



## Module 2: Core Java 8

### Module Overview

In this module, students will be able to familiarize with Java and Core Java. Apart from above, they will be able to understand Java Architecture and Advantages of the same.



## Module Objective

**At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:**

- Implementing OOPs features in Java
- Developing Java Desktop Applications
- Use of Core JDK 1.8 API
- Testing using Junit 4
- Implementing Multithreading



## Java Introduction

Java is a general-purpose, object-oriented programming language. Java was developed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. Originally, it was called '**Oak**' by James Gosling (one of the developers), then it was renamed as '**Java**', 1995. Java name is derived from Java Coffee, it comes from the Island of Java. During the 1600s, the Dutch introduced coffee to Southeast Asia. They brought coffee trees to places like Bali and Sumatra, where it's still grown today. Java has never been a popular name for coffee, although it's consistently been used, and most coffee drinkers are familiar with the term. Java may not be the most common name for coffee, but it stands alone as the only name that has inspired a computer programming language.

The Java team which included Patrick Naughton discovered that the existing languages like C and C++ had limitations



in terms of both reliability and portability. However, they modelled their new language Java on C and C++ but removed a number of features of C and C++ that were considered as sources of problems and thus made Java really simple, reliable, portable and powerful language.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics.

However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

The most striking feature of the language is that it is a **platform-neutral** language. Java is the first programming language that is not tied to any particular hardware or operating system. Programs developed in Java can be executed anywhere on any system. We can call Java as a revolutionary technology because it has brought in a fundamental shift in how we develop and use programs. Nothing like this has happened to the software industry before.



## Java Architecture

As one of the main advantages of Java is that it is platform independent. In Java, **JVM** (Java Virtual Machine) is a component which provides an environment for running Java Programs. At the runtime, JVM interprets the bytecode into machine code which will be executed the machine in which the Java program runs.

Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into bytecode. It is very important as a developer that we should know the Architecture of the JVM, as it enables us to write code more efficiently. Every Java developer knows that bytecode will be executed by JRE (Java Runtime Environment). But many don't know the fact that JRE is the implementation of Java Virtual Machine (JVM), which analyzes the bytecode, interprets the code, and executes it.

A JRE performs the following 3 main tasks respectively.

- 1) Class Loader Subsystem
- 2) Runtime Data Area
- 3) Execution Engine

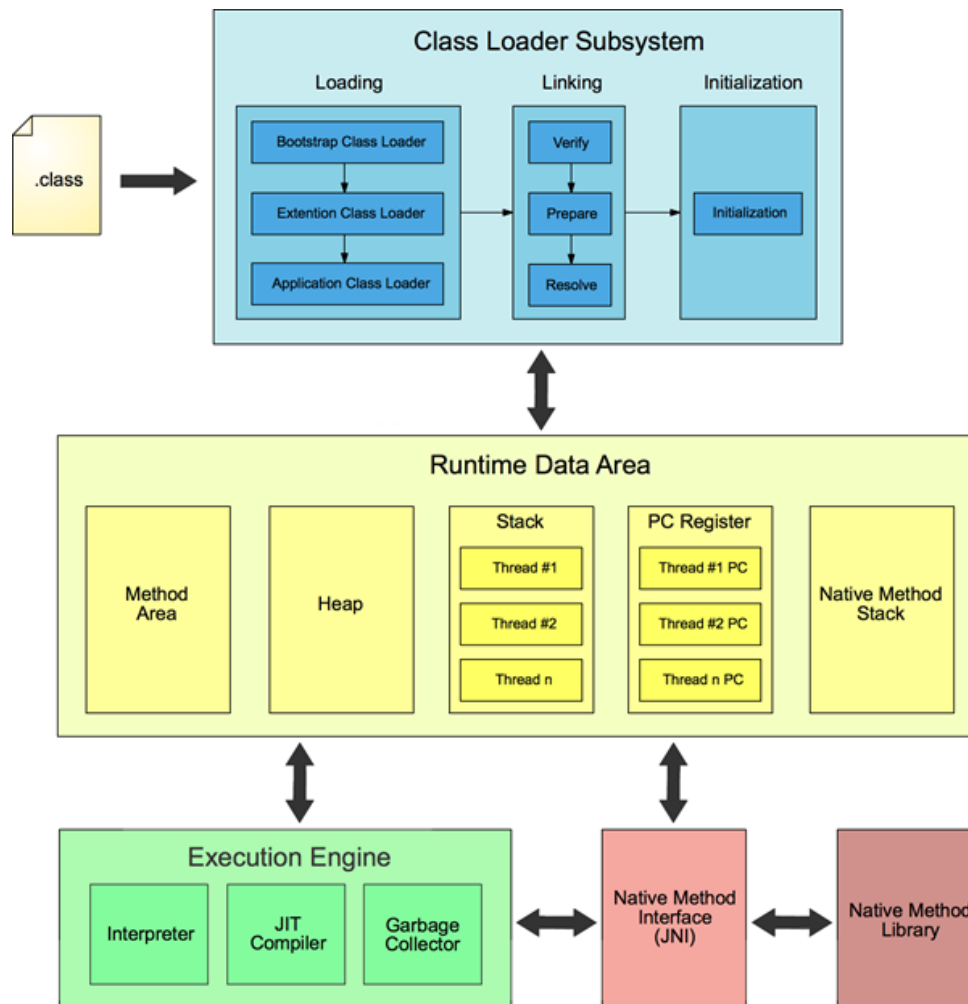


Fig. JVM Architecture Diagram

## 1. Class Loader Subsystem

Java's dynamic class loading functionality is handled by the class loader subsystem. It loads links and initializes the class file when it refers to a class for the first time at runtime, not compile time. The class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes

obtained through the network from the classes available locally. Once the bytecode is loaded successfully, the next step is bytecode verification by bytecode verifier.

## 2. Runtime Data Area

Runtime data area or Bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

- The code follows JVM specifications.
- There is no unauthorized access to memory.
- The code does not cause any stack overflows.
- There are no illegal data conversions in the code such as float to object references.

Further runtime data area is divided into 5 parts:

- Method Area :- All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
- Heap Area :- All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM.
- Stack Area :- For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame.
- PC Registers :- Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
- Native Method stacks :- Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

## 3. Execution Engine

As we saw earlier when the Java program is executed, the byte code is interpreted by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE comprises the component JIT compiler. JIT makes the execution faster. The bytecode which is assigned to the Runtime Data Area will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

Once the bytecode is compiled into that particular machine code, it is cached by the JIT (Just In Time) compiler and will be reused for the future needs. Hence the main performance improvement by using JIT compiler can be seen when the same code is executed again and again because JIT makes use of the machine code which is cached and stored.



## Advantages of Java

**Advantages of Java are:**

- Easy to learn

- Java was designed to be easy to write, compile, debug and learn other than other programming languages.
- Object Oriented Programming
  - This allows you to create modular programs and reusable code.
- Rich APIs
  - Java offers APIs for various activities like database connection, networking, I/O, XML parsing, utilities and much more.
- Platform Independent
  - One of the most significant advantages of Java is its ability to move easily from one computer system to another.
- Powerful Open source rapid development tools
  - Large number of open source development tools that use Java have made Java an even more powerful option for developers i.e. Eclipse and Netbeans.
- Open source Libraries
  - The large number of open source libraries and well matured with industrial support have also ensured that Java gets to be used everywhere.
- Free
  - When it comes to development, we are today faced with intense competition. Many small and medium enterprises want their software development and/or web app development done for their businesses, but they don't really have a big budget for that. With Java being free, it makes it cost effective.
- Community Support
  - Finally, there is extensive community support that Java has managed to muster. That goes a long way in helping new Java developers learn the art and become productive soon.

Congratulations!! Your Program Works!!



## Installing the Java Software/ Development Kit

Till now we got to know how Java was invented, what is its architecture and what are its advantages. Now, in here we will learn how to install JDK and Eclipse on your PC.

JRE is needed for running Java programs. JDK (Java Development Kit) which includes JRE plus development tools is needed for writing as well as reading Java Programs. We can also say that JRE is a subset of JDK.

When Java was created in the year 1991 with the name of Oak. However, Sun Microsystem announced Java on September 23, 1995, along with its framework JDK Alpha and Beta. After this, there were many changes made and many new features were in JDK. Every version had its own specifications and names such as JDK 1.0, JDK 1.1, J2SE 1.2, and so on. Currently, the version which is used is Java SE 14 (JDK 14) announced on March 17<sup>th</sup>, 2020.

- **Installation of JDK in Windows**

Let's get started with this....

The latest version of the JDK is Java SE 14, released in March 2020. However, we are going to use Eclipse 4.14 which is compatible with JDK 13. Hence, we are going to install JDK 13.



- ❖ You should create a NEW Java project for EACH of your Java application.
- ❖ Nonetheless, Eclipse allows you to keep more than one programs in a project, which is handy for writing toy programs.

To run a particular program, open and right-click on the source file ⇒ Run As ⇒ Java Application.

- ❖ Clicking the "Run" button (with a "Play" icon) runs the recently-run program (based on the previous configuration). Try clicking on the "down-arrow" besides the "Run" button.

- Implementation of these programming structures.



## **Variables**

A variable is a location in memory (storage area) to hold data. To indicate the storage area, each variable should be given a unique name (identifier).

### **How to declare variables in Java?**

Here's an example to declare a variable in Java.

```
int speedLimit = 80;
```

Here, speedLimit is a variable of int data type, and is assigned value 80. Meaning, the speedLimit variable can store integer values.

In the example, we have assigned value to the variable during declaration. However, it's not mandatory. You can declare variables without assigning the value, and later you can store the value as you wish. For example,

```
int speedLimit;
```

```
speedLimit = 80;
```

The value of a variable can be changed in the program, hence the name 'variable'. For example,

```
int speedLimit = 80;
```

```
.....
```

```
speedLimit = 80;
```

Java is a statically typed language. It means that all variables must be declared before they can be used.

Also, you cannot change the data type of a variable in Java within the same scope. What is variable scope? Don't worry about it for now. For now, just remember you cannot do something like this.

```
int speedLimit = 80;
```

```
.....
```

```
float speedLimit;
```

### **Rules for Naming Variables in Java**

Java programming language has its own set of rules and conventions for naming variables. Here's what you need to know:

- Variables in Java are case-sensitive.
- A variable's name is a sequence of Unicode letters and digits. It can begin with a letter, \$ or \_. However, it's convention to begin a variable name with a letter. Also, variable name cannot use whitespace in Java.

Variable Name	Remarks
<b>speed</b>	Valid variable name
<b>_speed</b>	Valid but bad variable name
<b>\$speed</b>	Valid but bad variable name
<b>speed1</b>	Valid variable name

<code>spe ed</code>	Invalid variable name
<code>spe'ed</code>	Invalid variable name

- When creating variables, choose a name that makes sense. For example, score, number, level makes more sense than variable name such as s, n, and l.
- If you choose one-word variable name, use all lowercase letters. For example, it's better to use speed rather than SPEED, or sPEED.
- If you choose variable name having more than one word, use all lowercase letters for the first word and capitalize the first letter of each subsequent word. For example, speedLimit.

There are 4 types of variables in Java programming language:

- Instance Variables (Non-Static Fields)
- Class Variables (Static Fields)
- Local Variables
- Parameters



## Datatypes

As mentioned above, Java is a statically typed language. This means that, all variables must be declared before they can be used.

```
int speed;
```

Here, speed is a variable, and the data type of the variable is int. The int data type determines that the speed variable can only contain integers.

In simple terms, a variable's data type determines the values a variable can store. There are 8 data types predefined in Java programming language, known as primitive data types.

In addition to primitive data types, there are also referenced data types in Java (you will learn about it in later chapters).

### 8 Primitive Data Types

#### a. boolean

- The boolean data type has two possible values, either true or false.
- Default value: false.
- They are usually used for true/false conditions.
- Example:

```
class BooleanExample {
    public static void main(String[] args) {
        boolean flag = true;
        System.out.println(flag);
    }
}
```



When you run the program, the output will be:

true

b. byte

- The byte data type can have values from -128 to 127 (8-bit signed two's complement integer).
- It's used instead of int or other integer data types to save memory if it's certain that the value of a variable will be within [-128, 127].
- Default value: 0
- Example:

```
class ByteExample {  
    public static void main(String[] args) {  
        byte range;  
        range 124;  
        System.out.println(range);  
        //Error code below. Why?  
        //range = 200  
    }  
}
```

When you run the program, the output will be:

124

c. short

- The short data type can have values from -32768 to 32767 (16-bit signed two's complement integer).
- It's used instead of other integer data types to save memory if it's certain that the value of the variable will be within [-32768, 32767].
- Default value: 0
- Example:

```
class ShortExample {  
    public static void main(String[] args) {  
        short temperature;  
        temeperature = -200;  
        System.out.println(temperature);  
    }  
}
```

When you run the program, the output will be:

`-200`

d. int

- The int data type can have values from -231 to 231-1 (32-bit signed two's complement integer).
- If you are using Java 8 or later, you can use unsigned 32-bit integer with minimum value of 0 and maximum value of 232-1.
- Default value: 0
- Example:

```
class IntExample {  
    public static void main(String[] args) {  
        int range = -4250000;  
        System.out.println(range);  
    }  
}
```

When you run the program, the output will be:

`-4250000`

e. long

- The long data type can have values from -263 to 263-1 (64-bit signed two's complement integer).
- If you are using Java 8 or later, you can use unsigned 64-bit integer with minimum value of 0 and maximum value of 264-1.
- Default value: 0
- Example:

```
class LongExample {  
    public static void main(String[] args) {  
        long range = -42332250000L;  
        System.out.println(range);  
    }  
}
```

When you run the program, the output will be:

```
-42332250000L
```

Notice, the use of *L* at the end of *-42332200000*. This represents that it's an integral literal of *long* type.

#### f. double

- The double data type is a double-precision 64-bit floating point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0d)
- Example:

```
class DoubleExample {  
    public static void main(String[] args) {  
        double number = -42.3;  
        System.out.println(number);  
    }  
}
```

When you run the program, the output will be:

```
-42.3
```

#### g. float

- The float data type is a single-precision 32-bit floating point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0f)
- Example:

```
class FloatExample {  
    public static void main(String[] args) {  
        float number = -42.3f;  
        System.out.println(number);  
    }  
}
```

When you run the program, the output will be:

`-42.3`

Notice that, we have used `-42.3f` instead of `-42.3` in the above program. It's because `-42.3` is a *double* literal. To tell compiler to treat `-42.3` as *float* rather than double, you need to use *f* or *F*.

#### h. `char`

- It's a 16-bit Unicode character.
- The minimum value of char data type is `'\u0000'` (0). The maximum value of char data type is `'\uffff'`.
- Default value: `'\u0000'`
- Example:

```
class CharExample {  
    public static void main(String[] args) {  
        char letter = '\u0051';  
        System.out.println(letter);  
    }  
}
```

When you run the program, the output will be:

`Q`

You get the output `Q` because the Unicode value of `Q` is `'\u0051'`.



