjectOverride()</u></a> This method is called by trusted subclasses of ObjectOutputStream that constructed ObjectOutputStream using the protected no-arg constructor.
20	<a href="#"><u>short readShort()</u></a> This method reads a 16 bit short.
21	<a href="#"><u>protected void readStreamHeader()</u></a> This method is provided to allow subclasses to read and verify their own stream headers.
22	<a href="#"><u>Object readUnshared()</u></a> This method reads an "unshared" object from the ObjectInputStream.
23	<a href="#"><u>int readUnsignedByte()</u></a> This method reads an unsigned 8 bit byte.
24	<a href="#"><u>int readUnsignedShort()</u></a> This method reads an unsigned 16 bit short.

25	<u><a href="#">String readUTF()</a></u> This method reads a String in modified UTF-8 format.
26	<u><a href="#">void registerValidation(ObjectInputValidation obj, int prio)</a></u> This method register an object to be validated before the graph is returned.
27	<u><a href="#">protected Class&lt;?&gt; resolveClass(ObjectStreamClass desc)</a></u> This method loads the local class equivalent of the specified stream class description.
28	<u><a href="#">protected Object resolveObject(Object obj)</a></u> This method will allow trusted subclasses of ObjectInputStream to substitute one object for another during deserialization.
29	<u><a href="#">protected Class&lt;?&gt; resolveProxyClass(String[] interfaces)</a></u> This method returns a proxy class that implements the interfaces named in a proxy class descriptor; subclasses may implement this method to read custom data from the stream along with the descriptors for dynamic proxy classes, allowing them to use an alternate loading mechanism for the interfaces and the proxy class.
30	<u><a href="#">int skipBytes(int len)</a></u> This method skips bytes.

## Java 'transient' Keyword

Java **transient** keyword is used in serialization. If you define any data member as **transient**, it will not be serialized.

```
import java.io.Serializable;

public class Student implements Serializable{
    int id;
    String name;
    transient int age;//Now it will not be serialized
    public Student(int id, String name,int age) {
        this.id = id;
        this.name = name;
        this.age=age;
    }
}
```

### A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**, and **double**. This is the information we want to save and restore.

A **FileOutputStream** is created that refers to a file named "**serial**", and an **ObjectOutputStream** is created for that file stream. The **writeObject()** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named "**serial**", and an **ObjectInputStream** is created for that file stream. The **readObject()** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be transient. That data is then not saved during serialization.

```

// A serialization demo.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

        // Object serialization

        try ( ObjectOutputStream objOStrm =
              new ObjectOutputStream(new FileOutputStream("serial")) )
        {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1: " + object1);

            objOStrm.writeObject(object1);
        }
        catch(IOException e) {
            System.out.println("Exception during serialization: " + e);
        }

        // Object deserialization

        try ( ObjectInputStream objIStrm =
              new ObjectInputStream(new FileInputStream("serial")) )
        {
            MyClass object2 = (MyClass)objIStrm.readObject();
            System.out.println("object2: " + object2);
        }
        catch(Exception e) {
            System.out.println("Exception during deserialization: " + e);
        }
    }
}

class MyClass implements Serializable {
    String s;
    int i;
    double d;

    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
    }

    public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
    }
}

```

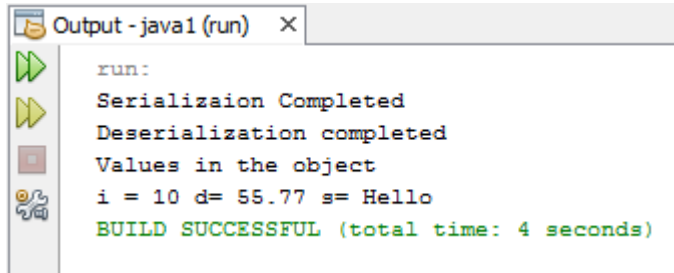
This program demonstrates that the instance variables of **object1** and **object2** are identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

### Another example to demonstrate serialization and deserialization process

```
package filehandling;
import java.io.*;
class Demo implements Serializable{
    int i =10;
    double d = 55.77;
    String s = "Hello";
}
public class SerDeserDemo {
    public static void main(String[] args) throws Exception {
        Demo obj = new Demo();
        //serialization
        FileOutputStream fos = new FileOutputStream("E:/pqr.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(obj);
        System.out.println("Serializaion Completed");
        //deserialization
        FileInputStream fis = new FileInputStream("E:/pqr.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Demo obj1= (Demo) ois.readObject();
        System.out.println("Deserialization completed ");
    }
}
```

```
System.out.println("Values in the object");  
  
System.out.println("i = "+ obj1.i+" d= "+ obj1.d+ " s= "+obj1.s);  
  
}  
  
}
```

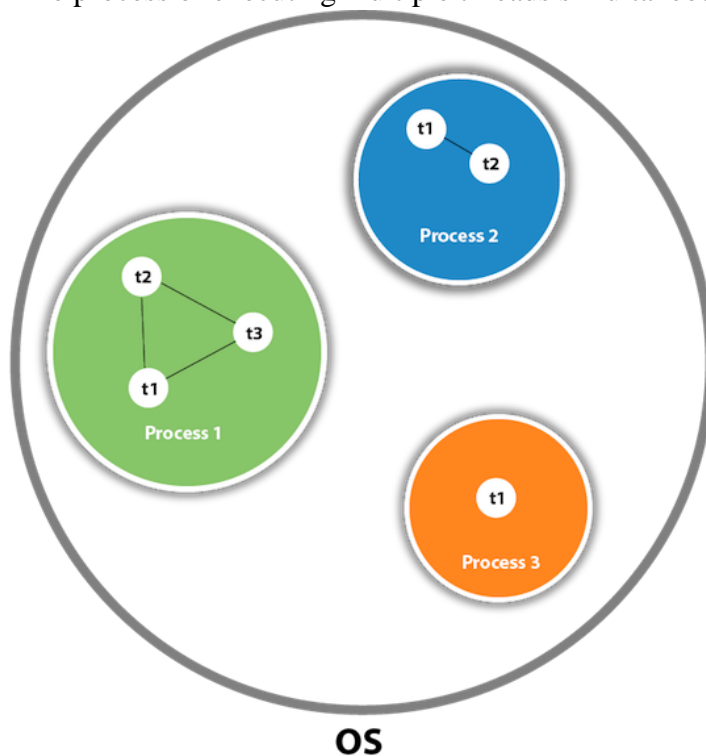


## Multithreading in Java

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.



- A **thread** is a light-weight smallest part of a process that can run concurrently with the other parts (other threads) of the same process. Threads are independent because they all have separate path of execution.
  - ✎ A thread is a lightweight subprocess, the smallest unit of processing. It has a separate path of execution.
  - ✎ Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.
- The process of executing multiple threads simultaneously is known as **multithreading**.





- As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.
- The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
- In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- ❖ Process-based Multitasking (Multiprocessing)
- ❖ Thread-based Multitasking (Multithreading)

### Process-based Multitasking (Multiprocessing)

- ✎ A process is, in essence, a program that is executing.
- ✎ Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.
- ✎ For example, processbased multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
- ✎ In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

### Thread-based Multitasking (Multithreading)

- ✎ In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
- ✎ This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

## Multiprocessing vs Multithreading

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's control. However, multithreaded multitasking is.

- ❖ The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
- ❖ Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.
- ❖ Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.
- ❖ By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.
- ❖ Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.
- ❖ Java Multithreading is mostly used in games, animation etc.

