Introduction to Java

JAVA was developed by James Gosling at Sun Microsystems Inc. in the year 1991, later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs.

Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to write once run anywhere that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

Application - According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

- Desktop Applications such as acrobat reader, media player, antivirus, etc.
- Web Applications such as irctc.co.in, javatpoint.com, etc.
- Enterprise Applications such as banking applications.
- Mobile
- Embedded System
- Smart Card
- Robotics
- Games, etc.

Types of Java Applications

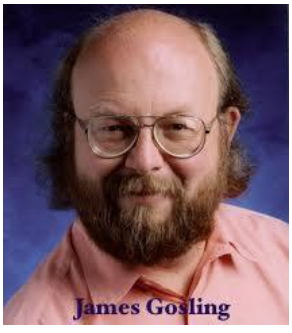
1) Standalone Application - Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application - An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application - An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application - An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

History of Java



The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

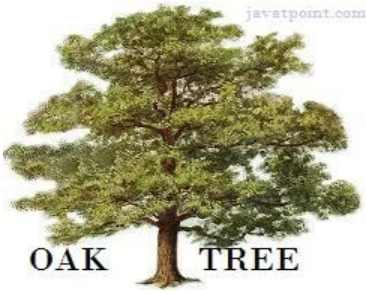
The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s. Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

1) **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.



Why Java was named as "Oak"?

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Features of Java

1. Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of bytecode. That is why we call java a platform-independent language.

2. Object-Oriented Programming Language: Organizing the program in the terms of collection of objects is a way of object-oriented programming, each of which represents an instance of the class. The four main concepts of Object-Oriented programming are: Abstraction, Encapsulation, Inheritance and Polymorphism.

3. Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and explicit memory allocation.

4. Robust: Java language is robust that means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

5. Secure: In java, we don't have pointers, and so we cannot access out-of-bound arrays i.e. it shows **ArrayIndexOutOfBound Exception** if we try to do so. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

6. Distributed: We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

7. Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

8. Portable: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

9. High Performance: Java architecture is defined in such a way that it reduces overhead during the runtime and at some time java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basics where it only compiles those methods that are called making applications to execute faster.

10. Dynamic flexibility: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes and even creating new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

11. Sandbox Execution: Java programs run in a separate space that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier. Bytecode verifier also provides additional security as its role is to check the code for any violation access.

12. Write Once Run Anywhere: As discussed above java application generates '.class' file which corresponds to our applications (program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

13. Power of compilation and interpretation: Most languages are designed with purpose either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

C++ vs Java

There are many differences and similarities between the C++ programming language and Java. A list of top differences between C++ and Java are given below:

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritances.	Java doesn't support multiple inheritances through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is

		platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

Note: Java doesn't support default arguments like C++. Java does not support header files like C++. Java uses the import keyword to include different classes and methods.

First Java Program | Hello World Example

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

```
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

Save the above file as Simple.java.

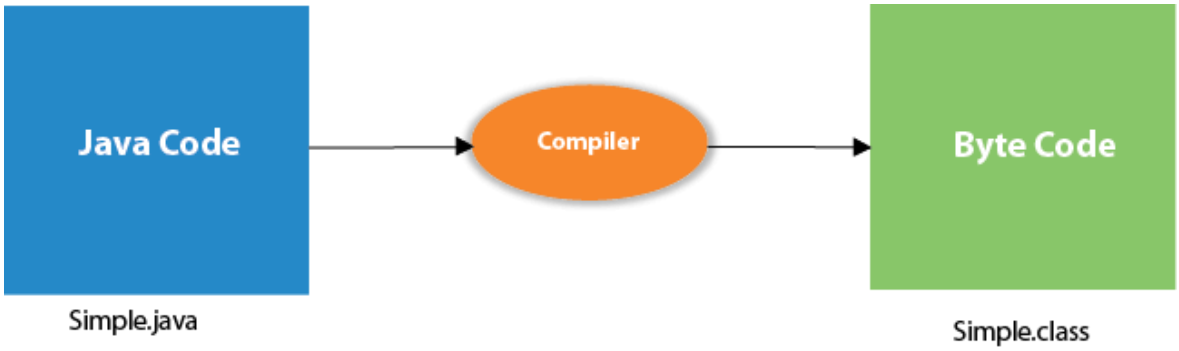
```
To compile: javac Simple.java
To execute: java Simple
```

The javac compiler creates a file called Sample.class that contains the bytecode version of the program. The Java bytecode is the intermediate representation of your program that contains instructions the JVM will execute. Thus, the output of javac is not code that can be directly executed.

To actually run the program you must use the Java application launcher, called java. To do so, pass the class name Sample as a command-link argument.

When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension. This is why it is a good idea to give your Java source files the same name as the class they contain – the name of othe source file will match the name of the .class file.

Compilation Flow: When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

In how many ways we can write a Java program?

There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

1) By changing the sequence of the modifiers, method prototype is not changed in Java.

```
static public void main(String args[])
```

2) The subscript notation in the Java array can be used after type, before the variable or after the variable.

```
public static void main(String[] args)
public static void main(String []args)
public static void main(String args[])
```

3) You can provide var-args support to the main() method by passing 3 ellipses (dots). Let's see the simple code of using var-args in the main() method. We will learn about var-args later in the Java New Features chapter.

```
public static void main(String... args)
```

4) Having a semicolon at the end of class is optional in Java. Let's see the simple code.

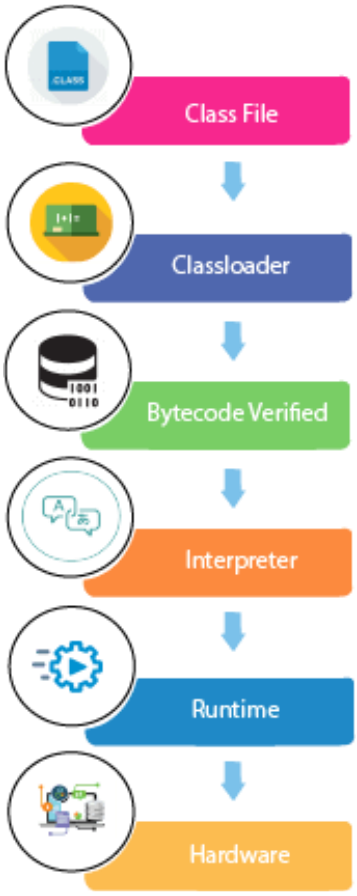
```
class A{
static public void main(String... args){
System.out.println("hello java4");
}
};
```

Valid Java main() method signature

```
public static void main(String[] args)
public static void main(String []args)
public static void main(String args[])
public static void main(String... args)
static public void main(String[] args)
public static final void main(String[] args)
final public static void main(String[] args)
final strictfp public static void main(String[] args)
```

Invalid Java main() method signature