## Assignments and Initializations

As we have seen before, variables are used to represent values that may change in the program. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is given as:

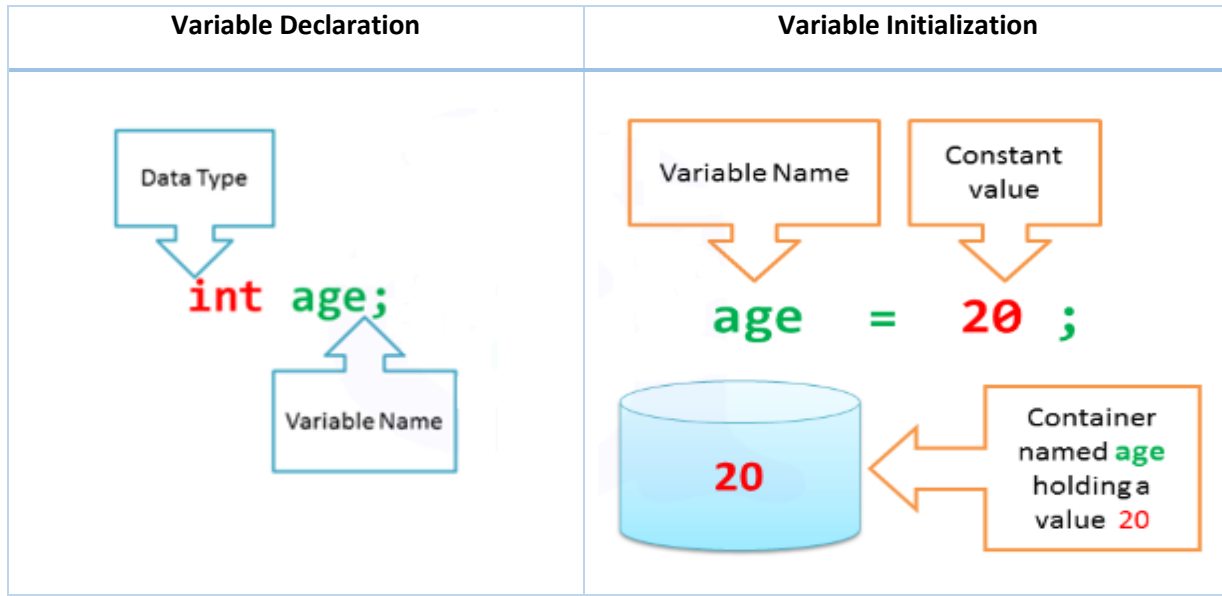
**`type variable_name; // this is a declaration`**

**`variable_name = value; // this is an assignment`**

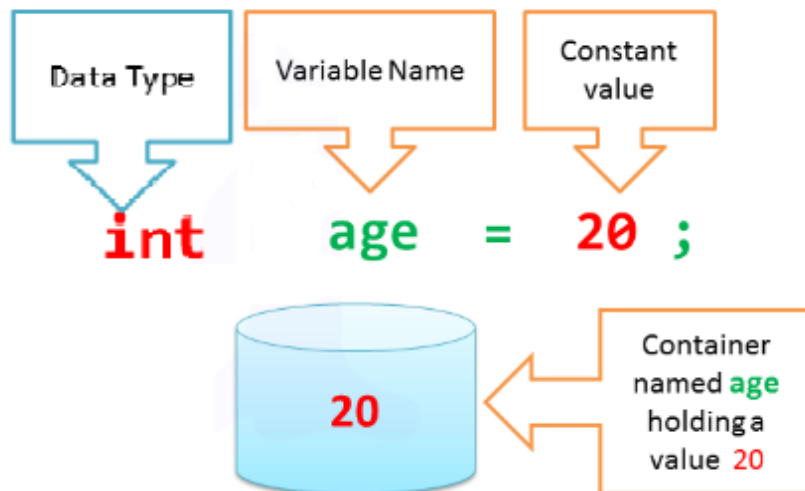
**`type variable_name = value; // this is an initialization`**

Once you declare variable, it is mandatory to explicitly initialize it by means of an assignment statement because uninitialized variables can never be used in any program. To assign a value there are two ways, i.e. static and dynamic.

- In Static assignment, the memory is determined for variables when the program starts.



- In Dynamic assignment, variables can be declared anywhere in the program, this is because when the statement is executed the memory is assigned to them.



## Java Operators

Operators are special symbols (characters) that carry out operations on operands (variables and values). For example, + is an operator that performs addition.

### Assignment Operator

Assignment operators are used in Java to assign values to variables. For example,

```
int age;  
age = 5;
```

The assignment operator assigns the value on its right to the variable on its left. Here, 5 is assigned to the variable age using = operator.

#### Example 1: Assignment Operator

```
class AssignmentOperator {  
    public static void main(String[] args) {  
  
        int number1, number2;  
  
        // Assigning 5 to number1  
        number1 = 5;  
        System.out.println(number1);  
  
        // Assigning value of variable number2 to number1  
        number2 = number1;  
        System.out.println(number2);  
    }  
}
```

When you run the program, the output will be:

```
5  
5
```

### Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Meaning
+	Addition (also used for string concatenation)
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator
%	Remainder Operator

## Example 2: Arithmetic Operator

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

        double number1 = 12.5, number2 = 3.5, result;

        // Using addition operator
        result = number1 + number2;
        System.out.println("number1 + number2 = " + result);

        // Using subtraction operator
        result = number1 - number2;
        System.out.println("number1 - number2 = " + result);

        // Using multiplication operator
        result = number1 * number2;
        System.out.println("number1 * number2 = " + result);

        // Using division operator
        result = number1 / number2;
        System.out.println("number1 / number2 = " + result);

        // Using remainder operator
        result = number1 % number2;
        System.out.println("number1 % number2 = " + result);
    }
}
```

When you run the program, the output will be:

```
number1 + number2 = 16.0
number1 - number2 = 9.0
number1 * number2 = 43.75
number1 / number2 = 3.5714285714285716
number1 % number2 = 2.0
```

In above example, all operands used are variables. However, it's not necessary at all. Operands used in arithmetic operators can be literals as well. For example,

```
result = number1 + 5.2;
result = 2.3 + 4.5;
number2 = number1 -2.9;
```

The + operator can also be used to concatenate two or more strings.

## Unary Operators

Unary operator performs operation on only one operand.

Operator	Meaning
+	Unary plus (not necessary to use since numbers are positive without using it)
-	Unary minus; inverts the sign of an expression
++	Increment operator; increments value by 1
--	decrement operator; decrements value by 1
!	Logical complement operator; inverts the value of a boolean

### Example 3: Unary Operator



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

        double number = 5.2, resultNumber;
        boolean flag = false;

        System.out.println("+number = " + ++number);
        // number is equal to 5.2 here.

        System.out.println("-number = " + --number);
        // number is equal to 5.2 here.

        // ++number is equivalent to number = number + 1
        System.out.println("number = " + ++number);
        // number is equal to 6.2 here.

        // -- number is equivalent to number = number - 1
        System.out.println("number = " + --number);
        // number is equal to 5.2 here.

        System.out.println("!flag = " + !flag);
        // flag is still false.
    }
}
```

When you run the program, the output will be:

```
+number = 5.2
-number = -5.2
number = 6.2
number = 5.2
!flag = true
```

You can also use ++ and -- operator as both prefix and postfix in Java. The ++ operator increases value by 1 and - operator decreases value by 1.

## **Equality and Relational Operators**

The equality and relational operators determine the relationship between two operands. It checks if an operand is greater than, less than, equal to, not equal to and so on. Depending on the relationship, it results to either true or false.

Operator	Description	Example
==	equal to	5 == 3 is evaluated to <code>false</code>
!=	not equal to	5 != 3 is evaluated to <code>true</code>
>	greater than	5 > 3 is evaluated to <code>true</code>
<	less than	5 < 3 is evaluated to <code>false</code>
>=	greater than or equal to	5 >= 5 is evaluated to <code>true</code>
<=	less then or equal to	5 <= 5 is evaluated to <code>true</code>

Equality and relational operators are used in decision making and loops (which will be discussed later). For now, check this simple example.

#### Example 4: Equality and Relational Operators

```
class RelationalOperator {
    public static void main(String[] args) {
        int number1 = 5, number2 = 6;
        if (number1 > number2)
        {
            System.out.println("number1 is greater than number2.");
        }
        else
        {
            System.out.println("number2 is greater than number1.");
        }
    }
}
```

When you run the program, the output will be:

```
number2 is greater than number1.
```

Here, we have used > operator to check if number1 is greater than number2 or not.

Since, number2 is greater than number1, the expression number1 > number2 is evaluated to false.

Hence, the block of code inside else is executed and the block of code inside if is skipped.

For now, just remember that the equality and relational operators compares two operands and is evaluated to either true or false.

In addition to relational operators, there is also a type comparison operator `instance of` which compares an object to a specified type. For example,

### Instance of Operator

Here's an example of instance of operator.

```
class instanceofOperator {  
    public static void main(String[] args) {  
  
        String test = "asdf";  
        boolean result;  
  
        result = test instanceof String;  
        System.out.println(result);  
    }  
}
```

When you run the program, the output will be true. It's because test is the instance of String class.

You will learn more about instance of operator works once you understand Java *Classes and Objects*.

### Logical Operators

The logical operators `||` (conditional-OR) and `&&` (conditional-AND) operates on boolean expressions. Here's how they work.

Operator	Description	Example
<code>  </code>	conditional-OR; true if either of the boolean expression is true	<code>false    true</code> is evaluated to true
<code>&amp;&amp;</code>	conditional-AND; true if all boolean expressions are true	<code>false &amp;&amp; true</code> is evaluated to false

## Example 5: Logical Operators

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

        int number1 = 1, number2 = 2, number3 = 9;
        boolean result;

        // At least one expression needs to be true for result to be true
        result = (number1 > number2) || (number3 > number1);
        // result will be true because (number1 > number2) is true
        System.out.println(result);

        // All expression must be true from result to be true
        result = (number1 > number2) && (number3 > number1);
        // result will be false because (number3 > number1) is false
        System.out.println(result);
    }
}
```

When you run the program, the output will be:

```
true
false
```

Logical operators are used in decision making and looping.

## Ternary Operator

The conditional operator or ternary operator `?:` is shorthand for `if-then-else` statement. The syntax of conditional operator is:

```
variable = Expression ? expression1 : expression2
```

Here's how it works.

- If the Expression is true, expression1 is assigned to variable.
- If the Expression is false, expression2 is assigned to variable.

## Example 6: Ternary Operator

```
class ConditionalOperator {  
    public static void main(String[] args) {  
  
        int februaryDays = 29;  
        String result;  
  
        result = (februaryDays == 28) ? "Not a leap year" : "Leap year";  
        System.out.println(result);  
    }  
}
```

When you run the program, the output will be:

```
Leap year
```

### **Bitwise and Bit Shift Operators**

To perform bitwise and bit shift operators in Java, these operators are used.

Operator	Description
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR
	Bitwise inclusive OR

These operators are not commonly used.

### **More Assignment Operators**

We have only discussed about one assignment operator = in the beginning of the article. Except this operator, there are quite a few assignment operators that helps us to write cleaner code.

Operator	Example	Equivalent to
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
<<=	x <<= 5	x = x << 5
>>=	x >>= 5	x = x >> 5
&=	x &= 5	x = x & 5
^=	x ^= 5	x = x ^ 5
=	x  = 5	x = x   5



## Java Strings

### What is String in Java? String is a data type?

String is a Class in java and defined in java.lang package. It's not a primitive data type like int and long. String class represents character Strings. String is used in almost all the Java applications and there are some interesting facts we should know about String. String is immutable and final in Java and JVM uses String Pool to store all the String objects.

Some other interesting things about String is the way we can instantiate a String object using double quotes and overloading of "+" operator for concatenation.

### What are different ways to create String Object?

We can create String object using new operator like any normal java class or we can use double quotes to create a String object. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.

```
String str = new String("abc");
```

```
String str1 = "abc";
```

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.

When we use new operator, JVM creates the String object but don't store it into the String Pool. We can use intern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.

### **Write a method to check if input String is Palindrome?**

A String is said to be Palindrome if its value is same when reversed. For example, "aba" is a Palindrome String.

String class doesn't provide any method to reverse the String but StringBuffer and StringBuilder class has reverse method that we can use to check if String is palindrome or not.

```
private static boolean isPalindrome(String str) {  
    if (str == null)  
        return false;  
      
    StringBuilder strBuilder = new StringBuilder(str);  
    strBuilder.reverse();  
    return strBuilder.toString().equals(str);  
}
```

Sometimes interviewer asks not to use any other class to check this, in that case we can compare characters in the String from both ends to find out if it's palindrome or not.

```
private static boolean isPalindromeString(String str) {  
    if (str == null)  
        return false;  
      
    int length = str.length();  
    System.out.println(length / 2);  
    for (int i = 0; i < length / 2; i++) {  
      
        if (str.charAt(i) != str.charAt(length - i - 1))  
            return false;  
    }  
}
```

```
}  
  
    return true;  
  
}
```

### **Write a method that will remove given character from the String?**

We can use replaceAll method to replace all the occurrence of a String with another String. The important point to note is that it accepts String as argument, so we will use Character class to create String and use it to replace all the characters with empty String.

```
private static String removeChar(String str, char c) {  
  
    if (str == null)  
  
        return null;  
  
    return str.replaceAll(Character.toString(c), "");  
  
}
```

### **How can we make String upper case or lower case?**

We can use String class toUpperCase and toLowerCase methods to get the String in all upper case or lower case. These methods have a variant that accepts Locale argument and use that locale rules to convert String to upper or lower case.

### **What is String subSequence method?**

Java 1.4 introduced CharSequence interface and String implements this interface, this is the only reason for the implementation of subSequence method in String class. Internally it invokes the String substring method.

### **How to compare two Strings in java program?**

Java String implements Comparable interface and it has two variants of compareTo() methods.

compareTo(String anotherString) method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns positive integer. It returns zero when both the String have same value, in this case equals(String str) method will also return true.

compareToIgnoreCase(String str): This method is similar to the first one, except that it ignores the case. It uses String CASE\_INSENSITIVE\_ORDER Comparator for case insensitive comparison. If the value is zero then equalsIgnoreCase(String str) will also return true.



### **How to convert String to char and vice versa?**

This is a tricky question because String is a sequence of characters, so we can't convert it to a single character. We can use charAt method to get the character at given index or we can use toCharArray() method to convert String to character array.

### **How to convert String to byte array and vice versa?**

We can use String getBytes() method to convert String to byte array and we can use String constructor new String(byte[] arr) to convert byte array to String.

### **Can we use String in switch case?**

This is a tricky question used to check your knowledge of current Java developments. Java 7 extended the capability of switch case to use Strings also, earlier java versions doesn't support this.

If you are implementing conditional flow for Strings, you can use if-else conditions and you can use switch case if you are using Java 7 or higher versions.

### **Write a program to print all permutations of String?**

This is a tricky question and we need to use recursion to find all the permutations of a String, for example "AAB" permutations will be "AAB", "ABA" and "BAA".

We also need to use Set to make sure there are no duplicate values.

### **Write a function to find out longest palindrome in a given string?**

A String can contain palindrome strings in it and to find longest palindrome in given String is a programming question.

### **Difference between String, StringBuffer and StringBuilder?**

String is immutable and final in java, so whenever we do String manipulation, it creates a new String. String manipulations are resource consuming, so java provides two utility classes for String manipulations – StringBuffer and StringBuilder.

StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So, when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.

StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

Check this post for extensive details about String vs StringBuffer vs StringBuilder.

### **Why String is immutable or final in Java**

There are several benefits of String because it's immutable and final.

String Pool is possible because String is immutable in java.

It increases security because any hacker can't change its value and it's used for storing sensitive information such as database username, password etc.

Since String is immutable, it's safe to use in multi-threading and we don't need any synchronization.

Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.

### **How to Split String in java?**

We can use `split(String regex)` to split the String into String array based on the provided regular expression.

### **Why Char array is preferred over String for storing password?**

String is immutable in java and stored in String pool. Once it's created it stays in the pool until unless garbage collected, so even though we are done with password it's available in memory for longer duration and there is no way to avoid it. It's a security risk because anyone having access to memory dump can find the password as clear text.

If we use char array to store password, we can set it to blank once we are done with it. So we can control for how long it's available in memory that avoids the security threat with String.

### **How do you check if two Strings are equal in Java?**

There are two ways to check if two Strings are equal or not – using “==” operator or using equals method. When we use “==” operator, it checks for value of String as well as reference but in our programming, most of the time we are checking equality of String for value only. So we should use equals method to check if two Strings are equal or not.

There is another function `equalsIgnoreCase` that we can use to ignore case.