### **Multitasking vs Multithreading vs Multiprocessing vs parallel processing**

**Multitasking:** Ability to execute more than one task at the same time is known as multitasking.

**Multithreading:** We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as *Thread-based Multitasking*.

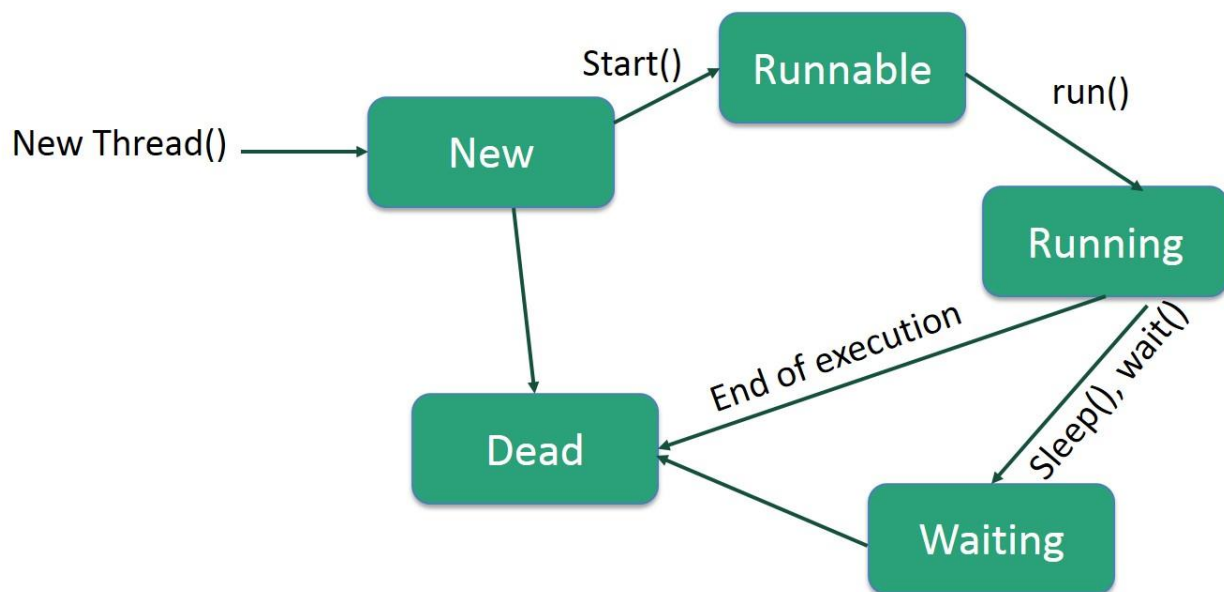
**Multiprocessing:** It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

**Parallel Processing:** It refers to the utilization of multiple CPUs in a single computer system.

## Life cycle of a Thread (Thread States)

A thread goes through various stages in its life cycle.

Threads exist in several states. Here is a general description. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily halts its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.



### New (Born) State

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

When the thread instance is created, it will be in “New” state.

### Runnable (Ready) State

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

When the thread is started, it is called “Runnable” state.

### Running State

The thread is in running state if the thread scheduler has selected it.

When the thread is running, it is called “Running” state

### Waiting (Blocked) State

This is the state when the thread is still alive, but is currently not eligible to run.

When the thread is put on hold or it is waiting for the other thread to complete, then that state will be known as “waiting” state.

### Terminated (Dead) State

A thread is in terminated or dead state when its `run()` method exits.

When the thread is dead, it will be known as “terminated” state

### The Thread Class and the Runnable Interface

Java’s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. Thread encapsulates a thread of execution. Since you can’t directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it.

- ✎ To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

### The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method `currentThread()`, which is a public static member of **Thread**. Its general form is shown here:

```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let’s begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep()** method in Thread might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when **t** is used as an argument to **println()**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and main is also the name of the

group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by Thread that are used in the program.

- ✎ The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in **milliseconds**. This method may throw an **InterruptedException**.

The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds) throws  
InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

- ✎ As the preceding program shows, you can set the name of a thread by using **setName( )**. You can obtain the name of a thread by calling **getName( )** (but note that this is not shown in the program). These methods are members of the Thread class and are declared like this:

```
final void setName(String threadName)
```

```
final String getName( )
```

Here, **threadName** specifies the name of the thread.

## Creating a Thread

In the most general sense, we create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- ✎ We can **implement the Runnable interface**.
- ✎ We can **extend the Thread class**, itself.

### Creating a Thread by implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run()**, which is declared like this:

```
public void run( )
```

Inside **run()**, you will define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, **threadOb** is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by **threadName**.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**. In essence, **start()** executes a call to **run()**. The **start()** method is shown here:

```
void start( )
```





If your class is intended to be executed as a thread then you can achieve this by implementing a `Runnable` interface. You will need to follow three basic steps –

### Step 1

As a first step, you need to implement a **`run()`** method provided by a **`Runnable`** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the **`run()`** method –

```
public void run( )
```

### Step 2

As a second step, you will instantiate a `Thread` object using the following constructor –

```
Thread(Runnable threadObj, String threadName)
```

Or

```
Thread(Runnable threadObj)
```

Where, **`threadObj`** is an instance of a class that implements the **`Runnable`** interface and **`threadName`** is the name given to the new thread.

### Step 3

Once a `Thread` object is created, you can start it by calling **`start()`** method, which executes a call to **`run()`** method. Following is a simple syntax of **`start()`** method –

```
void start();
```

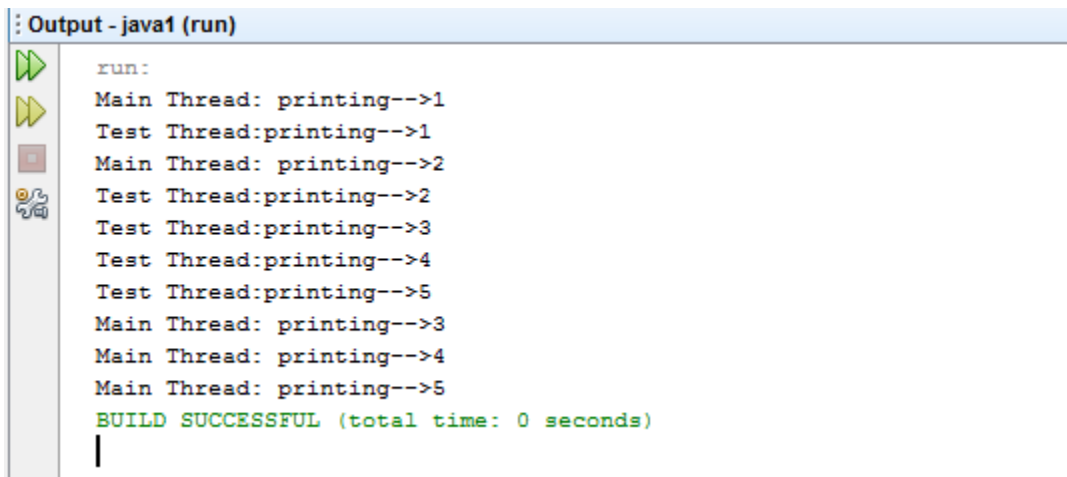
**Example1:**

```
//simple program to illustrate thread creation by
//implementing the Runnable Interface
package examples;

public class UsingRunnable implements Runnable {
    public void run(){
        for(int i =1;i<=5;i++)
        {
            System.out.println("Test Thread:printing-->" +i);
        }
    }

    public static void main(String[] args) {
        UsingRunnable r1 = new UsingRunnable();
        Thread t1 = new Thread(r1);
        t1.start();
        for(int i =1 ;i<=5;i++)
        {
            System.out.println("Main Thread: printing-->" +i);
        } } }