```
public void main(String[] args)
static void main(String[] args)
public void static main(String[] args)
abstract public static void main(String[] args)
```



Internal Details of Hello Java Program

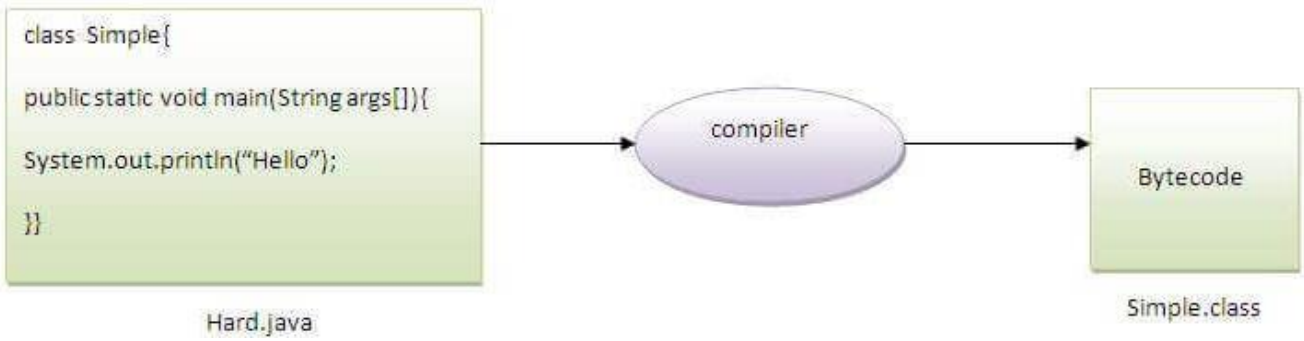
In the previous section, we have created Java Hello World program and learn how to compile and run a Java program. In this section, we are going to learn, what happens while we compile and run the Java program.

What happens at compile time? At compile time, the Java file is compiled by Java Compiler (It does not interact with OS) and converts the Java code into bytecode.

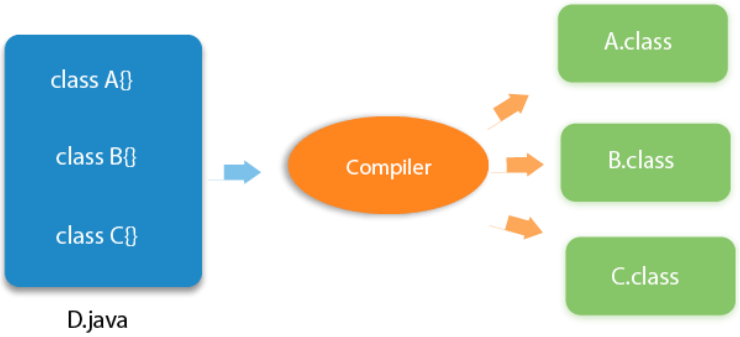
What happens at runtime? At runtime, the following steps are performed:

Classloader: It is the subsystem of JVM that is used to load class files.
Bytecode Verifier: Checks the code fragments for illegal code that can violate access rights to objects.
Interpreter: Read bytecode stream then execute the instructions.

Q) Can you save a Java source file by another name than the class name? Yes, if the class is not public. It is explained in the figure given below:



Q) Can you have multiple classes in a java source file? Yes, like the figure given below illustrates:

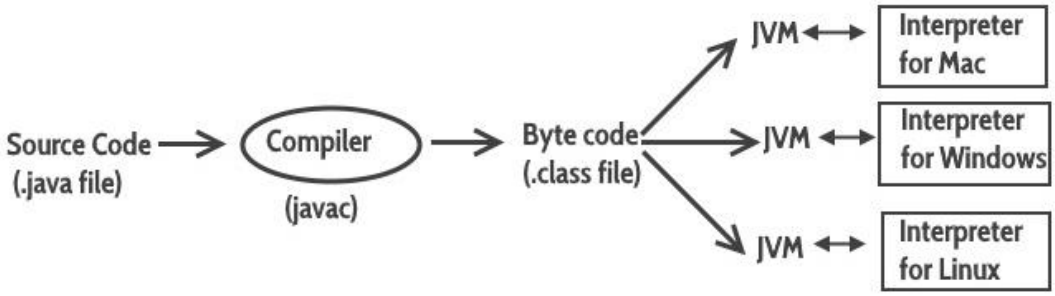


Difference between JDK, JRE, and JVM

Java is a high level programming language. A program written in high level language cannot be run on any machine directly. First, it needs to be translated into that particular machine language. The **javac compiler** does this thing, it takes java program (.java file containing source code) and translates it into machine code (referred as byte code or .class file).

Java Virtual Machine (JVM) is a virtual machine that resides in the real machine (your computer) and the **machine language for JVM is byte code**. This makes it easier for compiler as it has to generate byte code for JVM rather than different machine code for each type of machine. JVM executes the byte code generated by compiler and produce output. **JVM is the one that makes java platform independent**.

So, now we understood that the primary function of JVM is to execute the byte code produced by compiler. **Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.** Which means that the byte code generated on Windows can be run on Mac OS and vice versa. That is why we call java as platform independent language. The same thing can be seen in the diagram below:



JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

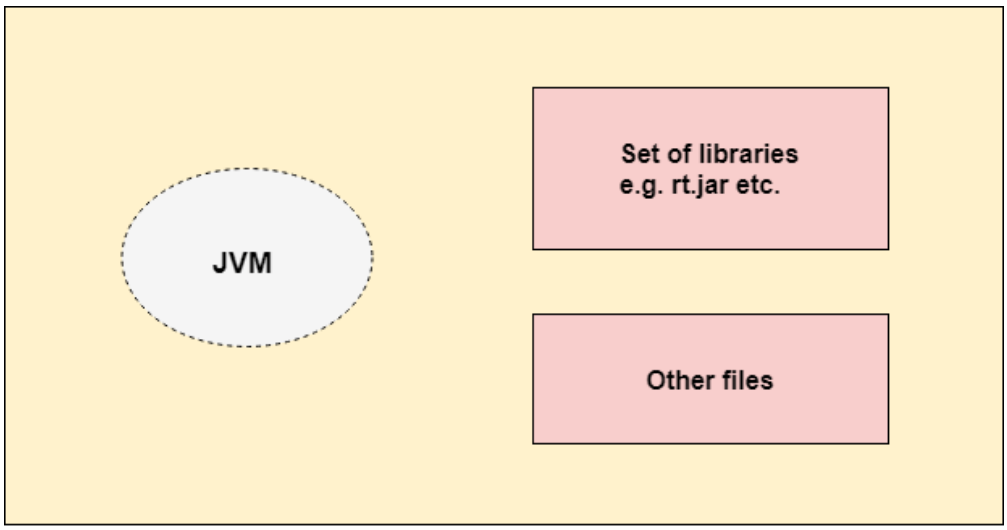
The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

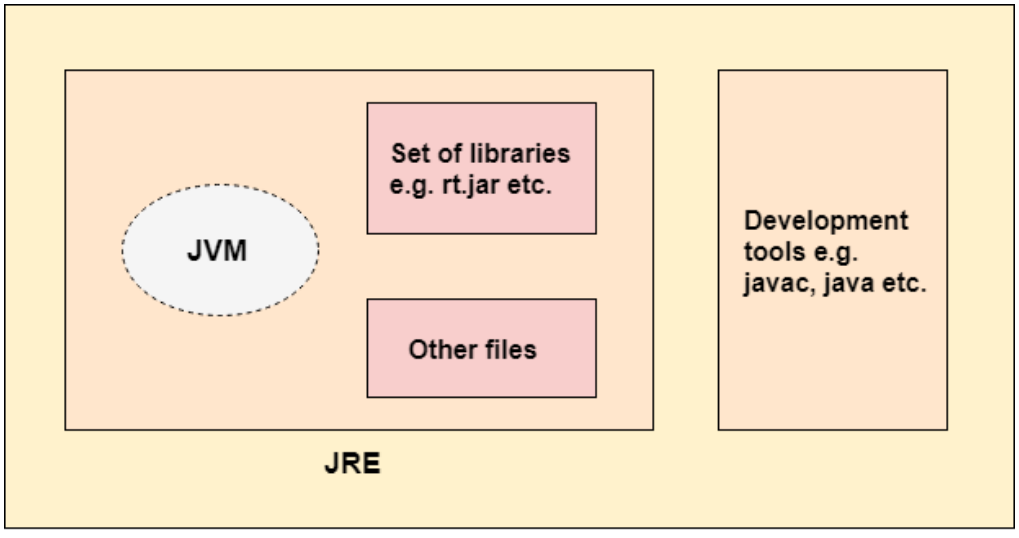


JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools. JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JVM (Java Virtual Machine) Architecture

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM? It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.

2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).

3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

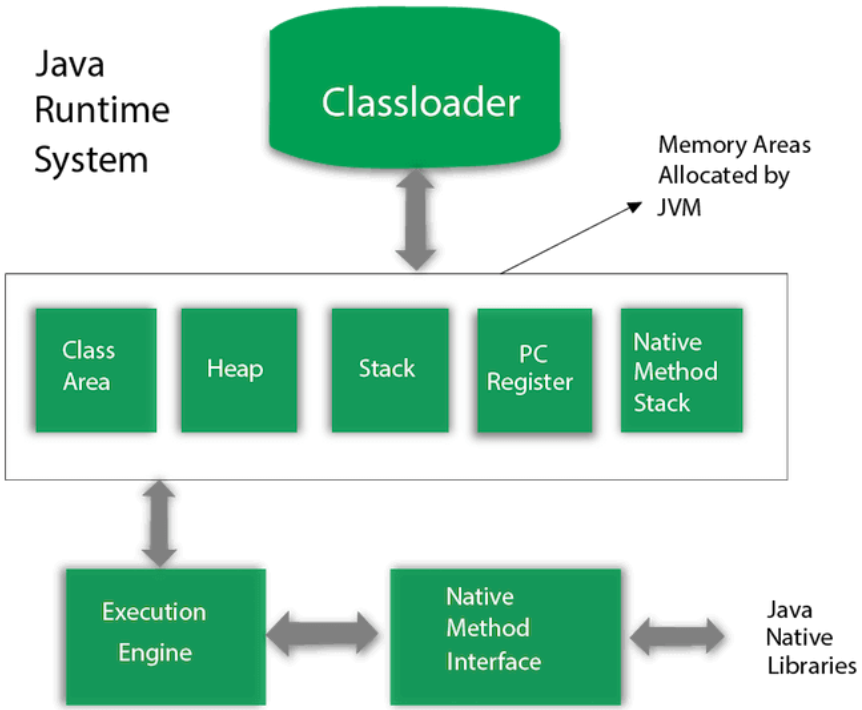
What it does - The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Architecture - Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader - Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.

2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.

3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