```
String s1 = "abc";  
String s2 = "abc";  
String s3= new String("abc");  
System.out.println("s1 == s2 ? "+(s1==s2)); //true  
System.out.println("s1 == s3 ? "+(s1==s3)); //false  
System.out.println("s1 equals s3 ? "+(s1.equals(s3))); //true
```

### **What is String Pool?**

As the name suggests, String Pool is a pool of Strings stored in Java heap memory. We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

### **What does String intern() method do?**

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

This method always returns a String that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

### **Does String is thread-safe in Java?**

Strings are immutable, so we can't change its value in program. Hence, it's thread-safe and can be safely used in multi-threaded environment.

### **Why String is popular HashMap key in Java?**

Since String is immutable, its hashCode is cached at the time of creation and it doesn't need to be calculated again. This makes it a great candidate for key in a Map and its processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.



## **Control Flow**

### **Java if, if...else Statement**

In programming, it's often desirable to execute a certain section of code based upon whether the specified condition is true or false (which is known only during the run time). For such cases, control flow statements are used.

#### **Java if (if-then) Statement**

The syntax of if-then statement in Java is:

```
if (expression) {  
    // statements  
}
```

### Example: Java if Statement

```
class IfStatement {  
    public static void main(String[] args) {  
        int number = 10;  
        if (number > 0) {  
            System.out.println("Number is positive.");  
        }  
        System.out.println("This statement is always executed.");  
    }  
}
```

When you run the program, the output will be:

```
Number is positive.  
This statement is always executed.
```

When number is 10, the test expression `number > 0` is evaluated to true. Hence, codes inside the body of if statements are executed.

Now, change the value of number to a negative integer. Let's say -5. The output in this case will be:

```
This statement is always executed.
```

When number is -5, the test expression `number > 0` is evaluated to false. Hence, Java compiler skips the execution of body of if statement.

### Java if...else (if-then-else) Statement

The if statement executes a certain section of code if the test expression is evaluated to true. The if statement can have optional else statement. Codes inside the body of else statement are executed if the test expression is false.

The syntax of if-then-else statement is:

```
if (expression) {  
    // codes  
}  
else {  
    // some other code  
}
```

### Java if..else..if Statement

In Java, it's possible to execute one block of code among many. For that, you can use if..else...if ladder.

```
if (expression1)  
{  
    // codes  
}  
else if(expression2)  
{  
    // codes  
}  
else if (expression3)  
{  
    // codes  
}  
.  
.  
else  
{  
    // codes  
}
```

The if statements are executed from the top towards the bottom. As soon as the test expression is true, codes inside the body of that if statement is executed. Then, the control of program jumps outside if-else-if ladder.

If all test expressions are false, codes inside the body of else is executed.

### Java Nested if..else Statement

It's possible to have if..else statements inside a if..else statement in Java. It's called nested if...else statement.

Here's a program to find largest of 3 numbers:

### **Example: Nested if...else Statement**

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

        Double n1 = -1.0, n2 = 4.5, n3 = -5.3, largestNumber;

        if (n1 >= n2) {
            if (n1 >= n3) {
                largestNumber = n1;
            } else {
                largestNumber = n3;
            }
        } else {
            if (n2 >= n3) {
                largestNumber = n2;
            } else {
                largestNumber = n3;
            }
        }

        System.out.println("Largest number is " + largestNumber);
    }
}
```

When you run the program, the output will be:

```
Largest number is 4.5
```

**Note:** In above programs, we have assigned value of variables ourselves to make this easier. However, in real world applications, these values may come from user input data, log files, form submission etc.

You should also check [ternary operator in Java](#), which is kind of shorthand notation of if...else statement.

### **Java switch Statement**

In Java, the [if..else..if ladder](#) executes a block of code among many blocks. The switch statement can a substitute for long if..else..if ladders which generally makes your code more readable.

The syntax of switch statement is:

```
switch (variable/expression) {  
  case value1:  
    // statements  
    break;  
  case value2:  
    // statements  
    break;  
    .. .. .  
    .. .. .  
  default:  
    // statements  
}
```

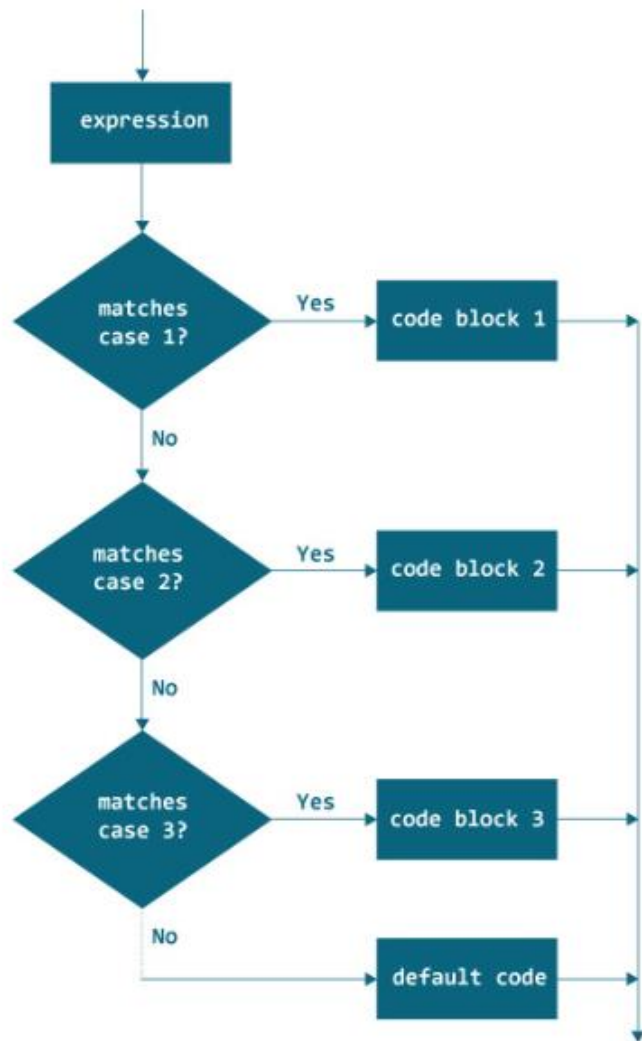
The switch statement evaluates its expression (mostly variable) and compares with values (can be expression) of each case label.

The switch statement executes all statements of the matching case label.

Suppose, the variable/expression is equal to value2. In this case, all statements of that matching case are executed.

Notice, the use of break statement. This statement terminates the execution of switch statement. The break statements are important because if they are not used, all statements after the matching case label are executed in sequence until the end of switch statement.

### **Flowchart of switch Statement**



It's also important to note that switch statement in Java only works with:

- Primitive data types: byte, short, char and int
- *Enumerated types (Java enums)*
- String class
- a few classes that wrap primitive types: Character, Byte, Short, and Integer.

Example : Java switch statement



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

        int week = 4;
        String day;

        switch (week) {
            case 1:
                day = "Sunday";
                break;
            case 2:
                day = "Monday";
                break;
            case 3:
                day = "Tuesday";
                break;
            case 4:
                day = "Wednesday";
                break;
            case 5:
                day = "Thursday";
                break;
            case 6:
                day = "Friday";
                break;
            case 7:
                day = "Saturday";
                break;
            default:
                day = "Invalid day";
                break;
        }
        System.out.println(day);
    }
}
```

When you run the program, the output will be:

```
Wednesday
```

### **Java for Loop**

Loop is used in programming to repeat a specific block of code. In this article, you will learn to create a for loop in Java programming.

Loop is used in programming to repeat a specific block of code until certain condition is met (test expression is false).

Loops are what makes computers interesting machines. Imagine you need to print a sentence 50 times on your screen. Well, you can do it by using print statement 50 times (without using loops). How about you need to print a sentence one million times? You need to use loops.

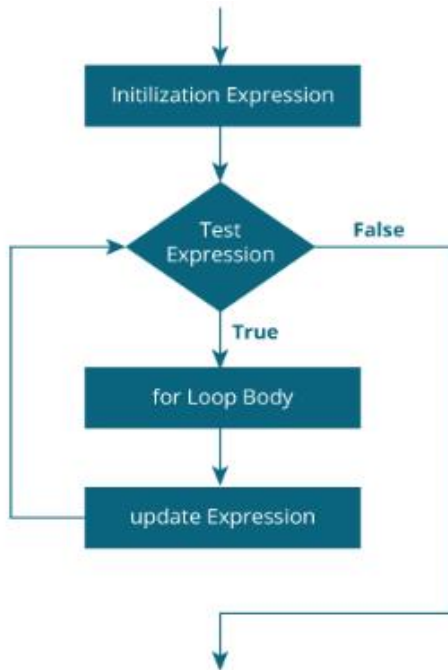
The syntax of for Loop in Java is:

```
for (initialization; testExpression; update)
{
    // codes inside for loop's body
}
```

#### How for loop works?

1. The initialization expression is executed only once.
2. Then, the test expression is evaluated. Here, test expression is a boolean expression.
3. If the test expression is evaluated to true,
  - Codes inside the body of for loop is executed.
  - Then the update expression is executed.
  - Again, the test expression is evaluated.
  - If the test expression is true, codes inside the body of for loop is executed and update expression is executed.
  - This process goes on until the test expression is evaluated to false.
4. If the test expression is evaluated to false, for loop terminates.

#### for Loop Flowchart



### Example: for Loop

```

// Program to find the sum of natural numbers from 1 to 1000.
class Number {
    public static void main(String[] args) {
        int sum = 0;

        for (int i = 1; i <= 1000; ++i) {
            sum += i;      // sum = sum + i
        }

        System.out.println("Sum = " + sum);
    }
}
  
```

When you run the program, the output will be:

```
Sum = 500500
```

Here, the variable sum is initialized to 0. Then, in each iteration of for loop, variable sum is assigned sum + i and the value of i is increased until i is greater than 1000. For better visualization,

```
1st iteration: sum = 0+1 = 1
2nd iteration: sum = 1+2 = 3
3rd iteration: sum = 3+3 = 6
4th iteration: sum = 6+4 = 10
... ..

999th iteration: sum = 498501 + 999 = 499500
1000th iteration: sum = 499500 + 1000 = 500500
```

## Java for-each Loop

In Java, there is an alternative syntax of for loop to work with *arrays* and *collections* (known as for-each loop).

### How for-each loop works?

Here's how the enhanced for loop works. For each iteration, for-each loop

- **iterates** through each item in the given collection or array (collection),
- **stores** each item in a variable (item)
- and **executes** the body of the loop.

Let's make it clear through an example.

### **Example: for-each loop**

The program below calculates the sum of all elements of an integer array.

```
class EnhancedForLoop {
    public static void main(String[] args) {
        int[] numbers = {3, 4, 5, -5, 0, 12};
        int sum = 0;

        for (int number: numbers) {
            sum += number;
        }

        System.out.println("Sum = " + sum);
    }
}
```

When you run the program, the output will be:

```
Sum = 19
```

In the above program, the execution of foreach loop looks as:

Iteration	Value of number	Value of sum
1	3	3
2	4	7
3	5	12
4	-5	7
5	0	7
6	12	19

You can see during each iteration, the for-each loop

- iterates through each element in the number's variable
- stores it in the number variable
- and executes the body, i.e. adds number to sum

## **Java while Loop**

The syntax of while loop is:

```
while (testExpression) {  
    // codes inside body of while loop  
}
```

## **How while loop works?**

The test expression inside parenthesis is a boolean expression.

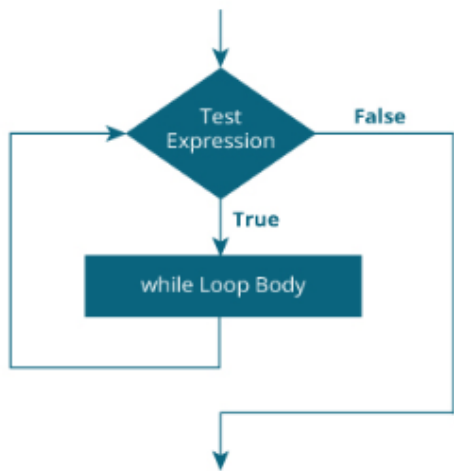
If the test expression is evaluated to true, statements inside the while loop are executed.

then, the test expression is evaluated again.

This process goes on until the test expression is evaluated to false.

If the test expression is evaluated to false, while loop is terminated.

#### Flowchart of while Loop



#### Example: Java while Loop

```
// Program to find the sum of natural numbers from 1 to 100.
class AssignmentOperator {
    public static void main(String[] args) {
        int sum = 0, i = 100;
        while (i != 0) {
            sum += i;    // sum = sum + i;
            --i;
        }
        System.out.println("Sum = " + sum);
    }
}
```

When you run the program, the output will be:

```
Sum = 5050
```

Here, the variable sum is initialized to 0 and i is initialized to 100. In each iteration of while loop, variable sum is assigned sum + i, and the value of i is decreased by 1 until i is equal to 0. For better visualization,

```
1st iteration: sum = 0+100 = 100, i = 99
2nd iteration: sum = 100+99 = 199, i = 98
3rd iteration: sum = 199+98 = 297, i = 97
... ..
... ..
99th iteration: sum = 5047+2 = 5049, i = 1
100th iteration: sum = 5049+1 = 5050, i = 0
```

### **Java do...while Loop**

The do...while loop is similar to while loop with one key difference. The body of do...while loop is executed for once before the test expression is checked.

The syntax of do..while loop is:

```
do {
    // codes inside body of do while loop
} while (testExpression);
```

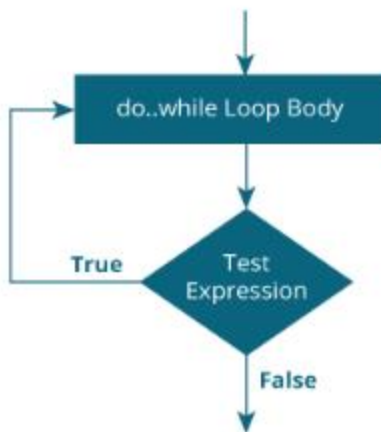
### **How do...while loop works?**

The body of do...while loop is executed once (before checking the test expression). Only then, the test expression is checked.

If the test expression is evaluated to true, codes inside the body of the loop are executed, and the test expression is evaluated again. This process goes on until the test expression is evaluated to false.

When the test expression is false, the do. While loop terminates.

### **Flowchart of do...while Loop**



### Example: do...while Loop

The program below calculates the sum of numbers entered by the user until user enters 0.

To take input from the user, Scanner object is used. Visit [Java Basic Input](#) to learn more on how to take input from the user.

```

import java.util.Scanner;

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

        Double number, sum = 0.0;
        Scanner input = new Scanner(System.in);

        do {
            System.out.print("Enter a number: ");
            number = input.nextDouble();
            sum += number;
        } while (number != 0.0);

        System.out.println("Sum = " + sum);

    }
}
  
```

When you run the program, the output will be:



```
Enter a number: 2.5
Enter a number: 23.3
Enter a number: -4.2
Enter a number: 3.4
Enter a number: 0
Sum = 25.0
```

### **Java Break Statement**

Suppose you are working with loops. It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

In such cases, break and continue statements are used.

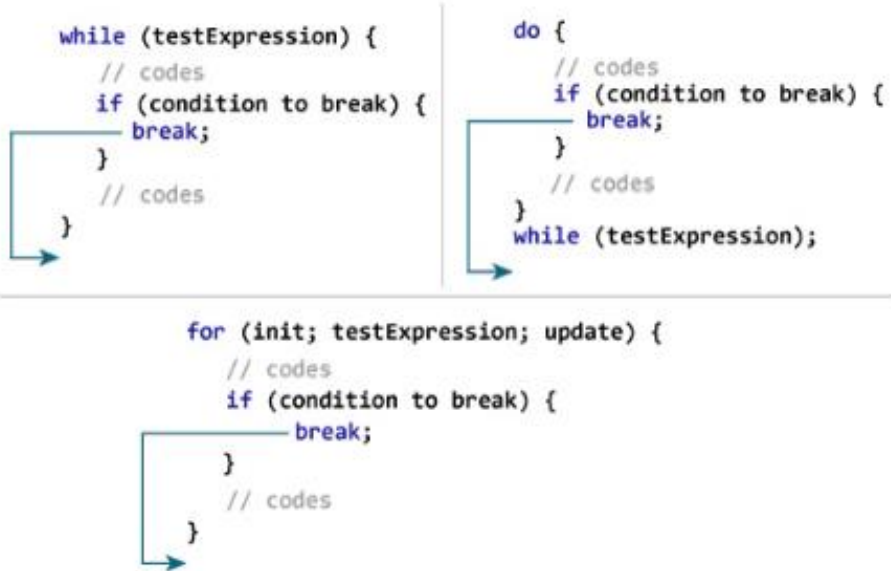
The break statement terminates the loop immediately, and the control of the program moves to the next statement following the loop.

It is almost always used with decision making statements (if...else Statement).

The syntax of a break statement is:

```
break;
```

### **How break statement works?**



### Example: Java break statement

The program below calculates the sum of numbers entered by the user until user enters a negative number. To take input from the user, Scanner object is used. Visit [Java Basic Input](#) to learn more on how to take input from the user.