//Output: (may vary in different executions and environment!)
```



```
Output - java1 (run)
run:
Main Thread: printing-->1
Test Thread:printing-->1
Main Thread: printing-->2
Test Thread:printing-->2
Test Thread:printing-->3
Test Thread:printing-->4
Test Thread:printing-->5
Main Thread: printing-->3
Main Thread: printing-->4
Main Thread: printing-->5
BUILD SUCCESSFUL (total time: 0 seconds)
```

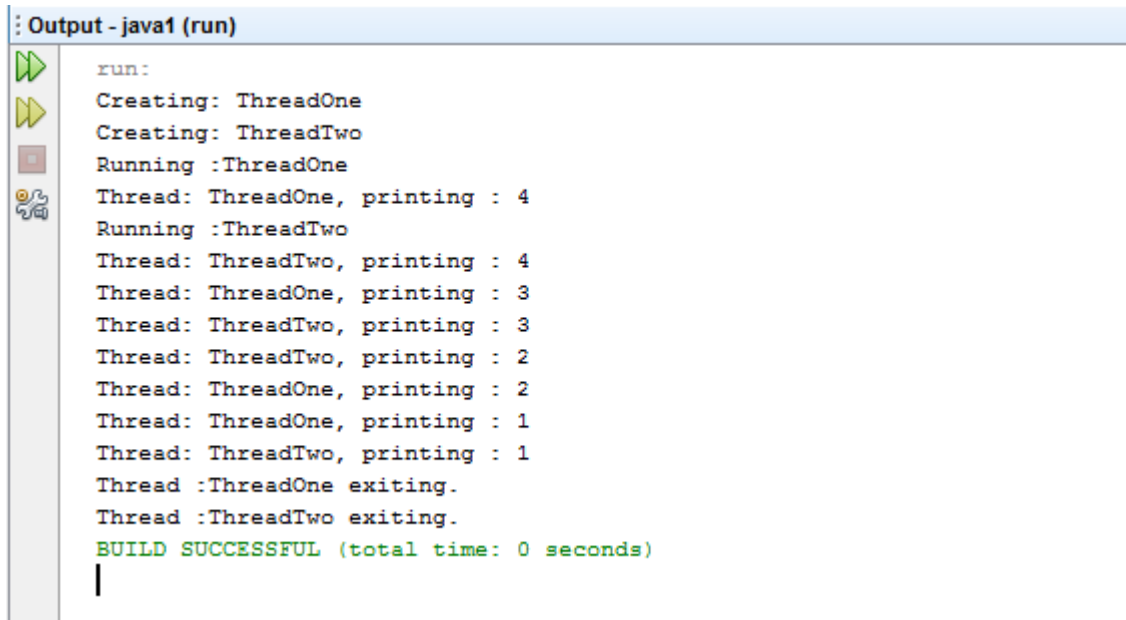
**Example2:**

//Another example for Thread creation using Runnable Interface  
package examples;

```
class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;
    RunnableDemo(String name) {
        threadName = name;
        System.out.println("Creating: " + threadName );
    }

    public void run() {
        System.out.println("Running : " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ",
printing : " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread : " + threadName + "
interrupted.");
        }
        System.out.println("Thread : " + threadName + "
exiting.");
    }
}
```

```
public class TestThread {  
    public static void main(String args[]) {  
        RunnableDemo r1 = new RunnableDemo( "ThreadOne");  
        Thread t1 = new Thread(r1);  
        t1.start();  
        RunnableDemo r2 = new RunnableDemo( "ThreadTwo");  
        Thread t2 = new Thread(r2);  
        t2.start();  
    }  
}
```



```
Output - java1 (run)  
run:  
Creating: ThreadOne  
Creating: ThreadTwo  
Running :ThreadOne  
Thread: ThreadOne, printing : 4  
Running :ThreadTwo  
Thread: ThreadTwo, printing : 4  
Thread: ThreadOne, printing : 3  
Thread: ThreadTwo, printing : 3  
Thread: ThreadTwo, printing : 2  
Thread: ThreadOne, printing : 2  
Thread: ThreadOne, printing : 1  
Thread: ThreadTwo, printing : 1  
Thread :ThreadOne exiting.  
Thread :ThreadTwo exiting.  
BUILD SUCCESSFUL (total time: 0 seconds)  
|
```

**Creating a Thread by extending the Thread Class**

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

**Step 1**

You will need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method –

```
public void run( )
```

**Step 2**

Once Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method –

```
void start( );
```

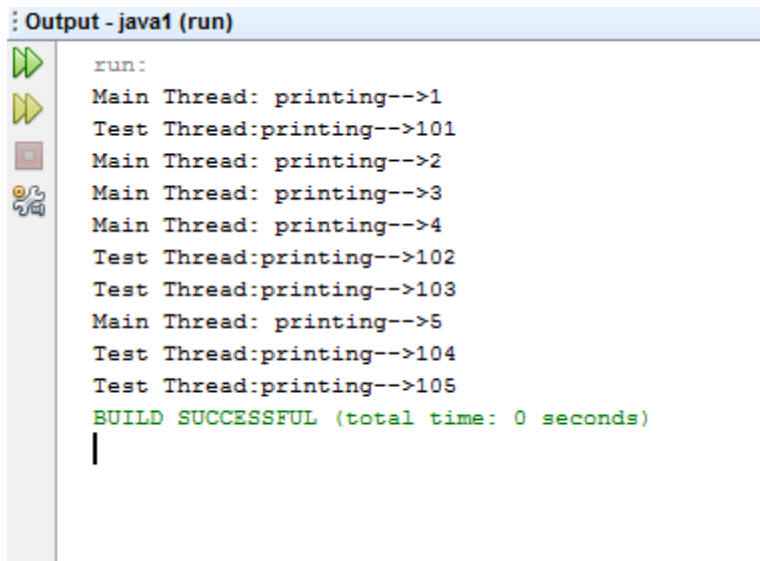
**Example1:**

```
//simple program to illustrate thread creation by
//extending the Thread Class
package examples;
public class UsingThread extends Thread {
    public void run(){
        for(int i =101;i<=105;i++)
        {
            System.out.println("Test Thread:printing-->" +i);
        }
    }
}
```

```

public static void main(String[] args) {
    UsingThread t1 = new UsingThread();
    t1.start();
    for(int i =1 ;i<=5;i++)
    {
        System.out.println("Main Thread: printing-->" +i);
    }
}

```



```

Output - java1 (run)
run:
Main Thread: printing-->1
Test Thread:printing-->101
Main Thread: printing-->2
Main Thread: printing-->3
Main Thread: printing-->4
Test Thread:printing-->102
Test Thread:printing-->103
Main Thread: printing-->5
Test Thread:printing-->104
Test Thread:printing-->105
BUILD SUCCESSFUL (total time: 0 seconds)
|

```

Example2:

//Another example for Thread creation by

//extending the Thread Class

package examples;

class ThreadDemo extends Thread {

private String threadName;

ThreadDemo(String name) {

threadName = name;

System.out.println("Creating: " + threadName );

}

```

public void run() {
    System.out.println("Running :" + threadName );
    try {
        for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", printing : " + i);
            // Let the thread sleep for a while.
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println("Thread :" + threadName + " interrupted.");
    }
    System.out.println("Thread :" + threadName + " exiting.");
}
}

public class TestingThread {
    public static void main(String args[]) {
        Thread t1 = new ThreadDemo("Thread111");
        t1.start();
        Thread t2 = new ThreadDemo("Thread222");
        t2.start();
    }
}

```