```
//Let's see an example to print the classloader name
public class ClassLoaderExample
{
    public static void main(String[] args)
    {
```

```
        // Let's print the classloader name of current class.
        //Application/System classloader will load this class
        Class c=ClassLoaderExample.class;
        System.out.println(c.getClassLoader());
        //If we print the classloader name of String, it will print null because it is an
        //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader
        System.out.println(String.class.getClassLoader());
    }
}
```

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

2) Class (Method) Area – Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap - It is the runtime data area in which objects are allocated.

4) Stack - Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register - PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack - It contains all the native methods used in the application.

7) Execution Engine - It contains:

- **A virtual processor**
- **Interpreter:** Read bytecode stream then execute the instructions.
- **Just-In-Time (JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8) Java Native Interface - Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

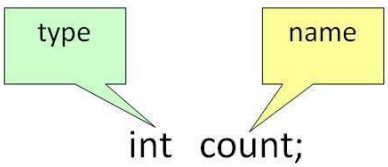
Java Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location. There are three types of variables in java: local, instance and static. There are two types of data types in Java: primitive and non-primitive.

Variable - A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

```
int data=50; // Here data is variable
```

How to declare variables? We can declare variables in java as pictorially depicted below as a visual aid. From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

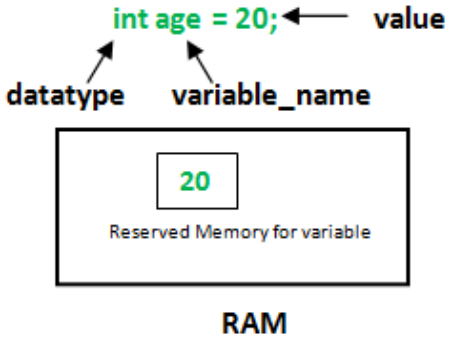


1. Datatype: Type of data that can be stored in this variable.

2. Dataname: Name was given to the variable.

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input



How to initialize variables? It can be perceived with the help of 3 components that are as follows:

- **datatype**: Type of data that can be stored in this variable.
- **variable_name**: Name given to the variable.
- **value**: It is the initial value stored in the variable.

Illustrations:

```
float simpleInterest;
// Declaring float variable
int time = 10, speed = 20;
// Declaring and Initializing integer variable
char var = 'h';
// Declaring and Initializing character variable
```

Types of Variables

1) Local Variable - A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

2) Instance Variable - A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static. It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable - A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
public class A
{
    static int m=100; //static variable
    void method()
    {
        int n=90; //local variable
    }
    public static void main(String args[])
    {
        int data=50; //instance variable
    }
} //end of class
```

Java Variable Example: Widening

```
public class Simple{
public static void main(String[] args){
  int a=10;
  float f=a;
  System.out.println(a); // 10
  System.out.println(f); // 10.0
}}
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{
public static void main(String[] args){
float f=10.5f;
```

```
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f); // 10.5
System.out.println(a); // 10
}}
```

Java Variable Example: Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);  // 130
System.out.println(b);  // -126
}}
```

Java Variable Example: Adding Lower Type

```
class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b;//Compile Time Error: because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);  // 20
}}
```

Differences between the Instance variable vs. the Static variables

- Each object will have its copy of the instance variable, whereas We can only have one copy of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of static, changes will be reflected in other objects as static variables are common to all objects of a class.
- We can access instance variables through object references, and Static Variables can be accessed directly using the class name**.**

The Scope and Lifetime of Variables - So far, all of the variables used have been declared at the start of the main( ) method. However, Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

```
public class Scope {
    public static void main(String args[]){
        int x; // known to all code within  main
        x = 10;
        if(x == 10){ // start new scope
            int y = 20; // known only to this block

            // x and y both are known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y not known  here
        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

Data Types in Java

There are majorly two types of languages.

- First, one is **Statically typed language** where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types. **Example:** C, C++, Java.

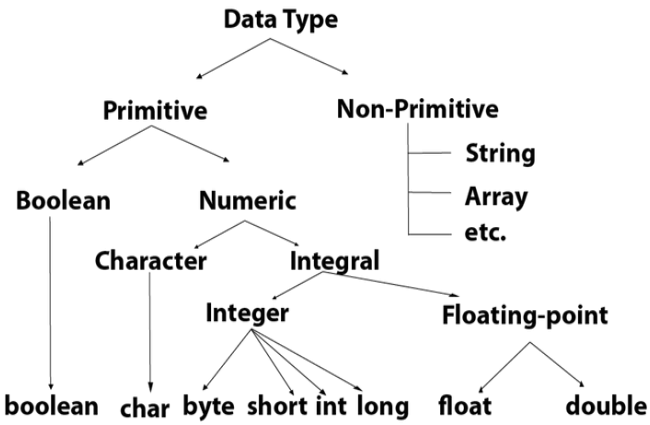
- The other is **Dynamically typed languages.** These languages can receive different data types over time. **Example:** Ruby, Python

Java is **statically typed and also a strongly typed language**.

Java is a Strongly Typed Language – It means: First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as is some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.



Java Primitive Data Types - In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name. There are 8 types of primitive data types:

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type - The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions. The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

```
Boolean one = false
```

Byte Data Type - The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

```
byte a = 10, byte b = -20

class Main {
  public static void main(String[] args) {
    byte range;
    range = 124;
    System.out.println(range);    // prints 124
  }
}

// Java program to demonstrate byte data type in Java
class GeeksforGeeks {
    public static void main(String args[])
    {
        byte a = 126;
        // byte is 8 bit value
        System.out.println(a);
        a++;
        System.out.println(a);

        // It overflows here because
        // byte can hold values from -128 to 127
        a++;
        System.out.println(a);
        // Looping back within the range
        a++;
        System.out.println(a);
    }
}
```

Output:

```
126
127
-128
-127
```

Short Data Type - The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

```
short s = 10000, short r = -5000

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

Int Data Type - The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0. The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

```
int a = 100000, int b = -200000

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

```
    int range = -4250000;
    System.out.println(range);  // print -4250000
  }
}
```

There are 4 types of integer literals in Java:

1. binary (base 2)
2. decimal (base 10)
3. octal (base 8)
4. hexadecimal (base 16)

```
// binary
int binaryNumber = 0b10010;
// octal
int octalNumber = 027;
// decimal
int decNumber = 34;
// hexadecimal
int hexNumber = 0x2F; // 0x represents hexadecimal
// binary
int binNumber = 0b10010; // 0b represents binary
```

In Java, binary starts with **0b**, octal starts with **0**, and hexadecimal starts with **0x**.

Note: Integer literals are used to initialize variables of integer types like byte, short, int, and long.

Long Data Type - The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is -9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

```
long a = 100000L, long b = -200000L

class LongExample {
  public static void main(String[] args) {
    long range = -42332200000L;
    System.out.println(range);    // prints -42332200000
  }
}
```

Float Data Type - The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

```
float f1 = 234.5f

class Main {
  public static void main(String[] args) {
    double myDouble = 3.4;
    float myFloat = 3.4F;
    // 3.445*10^2
    double myDoubleScientific = 3.445e2;
    System.out.println(myDouble);  // prints 3.4
    System.out.println(myFloat);    // prints 3.4
    System.out.println(myDoubleScientific);   // prints 344.5
  }
}
```

Double Data Type - The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

```
double d1 = 12.3

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

Char Data Type - The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

```
char letterA = 'A'

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

Why char uses 2 byte in java and what is \u0000 ? It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

```
class Main {
  public static void main(String[] args) {
    char letter1 = '9';
    System.out.println(letter1);  // prints 9
    char letter2 = 65;
    System.out.println(letter2);  // prints A

  }
}
```

Non-Primitive Data Type or Reference Data Types

The **Reference Data Types** will contain a memory address of variable value because the reference types won't store the variable value directly in memory. They are **strings**, **objects**, arrays, etc.

- **String**: Strings are defined as an array of characters. The difference between a character array and a string in Java is, the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char type entities.

- Unlike C/C++, Java strings are not terminated with a null character. Below is the basic syntax for declaring a string in Java programming language.

  Syntax: `<String_Type> <string_variable> = "<sequence_of_string>";`

Example:

```
// Declare String without using new operator
String s = "GeeksforGeeks";
// Declare String using new operator
String s1 = new String("GeeksforGeeks");
```

Class: A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

- **Modifiers**: A class can be public or has default access.

- **Class name:** The name should begin with a initial letter (capitalized by convention).

- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

- **Body:** The class body surrounded by braces, { }.

Object: It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Interface: Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, nobody).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.

- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.

- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Array: An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays.

- In Java, all arrays are dynamically allocated. (discussed below)
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using size.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each has an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions - When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types - For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form: `(target-type) value`

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// ...
b = (byte) a;
```

The following program demonstrate some type conversions that requires casts:

```
public class Conversion {
    public static void main(String args[]){
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);
        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d  and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Automatic Type Promotion in Expressions - In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a * b** is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable.

However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules - Java defines several type promotion rules that apply to expressions. They are as follows: First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double. The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
public static void main(String args[]) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}
```

Let's look closely at the type promotions that occur in this line from the program: `double result = (f * b) + (i / c) - (d * s);`

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System? Before Unicode, there were many language standards:

- ASCII (American Standard Code for Information Interchange) for the United States.
- ISO 8859-1 for Western European Language.
- KOI-8 for Russian.
- GB18030 and BIG-5 for Chinese, and so on.

Problem - A particular code value corresponds to different letters in the various language standards.

The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, other require two or more byte.

Solution - To solve these problems, a new language standard was developed i.e. Unicode System. In Unicode, character holds 2 byte, so java also uses 2 byte for characters.

```
lowest value:\u0000
highest value:\uFFFF
```

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc. There are many types of operators in Java which are given below:

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
	prefix	++expr --expr +expr -expr ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while * is also an operator used for multiplication.

Operators in Java can be classified into 5 types:

1. Java Arithmetic Operators - Arithmetic operators are used to perform arithmetic operations on variables and data. For example, a + b; Here, the + operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.

Operator	Operation
+	Addition
-	Subtraction
*****	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Example 1: Arithmetic Operators

```
class Main {
  public static void main(String[] args) {
    // declare variables
    int a = 12, b = 5;
    // addition operator
    System.out.println("a + b = " + (a + b)); // a + b = 17
    // subtraction operator
    System.out.println("a - b = " + (a - b)); // a - b = 7
    // multiplication operator
    System.out.println("a * b = " + (a * b)); // a * b = 60
    // division operator
    System.out.println("a / b = " + (a / b)); // a / b = 2
    // modulo operator
    System.out.println("a % b = " + (a % b)); // a % b = 2
  }
```

```
}
```

In the above example, we have used +, -, and * operators to compute addition, subtraction, and multiplication operations.

/ Division Operator - Note the operation, a / b in our program. The / operator is the division operator. If we use the division operator with two integers, then the resulting quotient will also be an integer. And, if one of the operands is a floating-point number, we will get the result will also be in floating-point.

In Java,

```
(9 / 2) is 4
(9.0 / 2) is 4.5
(9 / 2.0) is 4.5
(9.0 / 2.0) is 4.5
```

% Modulo Operator - The modulo operator % computes the remainder. When a = 7 is divided by b = 4, the remainder is **3**.

Note: The % operator is mainly used with integers.

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

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

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable age. Let's see some more assignment operators available in Java.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
***=**	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Example 2: Assignment Operators

```
class Main {
  public static void main(String[] args) {
    // create variables
    int a = 4;
    int var;
    // assign value using =
    var = a;
    System.out.println("var using =: " + var); // var using =: 4
    // assign value using =+
    var += a;
    System.out.println("var using +=: " + var); // var using +=: 8
    // assign value using =*
    var *= a;
    System.out.println("var using *=: " + var); // var using *=: 32
  }
}
```

3. Java Relational Operators - Relational operators are used to check the relationship between two operands. For example,

```
// check if a is less than b
a < b;
```

Here, < operator is the relational operator. It checks if a is less than b or not. It returns either true or false.

Operator	Description	Example

==	Is Equal To	3 == 5 returns false
!=	Not Equal To	3 != 5 returns true
>	Greater Than	3 > 5 returns false
<	Less Than	3 < 5 returns true
>=	Greater Than or Equal To	3 >= 5 returns false
<=	Less Than or Equal To	3 <= 5 returns true

Example 3: Relational Operators

```
class Main {
  public static void main(String[] args) {
    // create variables
    int a = 7, b = 11;
    // value of a and b
    System.out.println("a is " + a + " and b is " + b);
    // == operator
    System.out.println(a == b);  // false
    // != operator
    System.out.println(a != b);  // true
    // > operator
    System.out.println(a > b);  // false
    // < operator
    System.out.println(a < b);  // true
    // >= operator
    System.out.println(a >= b);  // false
    // <= operator
    System.out.println(a <= b);  // true
  }
}
```

Note: Relational operators are used in decision making and loops.

4. Java Logical Operators - Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	!expression	true if expression is false and vice versa

Example 4: Logical Operators

```
class Main {
  public static void main(String[] args) {
    // && operator
    System.out.println((5 > 3) && (8 > 5));  // true
    System.out.println((5 > 3) && (8 < 5));  // false
    // || operator
    System.out.println((5 < 3) || (8 > 5));  // true
    System.out.println((5 > 3) || (8 < 5));  // true
    System.out.println((5 < 3) || (8 < 5));  // false
    // ! operator
    System.out.println(!(5 == 3));  // true
    System.out.println(!(5 > 3));  // false
  }
}
```

Working of Program

- (5 > 3) && (8 > 5) returns true because both (5 > 3) and (8 > 5) are true.
- (5 > 3) && (8 < 5) returns false because the expression (8 < 5) is false.

- (5 < 3) || (8 > 5) returns true because the expression (8 > 5) is true.
- (5 > 3) && (8 > 5) returns true because the expression (5 > 3) is true.
- (5 > 3) && (8 > 5) returns false because both (5 < 3) and (8 < 5) are false.
- !(5 == 3) returns true because 5 == 3 is false.
- !(5 > 3) returns false because 5 > 3 is true.

Boolean Logical Operators - The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state: !true == false and !false == true. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Here is a program that is almost the same as the BitLogic example shown earlier, but it operates on boolean logical values instead of binary bits:

```
// Demonstrate the boolean logical operators.
class BoolLogic {
	public static void main(String args[]) {
		boolean a = true;
		boolean b = false;
		boolean c = a | b;
		boolean d = a & b;
		boolean e = a ^ b;
		boolean f = (!a & b) | (a & !b);
		boolean g = !a;
		System.out.println(" a = " + a);
		System.out.println(" b = " + b);
		System.out.println(" a|b = " + c);
		System.out.println(" a&b = " + d);
		System.out.println(" a^b = " + e);
		System.out.println("!a&b|a&!b = " + f);
		System.out.println(" !a = " + g);
	}
}
```

After running this program, you will see that the same logical rules apply to Boolean values as they did to bits. As you can see from the following output, the string representation of a Java boolean value is one of the literal values true or false:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

5. Java Unary Operators - Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**. Different types of unary operators are:

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

Increment and Decrement Operators - Java also provides increment and decrement operators: ++ and -- respectively. ++ increases the value of the operand by **1**, while -- decrease it by **1**. For example,

```
int num = 5;
// increase num by 1
++num;
```

Here, the value of num gets increased to **6** from its initial value of **5**.

```
class Main {
  public static void main(String[] args) {
    // declare variables
    int a = 12, b = 12;
    int result1, result2;
    // original value
    System.out.println("Value of a: " + a); // Value of a: 12
    // increment operator
    result1 = ++a;
    System.out.println("After increment: " + result1); // After increment: 13
    System.out.println("Value of b: " + b); // Value of b: 12
    // decrement operator
    result2 = --b;
    System.out.println("After decrement: " + result2); // After decrement: 11
  }
}
```

In the above program, we have used the ++ and -- operator as **prefixes (++a, --b)**. We can also use these operators as **postfix (a++, b++)**. There is a slight difference when these operators are used as prefix versus when they are used as a postfix.

6. Java Bitwise Operators - Bitwise operators in Java are used to perform operations on individual bits. For example,

```
Bitwise complement Operation of 35
35 = 00100011 (In Binary)
~ 00100011

  ________
   11011100  = 220 (In decimal)
```

Here, ~ is a bitwise operator. It inverts the value of each bit (**0** to **1** and **1** to **0**). The various bitwise operators present in Java are:

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

These operators are not generally used in Java. To learn more, visit Java Bitwise and Bit Shift Operators.

Other operators - Besides these operators, there are other additional operators in Java.

Java instanceof Operator - The instanceof operator checks whether an object is an instanceof a particular class. For example,

```
class Main {
  public static void main(String[] args) {
    String str = "Programiz";
    boolean result;

    // checks if str is an instance of
    // the String class
    result = str instanceof String;
    System.out.println("Is str an object of String? " + result); // Is str an object of String? true
  }
}
```

Here, str is an instance of the String class. Hence, the instanceof operator returns true. To learn more, visit Java instanceof.

Java Ternary Operator - The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

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

Here's how it works.

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

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

}
```

In the above example, we have used the ternary operator to check if the year is a leap year or not.

```
// Java program to illustrate
// max of three numbers using
// ternary operator.
public class operators {
    public static void main(String[] args)
    {
        int a = 20, b = 10, c = 30, result;
        // result holds max of three
        // numbers
        result = ((a > b)
                      ? (a > c)
                            ? a
```

```
                        : c
                    : (b > c)
                        ? b
                        : c);
        System.out.println("Max of three numbers = " + result);
    }
}
```

Java Basic Input and Output

In this tutorial, you will learn simple ways to display output to users and take input from users in Java.