```
import java.util.Scanner;

class UserInputSum {
    public static void main(String[] args) {
        Double number, sum = 0.0;
        Scanner input = new Scanner(System.in);

        while (true) {
            System.out.print("Enter a number: ");
            number = input.nextDouble();

            if (number < 0.0) {
                break;
            }

            sum += number;
        }
        System.out.println("Sum = " + sum);
    }
}
```

When you run the program, you will get similar output like this:

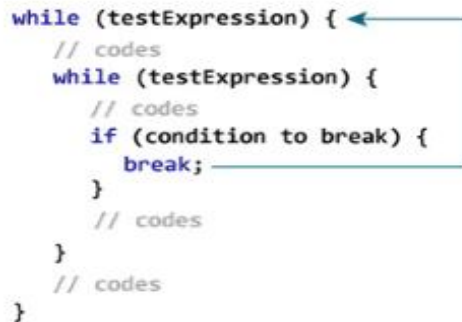
```
Enter a number: 3.2
Enter a number: 5
Enter a number: 2.3
Enter a number: 0
Enter a number: -4.5
Sum = 10.5
```

In the above program, the test expression of the while loop is always true.

Here, the while loop runs until user enters a negative number. If user inputs negative number, break statement inside the body of if statement is executed which terminates the while loop.

In case of nested loops, break terminates the innermost loop.

```
while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (condition to break) {
            break;
        }
        // codes
    }
    // codes
}
```



Here, the break statement terminates the innermost while loop, and control jumps to the outer loop.

### **Labeled break Statement**


The break statement we have discussed till now is unlabeled form of break statement, which terminates the innermost for, while, do..while and switch statement. There is another form of break statement, labeled break, that can be used to terminate the outer loop.

#### **How labeled break statement works?**

```

label:
for (int; testExpresison, update) {
    // codes
    for (int; testExpression; update) {
        // codes
        if (condition to break) {
            break label;
        }
        // codes
    }
    // codes
}

```



Here, label is an identifier. When break statement executes, it terminates the labeled statement, and control of the program jumps to the statement immediately following the labeled statement.

Here's another example:

```

while (testExpression) {
    // codes
    second:
    while (testExpression) {
        // codes
        while(testExpression) {
            // codes
            break second;
        }
    }
    // control jumps here
}

```

When break second; executes, control jumps to the statement following the labeled statement second.

### **Java continue Statement**

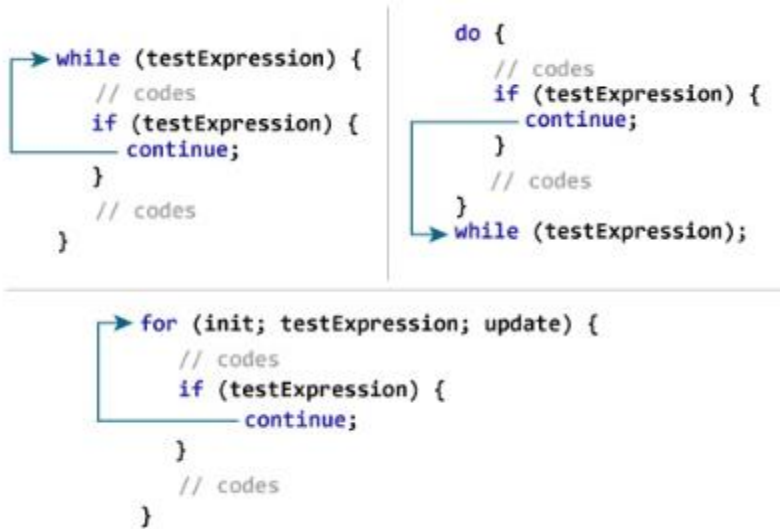
The continue statement skips the current iteration of a loop (for, while, and do...while loop). When continue statement is executed, control of the program jumps to the end of the loop. Then, the test expression that controls the loop is evaluated. In case of for loop, the update statement is executed before the test expression is evaluated.

It is almost always used with decision making statements (if...else Statement).

It's syntax is:

```
continue;
```

### How continue statement works?



### Example: Java continue statement

The program below calculates the sum of maximum of 5 positive numbers entered by the user. If the user enters negative number or zero, it is skipped from calculation.

To take input from the user, Scanner object is used. Visit [Java Basic Input](#) to learn more on how to take input from the user.

```
import java.util.Scanner;

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

        Double number, sum = 0.0;
        Scanner input = new Scanner(System.in);

        for (int i = 1; i < 6; ++i) {
            System.out.print("Enter a number: ");
            number = input.nextDouble();

            if (number <= 0.0) {
                continue;
            }

            sum += number;
        }
        System.out.println("Sum = " + sum);
    }
}
```

When you run the program, you will get similar output like this:

```
Enter a number: 2.2
Enter a number: 5.6
Enter a number: 0
Enter a number: -2.4
Enter a number: -3
Sum = 7.8
```

In case of nested loops, `continue` skips the current iteration of innermost loop.

### **Labelled continue Statement**

The continue statement we have discussed till now is unlabeled form of continue, which skips the execution of remaining statement(s) of innermost for, while and do..while loop.

There is another form of continue statement, labeled form, that can be used to skip the execution of statement(s) that lies inside the outer loop.

### **How labeled continue statement works?**

```

label:
while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (condition for continue) {
            continue label;
        }
        // codes
    }
    // codes
}

```

Here, label is an identifier.

The use of labeled `continue` is often discouraged as it makes your code hard to understand. If you are in a situation where you have to use labeled `continue`, refactor your code and try to solve it in a different way to make it more readable.



## Java Arrays

An array is a container that holds data (values) of one single type. For example, you can create an array that can hold 100 values of int type.

Array is a fundamental construct in Java that allows you to store and access large number of values conveniently.

### How to declare an array?

Here's how you can declare an array in Java:

```
dataType[] arrayName;
```

- dataType can be a primitive data type like: int, char, Double, byte etc. or an object.
- arrayName is an identifier.

Let's take the above example again.

```
data = new Double[10];
```

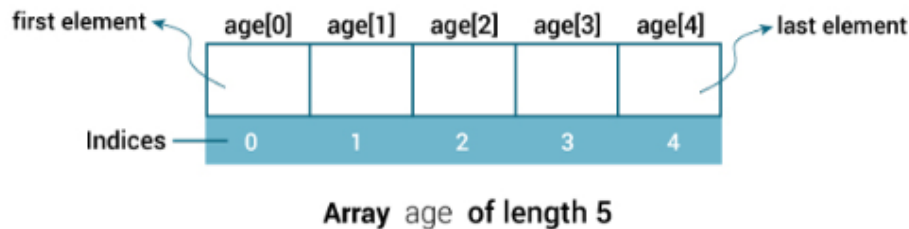
The length of data array is 10. Meaning, it can hold 10 elements (10 Double values in this case).

Note, once the length of the array is defined, it cannot be changed in the program.

### Java Array Index

You can access elements of an array by using indices.

```
int[] age = new int[5];
```



The first element of array is `age[0]`, second is `age[1]` and so on.

If the length of an array is `n`, the last element will be `arrayName[n-1]`. Since the length of `age` array is 5, the last element of the array is `age[4]` in the above example.

The default initial value of elements of an array is 0 for numeric types and false for boolean. We can demonstrate this:

```
class ArrayExample {  
    public static void main(String[] args) {  
        int[] age = new int[5];  
  
        System.out.println(age[0]);  
        System.out.println(age[1]);  
        System.out.println(age[2]);  
        System.out.println(age[3]);  
        System.out.println(age[4]);  
    }  
}
```



When you run the program, the output will be:

```
0
0
0
0
0
```

There is a better way to access elements of an array by using looping construct (generally for loop is used).

```
class ArrayExample {
    public static void main(String[] args) {
        int[] age = new int[5];
        for (int i = 0; i < 5; ++i) {
            System.out.println(age[i]);
        }
    }
}
```

### How to initialize arrays in Java?

In Java, you can initialize arrays during declaration or you can initialize (or change values) later in the program as per your requirement.

### **Initialize an Array during Declaration**

Here's how you can initialize an array during declaration.

```
int[] age = {12, 4, 5, 2, 5};
```

This statement creates an array and initializes it during declaration.

The length of the array is determined by the number of values provided which is separated by commas. In our example, the length of age array is 5.

### How to access array elements?

You can easily access and alter array elements by using its numeric index. Let's take an example.

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

        int[] age = new int[5];

        // insert 14 to third element
        age[2] = 14;

        // insert 34 to first element
        age[0] = 34;

        for (int i = 0; i < 5; ++i) {
            System.out.println("Element at index " + i + ": " + age[i]);
        }
    }
}
```

When you run the program, the output will be:

```
Element at index 0: 34
Element at index 1: 0
Element at index 2: 14
Element at index 3: 0
Element at index 4: 0
```

Couple of things here.

- The `for..each` loop is used to access each elements of the array. Learn more on [how for...each loop works in Java](#).
- To calculate the average, `int` values `sum` and `arrayLength` are converted into `double` since `average` is `double`. This is type casting.
- To get length of an array, `length` attribute is used. Here, `numbers.length` returns the length of `numbers` array.

## Multidimensional Arrays

Arrays we have mentioned till now are called one-dimensional arrays. In Java, you can declare an array of arrays known as multidimensional array. Here's an example to declare and initialize multidimensional array.

```
Double[][] matrix = {{1.2, 4.3, 4.0},
                     {4.1, -1.1}
};
```

Here, `matrix` is a 2-dimensional array.



## Maven Fundamentals

Maven is a tool for project management and build automation. Maven is not Ant++. Maven serves a similar purpose to the Apache Ant tool, but it is based on different concepts and works in a profoundly different manner. Maven is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project.

It is pronounced as “May-ven” and Maven is a Yiddish (Jewish Lang) word which means “accumulator of knowledge”. Actually, there is an open source project named Jakarta Turbine which was facing complexity issues with module build processes and atleast the Apache team came up with a solution to simplify this build process which was then termed as Apache Maven.

Maven provides features such as:

- Build tool Capabilities
- Run Reports
- Generate a website
- Dependency Management
- Repositories ( Reusable Plug-ins )
- Continuous Integration build systems
- Portable
- Building configuration using maven are portable to another machine without any effort

Maven allows to comprehend the complete state of a development effort in the shortest period of time. To attain this goal, Maven deals with:

### Making the build process easy:

While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does provide a lot of shielding from the details.

### Providing a uniform build system:

Maven allows a project to build using its project object model (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform build system. Once you familiarize yourself with how one Maven project builds you automatically know how all Maven projects build saving you immense amounts of time when trying to navigate many projects.

### Providing quality project information:

Maven provides plenty of useful project information that is in part taken from your POM and in part generated from your project's sources.

For example,

Maven can provide:

- Change log document created directly from source control
- Cross referenced sources
- Mailing lists
- Dependency list
- Unit test reports including coverage

### **Maven Principles:**

*Conventions over configuration* used to facilitate a uniform build system that speeds up the development cycle. There are three primary conventions that Maven employs to promote a standardized development environment:

#### Standard Project Layout:

Having a common directory layout would allow for users familiar with one Maven project to immediately feel at home in another Maven project. The advantages are analogous to adopting a site-wide look-and-feel.

#### Standard Naming conventions:

If multiple projects are involved, standard naming convention for directories provide clarity and immediate comprehension.

#### One Primary Output per Project:

Instead of producing a single JAR file for entire project, Maven would encourage you to have three, separate projects: a project for the client portion of the application, a project for the server portion of the application, and a project for the shared utility code portion.

This separation of concerns (SoC) principle used to achieve the required engineering quality factors such as adaptability, maintainability, extendibility and reusability.

If you follow the convention, Maven will require almost zero effort - just put your source in the correct directory and Maven will take care of the rest.

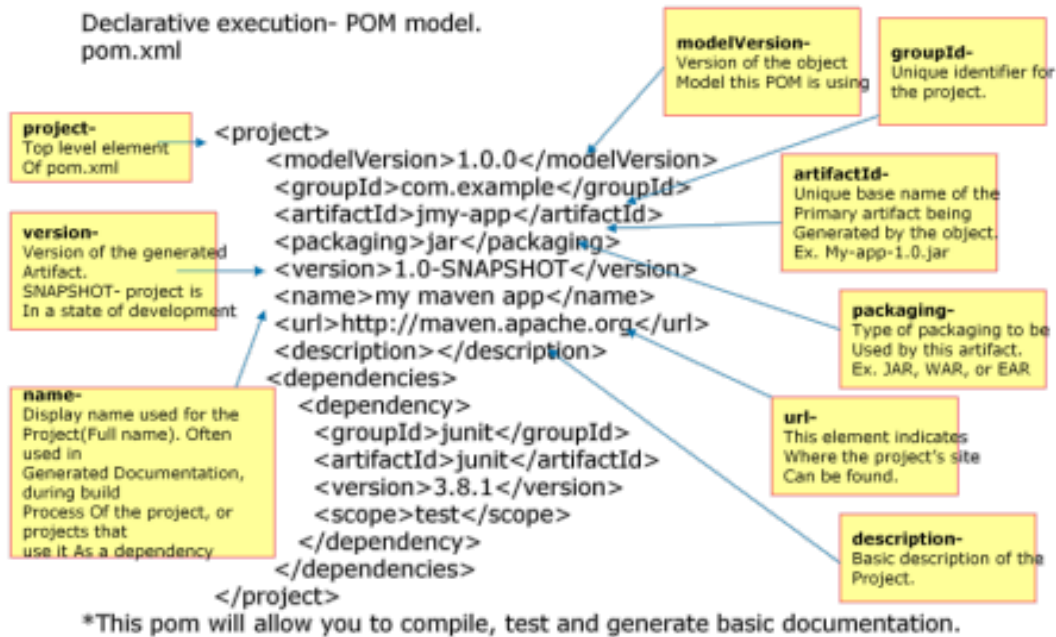
Maven is driven in a declarative fashion using Maven's Project Object Model (POM) and specifically the plug-in configurations contained in the POM.

Maven uses SoC principle to promote reuse of build logic by encapsulating build logic into coherent modules called plugins. In Maven, there is a plug-in for compiling source code, a plug-in for running tests, a plug-in for creating JARs, a plug-in for creating Javadocs, and many other functions. Plugins are the key building blocks for everything in Maven.

When a dependency is declared within the context of your project, Maven tries to satisfy that dependency by looking in repositories. Maven uses a local repository to resolve its dependencies. If not found one or more remote repositories are consulted to find a dependency. If found, the dependency is downloaded to the local repository and used from the local repository.

Maven can provide benefits for your build process by employing standard conventions and practices to accelerate your development cycle while at the same time helping you achieve a higher rate of success.

## Project Object Model (POM)



The `<project>` element is the root of the project descriptor.

`<modelVersion>` : Declares to which version of project descriptor this POM conforms.

`<groupId>` : A universally unique identifier for a project. It is normal to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. org.apache.maven).

`<artifactId>` : The identifier for this artifact that is unique within the group given by the group ID. An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include:

JARs, source and binary distributions, and WARs.

`<packaging>` : The type of artifact this project produces, for example jar war ear pom. Plugins can create their own packaging, and therefore their own packaging types, so this list does not contain all possible types.

Default value is: jar.

`<Name>` : The full name of the project.

`<url>` : The URL to the project's homepage.

`<Version>` : The current version of the artifact produced by this project.

**<description>** : A detailed description of the project, used by Maven whenever it needs to describe the project, such as on the web site.

**<dependencies>**: This element describes all of the dependencies associated with a project. These dependencies are used to construct a class path for your project during the build process. They are automatically downloaded from the repositories defined in this project.

### Advantages:

- A developer familiar with Maven will quickly get familiar with a new project.
- No time wasted on re-inventing directory structures and conventions.