```

Output - java1 (run)
run:
Creating: Thread111
Creating: Thread222
Running :Thread111
Thread: Thread111, printing : 4
Running :Thread222
Thread: Thread222, printing : 4
Thread: Thread111, printing : 3
Thread: Thread222, printing : 3
Thread: Thread111, printing : 2
Thread: Thread222, printing : 2
Thread: Thread111, printing : 1
Thread: Thread222, printing : 1
Thread :Thread222 exiting.
Thread :Thread111 exiting.
BUILD SUCCESSFUL (total time: 4 seconds)

```

### Which approach to follow? (Implementing Runnable or extending Thread)

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is **run ( )**. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**. Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class. Ultimately, which approach to use is up to you.



## Creating multiple threads

A program can spawn as many threads as it needs. For example, the following program creates three **child threads**:

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

**The output from this program is shown here:**

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to `sleep(10000)` in `main( )`. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

## Thread Methods

Following is the list of important methods available in the Thread class. (**non-static methods** )

Sr.No.	Method & Description
1	<b>public void start()</b> Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	<b>public void run()</b> If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	<b>public final void setName(String name)</b> Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	<b>public final void setPriority(int priority)</b> Sets the priority of this Thread object. The possible values are between 1 and 10.
5	<b>public final void setDaemon(boolean on)</b> A parameter of true denotes this Thread as a daemon thread.
6	<b>public final void join(long millisec)</b> The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	<b>public void interrupt()</b> Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	<b>public final boolean isAlive()</b> Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the **static methods** performs the operation on the currently running thread.

**(static methods )**

Sr.No.	Method & Description
1	<b>public static void yield()</b> Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	<b>public static void sleep(long millisec)</b> Causes the currently running thread to block for at least the specified number of milliseconds.
3	<b>public static boolean holdsLock(Object x)</b> Returns true if the current thread holds the lock on the given Object.
4	<b>public static Thread currentThread()</b> Returns a reference to the currently running thread, which is the thread that invokes this method.
5	<b>public static void dumpStack()</b> Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

### Using `isAlive()` and `join()`

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling `sleep()` within `main()`, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, `Thread` provides a means by which you can answer this question.

- Two ways exist to determine whether a thread has finished. First, you can call `isAlive()` on the thread. This method is defined by **`Thread`**, and its general form is shown here:

```
final boolean isAlive( )
```

The `isAlive()` method returns **`true`** if the thread upon which it is called is still running. It returns **`false`** otherwise.

- While `isAlive()` is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called `join()`, shown here:

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of `join()` allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
//To Demonstrate isAlive() and join() methods
package examples;

class First extends Thread{
    public void run(){
        try{
            for(int i=0;i<5;i++){
                System.out.println("WOW");
                Thread.sleep(1000);
            }
        }catch(Exception e){
            System.out.println(e);
        }
    }
}
```

```

class Second extends Thread{
public void run(){
    try{
        for(int i=0;i<5;i++){
            System.out.println("AWESOME");
            Thread.sleep(1000);
        }
    }catch(Exception e){
        System.out.println(e);
    }
}
}

public class IsAliveAndJoin{
    public static void main(String[] args) {
        First t1 = new First();
        Second t2 = new Second();
        t1.start();
        System.out.println("Is t1 alive?? "+t1.isAlive());
        t2.start();
        System.out.println("Is t2 alive?? "+t2.isAlive());
        System.out.println("I am from main thread and I can
appear anywhere");
        try{
            t1.join();//main thread waits until t1 finishes its
task
            t2.join();//main thread waits until t2 finishes its
task
        }catch(Exception e){
            System.out.println(e);
        }
    }
}

```

```

    }

    System.out.println("Is t1 alive?? "+t1.isAlive());

    System.out.println("Is t2 alive?? "+t2.isAlive());

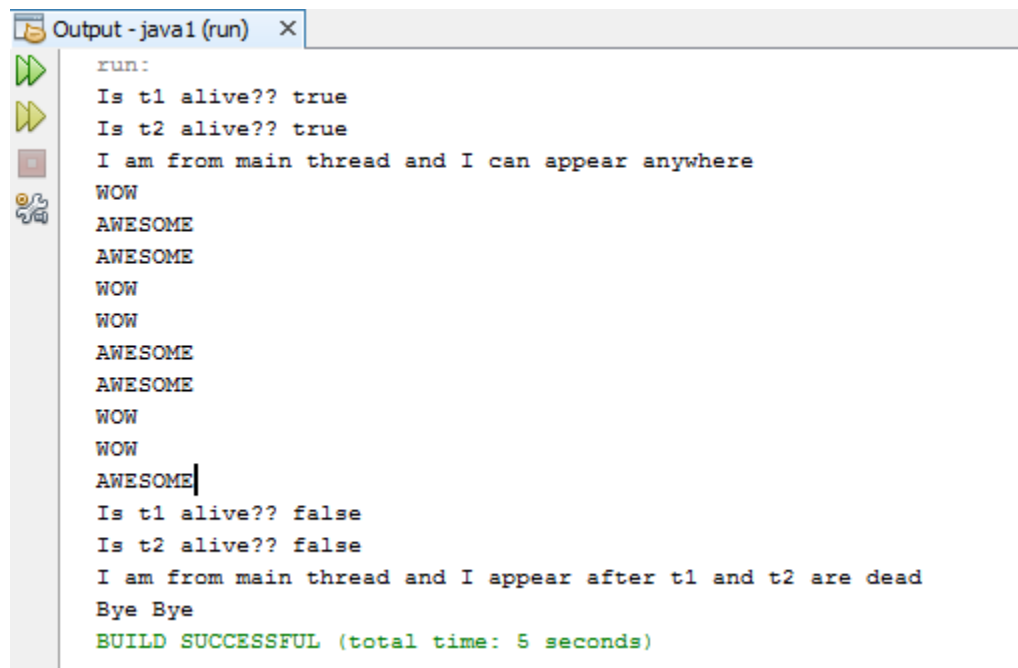
    System.out.println("I am from main thread and I appear
after t1 and t2 are dead");

    System.out.println("Bye Bye");

}

}

```



```

Output - java1 (run) X
run:
Is t1 alive?? true
Is t2 alive?? true
I am from main thread and I can appear anywhere
WOW
AWESOME
AWESOME
WOW
WOW
AWESOME
AWESOME
WOW
WOW
AWESOME
Is t1 alive?? false
Is t2 alive?? false
I am from main thread and I appear after t1 and t2 are dead
Bye Bye
BUILD SUCCESSFUL (total time: 5 seconds)

```

## Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.

Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a **context switch**. The rules that determine when a context switch takes place are simple:

- ✎ **A thread can voluntarily relinquish control.** This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- ✎ **A thread can be preempted by a higher-priority thread.** In this case, a lower-priority thread that does not yield the processor is simply preempted - no matter what it is doing by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

*Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.*

*Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.*

**Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.**

**Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.** In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread. In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.

- ✎ To set a thread's priority, use the **setPriority()** method, which is a member of Thread.

This is its general form:



```
final void setPriority(int level)
```

Here, **level** specifies the new priority setting for the calling thread. The value of level must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`.

Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5. These priorities are defined as static final variables within `Thread`.

- ✎ You can obtain the current priority setting by calling the **getPriority( )** method of `Thread`, shown here:

```
final int getPriority( )
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

- ✎ In a Multi-threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.
- ✎ Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in `Thread` class for priority.
  - **public static int MIN\_PRIORITY**: This is minimum priority that a thread can have. Value for this is 1.
  - **public static int NORM\_PRIORITY**: This is default priority of a thread if do not explicitly define it. Value for this is 5
  - **public static int MAX\_PRIORITY**: This is maximum priority of a thread. Value for this is 10.

#### Get and Set Thread Priority:

**public final int getPriority():** this method returns priority of given thread.

**public final void setPriority(int newPriority):** this method changes the priority of thread to the value **newPriority**. This method throws **IllegalArgumentException** if value of parameter **newPriority** goes beyond minimum(1) and maximum(10) limit.

**Example**

```

package examples;

class ThreadPriorityDemo extends Thread{

    ThreadPriorityDemo(String n){

        super(n);

    }

    public void run(){

        //getting thread names and priority values and printing
        them

        System.out.println("running thread name
is:"+Thread.currentThread().getName()

        +"\t thread priority
is:"+Thread.currentThread().getPriority());

    }

    public static void main(String args[]){

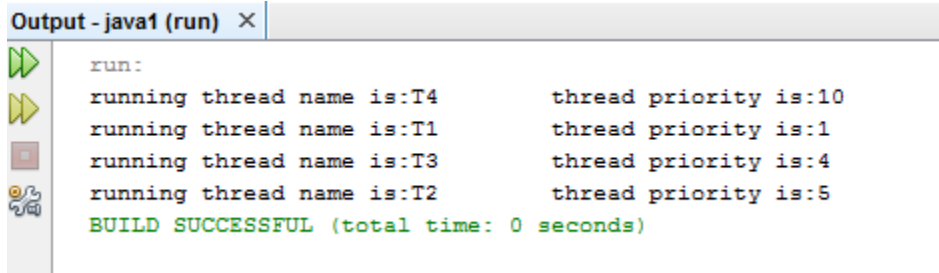
        ThreadPriorityDemo tpd1=new ThreadPriorityDemo("T1");
        ThreadPriorityDemo tpd2=new ThreadPriorityDemo("T2");
        ThreadPriorityDemo tpd3=new ThreadPriorityDemo("T3");
        ThreadPriorityDemo tpd4=new ThreadPriorityDemo("T4");