Java Output - In Java, you can simply use

```
System.out.println(); or
System.out.print(); or
System.out.printf();
```

to send output to standard output (screen). Here,

- System is a class
- out is a public static field: it accepts output data.

Don't worry if you don't understand it. We will discuss class, public, and static in later chapters. Let's take an example to output a line.

```
class AssignmentOperator {
    public static void main(String[] args) {
        System.out.println("Java programming is interesting.");// Java programming is interesting.
    }
}
```

Here, we have used the println() method to display the string.

Difference between println(), print() and printf()

- print() - It prints string inside the quotes.
- println() - It prints string inside the quotes similar like print() method. Then the cursor moves to the beginning of the next line.
- printf() - It provides string formatting (similar to printf in C/C++ programming).

Example: print() and println()

```
class Output {
    public static void main(String[] args) {
        System.out.println("1. println ");
        System.out.println("2. println ");
        System.out.print("1. print ");
        System.out.print("2. print");
    }
}
```

Output:

```
1. println
2. println
1. print 2. print
```

In the above example, we have shown the working of the print() and println() methods. To learn about the printf() method, visit Java printf().

Example: Printing Variables and Literals

```
class Variables {
    public static void main(String[] args) {
        Double number = -10.6;
        System.out.println(5); // 5
        System.out.println(number); // -10.6
    }
}
```

Here, you can see that we have not used the quotation marks. It is because to display integers, variables and so on, we don't use quotation marks.

Example: Print Concatenated Strings

```
class PrintVariables {
    public static void main(String[] args) {
        Double number = -10.6;
        System.out.println("I am " + "awesome.");
        System.out.println("Number = " + number);
    }
}
```

Output:

```
I am awesome.
Number = -10.6
```

In the above example, notice the line, `System.out.println("I am " + "awesome.");`
Here, we have used the + operator to concatenate (join) the two strings: "I am " and "awesome.".
And also, the line, `System.out.println("Number = " + number);`
Here, first the value of variable number is evaluated. Then, the value is concatenated to the string: "Number = ".

Java Input - Java provides different ways to get input from the user. However, in this tutorial, you will learn to get input from user using the object of Scanner class. In order to use the object of Scanner, we need to import java.util.Scanner package.

```
import java.util.Scanner;
```

To learn more about importing packages in Java, visit Java Import Packages. Then, we need to create an object of the Scanner class. We can use the object to take input from the user.

```
// create an object of Scanner
Scanner input = new Scanner(System.in);
// take input from the user
int number = input.nextInt();
```

Example: Get Integer Input From the User

```
import java.util.Scanner;
class Input {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("You entered " + number);
        // closing the scanner object
        input.close();
    }
}
```

Output:

```
Enter an integer: 23
You entered 23
```

In the above example, we have created an object named input of the Scanner class. We then call the nextInt() method of the Scanner class to get an integer input from the user.

Similarly, we can use nextLong(), nextFloat(), nextDouble(), and next() methods to get long, float, double, and string input respectively from the user.

Note: We have used the close() method to close the object. It is recommended to close the scanner object once the input is taken.

Example: Get float, double and String Input

```
import java.util.Scanner;
class Input {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        // Getting float input
        System.out.print("Enter float: ");
        float myFloat = input.nextFloat();
        System.out.println("Float entered = " + myFloat);

        // Getting double input
        System.out.print("Enter double: ");
        double myDouble = input.nextDouble();
        System.out.println("Double entered = " + myDouble);

        // Getting String input
        System.out.print("Enter text: ");
        String myString = input.next();
        System.out.println("Text entered = " + myString);
    }
}
```

Output:

```
Enter float: 2.343
Float entered = 2.343
Enter double: -23.4
Double entered = -23.4
Enter text: Hey!
Text entered = Hey!
```

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords

1. **abstract:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case:** Java case keyword is used with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.

11. **do:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if:** Java if keyword tests the condition. It executes the if block if the condition is true.
21. **implements:** Java implements keyword is used to implement an interface.
22. **import:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

Java Expressions, Statements and Blocks

Java Expressions - A Java expression consists of variables, operators, literals, and method calls. For example,

```
int score;
score = 90;
```

Here, score = 90 is an expression that returns an int. Consider another example,

```
Double a = 2.2, b = 3.4, result;
result = a + b - 3.4;
```

Here, a + b - 3.4 is an expression.

```
if (number1 == number2)
    System.out.println("Number 1 is larger than number 2");
```

Here, number1 == number2 is an expression that returns a boolean value. Similarly, "Number 1 is larger than number 2" is a string expression.

Java Statements - In Java, each statement is a complete unit of execution. For example,

int `score = 9*5;`

Here, we have a statement. The complete execution of this statement involves multiplying integers 9 and 5 and then assigning the result to the variable score. In the above statement, we have an expression 9 * 5. In Java, expressions are part of statements.

Expression statements - We can convert an expression into a statement by terminating the expression with a ;. These are known as expression statements. For example,

```
// expression
number = 10
// statement
number = 10;
```

In the above example, we have an expression number = 10. Here, by adding a semicolon (;), we have converted the expression into a statement (number = 10;). Consider another example,

```
// expression
++number
// statement
++number;
```

Similarly, ++number is an expression whereas ++number; is a statement.

Declaration Statements - In Java, declaration statements are used for declaring variables. For example,

```
Double tax = 9.5;
```

The statement above declares a variable tax which is initialized to 9.5.

Note: There are control flow statements that are used in decision making and looping in Java. You will learn about control flow statements in later chapters.

Java Blocks - A block is a group of statements (zero or more) that is enclosed in curly braces { }. For example,

```
class Main {
    public static void main(String[] args) {
        String band = "Beatles";
        if (band == "Beatles") { // start of block
            System.out.print("Hey ");
            System.out.print("Jude!");
        } // end of block
    }
}
```

Output: Hey Jude!

In the above example, we have a block if {....}. Here, inside the block we have two statements:

```
System.out.print("Hey ");
System.out.print("Jude!");
```

However, a block may not have any statements. Consider the following examples,

```
class Main {
    public static void main(String[] args) {
        if (10 > 5) { // start of block
        } // end of block
    }
}
```

This is a valid Java program. Here, we have a block if {...}. However, there is no any statement inside this block.

```
class AssignmentOperator {
    public static void main(String[] args) {  // start of block
    } // end of block
}
```

Here, we have block public static void main() {...}. However, similar to the above example, this block does not have any statement.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program. Java provides three types of control flow statements.

1. **Decision Making statements** - if statements, switch statement
2. **Loop statements -** do while loop**,** while loop**,** for loop**,** for-each loop
3. **Jump statements -** break statement**,** continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1) Simple if statement: It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
```

```
if(x+y > 20) {
System.out.println("x + y is greater than 20");// x + y is greater than 20
}
}
}
```

2) if-else statement - The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output: x + y is greater than 20

3) if-else-if ladder: The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output: Delhi

4. Nested if-statement - In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output: Delhi

Example: Nested if...else Statement

```
class Main {
  public static void main(String[] args) {
    // declaring double type variables
    Double n1 = -1.0, n2 = 4.5, n3 = -5.3, largest;

    // checks if n1 is greater than or equal to n2
    if (n1 >= n2) {
      // if...else statement inside the if block
      // checks if n1 is greater than or equal to n3
      if (n1 >= n3) {
        largest = n1;
      }
      else {
        largest = n3;
      }
    } else {
      // if..else statement inside else block
      // checks if n2 is greater than or equal to n3
      if (n2 >= n3) {
        largest = n2;
      }
      else {
        largest = n3;
      }
    }

    System.out.println("Largest Number: " + largest);
  }
```

```
}
```

Output: Largest Number: 4.5

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

```
switch (expression){
    case value1:
     statement1;
     break;
    .
    .
    .
    case valueN:
     statementN;
     break;
    default:
     default statement;
}
```

Consider the following example to understand the flow of the switch statement.

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output: 2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used. Here is a simple example that uses a switch statement:

```
public class SampleSwitch {
    public static void main(String args[]){
        for(int i = 0; i < 6; i++){
            switch(i){
                case 0:
```

```
                System.out.println("i is zero. ");
                break;
                case 1:
                System.out.println("i is one.");
                break;
                case 2:
                System.out.println("i is two.");
                break;
                case 3:
                System.out.println("i is three.");
                break;
                default:
                System.out.println("i is greater than 3.");
            }
        }
    }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

As you can see, each time through the loop, the statements associated with the case constant that matches i are executed. All others are bypassed. After i is greater than 3, no case statements match, so the default statement is executed.