### Standard Directory Layout:

The src directory has a number of subdirectories, each of which has a clearly defined purpose. Listing out few subdirectories in src directory:

src/main/java - Contains the deliverable Java source code for the project.

src/main/resources - Contains the deliverable resources for the project, such as property files.

src/test/java - Contains the testing classes (JUnit or TestNG test cases, for example) for the project.

src/test/resources - Contains resources necessary for testing.

src/site - Contains files used to generate the Maven project website.



## TDD with Junit 5

### What is JUnit?

JUnit is a unit testing framework for Java programming language. JUnit has been important in the development of test-driven development and is one of a family of unit testing frameworks collectively known as *xUnit*, that originated with JUnit.

It is an open-source testing framework for java programmers. The java programmer can create test cases and test his/her own code.

It is one of the unit testing framework. Current version is JUnit 4.

To perform unit testing, we need to create test cases. The unit test case is a code which ensures that the program logic works as expected.

The *org.junit* package contains many interfaces and classes for JUnit testing such as Assert, Test, Before, After etc.

### Features of JUnit:

- JUnit is an open source framework, which is used for writing and running tests.
- Provides annotations to identify test methods.
- Provides assertions for testing expected results.
- Provides test runners for running tests.
- JUnit tests allow you to write codes faster, which increases quality.
- JUnit is elegantly simple. It is less complex and takes less time.
- JUnit tests can be run automatically, and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

### What is Unit Test Case?

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected. To achieve the desired results quickly, a test framework is required. JUnit is a perfect unit test framework for Java programming language.

A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

There must be at least two-unit test cases for each requirement – one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.



### Types of Testing

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. Unit testing comes into picture at the developers' level; it is the testing of single entity (class or method). Unit testing plays a critical role in helping a software company deliver quality products to its customers.

Unit testing can be done in two ways – Manual Testing and Automated Testing.

Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.
<b>Time-consuming and tedious</b> – Since test cases are executed by human resources, it is very slow and tedious.	<b>Fast</b> – Automation runs test cases significantly faster than human resources.
<b>Huge investment in human resources</b> – As test cases need to be executed manually, more testers are required in manual testing.	<b>Less investment in human resources</b> – Test cases are executed using automation tools, so less number of testers are required in automation testing.
<b>Less reliable</b> – Manual testing is less reliable, as it has to account for human errors.	<b>More reliable</b> – Automation tests are precise and reliable.
<b>Non-programmable</b> – No programming can be done to write sophisticated tests to fetch hidden information.	<b>Programmable</b> – Testers can program sophisticated tests to bring out hidden information.

## JUnit: Environment Setup

JUnit is a framework for Java, so very first requirement is to install JDK in the system and then set up its environment. After this follow the below steps to install and set up JUnit in the system.

### Step 1: Download JUnit Archive

Download the latest version of JUnit jar file from <http://www.junit.org>. At the time of writing this tutorial, we have downloaded Junit-4.12.jar and copied it into C:\>JUnit folder.

OS	Archive name
Windows	junit4.12.jar



Linux	junit4.12.jar
Mac	junit4.12.jar

## Step 2: Set JUnit Environment

Set the `JUNIT_HOME` environment variable to point to the base directory location where JUNIT jar is stored on your machine. Let's assuming we've stored **junit4.12.jar** in the **JUNIT** folder.

Sr. No.	OS & Description
1	<b>Windows</b> Set the environment variable JUNIT_HOME to C:\JUNIT
2	<b>Linux</b> export JUNIT_HOME = /usr/local/JUNIT
3	<b>Mac</b> export JUNIT_HOME = /Library/JUNIT

## Step 3: Set CLASSPATH Variable

Set the **CLASSPATH** environment variable to point to the JUNIT jar location.

Sr.No	OS & Description
1	<b>Windows</b> Set the environment variable CLASSPATH to %CLASSPATH%;%JUNIT_HOME%\junit4.12.jar;;
2	<b>Linux</b> export CLASSPATH = \$CLASSPATH:\$JUNIT_HOME/junit4.12.jar:.

3	<b>Mac</b>  export CLASSPATH = \$CLASSPATH:\$JUNIT_HOME/junit4.12.jar:.
---	-------------------------------------------------------------------------------

#### Step 4: Test JUnit Setup

Create a java class file name TestJUnit in **C:\>JUNIT\_WORKSPACE**

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJUnit {
    @Test
    public void testAdd() {
        String str = "JUnit is working fine";
        assertEquals("JUnit is working fine",str);
    }
}
```

Create a java class file name *TestRunner* in **C:\>JUNIT\_WORKSPACE** to execute test case(s).

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJUnit.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

### Step 5: Verify the Result

Compile the classes using **javac** compiler as follows –

```
C:\JUNIT_WORKSPACE>javac TestJunit.java TestRunner.java
```

Now run the Test Runner to see the result as follows –

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output:

```
true
```

### Annotations used in Junit

Annotation is a special form of **syntactic meta-data** that can be added to Java source code for better code readability and structure. Variables, parameters, packages, methods and classes can be annotated.

Some of the JUnit annotations which can be useful are –

- Before
- After
- BeforeClass
- AfterClass
- Ignore
- RunWith
- Test

Annotations were introduced in Junit4, which makes Java code more readable and simpler. This is the big difference between Junit3 and Junit4 that Junit4 is annotation based.

With the knowledge of annotations in Junit4, one can easily learn and implement a JUnit test.

Below is the list of important and frequently used annotations:

S.No.	Annotations	Description
1.	@Test	This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.

2.	@Before	This annotation is used if you want to execute some statement such as preconditions before each test case.
3.	@BeforeClass	This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
4.	@After	This annotation can be used if you want to execute some statements after eachTest Case for e.g. resetting variables, deleting temporary files ,variables, etc.
5.	@AfterClass	This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
6.	@Ignores	This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
7.	@Test(timeout=500)	This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.
8.	@Test(expected=IllegalArgumentException.class)	This annotation can be used if you want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not.

Let's create a class covering important JUnit annotations with simple print statements and execute it with a test runner class:

**Step 1:** Consider below java class having various methods which are attached to above-listed annotations –

[JUnitAnnotationsExample.java](#)

```
package guru99.junit;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

import java.util.ArrayList;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class JunitAnnotationsExample {

    private ArrayList<String> list;

    @BeforeClass
    private ArrayList<String> list;
    @BeforeClass
    public static void m1() {
        System.out.println("Using @BeforeClass , executed before all test cases ");
    }

    @Before
    public void m2() {
        list = new ArrayList<String>();
        System.out.println("Using @Before annotations ,executed before each test cases ");
    }

    @AfterClass
    public static void m3() {
        System.out.println("Using @AfterClass ,executed after all test cases");
    }

}
```

```
@After
public void m4() {
    list.clear();
    System.out.println("Using @After ,executed after each test cases");
}

@Test
public void m5() {
    list.add("test");
    assertFalse(list.isEmpty());
    assertEquals(1, list.size());
}

@Ignore
public void m6() {
    System.out.println("Using @Ignore , this execution is ignored");
}

@Test(timeout = 10)
public void m7() {
    System.out.println("Using @Test(timeout),it can be used to enforce timeout in
JUnit4 test case");
}

@Test(expected = NoSuchMethodException.class)

public void m8() {
    System.out.println("Using @Test(expected) ,it will check for specified exception
during its execution");
}

}

list.add("test");
assertFalse(list.isEmpty());
assertEquals(1, list.size());
}

@Ignore
public void m6() {
    System.out.println("Using @Ignore , this execution is ignored");
}
```

```
}
@Test(timeout = 10)
public void m7() {
    System.out.println("Using @Test(timeout),it can be used to enforce timeout in
JUnit4 test case");
}
@Test(expected = NoSuchMethodException.class)

public void m8() {
    System.out.println("Using @Test(expected) ,it will check for specified exception
during its execution");
}
}
```

**Step 2:** let's create a test runner class to execute above test:

TestRunner.java

```
package guru99.junit;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

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

        Result result = JUnitCore.runClasses(JunitAnnotationsExample.class);

        for (Failure failure : result.getFailures()) {

            System.out.println(failure.toString());
        }
        System.out.println("Result==" + result.wasSuccessful());

    }
}
```

### Expected Result:

- All the test cases will be executed one by one, and all print statement can be seen on a console.
- As discussed in above table @Before, @BeforeClass [ method m1() and m2() ] will be executed before each and before all test cases respectively.
- In the same way @after,@afterClass (method m3() and m4()) will be executed after each and after all test cases respectively. @ignore (method m6())will be treated as ignoring the test.

Let's analyse test cases used in above java class in detail:

#### I. Consider method m5() as given below :

```
@Test
public void m5() {
    list.add("test");
    assertFalse(list.isEmpty());
    assertEquals(1, list.size());
}
```

In above method as you are adding a string in the variable "list" so

- *list.isEmpty()* will return false.
- *assertFalse(list.isEmpty())* must return true.
- As a result, the test case will pass.

As you have added only one string in the list, so the size is one.

- *list.size()* must return int value as "1" .
- So *assertEquals(1, list.size())* must return true.
- As a result, the test case will **pass**.

#### II. Consider method m7() as given below :

```
@Test(timeout = 10)
public void m7() {
    System.out.println("Using @Test(timeout),it can be used to enforce
    timeout in JUnit4 test case");
}
```

As discussed above *@Test(timeout = 10)*annotation is used to enforce timeout in the test case.



III. Consider method `m8()` as given below :

```
@Test(expected = NoSuchMethodException.class)

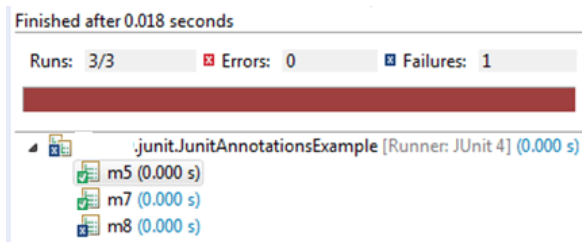
public void m8() {
    System.out.println("Using @Test(expected) ,it will check for specified
exception during its execution");
}
```

As discussed above `@Test(expected)` will check for specified exception during its execution so method `m8()` will throw "No Such Method Exception." As a result, the test will be executed with an exception.

As all test cases are passed, this results in a successful test execution.

#### Actual Result:

As there are three test cases in above example, all test cases will be executed one by one. See output below:



#### **Assert Class**

Assert is a method useful in determining Pass or Fail status of a test case, the assert methods are provided by the class `org.junit.Assert` which extends `java.lang.Object` class.

There are various types of assertions like Boolean, Null, Identical etc.

JUnit provides a class named Assert, which provides a bunch of assertion methods useful in writing test cases and to detect test failure

The assert methods are provided by the class `org.junit.Assert` which extends `java.lang.Object` class.

All the assertions are in the Assert class.

```
public class Assert extends java.lang.Object
```

This class provides a set of assertion methods, useful for writing tests. Only failed assertions are recorded. Some of the important methods of Assert class are as follows –

Sr.No.	Methods & Description
1	<b>void assertEquals(boolean expected, boolean actual)</b> Checks that two primitives/objects are equal.
2	<b>void assertTrue(boolean condition)</b> Checks that a condition is true.
3	<b>void assertFalse(boolean condition)</b> Checks that a condition is false.
4	<b>void assertNotNull(Object object)</b> Checks that an object isn't null.
5	<b>void assertNull(Object object)</b> Checks that an object is null.
6	<b>void assertSame(object1, object2)</b> The assertSame() method tests if two object references point to the same object.
7	<b>void assertNotSame(object1, object2)</b> The assertNotSame() method tests if two object references do not point to the same object.
8	<b>void assertEquals(expectedArray, resultArray);</b> The assertEquals() method will test whether two arrays are equal to each other.

Let's use some of the above-mentioned methods in an example.

Create a java class file named TestAssertions.java in C:\>JUNIT\_WORKSPACE.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestAssertions {
    @Test
    public void testAssertions() {
//test data
        String str1 = new String ("abc");
        String str2 = new String ("abc");
        String str3 = null;
        String str4 = "abc";
        String str5 = "abc";

        int val1 = 5;
        int val2 = 6;
        String[] expectedArray = {"one", "two", "three"};
        String[] resultArray = {"one", "two", "three"};

        //Check that two objects are equal
        assertEquals(str1, str2);

        //Check that a condition is true
        assertTrue (val1 < val2);

        //Check that a condition is false
        assertFalse(val1 > val2);

        //Check that an object isn't null
        assertNotNull(str1);

        //Check that an object is null
        assertNull(str3);

        //Check if two object references point to the same object
        assertSame(str4,str5);

        //Check if two object references not point to the same object
        assertNotSame(str1,str3);

        //Check whether two arrays are equal to each other.
        assertEquals(expectedArray, resultArray);
    } }
```

Next, create a java class file named TestRunner.java in C:\>JUNIT\_WORKSPACE to execute test case(s).

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner2 {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestAssertions.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

Compile the Test case and Test Runner classes using javac.

```
C:\JUNIT_WORKSPACE>javac TestAssertions.java TestRunner.java
```

Now run the Test Runner, which will run the test case defined in the provided Test Case class.

```
C:\JUNIT_WORKSPACE>javac TestRunner
```

Verify the output.

```
true
```

## Test Driver Development

Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.

- Tests are non-production code written in the same language as the application.
- Tests return a simple pass or fail, giving the developer immediate feedback.

Test-Driven Development starts with designing and developing tests for every small functionality of an application. In TDD approach, first, the test is developed which specifies and validates what the code will do.

## Why TDD?

A significant advantage of TDD is that it enables you to take small steps when writing software.

- TDD is done at Unit level - i.e. testing the internals of a class
- Tests are written for every function
- Mostly written by developers using one of the tool specific to the application.

## Testing Frameworks and Tools

The following Tools are available for Unit Testing:

- |                    |                   |
|--------------------|-------------------|
| • cputest          | • JUnit           |
| • csUnit (.Net)    | • NDbUnit         |
| • CUnit            | • NUnit           |
| • DUnit (Delphi)   | • OUnit           |
| • DBFit            | • PHPUnit         |
| • DBUnit           | • PyUnit (Python) |
| • DocTest (Python) | • SimpleTest      |
| • Googletest       | • TestNG          |
| • HTMLUnit         | • VUnit           |
| • HTTPUnit         | • XUnit           |
| • JMock            | • xUnit.net       |

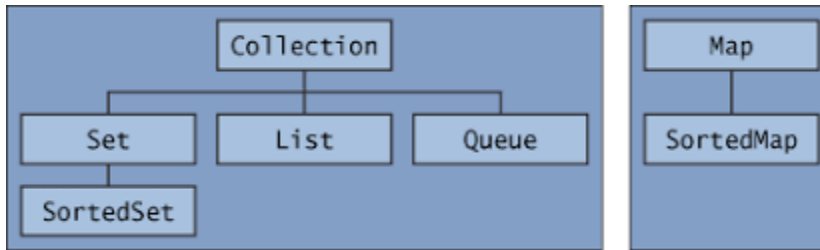


## Collections

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

The collections framework defines several interfaces. This section provides an overview of each interface –

Interface	Description
<b>The Collection Interface</b>	This enables you to work with groups of objects; it is at the top of the collection's hierarchy.
<b>The List Interface</b>	This extends Collection and an instance of List stores an ordered collection of elements.
<b>The Set</b>	This extends Collection to handle sets, which must contain unique elements.
<b>The SortedSet</b>	This extends Set to handle sorted sets.
<b>The Map</b>	This maps unique keys to values.
<b>The Map.Entry</b>	This describes an element (a key/value pair) in a map. This is an inner class of Map.
<b>The SortedMap</b>	This extends Map so that the keys are maintained in an ascending order.
<b>The Enumeration</b>	This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

Rather than getting into more details about all the interfaces, we thought it would be helpful to first discuss the concrete data structures that the Java library supplies. Once we have thoroughly described the classes you might want to use, we will return to abstract considerations and see how the collections framework organizes these classes.

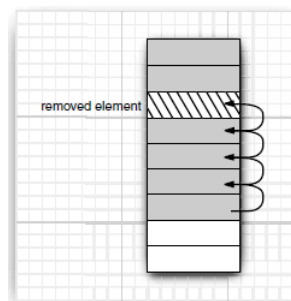
The collections in the Java library and briefly describes the purpose of each collection class shows below.

All classes in Table below implement the Collection interface, with the exception of the classes with names ending in Map. Those classes implement the Map interface instead.