        //setting thread priorities

        tpd1.setPriority(Thread.MIN_PRIORITY);
        tpd2.setPriority(Thread.NORM_PRIORITY);
        tpd3.setPriority(4);
        tpd4.setPriority(Thread.MAX_PRIORITY);
    }
}

```

```
tpd1.start();  
tpd2.start();  
tpd3.start();  
tpd4.start();  
}  
}
```



```
run:  
running thread name is:T4      thread priority is:10  
running thread name is:T1      thread priority is:1  
running thread name is:T3      thread priority is:4  
running thread name is:T2      thread priority is:5  
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Thread Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

- **Synchronization in java is the capability to control the access of multiple threads to any shared resource.**
- **Java Synchronization is better option where we want to allow only one thread to access the shared resource.**
- **The synchronization is mainly used to:**
  - **To prevent thread interference.**
  - **To prevent consistency problem.**
- **Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.**

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor. A monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways.

1. Using `Synchronized_Methods`
2. Using `Synchronized Statement`

### Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

- ✂ **If you declare any method as synchronized, it is known as synchronized method.**
- ✂ **Synchronized method is used to lock an object for any shared resource.**
- ✂ **When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.**

```
//This program is not synchronized
package examples;
class MulTable{
void printTable(int n){//method not synchronized
    System.out.println("Table of " + n);
    for(int i=1;i<=10;i++){
        System.out.println(n*i);
        try{
            Thread.sleep(400);
        }catch(InterruptedException e){
            System.out.println(e);
        }
    }
}
```

```

    }
}

class MyThread1 extends Thread{
    MulTable t;
    MyThread1(MulTable t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }
}

class MyThread2 extends Thread{
    MulTable t;
    MyThread2(MulTable t){
        this.t=t;
    }
    public void run(){
        t.printTable(100);
    }
}

public class SynMethodDemo{
    public static void main(String args[]){
        //creating MulTable object
        MulTable obj = new MulTable();
    }
}

```

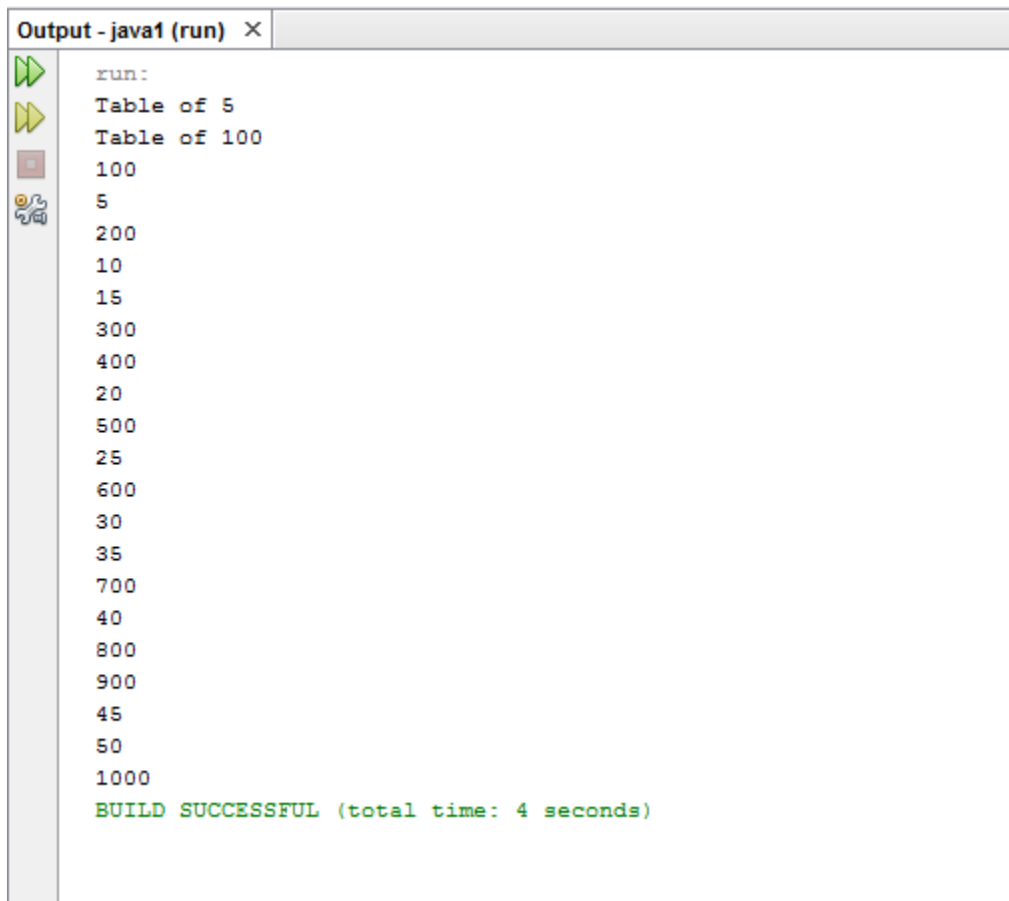
```
//creating threads which shares same object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);

//statrting threads
t1.start();
t2.start();

}

}
```

Output :



```
Output - java1 (run) X
run:
Table of 5
Table of 100
100
5
200
10
15
300
400
20
500
25
600
30
35
700
40
800
900
45
50
1000
BUILD SUCCESSFUL (total time: 4 seconds)
```

After adding synchronized keyword in method .....

```

//This program is synchronized
package examples;

class MulTable{
synchronized void printTable(int n) { //method is now synchronized
    System.out.println("Table of " + n);
    for(int i=1;i<=10;i++){
        System.out.println(n*i);
        try{
            Thread.sleep(400);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

class MyThread1 extends Thread{
    MulTable t;
    MyThread1(MulTable t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }
}

class MyThread2 extends Thread{

```



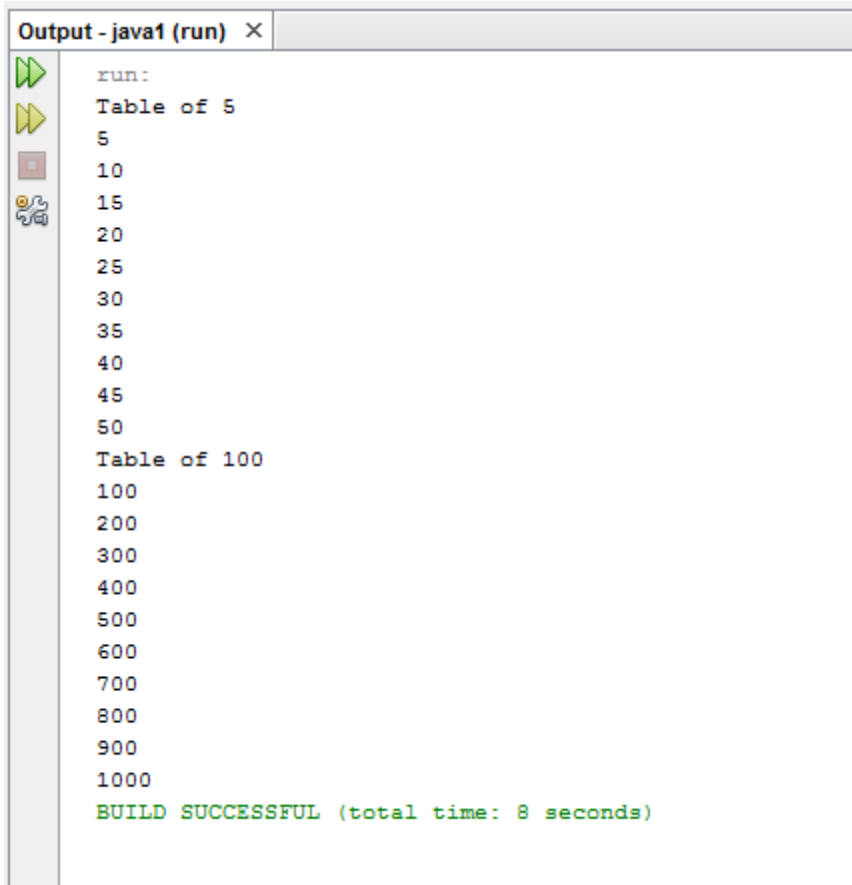
```

MulTable t;
MyThread2(MulTable t){
    this.t=t;
}
public void run(){
    t.printTable(100);
}
}

public class SynMethodDemo{
    public static void main(String args[]){
        //creating MulTable object
        MulTable obj = new MulTable();
        //creating threads which shares same object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        //statrting threads
        t1.start();
        t2.start();
    }
}