The break statement is optional. If you omit the break, execution will continue on into the next case. It is sometimes desirable to have multiple cases without break statements between them. For example, consider the following program:

```
public class MissingBreak {
    public static void main(String args[]){
        for(int i = 0; i < 12; i++){
            switch(i){
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i is less than 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                    System.out.println("i is less than 10");;
                    break;
                default:
                    System.out.println("i is 10 or more.");
            }
        }
    }
}
```

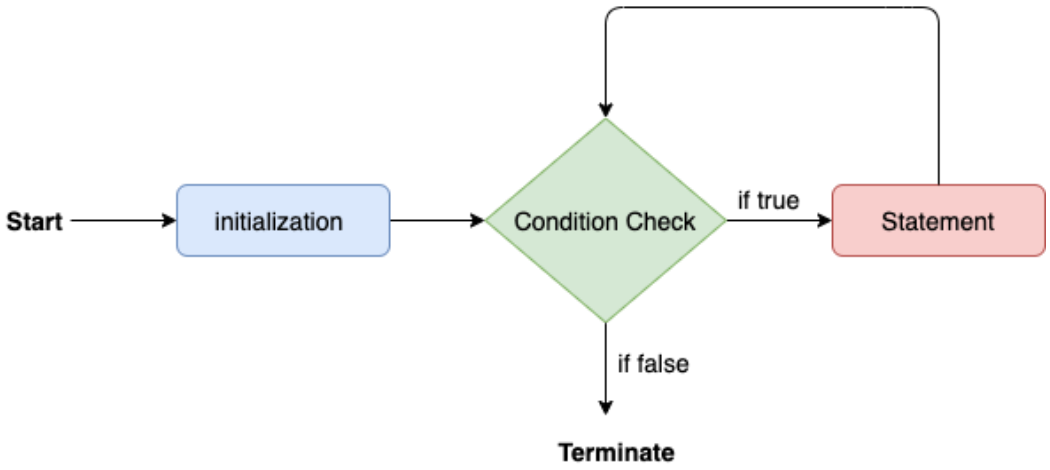
Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition. In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```



Consider the following example to understand the proper functioning of the for loop in java.

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);  // The sum of first 10 natural
numbers is 55
}
}
```

Test for primes

```
public class FindPrime {
    public static void main(String args[]){
        int num;
        boolean isPrime = true;

        num = 14;
        for(int i = 2; i <= num/2; i++){
            if((num % i) == 0){
                isPrime = false;
                break;
            }
        }
        if(isPrime) System.out.println("Prime");
        else System.out.println("Not Prime");
    }
}
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

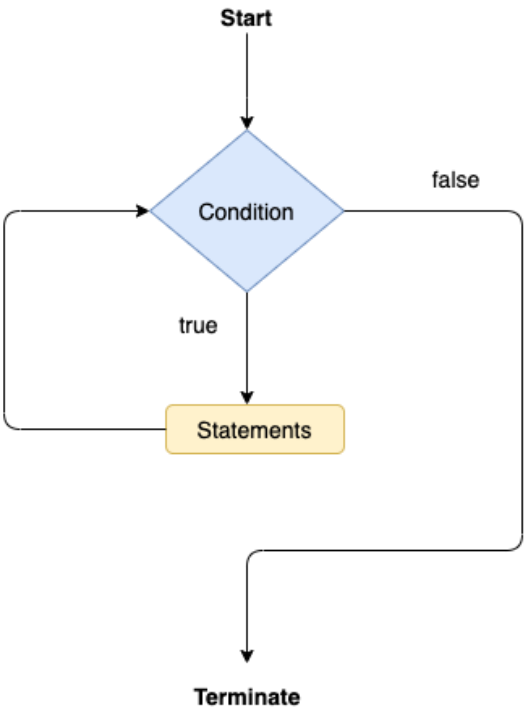
```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Use for-each style for on a two-dimensional array

```
public class ForEach {
    public static void main(String args[]){
        int sum = 0;
        int nums[][] = new int[3][5];
        // give nums some values
        for(int i = 0; i < 3; i++){
            for(int j = 0; j < 5; j++){
                nums[i][j] = (i + 1) * (j + 1);
            }
        }

        // use for-each for to display and sum the values
        for(int x[] : nums){
            for(int y : x){
                System.out.println("Value is: " + y);
                sum += y;
            }
        }
        System.out.println("Summation: " + sum);
    }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```



Java while loop

The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

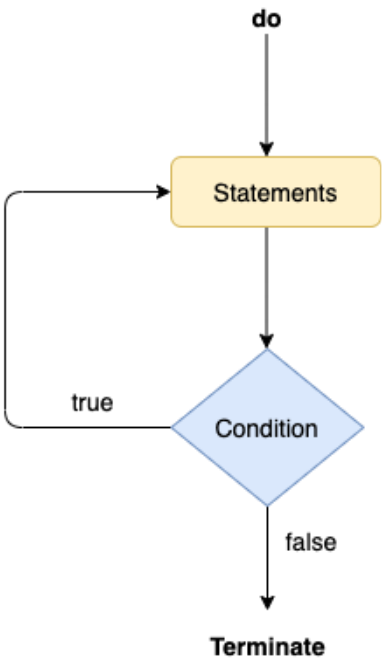
```
while(condition){
//looping statements
}
```

Consider the following example.

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop. It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.




```
do
{
//statements
} while (condition);
```

Consider the following example to understand the functioning of the do-while loop in Java.

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop. The

break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

```
public class BreakExample {
public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
for(int j = 0; j<=15;j++) {
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}
}
}
}
}
}
```

Output:

```
0
1
2
3
4
5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately. Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {
public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 2; i++) {
for (int j = i; j<=5; j++) {
if(j == 4) {
```

```
continue;
}
System.out.println(j);
}
}
}

}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

```
if(condition){
//code to be executed
}
```

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Output: Age is greater than 18

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
```

```
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Output: odd number

Leap Year Example: A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
public class LeapYearExample {
public static void main(String[] args) {
    int year=2020;
    if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
        System.out.println("LEAP YEAR");
    }
    else{
        System.out.println("COMMON YEAR");
    }
}
}
```

Output: LEAP YEAR

Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

```
public class IfElseTernaryExample {
public static void main(String[] args) {
    int number=13;
    //Using ternary operator
    String output=(number%2==0)?"even number":"odd number";
    System.out.println(output); // odd number
}
}
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}

//Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
public class IfElseIfExample {
public static void main(String[] args) {
```

```
    int marks=65;
    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Output: C grade

Program to check POSITIVE, NEGATIVE or ZERO:

```
public class PositiveNegativeExample {
public static void main(String[] args) {
    int number=-13;
    if(number>0){
    System.out.println("POSITIVE");
    }else if(number<0){
    System.out.println("NEGATIVE");
    }else{
    System.out.println("ZERO");
   }
}
}
```

Output: NEGATIVE

Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

```
if(condition){
     //code to be executed
          if(condition){
             //code to be executed
    }
}
```

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        }
```

```
    }
}}
```

Output: You are eligible to donate blood

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample2 {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=25;
    int weight=48;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        } else{
            System.out.println("You are not eligible to donate blood");
        }
    } else{
      System.out.println("Age must be greater than 18");
    }
}  }
```

Output: You are not eligible to donate blood

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement. In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of byte, short, int, long (with its Wrapper type), enums and string.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......
default:
  code to be executed if all cases are not matched;
}
```

Example:

```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
```

```
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Output: 20

```
//Java Program to demonstrate the example of Switch statement
//where we are printing month name for the given number
public class SwitchMonthExample {
public static void main(String[] args) {
    //Specifying month number
    int month=7;
    String monthString="";
    //Switch statement
    switch(month){
    //case statements within the switch block
    case 1: monthString="1 - January";
    break;
    case 2: monthString="2 - February";
    break;
    case 3: monthString="3 - March";
    break;
    case 4: monthString="4 - April";
    break;
    case 5: monthString="5 - May";
    break;
    case 6: monthString="6 - June";
    break;
    case 7: monthString="7 - July";
    break;
    case 8: monthString="8 - August";
    break;
    case 9: monthString="9 - September";
    break;
    case 10: monthString="10 - October";
    break;
    case 11: monthString="11 - November";
    break;
    case 12: monthString="12 - December";
    break;
    default:System.out.println("Invalid Month!");
    }
    //Printing month of the given number
    System.out.println(monthString);
}
}
```

Output: 7 - July

Program to check Vowel or Consonant: - If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