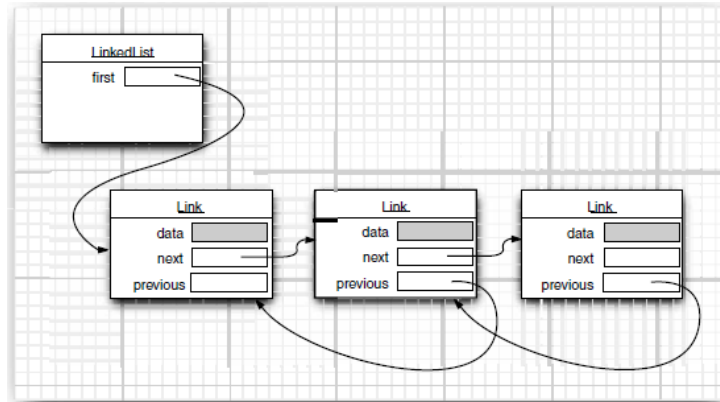
Collection Type	Description
<b>ArrayList</b>	An indexed sequence that grows and shrinks dynamically
<b>LinkedList</b>	An ordered sequence that allows efficient insertions and removal at any location
<b>HashSet</b>	An unordered collection that rejects duplicates
<b>TreeSet</b>	A sorted set
<b>EnumSet</b>	A set of enumerated type values
<b>LinkedHashSet</b>	A set that remembers the order in which elements were inserted
<b>PriorityQueue</b>	A collection that allows efficient removal of the smallest element
<b>HashMap</b>	A data structure that stores key/value associations
<b>TreeMap</b>	A map in which the keys are sorted
<b>EnumMap</b>	A map in which the keys belong to an enumerated type
<b>LinkedHashMap</b>	A map that remembers the order in which entries were added
<b>WeakHashMap</b>	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere
<b>IdentityHashMap</b>	A map with keys that are compared by ==, not equals

## Linked Lists

We used arrays and their dynamic cousin, the *ArrayList* class, for many examples in Volume 1. However, arrays and array lists suffer from a major drawback. Removing an element from the middle of an array is expensive since all array elements beyond the removed one must be moved toward the beginning of the array (see Figure). The same is true for inserting elements in the middle.



Another well-known data structure, the linked list, solves this problem. Whereas an array stores object references in consecutive memory locations, a linked list stores each object in a separate link. Each link also stores a reference to the next link in the sequence. In the Java programming language, all linked lists are actually doubly linked; that is, each link also stores a reference to its predecessor.



### Doubly LinkedList

Removing an element from the middle of a linked list is an inexpensive operation only the links around the element to be removed need to be updated. Perhaps you once took a data structures course in which you learned how to implement linked lists.

You may have bad memories of tangling up the links when removing or adding elements in the linked list. If so, you will be pleased to learn that the Java collections library supplies a class `LinkedList` ready for you to use.

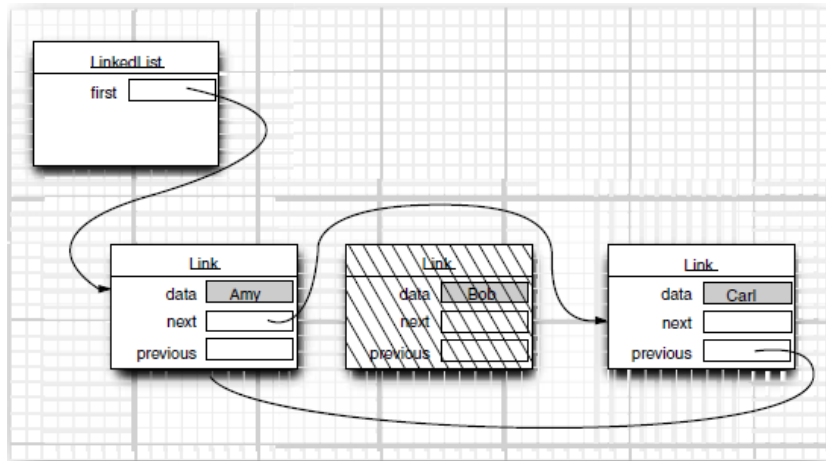
The following code example adds three elements and then removes the second one:

```
interface Queue<E>
List<String> staff = new LinkedList<String>(); // LinkedList
implements List
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

There is, however, an important difference between linked lists and generic collections. A linked list is an ordered collection in which the position of the objects matters.



The LinkedList.add method adds the object to the end of the list. But you often want to add objects somewhere in the middle of a list. This position-dependent add method is the



### Removing an element from a linked list

Responsibility of an iterator, since iterators describe positions in collections. Using iterators to add elements makes sense only for collections that have a natural ordering. For example, the set data type that we discuss in the next section does not impose any ordering on its elements.

Therefore, there is no add method in the Iterator interface. Instead, the collections library supplies a sub-interface *ListIterator* that contains an add method:

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    ...
}
```

Unlike Collection.add, this method does not return a boolean it is assumed that the add operation always modifies the list. In addition, the ListIterator interface has two methods that you can use for traversing a list backwards.

*E previous()*

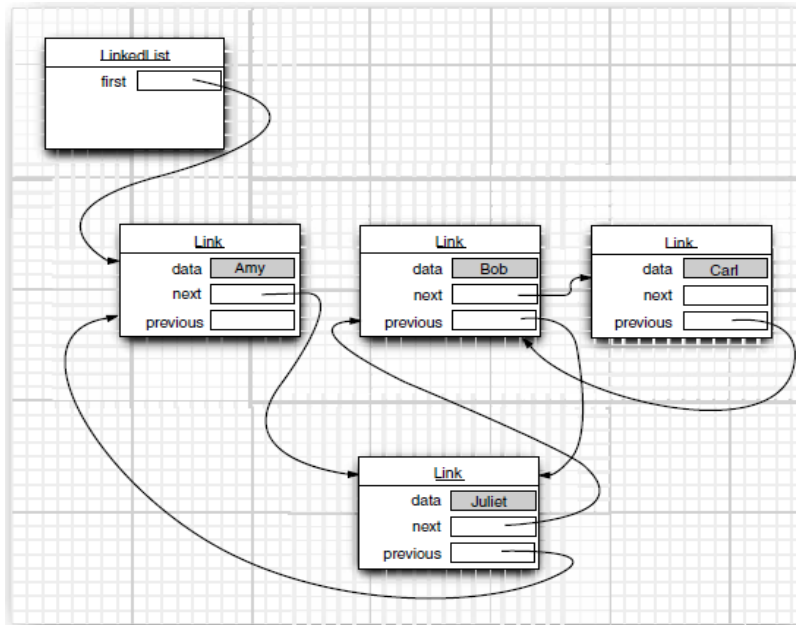
*boolean hasPrevious()*

Like the next method, the previous method returns the object that it skipped over. The listIterator method of the LinkedList class returns an iterator object that implements the ListIterator interface.

*ListIterator<String> iter = staff.listIterator();*

The add method adds the new element before the iterator position. For example, the following code skips past the first element in the linked list and adds "Juliet" before the second element :

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```



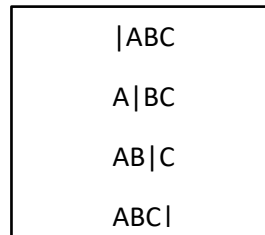
### Adding an element to a linked list

If you call the add method multiple times, the elements are simply added in the order in which you supplied them. They are all added in turn before the current iterator position.

When you use the add operation with an iterator that was freshly returned from the list- iterator method and that points to the beginning of the linked list, the newly added element becomes the new head of the list.

When the iterator has passed the last element of the list (that is, when hasNext returns false), the added element becomes the new tail of the list. If the linked list has n elements, there are n + 1 spots for adding a new element.

These spots correspond to the  $n + 1$  possible positions of the iterator. For example, if a linked list contains three elements, A, B, and C, then there are four possible positions (marked as |) for inserting a new element:



Finally, a set method replaces the last element returned by a call to next or previous with a new element. For example, the following code replaces the first element of a list with a new value:

```
ListIterator<String> iter = list.listIterator();  
  
String oldValue = iter.next(); // returns first element  
  
iter.set(newValue); // sets first element to newValue
```

As you might imagine, if an iterator traverses a collection while another iterator is modifying it, confusing situations can occur. For example, suppose an iterator points before an element that another iterator has just removed. The iterator is now invalid and should no longer be used.

The linked list iterators have been designed to detect such modifications. If an iterator finds that its collection has been modified by another iterator or by a method of the collection itself, then it throws a *ConcurrentModificationException*.

For example, consider the following code:

```
List<String> list = . . .;  
ListIterator<String> iter1 = list.listIterator();  
ListIterator<String> iter2 = list.listIterator();  
iter1.next();  
iter1.remove();  
iter2.next(); // throws ConcurrentModificationException
```

The call to *iter2.next* throws a *ConcurrentModificationException* since *iter2* detects that the list was modified externally. To avoid concurrent modification exceptions, follow this simple rule:

You can attach as many iterators to a collection as you like, provided that all of them are only readers. Alternatively, you can attach a single iterator that can both read and write.

Concurrent modification detection is achieved in a simple way. The collection keeps track of the number of mutating operations (such as adding and removing elements). Each iterator keeps a separate count of the number of mutating

operations that it was responsible for. At the beginning of each iterator method, the iterator simply checks whether its own mutation count equals that of the collection. If not, it throws a *ConcurrentModificationException*.

Now you have seen the fundamental methods of the `LinkedList` class. You use a *ListIterator* to traverse the elements of the linked list in either direction and to add and remove elements.

As you saw in the preceding section, many other useful methods for operating on linked lists are declared in the **Collection interface**. These are, for the most part, implemented in the *AbstractCollection* superclass of the `LinkedList` class. For example, the **toString** method invokes `toString` on all elements and produces one long string of the format `[A, B, C]`.

This is handy for debugging. Use the `contains` method to check whether an element is present in a linked list. For example, the call `staff.contains("Harry")` returns true if the linked list already contains a string that is equal to the string "Harry".

The library also supplies a number of methods that are, from a theoretical perspective, somewhat dubious. **Linked lists do not support fast random access**. If you want to see the *n*th element of a linked list, you have to start at the beginning and skip past the first *n* – 1 elements first. There is no shortcut. For that reason, programmers don't usually use linked lists in programming situations in which elements need to be accessed by an integer index.

Nevertheless, the `LinkedList` class supplies a `get` method that lets you access a particular element:

```
LinkedList<String> list = . . .;  
String obj = list.get(n);
```

Of course, this method is not very efficient. If you find yourself using it, you are probably using the wrong data structure for your problem.

You should never use this illusory random-access method to step through a linked list. The code

```
for (int i = 0; i < list.size(); i++)  
do something with list.get(i);
```

is staggeringly inefficient. Each time you look up another element, the search starts again from the beginning of the list. The `LinkedList` object makes no effort to cache the position information.

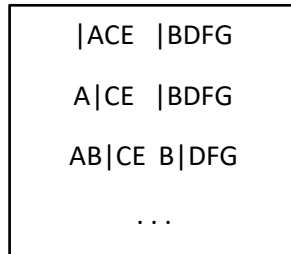
The list iterator interface also has a method to tell you the index of the current position. In fact, because Java iterators conceptually point between elements, it has two of them: The *nextIndex* method returns the integer index of the element that would be returned by the next call to `next`; the *previousIndex* method returns the index of the element that would be returned by the next call to `previous`. Of course, that is simply one less than *nextIndex*.

These methods are efficient the iterators keep a count of the current position. Finally, if you have an integer index  $n$ , then `list.listIterator(n)` returns an iterator that points just before the element with index  $n$ . That is, calling `next` yields the same element as `list.get(n)`; obtaining that iterator is inefficient.

If you have a linked list with only a handful of elements, then you don't have to be overly paranoid about the cost of the `get` and `set` methods. But then why use a linked list in the first place? The only reason to use a linked list is to minimize the cost of insertion and removal in the middle of the list. If you have only a few elements, you can just use an `ArrayList`.

We recommend that you simply stay away from all methods that use an integer index to denote a position in a linked list. If you want random access into a collection, use an array or `ArrayList`, not a linked list.

The program in Listing below puts linked lists to work. It simply creates two lists, merges them, then removes every second element from the second list, and finally tests the `removeAll` method. We recommend that you trace the program flow and pay special attention to the iterators. You may find it helpful to draw diagrams of the iterator positions, like this:



Note that the call

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

Prints all elements in the linked list by invoking the `toString` method in `AbstractCollection`.

LinkedListTest.java :

```
import java.util.*;
// This program demonstrates operations on linked lists.
public class LinkedListTest
{
    public static void main(String[] args)
    {
        List<String> a = new LinkedList<String>();
```

```

a.add("Amy");
a.add("Carl");
a.add("Erica");
List<String> b = new LinkedList<String>();
b.add("Bob"); b.add("Doug");
b.add("Frances");
b.add("Gloria");
// merge the words from b into a

ListIterator<String> alter = a.listIterator();
Iterator<String> blter = b.iterator();
while (blter.hasNext())
{
if (alter.hasNext()) alter.next(); alter.add(blter.next());
}
System.out.println(a);
// remove every second word from b
blter = b.iterator();
while (blter.hasNext())
{
blter.next(); // skip one element
if (blter.hasNext())
{
blter.next(); // skip next element
blter.remove(); // remove that element
}
}
System.out.println(b);
// bulk operation: remove all words in b from a
a.removeAll(b);
System.out.println(a);
}
}

```

#### java.util.List<E> 1.2 :

- *ListIterator<E> listIterator()*  
returns a list iterator for visiting the elements of the list.
- *ListIterator<E> listIterator(int index)*

returns a list iterator for visiting the elements of the list whose first call to next will return the element with the given index.

- *void add(int i, E element)*  
adds an element at the specified position.
- *void addAll(int i, Collection<? extends E> elements)*  
adds all elements from a collection to the specified position.
- *E remove(int i)*  
removes and returns an element at the specified position.
- *E set(int i, E element)*  
replaces the element at the specified position with a new element and returns the old element.
- *int indexOf(Object element)*  
returns the position of the first occurrence of an element equal to the specified element, or -1 if no matching element is found.
- *int lastIndexOf(Object element)*  
returns the position of the last occurrence of an element equal to the specified element, or -1 if no matching element is found.

#### **java.util.ListIterator<E> 1.2**

- *void add(E newElement)*  
adds an element before the current position.
- *void set(E newElement)*  
replaces the last element visited by next or previous with a new element. Throws an IllegalStateException if the list structure was modified since the last call to next or previous.
- *boolean hasPrevious()*  
returns true if there is another element to visit when iterating backwards through the list.
- *E previous()*  
returns the previous object. Throws a NoSuchElementException if the beginning of the list has been reached.
- *int nextIndex()*  
returns the index of the element that would be returned by the next call to next.

- *int previousIndex()*  
returns the index of the element that would be returned by the next call to *previous*.

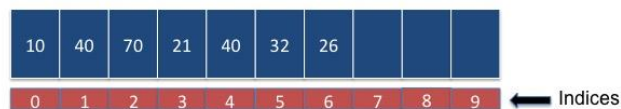
#### **java.util.LinkedList<E> 1.2**

- *LinkedList()*  
constructs an empty linked list.
- *LinkedList(Collection<? extends E> elements)*  
constructs a linked list and adds all elements from a collection.
- *void addFirst(E element)*
- *void addLast(E element)*  
add an element to the beginning or the end of the list.
- *E getFirst()*
- *E getLast()*  
return the element at the beginning or the end of the list.
- *E removeFirst()*
- *E removeLast()*  
remove and return the element at the beginning or the end of the list.

### **Array Lists**

In the preceding section, you saw the *List* interface and the *LinkedList* class that implements it. The *List* interface describes an ordered collection in which the position of elements matters. There are two protocols for visiting the elements: through an iterator and by random access with methods *get* and *set*. The latter is not appropriate for linked lists, but of course *get* and *set* make a lot of sense for arrays. The collections library supplies the familiar *ArrayList* class that also implements the *List* interface. An *ArrayList* encapsulates a dynamically reallocated array of objects.

Default size of *ArrayList* is 10



### **ArrayList**

### **Hash Sets**

Linked lists and arrays let you specify the order in which you want to arrange the elements. However, if you are looking for a particular element and you don't remember its position, then you need to visit all elements until you find a match. That can be time consuming if the collection contains many elements.