```

Output



```

Output - java1 (run) X
run:
Table of 5
5
10
15
20
25
30
35
40
45
50
Table of 100
100
200
300
400
500
600
700
800
900
1000
BUILD SUCCESSFUL (total time: 8 seconds)

```

### The synchronized Statement (Synchronized block)

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized block**.

This is the general form of the synchronized statement:

```

synchronized(objRef) {
    // statements to be synchronized
}

```

Here, **objRef** is a reference to the object being synchronized. A **synchronized block** ensures that a call to a synchronized method that is a member of **objRef**'s class occurs only after the current thread has successfully entered **objRef**'s monitor.

- **Synchronized block can be used to perform synchronization on any specific resource of the method.**
- **Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.**
- **If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.**
- **Synchronized block is used to lock an object for any shared resource.**
- **Scope of synchronized block is smaller than the method.**

```

//This program is to show the multithreading example with
//synchronization using synchronized block.

package examples;

class Table{

void printTable(int n){

    synchronized(this){//synchronized block

        System.out.println("Table of " + n);

        for(int i=1;i<=10;i++){

            System.out.println(n*i);

            try{

                Thread.sleep(400);

            }catch(InterruptedException e){

                System.out.println(e);

            }

        }

    }

}

}

}

}

class MyThread11 extends Thread{

    Table t;

    MyThread11(Table t){

        this.t=t;

    }

    public void run(){

        t.printTable(5);

    }

}

```

```

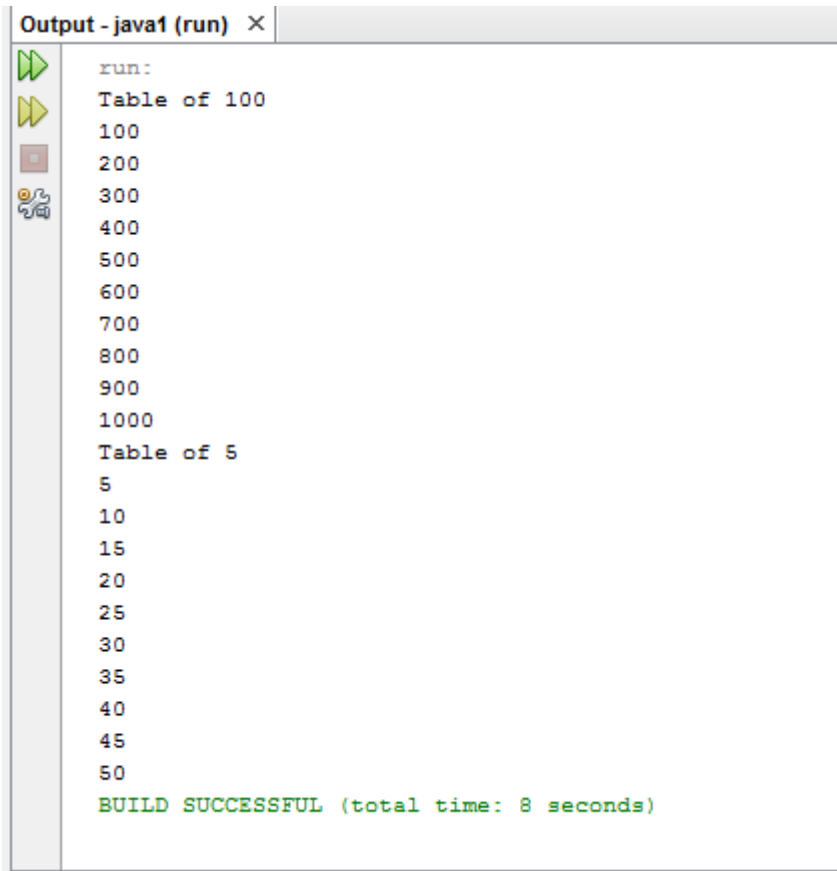
    }

    class MyThread22 extends Thread{
        Table t;
        MyThread22(Table t){
            this.t=t;
        }
        public void run(){

            t.printTable(100);
        }
    }

    public class SynBlockDemo{
        public static void main(String args[]){
            //creating Table object
            Table obj = new Table();
            //creating threads which shares same object
            MyThread11 t1=new MyThread11(obj);
            MyThread22 t2=new MyThread22(obj);
            //statrting threads
            t1.start();
            t2.start();
        }
    }

```



The image shows a screenshot of an IDE's output window. The window has a title bar that reads "Output - java1 (run) X". On the left side of the window, there is a vertical toolbar with four icons: a green play button, a yellow play button, a red stop button, and a blue and yellow refresh button. The main area of the window contains the following text:

```
run:
Table of 100
100
200
300
400
500
600
700
800
900
1000
Table of 5
5
10
15
20
25
30
35
40
45
50
BUILD SUCCESSFUL (total time: 8 seconds)
```

**Interthread communication**

If you are aware of interprocess communication then it will be easy for you to understand interthread communication. Interthread communication is important when you develop an application where two or more threads exchange some information.

There are three simple methods and a little trick which makes thread communication possible. All the three methods are listed below –

S.No.	Method & Description
1	<code>public void wait()</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> .
2	<code>public void notify()</code> Wakes up a single thread that is waiting on this object's monitor.
3	<code>public void notifyAll()</code> Wakes up all the threads that called <code>wait( )</code> on the same object.

These methods have been implemented as final methods in `Object`, so they are available in all the classes. All three methods can be called only from within a synchronized context.

**Understanding the process of inter-thread communication**

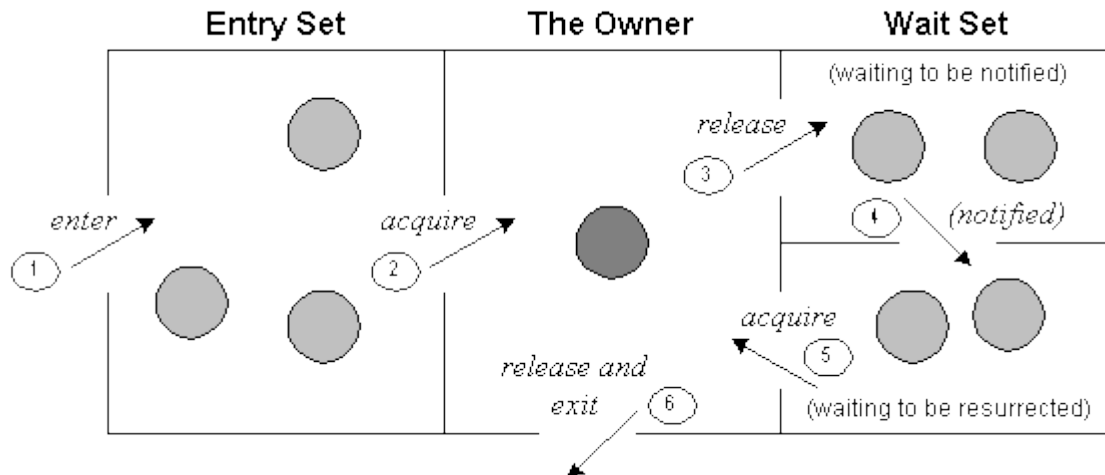


Figure 1: Inter-thread Communication

The point to point explanation of the above diagram is as follows:

- Threads enter to acquire lock.
- Lock is acquired by one thread.
- Now thread goes to waiting state if you call **wait()** method on the object. Otherwise it releases the lock and exits.
- If you call **notify()** or **notifyAll()** method, thread moves to the notified state (runnable state).
- Now thread is available to acquire lock.
- After completion of the task, thread releases the lock and exits the monitor state of the object.

**Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?**

- *It is because they are related to lock and object has a lock.*

**Difference between wait and sleep?**



<b>wait()</b>	<b>sleep()</b>
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

## Thread Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example.

```
public class TestThread {

    public static Object Lock1 = new Object();

    public static Object Lock2 = new Object();

    public static void main(String args[]) {

        ThreadDemo1 T1 = new ThreadDemo1();

        ThreadDemo2 T2 = new ThreadDemo2();

        T1.start();

        T2.start();

    }

    private static class ThreadDemo1 extends Thread {

        public void run() {

            synchronized (Lock1) {

                System.out.println("Thread 1: Holding lock 1...");

                try { Thread.sleep(10); }

                catch (InterruptedException e) {}

            }

        }

    }

}
```

```
        System.out.println("Thread 1: Waiting for lock 2...");

        synchronized (Lock2) {

            System.out.println("Thread 1: Holding lock 1 & 2...");

        }

    }

}

private static class ThreadDemo2 extends Thread {

    public void run() {

        synchronized (Lock2) {

            System.out.println("Thread 2: Holding lock 2...");

            try { Thread.sleep(10); }

            catch (InterruptedException e) {}

            System.out.println("Thread 2: Waiting for lock 1...");

            synchronized (Lock1) {

                System.out.println("Thread 2: Holding lock 1 & 2...");

            }

        }

    }

}
```

```

    }
}

```

When you compile and execute the above program, you find a deadlock situation and following is the output produced by the program –

### Output

```

Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...

```

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL+C.

### Deadlock Solution Example

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other –

```

public class TestThread {

    public static Object Lock1 = new Object();

    public static Object Lock2 = new Object();

    public static void main(String args[]) {

        ThreadDemo1 T1 = new ThreadDemo1();

        ThreadDemo2 T2 = new ThreadDemo2();

        T1.start();
    }
}

```

```

    T2.start();

}

private static class ThreadDemo1 extends Thread {

    public void run() {

        synchronized (Lock1) {

            System.out.println("Thread 1: Holding lock 1...");

            try {

                Thread.sleep(10);

            } catch (InterruptedException e) {}

            System.out.println("Thread 1: Waiting for lock 2...");

            synchronized (Lock2) {

                System.out.println("Thread 1: Holding lock 1 & 2...");

            }

        }

    }

}

private static class ThreadDemo2 extends Thread {

    public void run() {

        synchronized (Lock1) {

```

```

        System.out.println("Thread 2: Holding lock 1...");

        try {

            Thread.sleep(10);

        } catch (InterruptedException e) {}

        System.out.println("Thread 2: Waiting for lock 2...");

        synchronized (Lock2) {

            System.out.println("Thread 2: Holding lock 1 & 2...");

        }

    }

}

```

So just changing the order of the locks prevent the program in going into a deadlock situation and completes with the following result –

### Output

```

Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...

```

The above example is to just make the concept clear, however, it is a complex concept and you should deep dive into it before you develop your applications to deal with deadlock situations.

## Interrupting a Thread

If any thread is in sleeping or waiting state (i.e. `sleep()` or `wait()` is invoked), calling the `interrupt()` method on the thread, breaks out the sleeping or waiting state throwing `InterruptedException`. If the thread is not in the sleeping or waiting state, calling the `interrupt()` method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

### The 3 methods provided by the Thread class for interrupting a thread

- `public void interrupt()`
- `public static boolean interrupted()`
- `public boolean isInterrupted()`

## Advantages of Java Multithreading

- It doesn't block the user because threads are independent and you can perform multiple operations at same time.
- You can perform many operations together so it saves time.
- Threads are independent so it doesn't affect other threads if exception occur in a single thread.

## Java Wrapper Classes

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the **java.lang** package, which is imported by default into all Java programs.

The wrapper classes in java servers two primary purposes.

- To provide a mechanism to 'wrap' primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, HashSet, HashMap etc. collection.
- To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int    x = 25;  
Integer y = new Integer(33);
```

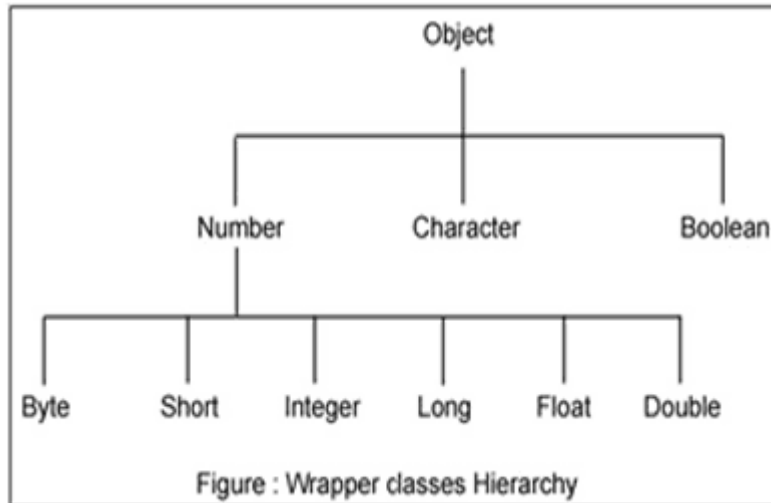
The first statement declares an **int variable** named x and initializes it with the value 25. The second statement instantiates an **Integer object**. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.



Below table lists wrapper classes in Java API with constructor details.

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API



As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in the constructor. For example to create Integer object we can have the following syntax.

```
Integer intObj = new Integer (25);
Integer intObj2 = new Integer ("25");
```

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

```
Integer intObj3 = new Integer ("Two");
```

The following discussion focuses on the Integer wrapperclass, but applies in a general sense to all eight wrapper classes.

Method	Purpose
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i

<code>byteValue()</code>	returns the value of this Integer as a byte
<code>doubleValue()</code>	returns the value of this Integer as a double
<code>floatValue()</code>	returns the value of this Integer as a float
<code>intValue()</code>	returns the value of this Integer as an int
<code>shortValue()</code>	returns the value of this Integer as a short
<code>longValue()</code>	returns the value of this Integer as a long
<code>int compareTo(int i)</code>	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
<code>static int compare(int num1, int num2)</code>	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
<code>boolean equals(Object intObj)</code>	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

The most common methods of the Integer wrapper class are summarized in above table. Similar methods for the other wrapper classes are found in the Java API documentation.

Let's see java program which explains few wrapper classes methods.

```

public class WrapperDemo {

    public static void main (String args[]){

        Integer intObj1 = new Integer (25);

        Integer intObj2 = new Integer ("25");

        Integer intObj3= new Integer (35);

        //compareTo demo

        System.out.println("Comparing using compareTo Obj1 and Obj2: " +
intObj1.compareTo(intObj2));

        System.out.println("Comparing using compareTo Obj1 and Obj3: " +
intObj1.compareTo(intObj3));

        //Equals demo

        System.out.println("Comparing using equals Obj1 and Obj2: " +
intObj1.equals(intObj2));

        System.out.println("Comparing using equals Obj1 and Obj3: " +
intObj1.equals(intObj3));

        Float f1 = new Float("2.25f");

        Float f2 = new Float("20.43f");

        Float f3 = new Float(2.25f);

        System.out.println("Comparing using compare f1 and f2: "
+Float.compare(f1,f2));

        System.out.println("Comparing using compare f1 and f3: "
+Float.compare(f1,f3));

        //Addition of Integer with Float

        Float f = intObj1.floatValue() + f1;

        System.out.println("Addition of intObj1 and f1: "+ intObj1 +"+"
+f1+"=" +f );

    }

}

```

**valueOf (), toHexString(), toOctalString() and toBinaryString() Methods:**

This is another approach to creating wrapper objects. We can convert from binary or octal or hexadecimal before assigning a value to wrapper object using two argument constructor. Below program explains the method in details

```
public class ValueOfDemo {
    public static void main(String[] args) {
        Integer intWrapper = Integer.valueOf("12345");
        //Converting from binary to decimal
        Integer intWrapper2 = Integer.valueOf("11011", 2);
        //Converting from hexadecimal to decimal
        Integer intWrapper3 = Integer.valueOf("D", 16);
        System.out.println("Value of intWrapper Object: "+
intWrapper);
        System.out.println("Value of intWrapper2 Object: "+
intWrapper2);
        System.out.println("Value of intWrapper3 Object: "+
intWrapper3);
        System.out.println("Hex value of intWrapper: " +
Integer.toHexString(intWrapper));
        System.out.println("Binary Value of intWrapper2: "+
Integer.toBinaryString(intWrapper2));
    }
}
```

- ❖ Each of primitive data types has dedicated class in java library.
- ❖ Wrapper class provides many methods while using collections like sorting, searching etc.



## Java Strings

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

### Creating Strings

The most direct way to create a string is to write –

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!".

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

```
public class StringDemo {  
  
    public static void main(String args[]) {  
  
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };  
  
        String helloString = new String(helloArray);  
  
        System.out.println( helloString );  
  
    }  
}
```

Note – *The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.*

### String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the **length()** method, which returns the number of characters contained in the string object.

The following program is an example of length(), method String class.

```
public class StringDemo {  
  
    public static void main(String args[]) {  
  
        String palindrome = "Dot saw I was Tod";  
  
        int len = palindrome.length();  
  
        System.out.println( "String Length is : " + len );  
  
    }  
}
```

This will produce the following result –

### Output

```
String Length is : 17
```

### Concatenating Strings

The String class includes a method for concatenating two strings –

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in –



```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in –

```
"Hello," + " world" + "!"
```

which results in –

```
"Hello, world!"
```

Let us look at the following example –

```
public class StringDemo {  
  
    public static void main(String args[]) {  
  
        String string1 = "saw I was ";  
  
        System.out.println("Dot " + string1 + "Tod");  
  
    }  
}
```

This will produce the following result –

Output:

```
Dot saw I was Tod
```

## Creating Format Strings

You have **printf()** and **format()** methods to print output with formatted numbers. The **String** class has an equivalent class method, **format()**, that returns a **String** object rather than a **PrintStream** object.

Using **String**'s static **format()** method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of –

## Example

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

You can write –

```
String fs;

fs = String.format("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);

System.out.println(fs);
```

## String Methods

Here is the list of methods supported by **String** class –

S.No.	Method & Description
-------	----------------------

1	<code>char charAt(int index)</code> Returns the character at the specified index.
2	<code>int compareTo(Object o)</code> Compares this String to another Object.
3	<code>int compareTo(String anotherString)</code> Compares two strings lexicographically.
4	<code>int compareToIgnoreCase(String str)</code> Compares two strings lexicographically, ignoring case differences.
5	<code>String concat(String str)</code> Concatenates the specified string to the end of this string.
6	<code>boolean contentEquals(StringBuffer sb)</code> Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	<code>static String copyValueOf(char[] data)</code> Returns a String that represents the character sequence in the array specified.
8	<code>static String copyValueOf(char[] data, int offset, int count)</code> Returns a String that represents the character sequence in the array specified.
9	<code>boolean endsWith(String suffix)</code>

	Tests if this string ends with the specified suffix.
10	<code>boolean equals(Object anObject)</code> Compares this string to the specified object.
11	<code>boolean equalsIgnoreCase(String anotherString)</code> Compares this String to another String, ignoring case considerations.
12	<code>byte getBytes()</code> Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	<code>byte[] getBytes(String charsetName)</code> Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	<code>void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)</code> Copies characters from this string into the destination character array.
15	<code>int hashCode()</code> Returns a hash code for this string.
16	<code>int indexOf(int ch)</code> Returns the index within this string of the first occurrence of the specified character.
17	<code>int indexOf(int ch, int fromIndex)</code>

	Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	<code>int indexOf(String str)</code> Returns the index within this string of the first occurrence of the specified substring.
19	<code>int indexOf(String str, int fromIndex)</code> Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	<code>String intern()</code> Returns a canonical representation for the string object.
21	<code>int lastIndexOf(int ch)</code> Returns the index within this string of the last occurrence of the specified character.
22	<code>int lastIndexOf(int ch, int fromIndex)</code> Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	<code>int lastIndexOf(String str)</code> Returns the index within this string of the rightmost occurrence of the specified substring.
24	<code>int lastIndexOf(String str, int fromIndex)</code> Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

25	<code>int length()</code>  Returns the length of this string.
26	<code>boolean matches(String regex)</code>  Tells whether or not this string matches the given regular expression.
27	<code>boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)</code>  Tests if two string regions are equal.
28	<code>boolean regionMatches(int toffset, String other, int ooffset, int len)</code>  Tests if two string regions are equal.
29	<code>String replace(char oldChar, char newChar)</code>  Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	<code>String replaceAll(String regex, String replacement)</code>  Replaces each substring of this string that matches the given regular expression with the given replacement.
31	<code>String replaceFirst(String regex, String replacement)</code>  Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	<code>String[] split(String regex)</code>  Splits this string around matches of the given regular expression.

33	<p><code>String[] split(String regex, int limit)</code></p> <p>Splits this string around matches of the given regular expression.</p>
34	<p><code>boolean startsWith(String prefix)</code></p> <p>Tests if this string starts with the specified prefix.</p>
35	<p><code>boolean startsWith(String prefix, int toffset)</code></p> <p>Tests if this string starts with the specified prefix beginning a specified index.</p>
36	<p><code>CharSequence subSequence(int beginIndex, int endIndex)</code></p> <p>Returns a new character sequence that is a subsequence of this sequence.</p>
37	<p><code>String substring(int beginIndex)</code></p> <p>Returns a new string that is a substring of this string.</p>
38	<p><code>String substring(int beginIndex, int endIndex)</code></p> <p>Returns a new string that is a substring of this string.</p>
39	<p><code>char[] toCharArray()</code></p> <p>Converts this string to a new character array.</p>
40	<p><code>String toLowerCase()</code></p> <p>Converts all of the characters in this String to lower case using the rules of the default locale.</p>
41	<p><code>String toLowerCase(Locale locale)</code></p>

	Converts all of the characters in this String to lower case using the rules of the given Locale.
42	String toString()  This object (which is already a string!) is itself returned.
43	String toUpperCase()  Converts all of the characters in this String to upper case using the rules of the default locale.
44	String toUpperCase(Locale locale)  Converts all of the characters in this String to upper case using the rules of the given Locale.
45	String trim()  Returns a copy of the string, with leading and trailing whitespace omitted.
46	static String valueOf(primitive data type x)  Returns the string representation of the passed data type argument.

## Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.



- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

### Constructors of Java ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

### Methods of Java ArrayList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.

<code>int lastIndexOf(Object o)</code>	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
<code>Object[] toArray()</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>Object[] toArray(Object[] a)</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>boolean add(Object o)</code>	It is used to append the specified element to the end of a list.
<code>boolean addAll(int index, Collection c)</code>	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
<code>Object clone()</code>	It is used to return a shallow copy of an ArrayList.
<code>int indexOf(Object o)</code>	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
<code>void trimToSize()</code>	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Example:

```
import java.util.*;
class TestCollection1{
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");
        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```