If you don't care about the ordering of the elements, then there are data structures that let you find elements much faster. The drawback is that those data structures give you no control over the order in which the elements appear. The data structures organize the elements in an order that is convenient for their own purposes.

A well-known data structure for finding objects quickly is the hash table. A hash table computes an integer, called the hash code, for each object. A hash code is an integer that is somehow derived from the instance fields of an object, preferably such that objects with different data yield different codes.

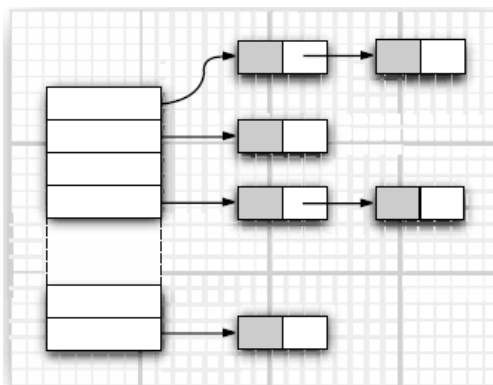
Table below lists a few examples of hash codes that result from the *hashCode* method of the *String* class:

String	Hash Code
"Lee"	76268
"lee"	107020
"eel"	100300

If you define your own classes, you are responsible for implementing your own *hashCode* method with the equals method: If *a.equals(b)*, then a and b must have the same hash code. What's important for now is that hash codes can be computed quickly and that the computation depends only on the state of the object that needs to be hashed, and not on the other objects in the hash table.

In Java, hash tables are implemented as arrays of linked lists. Each list is called a **bucket** (see Figure below). To find the place of an object in the table, compute its hash code and reduce it modulo the total number of buckets. The resulting number is the index of the bucket that holds the element.

For example, if an object has hash code 76268 and there are 128 buckets, then the object is placed in bucket 108 (because the remainder  $76268 \% 128$  is 108). Perhaps you are lucky and there is no other element in that bucket. Then, you simply insert the element into that bucket. Of course, it is inevitable that you sometimes hit a bucket that is already filled. This is called a **hash collision**.



Hash table

Then, you compare the new object with all objects in that bucket to see if it is already present. Provided that the hash codes are reasonably randomly distributed, and the number of buckets is large enough, only a few comparisons should be necessary.

Hash tables can be used to implement several important data structures. The simplest among them is the set type. A set is a collection of elements without duplicates. The add method of a set first tries to find the object to be added and adds it only if it is not yet present.

The program reads all words from the input and adds them to the hash set. It then iterates through the unique words in the set and finally prints out a count. (Alice in Wonderland has 5,909 unique words, including the copyright notice at the beginning.) The words appear in random order.

SetTest.java :

```
import java.util.*;
// This program uses a set to print all unique words in System.in.
public class SetTest
{
    public static void main(String[] args)
    {
        Set<String> words = new HashSet<String>(); // HashSet implements Set
        long totalTime = 0;
        Scanner in = new Scanner(System.in);
        while (in.hasNext())
        {
            String word = in.next();
            long callTime = System.currentTimeMillis();
            words.add(word);
            callTime = System.currentTimeMillis() - callTime;
            totalTime += callTime;
        }
        Iterator<String> iter = words.iterator();
        for (int i = 1; i <= 20; i++)
            System.out.println(iter.next());
        System.out.println("...");
        System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
    }
}
```

## java.util.HashSet<E> 1.2

- *HashSet()*

constructs an empty hash set.

- *HashSet(Collection<? extends E> elements)*  
constructs a hash set and adds all elements from a collection.
- *HashSet(int initialCapacity)*  
constructs an empty hash set with the specified capacity (number of buckets).
- *HashSet(int initialCapacity, float loadFactor)*  
constructs an empty hash set with the specified capacity and load factor (a number between 0.0 and 1.0 that determines at what percentage of fullness the hash table will be rehashed into a larger one).  
java.lang.Object 1.0
- *int hashCode()*  
returns a hash code for this object. A hash code can be any integer, positive or negative. The definitions of *equals* and *hashCode* must be compatible: If *x.equals(y)* is true, then *x.hashCode()* must be the same value as *y.hashCode()*.

## Tree Sets

The *TreeSet* class is similar to the hash set, with one added improvement. A tree set is a sorted collection. You insert elements into the collection in any order. When you iterate through the collection, the values are automatically presented in sorted order.

For example, suppose you insert three strings and then visit all elements that you added.

```
SortedSet<String> sorter = new TreeSet<String>();  
// TreeSet implements SortedSet  
sorter.add("Bob");  
sorter.add("Amy");  
sorter.add("Carl");  
for (String s : sorter) System.println(s);
```

Then, the values are printed in sorted order: *Amy Bob Carl*. As the name of the class suggests, the sorting is accomplished by a tree data structure. (The current implementation uses a red-black tree. For a detailed description of red-black trees, see, for example, Introduction to Algorithms by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein [The MIT Press 2001].)

Every time an element is added to a tree, it is placed into its proper sorting position. Therefore, the iterator always visits the elements in sorted order.

Adding an element to a tree is slower than adding it to a hash table, but it is still much faster than adding it into the right place in an array or linked list. If the tree contains  $n$  elements, then an average of  $\log_2 n$  comparisons are required to find the correct position for the new element.

For example, if the tree already contains 1,000 elements, then adding a new element requires about 10 comparisons.

Thus, adding elements into a *TreeSet* is somewhat slower than adding into a *HashSet*. for a comparison—but the *TreeSet* automatically sorts the elements.

Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	34545	75 sec	98 sec

- `TreeSet()`  
constructs an empty tree set.
- `TreeSet(Collection<? extends E> elements)`  
constructs a tree set and adds all elements from a collection.

#### Object Comparison

How does the *TreeSet* know how you want the elements sorted? By default, the tree set assumes that you insert elements that implement the *Comparable* interface. That interface defines a single method:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

The call *a.compareTo(b)* must return 0 if *a* and *b* are equal, a negative integer if *a* comes before *b* in the sort order, and a positive integer if *a* comes after *b*. The exact value does not matter; only its sign (>0, 0, or < 0) matters. Several standard Java platform classes implement the *Comparable* interface. One example is the *String* class. Its *compareTo* method compares strings in dictionary order (sometimes called lexicographic order).

If you insert your own objects, you must define a sort order yourself by implementing the *Comparable* interface. There is no default implementation of *compareTo* in the *Object* class.

For example, here is how you can sort *Item* objects by part number.

```
class Item implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        return partNumber - other.partNumber;
    }
    ...
}
```

If you compare two positive integers, such as part numbers in our example, then you can simply return their difference—it will be negative if the first item should come before the second item, zero if the part numbers are identical, and positive otherwise.

This trick only works if the integers are from a small enough range. If  $x$  is a large positive integer and  $y$  is a large negative integer, then the difference  $x - y$  can overflow.

However, using the *Comparable* interface for defining the sort order has obvious limitations. A given class can implement the interface only once. But what can you do if you need to sort a bunch of items by part number in one collection and by description in another? Furthermore, what can you do if you need to sort objects of a class whose creator didn't bother to implement the *Comparable* interface?

In those situations, you tell the tree set to use a different comparison method, by passing a *Comparator* object into the *TreeSet* constructor. The *Comparator* interface declares a *compare* method with two explicit parameters:

```
public interface Comparator<T>
{
    int compare(T a, T b);
}
```

Just like the *compareTo* method, the *compare* method returns a negative integer if *a* comes before *b*, zero if they are identical, or a positive integer otherwise.

To sort items by their description, simply define a class that implements the *Comparator* interface:

```
class ItemComparator implements
Comparator<Item>
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    } }
}
```

You then pass an object of this class to the tree set constructor:

```
ItemComparator comp = new ItemComparator();  
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

If you construct a tree with a comparator, it uses this object whenever it needs to compare two elements.

Note that this item comparator has no data. It is just a holder for the comparison method. Such an object is sometimes called a **function object**.

Function objects are commonly defined “on the fly,” as instances of anonymous inner classes:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new  
    Comparator<Item>()  
    {  
        public int compare(Item a, Item b)  
        {  
            String descrA = a.getDescription();
```

```
            String descrB = b.getDescription();  
            return descrA.compareTo(descrB);  
        }  
    });
```

If you look back at Table above, you may well wonder if you should always use a tree set instead of a hash set. After all, adding elements does not seem to take much longer, and the elements are automatically sorted. The answer depends on the data that you are collecting. If you don’t need the data sorted, there is no reason to pay for the sorting overhead.

More important, with some data it is much more difficult to come up with a sort order than a hash function. A hash function only needs to do a reasonably good job of scrambling the objects, whereas a comparison function must tell objects apart with complete precision.

To make this distinction more concrete, consider the task of collecting a set of rectangles. If you use a *TreeSet*, you need to supply a *Comparator<Rectangle>*. How do you compare two rectangles? By area? That doesn’t work. You can have two different rectangles with different coordinates but the same area. The sort order for a tree must be a total ordering.

Any two elements must be comparable, and the comparison can only be zero if the elements are equal. There is such a sort order for rectangles (the lexicographic ordering on its coordinates), but it is unnatural and cumbersome to compute. In contrast, a hash function is already defined for the Rectangle class. It simply hashes the coordinates.

### **java.lang.Comparable<T> 1.2**

- `int compareTo(T other)`  
compares this object with another object and returns a negative value if this comes before other, zero if they are considered identical in the sort order, and a positive value if this comes after other.

### **java.util.Comparator<T> 1.2**

- `int compare(T a, T b)`  
compares two objects and returns a negative value if a comes before b, zero if they are considered identical in the sort order, and a positive value if a comes after b.

### **java.util.SortedSet<E> 1.2**

- `Comparator<? super E> comparator()`  
returns the comparator used for sorting the elements, or null if the elements are compared with the `compareTo` method of the Comparable interface.
- `E first()`
- `E last()`  
returns the smallest or largest element in the sorted set.

### **java.util.NavigableSet<E> 6**

- `E higher(E value)`
- `E lower(E value)`  
returns the least element  $>$  value or the largest element  $<$  value, or null if there is no such element.
- `E ceiling(E value)`
- `E floor(E value)`  
returns the least element  $\geq$  value or the largest element  $\leq$  value, or null if there is no such element.
- `E pollFirst()`
- `E pollLast()`  
removes and returns the smallest or largest element in this set, or null if the set is empty.
- `Iterator<E> descendingIterator()`  
returns an iterator that traverses this set-in descending direction.

### **java.util.TreeSet<E> 1.2**

- `TreeSet()`  
constructs a tree set for storing Comparable objects.
- `TreeSet(Comparator<? super E> c)`  
constructs a tree set and uses the specified comparator for sorting its elements.
- `TreeSet(SortedSet<? extends E> elements)`  
constructs a tree set, adds all elements from a sorted set, and uses the same element comparator as the given sorted set.

## The Collection Framework

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hash tables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:

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

class Test
{
    public static void main (String[] args)
    {
        // Creating instances of array, vector and hashtable
        int arr[] = new int[] {1, 2, 3, 4};

        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();
        v.addElement(1);
        v.addElement(2);
        h.put(1,"geeks");
        h.put(2,"4geeks");

        // Array instance creation requires [], while Vector
        // and hashtable require ()
        // Vector element insertion requires addElement(), but
        // hashtable element insertion requires put()
```



```
// Accessing first element of array, vector and hashtable
System.out.println(arr[0]);
System.out.println(v.elementAt(0));
System.out.println(h.get(1));

// Array elements are accessed using [], vector elements
// using elementAt() and hashtable elements using get()
}
}
```

Output:

```
1
1
Geek
```

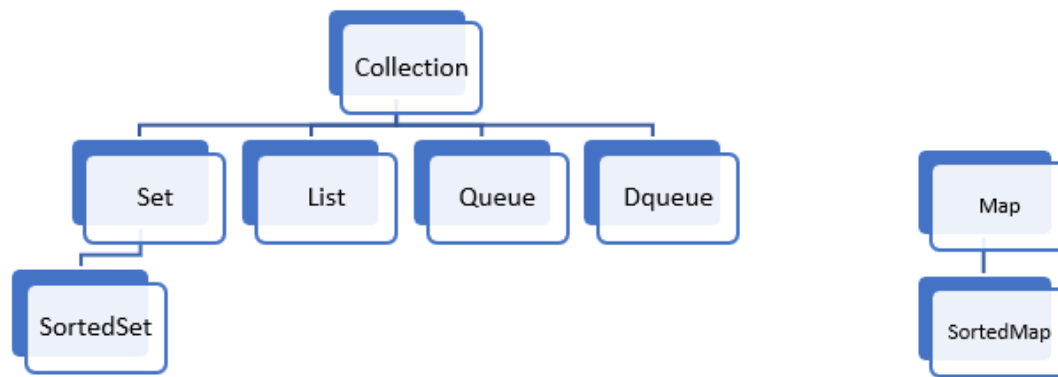
As we can see, none of these collections (Array, Vector or Hash table) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection.

**Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2.**

Both legacy Vectors and Hash tables were modified to conform to the Collection Framework.

**Advantages of Collection Framework:**

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
2. Reduces programming effort: A programmer doesn't have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.



**Fig. Core Interfaces in Collections**

Collection : Root interface with basic methods like add(), remove(), contains(), isEmpty(), addAll(), ... etc.

Set : Doesn't allow duplicates. Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based). Note that TreeSet implements SortedSet.

List : Can contain duplicates and elements are ordered. Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

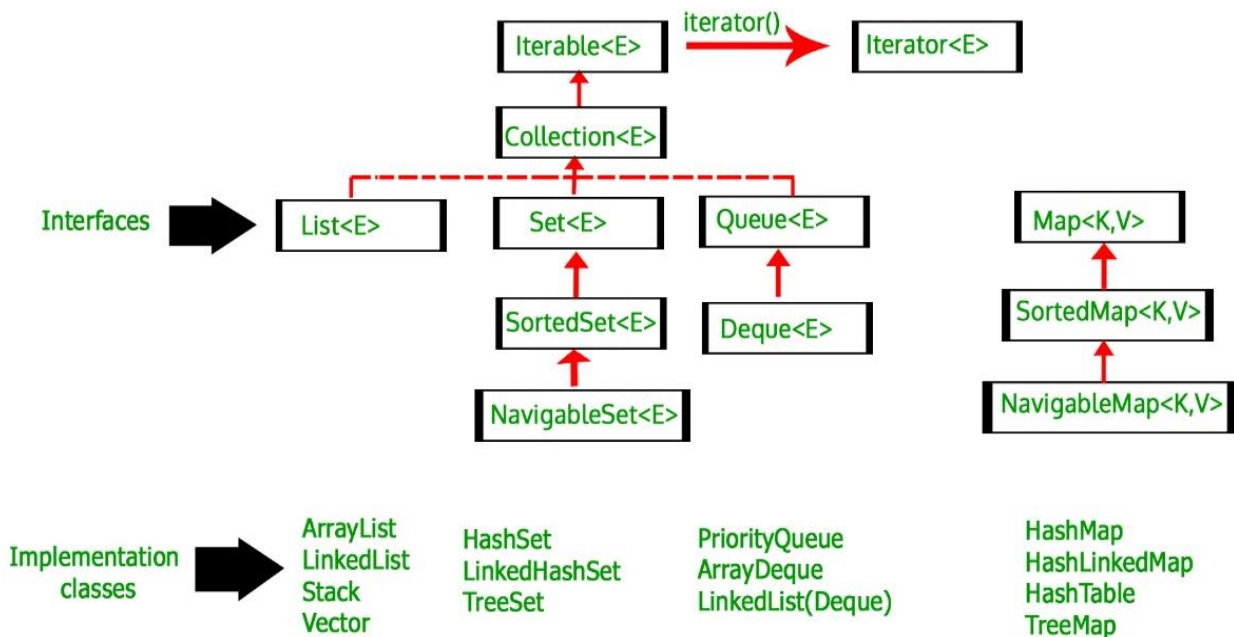
Queue : Typically order elements in FIFO order except exceptions like PriorityQueue.

Deque : Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.

Map : Contains Key value pairs. Doesn't allow duplicates. Example implementation are HashMap and TreeMap. TreeMap implements SortedMap.

The difference between Set and Map interface is that in Set we

have only keys, whereas in Map, we have key, value pairs.



**Fig. Detailed Core Interfaces in Collections**

## Collection Algorithms

The collections framework defines several algorithms that can be applied to collections and maps.

These algorithms are defined as static methods within the Collections class. Several of the methods can throw a *ClassCastException*, which occurs when an attempt is made to compare incompatible types, or an *UnsupportedOperationException*, which occurs when an attempt is made to modify an unmodifiable collection.

The methods defined in collection framework's algorithm are summarized in the following table –

Sr. No.	Methods & Description
1	<b>static int binarySearch(List list, Object value, Comparator c)</b> Searches for value in the list ordered according to <b>c</b> . Returns the position of value in list, or -1 if value is not found.

2	<b>static int binarySearch(List list, Object value)</b>  Searches for value in the list. The list must be sorted. Returns the position of value in list, or -1 if value is not found.
3	<b>static void copy(List list1, List list2)</b>  Copies the elements of list2 to list1.
4	<b>static Enumeration enumeration(Collection c)</b>  Returns an enumeration over c.
5	<b>static void fill(List list, Object obj)</b>  Assigns obj to each element of the list.
6	<b>static int indexOfSubList(List list, List subList)</b>  Searches list for the first occurrence of subList. Returns the index of the first match, or .1 if no match is found.
7	<b>static int lastIndexOfSubList(List list, List subList)</b>  Searches list for the last occurrence of subList. Returns the index of the last match, or .1 if no match is found.
8	<b>static ArrayList list(Enumeration enum)</b>  Returns an ArrayList that contains the elements of enum.
9	<b>static Object max(Collection c, Comparator comp)</b>  Returns the maximum element in c as determined by comp.
10	<b>static Object max(Collection c)</b>  Returns the maximum element in c as determined by natural ordering. The collection need not be sorted.

11	<b>static Object min(Collection c, Comparator comp)</b>  Returns the minimum element in <b>c</b> as determined by <b>comp</b> . The collection need not be sorted.
12	<b>static Object min(Collection c)</b>  Returns the minimum element in <b>c</b> as determined by natural ordering.
13	<b>static List nCopies(int num, Object obj)</b>  Returns num copies of <b>obj</b> contained in an immutable list. num must be greater than or equal to zero.
14	<b>static boolean replaceAll(List list, Object old, Object new)</b>  Replaces all occurrences of <b>old</b> with <b>new</b> in the list. Returns true if at least one replacement occurred. Returns false, otherwise.
15	<b>static void reverse(List list)</b>  Reverses the sequence in list.
16	<b>static Comparator reverseOrder( )</b>  Returns a reverse comparator.
17	<b>static void rotate(List list, int n)</b>  Rotates list by <b>n</b> places to the right. To rotate left, use a negative value for <b>n</b> .
18	<b>static void shuffle(List list, Random r)</b>  Shuffles (i.e., randomizes) the elements in the list by using <b>r</b> as a source of random numbers.
19	<b>static void shuffle(List list)</b>  Shuffles (i.e., randomizes) the elements in list.

20	<b>static Set singleton(Object obj)</b> Returns obj as an immutable set. This is an easy way to convert a single object into a set.
21	<b>static List singletonList(Object obj)</b> Returns obj as an immutable list. This is an easy way to convert a single object into a list.
22	<b>static Map singletonMap(Object k, Object v)</b> Returns the key/value pair k/v as an immutable map. This is an easy way to convert a single key/value pair into a map.
23	<b>static void sort(List list, Comparator comp)</b> Sorts the elements of list as determined by comp.
24	<b>static void sort(List list)</b> Sorts the elements of the list as determined by their natural ordering.
25	<b>static void swap(List list, int idx1, int idx2)</b> Exchanges the elements in the list at the indices specified by idx1 and idx2.
26	<b>static Collection synchronizedCollection(Collection c)</b> Returns a thread-safe collection backed by c.
27	<b>static List synchronizedList(List list)</b> Returns a thread-safe list backed by list.
28	<b>static Map synchronizedMap(Map m)</b> Returns a thread-safe map backed by m.
29	<b>static Set synchronizedSet(Set s)</b> Returns a thread-safe set backed by s.

30	<b>static SortedMap synchronizedSortedMap(SortedMap sm)</b> Returns a thread-safe sorted set backed by <b>sm</b> .
31	<b>static SortedSet synchronizedSortedSet(SortedSet ss)</b> Returns a thread-safe set backed by <b>ss</b> .
32	<b>static Collection unmodifiableCollection(Collection c)</b> Returns an unmodifiable collection backed by <b>c</b> .
33	<b>static List unmodifiableList(List list)</b> Returns an unmodifiable list backed by the list.
34	<b>static Map unmodifiableMap(Map m)</b> Returns an unmodifiable map backed by <b>m</b> .
35	<b>static Set unmodifiableSet(Set s)</b> Returns an unmodifiable set backed by <b>s</b> .
36	<b>static SortedMap unmodifiableSortedMap(SortedMap sm)</b> Returns an unmodifiable sorted map backed by <b>sm</b> .
37	<b>static SortedSet unmodifiableSortedSet(SortedSet ss)</b> Returns an unmodifiable sorted set backed by <b>ss</b> .

Following is an example, which demonstrates various algorithms.

```
import java.util.*;
public class AlgorithmsDemo {
    public static void main(String args[]) { // Create and initialize linked list
        LinkedList ll = new LinkedList();
        ll.add(new Integer(-8));
        ll.add(new Integer(20));
    }
}
```

```
ll.add(new Integer(-20));
ll.add(new Integer(8));
// Create a reverse order comparator
Comparator r = Collections.reverseOrder();
// Sort list by using the comparator
Collections.sort(ll, r);
// Get iterator
Iterator li = ll.iterator();
System.out.print("List sorted in reverse: ");
while(li.hasNext()) {
    System.out.print(li.next() + " ");
}
System.out.println();
Collections.shuffle(ll);
// display randomized list
li = ll.iterator();
System.out.print("List shuffled: ");
while(li.hasNext()) {
    System.out.print(li.next() + " ");
}
System.out.println();
System.out.println("Minimum: " + Collections.min(ll));
System.out.println("Maximum: " + Collections.max(ll));
}
}
```

This will produce the following result –

Output :

```
List sorted in reverse: 20 8 -8 -20

List shuffled: 20 -20 8 -8

Minimum: -20

Maximum: 20
```



## What are Threads?

You are probably familiar with multitasking: the ability to have more than one program working at what seems like the same time. For example, you can print while editing or sending fax. Of course, unless you have a multiple-processor machine, what is really going on is that the operating system is doling out resources to each program, giving the impression of parallel activity.

This resource distribution is possible because while you may think you are keeping the computer busy by, for example, entering data, most of the CPU's time will be idle.

Multitasking can be done in two ways, depending on whether the operating system interrupts programs without consulting with them first, or whether programs are only interrupted when they are willing to yield control.

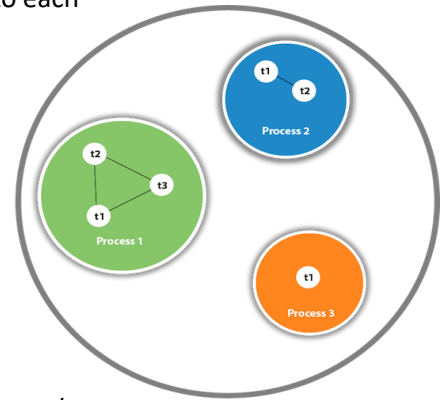
The former is called pre-emptive multitasking; the latter is called co-operative multitasking. Windows 3.1 and Mac OS 9 are co-operative multitasking systems, and UNIX/LINUX, Windows NT and OS X are pre-emptive.

Multithreading refers to two or more tasks executing concurrently within a single program. A thread is an independent path of execution within a program. Many threads can run concurrently within a program.

Every thread in Java is created and controlled by the **java.lang.Thread** class. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.

Multithreading has several advantages over Multiprocessing such as:

- Threads are **lightweight** compared to processes
- Threads share the same address space and therefore can share both data and code
- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.



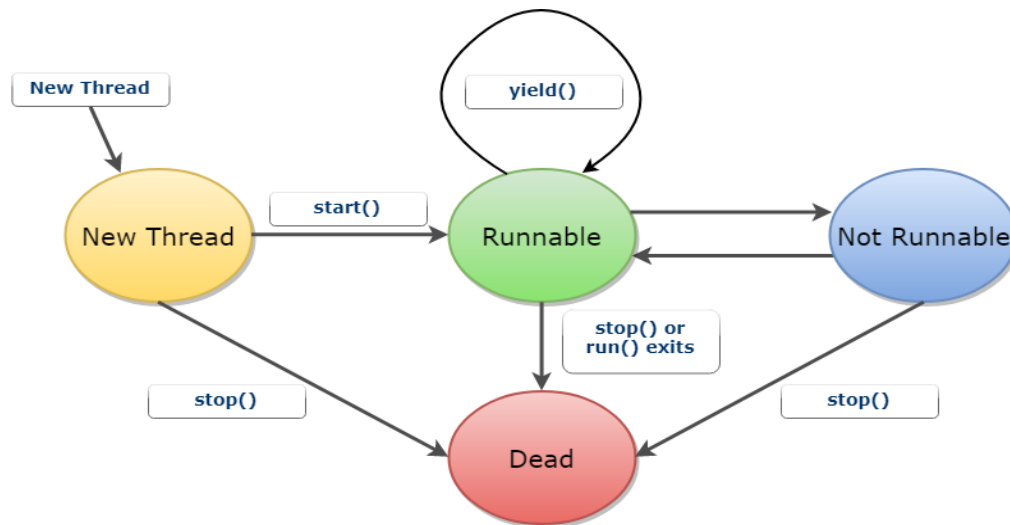
Object	Thread
<code>notify()</code> <code>notifyAll()</code> <code>wait()</code>	<code>sleep()</code> <code>yield()</code>

**Fig. Methods of Object and Thread Class**

Thread creation can be performed in two ways in Java:

- Implement the Runnable interface (java.lang.Runnable)
- By Extending the Thread class (java.lang.Thread)

A thread goes through various stages of its life cycle. Example, first of all, a thread is born, started its tasks, run a sequence of tasks concurrently, and then dies. Here is the diagram of the various stages of a life cycle.



**Fig. Life Cycle of a Thread in Java**

1. **New Thread:** A new thread begins its life cycle in the new state. The process remains in this condition until the program starts the thread.
2. **Runnable:** As soon as the new thread starts, the thread status becomes Runnable. At this stage, a thread is considered to execute its function or working.
3. **Not Runnable:** A Runnable thread when entered the time of waiting for the state for a specific interval of time. That time, the thread is not in Runnable condition.
4. **Dead / Terminated:** The Runnable thread enters the end stage when it completes its tasks.



## Thread Properties

Java assigns each thread a priority that concludes that how a thread will be treated concerning others. Thread priorities are integers that specify the relative priority of one thread with another. Thread priorities are used for deciding when to switch from one running thread to another. It is called a context switch.

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

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

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.

### **Implementing the Runnable Interface:**

```
public interface Runnable {  
    void run();  
}
```

One way to create thread in java is to implement the Runnable Interface and then instantiate an object of the class. You need to override the run() method into the class which is the only method that needs to be implemented. The run() method contains the logic of the thread.

Following is the program that illustrates instantiation and running of threads:

```
class RunnableThread implements Runnable {  
  
    Thread runner;  
    public RunnableThread() {  
    }  
    public RunnableThread(String threadName) {  
        runner = new Thread(this, threadName); // (1) Create a new thread.  
        System.out.println(runner.getName());  
        runner.start(); // (2) Start the thread.  
    }  
    public void run() {  
        //Display info about this particular thread  
        System.out.println(Thread.currentThread());  
    }  
}  
  
public class RunnableExample {  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(new RunnableThread(), "thread1");  
        Thread thread2 = new Thread(new RunnableThread(), "thread2");  
        RunnableThread thread3 = new RunnableThread("thread3");  
    }  
}
```

```
//Start the threads
    thread1.start();
    thread2.start();
    try {
        //delay for one second
        Thread.currentThread().sleep(1000);
    } catch (InterruptedException e) {
    }
    //Display info about the main thread
    System.out.println(Thread.currentThread());
}
```

#### Output:

```
thread3
Thread[thread1,5,main] Thread[thread2,5,main]
Thread[thread3,5,main] Thread[main,5,main]private
```

This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class.

### **Synchronization**

In the multithreaded programming, multiple threads run simultaneously and access common resources. To prevent deadlock, we must ensure that a resource must be shared by one thread at a time or else it may produce a weird or unforeseen result. This process is known as **Synchronization**.

#### **How to achieve thread synchronization in Java?**

Let us understand this with the help of an example!!

Assume, there are two threads which are accessing and writing to common file *output.txt*. If no synchronization is used then, one thread writes few words in the file and meanwhile another thread starts writing to the file. The resulting file will contain random contents written by both threads. With synchronization, if one thread is writing to the file, the file will be locked (in LOCK mode) and no other thread or process can access it until first thread completed its work.

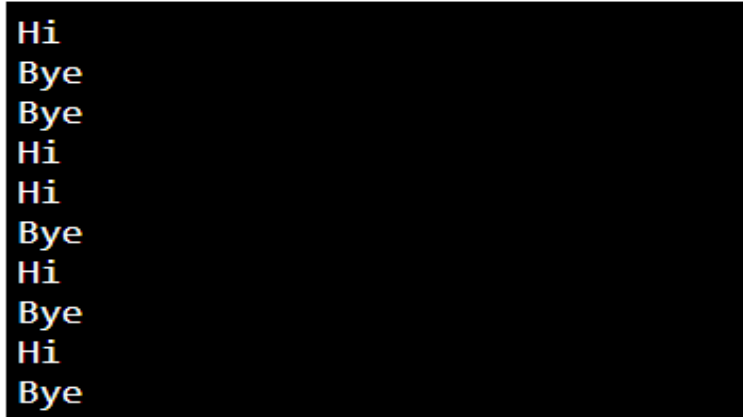
```
class print{
    public void printMSG(String s){
        for(int i=1;i<=5;i++) {
            System.out.println(s);
            try {
                Thread.sleep(1000);    // used to sleep current execution for 1s
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}

class one extends Thread{
    print t;
    one(print t){
        this.t=t;
    }
    public void run(){
        t.printMSG("Hi");
    }
}

class two extends Thread{
    print t;
    two(print t){
        this.t=t;
    }
    public void run() {
        t.printMSG("Bye");
    }
}

public class ExSynchronization {
    public static void main(String[] args) {
        print t=new print();
        one ob=new one(t);
        two o2=new two(t);
        ob.start();
        o2.start();
    }
}
```

Output:



```
Hi
Bye
Bye
Hi
Hi
Bye
Hi
Bye
Hi
Bye
```

In this program, we have designed two thread which are accessing a common function *printMSG()*. When we run this code, we may get the unwanted output as seen above.

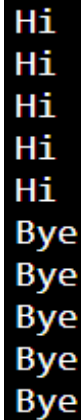
To synchronise the output, we will use synchronized method. This will lock the object for a shared resource. We can do this by adding synchronized keyword to the shared method.

Below is the synchronized code:

```
class print{
    synchronized public void printMSG(String s){
        for(int i=1;i<=5;i++) {
            System.out.println(s);
            try {
                Thread.sleep(1000); // used to sleep current execution for 1s
            } catch (Exception e) {
                System.out.println(e);
            }
        } } }
class one extends Thread{
    print t;
    one(print t){
        this.t=t;
    }
    public void run(){
        t.printMSG("Hi");
    }
}
```

```
class two extends Thread{
    print t;
    two(print t){
        this.t=t;
    }
    public void run() {
        t.printMSG("Bye");
    }
}
public class ExSynchronization {
    public static void main(String[] args) {
        print t=new print();
        one ob=new one(t);
        two o2=new two(t);
        ob.start();
        o2.start();
    }
}
```

Output:



```
Hi
Hi
Hi
Hi
Hi
Bye
Bye
Bye
Bye
Bye
```

## 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.

Below is the 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...
```

Now let us change the order of the lock and run 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...
```