```
public class SwitchVowelExample {
public static void main(String[] args) {
    char ch='O';
    switch(ch)
    {
```

```
        case 'a':
            System.out.println("Vowel");
            break;
        case 'e':
            System.out.println("Vowel");
            break;
        case 'i':
            System.out.println("Vowel");
            break;
        case 'o':
            System.out.println("Vowel");
            break;
        case 'u':
            System.out.println("Vowel");
            break;
        case 'A':
            System.out.println("Vowel");
            break;
        case 'E':
            System.out.println("Vowel");
            break;
        case 'I':
            System.out.println("Vowel");
            break;
        case 'O':
            System.out.println("Vowel");
            break;
        case 'U':
            System.out.println("Vowel");
            break;
        default:
            System.out.println("Consonant");
    }
}
}
```

Output: Vowel

Java Switch Statement is fall-through - The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

```
//Java Switch Example where we are omitting the
//break statement
public class SwitchExample2 {
public static void main(String[] args) {
    int number=20;
    //switch expression with int value
    switch(number){
    //switch cases without break statements
    case 10: System.out.println("10");
    case 20: System.out.println("20");
    case 30: System.out.println("30");
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Output:

```
20
30
Not in 10, 20 or 30
```

Java Switch Statement with String - Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

```
//Java Program to demonstrate the use of Java Switch
//statement with String
public class SwitchStringExample {
public static void main(String[] args) {
    //Declaring String variable
    String levelString="Expert";
    int level=0;
    //Using String in Switch expression
    switch(levelString){
    //Using String Literal in Switch case
    case "Beginner": level=1;
    break;
    case "Intermediate": level=2;
    break;
    case "Expert": level=3;
    break;
    default: level=0;
    break;
    }
    System.out.println("Your Level is: "+level);
}
}
```

Output: Your Level is: 3

Java Nested Switch Statement - We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

```
//Java Program to demonstrate the use of Java Nested Switch
public class NestedSwitchExample {
    public static void main(String args[])
      {
        // C - CSE, E - ECE, M - Mechanical
        char branch = 'C';
        int collegeYear = 4;
        switch( collegeYear )
        {
            case 1:
                System.out.println("English, Maths, Science");
                break;
            case 2:
                switch( branch )
                {
                    case 'C':
                        System.out.println("Operating System, Java, Data Structure");
                        break;
                    case 'E':
                        System.out.println("Micro processors, Logic switching theory");
                        break;
                    case 'M':
                        System.out.println("Drawing, Manufacturing Machines");
                        break;
                }
                break;
            case 3:
                switch( branch )
                {
                    case 'C':
                        System.out.println("Computer Organization, MultiMedia");
                        break;
                    case 'E':
                        System.out.println("Fundamentals of Logic Design, Microelectronics");
                        break;
                    case 'M':
                        System.out.println("Internal Combustion Engines, Mechanical Vibration");
```

```
                        break;
                }
                break;
            case 4:
                switch( branch )
                {
                    case 'C':
                        System.out.println("Data Communication and Networks, MultiMedia");
                        break;
                    case 'E':
                        System.out.println("Embedded System, Image Processing");
                        break;
                    case 'M':
                        System.out.println("Production Technology, Thermal Engineering");
                        break;
                }
                break;
        }
    }
}
```

Output: Data Communication and Networks, MultiMedia

Java Enum in Switch Statement - Java allows us to use enum in switch statement. Java enum is a class that represents the group of constants. (immutable such as final variables). We use the keyword enum and put the constants in curly braces separated by comma.

```
//Java Program to demonstrate the use of Enum
//in switch statement
public class JavaSwitchEnumExample {
       public enum Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }
       public static void main(String args[])
       {
         Day[] DayNow = Day.values();
           for (Day Now : DayNow)
           {
                switch (Now)
                {
                    case Sun:
                        System.out.println("Sunday");
                        break;
                    case Mon:
                        System.out.println("Monday");
                        break;
                    case Tue:
                        System.out.println("Tuesday");
                        break;
                    case Wed:
                        System.out.println("Wednesday");
                        break;
                    case Thu:
                        System.out.println("Thursday");
                        break;
                    case Fri:
                        System.out.println("Friday");
                        break;
                    case Sat:
                        System.out.println("Saturday");
                        break;
                }
            }
       }
}
```

Output:

```
Sunday
Monday
Twesday
Wednesday
Thursday
Friday
Saturday
```

Java Wrapper in Switch Statement - Java allows us to use four wrapper classes: Byte, Short, Integer and Long in switch statement.

```
//Java Program to demonstrate the use of Wrapper class
//in switch statement
public class WrapperInSwitchCaseExample {
       public static void main(String args[])
       {
            Integer age = 18;
            switch (age)
            {
                case (16):
                    System.out.println("You are under 18.");
                    break;
                case (18):
                    System.out.println("You are eligible for vote.");
                    break;
                case (65):
                    System.out.println("You are senior citizen.");
                    break;
                default:
                    System.out.println("Please give the valid age.");
                    break;
            }
        }
}
```

Output: You are eligible for vote.

Loops in Java

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

Java Simple for Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return Boolean value either true or false. It is an optional condition.

3. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

4. **Statement**: The statement of the loop is executed each time until the second condition is false.

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
}
```

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
```

```
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

```
public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
        System.out.println(i+" "+j);
}//end of i
}//end of j
}
}
```

```
public class PyramidExample {
public static void main(String[] args) {
for(int i=1;i<=5;i++){
for(int j=1;j<=i;j++){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

```
public class PyramidExample2 {
public static void main(String[] args) {
int term=6;
for(int i=1;i<=term;i++){
for(int j=term;j>=i;j--){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```

Output:

```
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

```
for(data_type variable : array_name){
//code to be executed
}
```

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
    //Declaring an array
    int arr[]={12,23,44,56,78};
    //Printing array using for-each loop
    for(int i:arr){
        System.out.println(i);
    }
}
}
```

Output:

```
12
23
44
56
78
```

```
// Calculate the sum of all elements of an array
class Main {
 public static void main(String[] args) {
   // an array of numbers
   int[] numbers = {3, 4, 5, -5, 0, 12};
   int sum = 0;
   // iterating through each element of the array
   for (int number: numbers) {
     sum += number;
   }
   System.out.println("Sum = " + sum);
 }
}
```

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop. Note: The break and continue keywords breaks or continues the innermost for loop respectively.

```
labelname:
for(initialization; condition; increment/decrement){
//code to be executed
}
```

Example:

```
class Main {
 public static void main(String[] args) {
   char[] vowels = {'a', 'e', 'i', 'o', 'u'};
   // iterating through an array using a for loop
   for (int i = 0; i < vowels.length; ++ i) {
     System.out.println(vowels[i]);
```

```
   }
 }
}

//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
    //Using Label for outer and for loop
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break aa;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
```

If you use **break bb;**, it will break inner loop only which is the default behaviour of any loop.

```
public class LabeledForExample2 {
public static void main(String[] args) {
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break bb;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Java Infinitive for Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

```
for(;;){
//code to be executed
}

//Java program to demonstrate the use of infinite for loop
//which prints an statement
```

```
public class ForExample {
public static void main(String[] args) {
    //Using no condition in for loop
    for(;;){
        System.out.println("infinitive loop");
    }
}
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c
```

Now, you need to press ctrl+c to exit from the program.

Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax	`for(init;condition;incr/decr){`
	`// code to be executed`
	`}`
		`while(condition){`
		`//code to be executed`
		`}`
			`do{`
			`//code to be executed`
			`}while(condition);`

Example	`//for loop`
	`for(int i=1;i<=10;i++){`
	`System.out.println(i);`
	`}`
		`//while loop`
		`int i=1;`
		`while(i<=10){`
		`System.out.println(i);`
		`i++;`
		`}`
			`//do-while loop`
			`int i=1;`
			`do{`
			`System.out.println(i);`
			`i++;`
			`}while(i<=10);`

Syntax for infinitive loop	`for(;;){`
	`//code to be executed`
	`}`
		`while(true){`
		`//code to be executed`
		`}`
			`do{`
			`//code to be executed`
			`}while(true);`

Java While Loop

The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops. The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

```
while (condition){
//code to be executed
Increment / decrement statement
}
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop. Example: `i <=100`

2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable. Example: `i++;`

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.

Example: In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

```
public class WhileExample {
public static void main(String[] args) {
    int i=1;
    while(i<=10){
        System.out.println(i);
    i++;
    }
}
}
```

Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

```
while(true){
//code to be executed
}
```

```
public class WhileExample2 {
public static void main(String[] args) {
 // setting the infinite while loop by passing true to the condition
    while(true){
        System.out.println("infinitive while loop");
    }
}
}
```

Output:

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c
```

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

```
// Java program to find the sum of positive numbers
import java.util.Scanner;
class Main {
  public static void main(String[] args) {
    int sum = 0;

    // create an object of Scanner class
    Scanner input = new Scanner(System.in);

    // take integer input from the user
    System.out.println("Enter a number");
    int number = input.nextInt();

    // while loop continues
    // until entered number is positive
    while (number >= 0) {
      // add only positive numbers
```

```
      sum += number;
      System.out.println("Enter a number");
      number = input.nextInt();
    }
    System.out.println("Sum = " + sum);
    input.close();
  }
}
```

Output

```
Enter a number 25
Enter a number 9
Enter a number 5
Enter a number -3
Sum = 39
```

Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop. Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically. i <=100

2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable. i++;

Note: The do block is executed at least once, even if the condition is false.

Example: In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Java Infinitive do-while Loop - If you pass **true** in the do-while loop, it will be infinitive do-while loop.

```
do{
//code to be executed
}while(true);
```

```
public class DoWhileExample2 {
public static void main(String[] args) {
    do{
        System.out.println("infinitive do while loop");
    }while(true);
}
}
```

Output:

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
ctrl+c
```

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

```
// Java program to find the sum of positive numbers
import java.util.Scanner;
class Main {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;
    // create an object of Scanner class
    Scanner input = new Scanner(System.in);

    // do...while loop continues
    // until entered number is positive
    do {
      // add only positive numbers
      sum += number;
      System.out.println("Enter a number");
      number = input.nextInt();
    } while(number >= 0);

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

Output 1

```
Enter a number 25
Enter a number 9
Enter a number 5
Enter a number -3
Sum = 39
```

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop. We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

```
jump-statement;
break;
```

```
//Java Program to demonstrate the use of break statement
//inside the for loop.
public class BreakExample {
public static void main(String[] args) {
    //using for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //breaking the loop
            break;
        }
        System.out.println(i);
    }
```

```
}
}
```

Output:

```
1
2
3
4
```

Java Break Statement with Inner Loop - It breaks inner loop only if you use break statement inside the inner loop.

```
//Java Program to illustrate the use of break statement
//inside an inner loop
public class BreakExample2 {
public static void main(String[] args) {
            //outer loop
            for(int i=1;i<=3;i++){
                    //inner loop
                    for(int j=1;j<=3;j++){
                        if(i==2&&j==2){
                            //using break statement inside the inner loop
                            break;
                        }
                        System.out.println(i+" "+j);
                    }
            }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Java Break Statement with Labeled For Loop - We can use break statement with a label. The feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer or inner loop.

```
//Java Program to illustrate the use of continue statement
//with label inside an inner loop to break outer loop
public class BreakExample3 {
public static void main(String[] args) {
            aa:
            for(int i=1;i<=3;i++){
                    bb:
                    for(int j=1;j<=3;j++){
                        if(i==2&&j==2){
                            //using break statement with label
                            break aa;
                        }
                        System.out.println(i+" "+j);
                    }
            }
}
}
```

Output:

```
1 1
1 2
```

```
1 3
2 1
```

```
//Java Program to demonstrate the use of break statement
//inside the while loop.
public class BreakWhileExample {
public static void main(String[] args) {
    //while loop
    int i=1;
    while(i<=10){
        if(i==5){
            //using break statement
            i++;
            break;//it will break the loop
        }
        System.out.println(i);
        i++;
    }
}
}
```

Output:

```
1
2
3
4
```

```
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
           //using break statement
           i++;
           break;//it will break the loop
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Output:

```
1
2
3
4
```

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, we have used the Scanner object.

```
import java.util.Scanner;
class UserInputSum {
    public static void main(String[] args) {
        Double number, sum = 0.0;
        // create an object of Scanner
        Scanner input = new Scanner(System.in);

        while (true) {
```

```
            System.out.print("Enter a number: ");
            // takes double input from user
            number = input.nextDouble();
            // if number is negative the loop terminates
            if (number < 0.0) {
                break;
            }
           sum += number;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Output:

```
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
```

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop. The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only. We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

```
jump-statement;
continue;
```

```
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
    //for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //using continue statement
            continue;//it will skip the rest statement
        }
        System.out.println(i);
    }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

Java Continue Statement with Inner Loop - It continues inner loop only if you use the continue statement inside the inner loop.

```
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
public static void main(String[] args) {
            //outer loop
            for(int i=1;i<=3;i++){
                    //inner loop
                    for(int j=1;j<=3;j++){
                        if(i==2&&j==2){
                            //using continue statement inside inner loop
                            continue;
                        }
                        System.out.println(i+" "+j);
                    }
            }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

Java Continue Statement with Labelled For Loop - We can use continue statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

```
//Java Program to illustrate the use of continue statement
//with label inside an inner loop to continue outer loop
public class ContinueExample3 {
public static void main(String[] args) {
            aa:
            for(int i=1;i<=3;i++){
                    bb:
                    for(int j=1;j<=3;j++){
                        if(i==2&&j==2){
                            //using continue statement with label
                            continue aa;
                        }
                        System.out.println(i+" "+j);
                    }
            }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Java Continue Statement in while loop

```
//Java Program to demonstrate the use of continue statement
//inside the while loop.
public class ContinueWhileExample {
```

```
public static void main(String[] args) {
    //while loop
    int i=1;
    while(i<=10){
        if(i==5){
            //using continue statement
            i++;
            continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Java Continue Statement in do-while Loop

```
//Java Program to demonstrate the use of continue statement
//inside the Java do-while loop.
public class ContinueDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
                //using continue statement
                 i++;
            continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Example: Compute the sum of 5 positive numbers

```
import java.util.Scanner;
class Main {
  public static void main(String[] args) {
```

```
    Double number, sum = 0.0;
    // create an object of Scanner
    Scanner input = new Scanner(System.in);

    for (int i = 1; i < 6; ++i) {
      System.out.print("Enter number " + i + " : ");
      // takes input from the user
      number = input.nextDouble();
      // if number is negative
      // continue statement is executed
      if (number <= 0.0) {
        continue;
      }
      sum += number;
    }
    System.out.println("Sum = " + sum);
    input.close();
  }
}
```

Output:

```
Enter number 1: 2.2
Enter number 2: 5.6
Enter number 3: 0
Enter number 4: -2.4
Enter number 5: -3
Sum = 7.8
```

Java Comments

The Java comments are the statements in a program that are not executed by the compiler and interpreter.

Why do we use comments in a code?

- Comments are used to make the program more readable by adding the details of the code.
- It makes easy to maintain the code and to find the errors easily.
- The comments can be used to provide information or explanation about the variable, method, class, or any statement.
- It can also be used to prevent the execution of program code while testing the alternative code.

Types of Java Comments

1) Java Single Line Comment - The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements. Single line comments starts with two forward slashes **(//)**. Any text in front of // is not executed by Java.

```
//This is single line comment
```

2) Java Multi Line Comment - The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there). Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

```
/*
This
is
multi line
comment
*/
```

Note: Usually // is used for short comments and /* */ is used for longer comments.

3) Java Documentation Comment - Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods,

arguments, etc., are used in the code. To create documentation API, we need to use the **javadoc tool**. The documentation comments are placed between /** and */.

```
/**
*
*We can use various tags to depict the parameter
*or heading or author name
*We can also use HTML tags
*
*/
```

javadoc tags

Some of the commonly used tags in documentation comments:

Tag	Syntax	Description
{@docRoot}	{@docRoot}	to depict relative path to root directory of generated document from any page.
@author	@author name - text	To add the author of the class.
@code	{@code text}	To show the text in code font without interpreting it as html markup or nested javadoc tag.
@version	@version version-text	To specify "Version" subheading and version-text when -version option is used.
@since	@since release	To add "Since" heading with since text to generated documentation.
@param	@param parameter-name description	To add a parameter with given name and description to 'Parameters' section.
@return	@return description	Required for every method that returns something (except void)

Let's use the Javadoc tag in a Java program.

Calculate.java

```
import java.io.*;

/**
 * <h2> Calculation of numbers </h2>
 * This program implements an application
 * to perform operation such as addition of numbers
 * and print the result
 * <p>
 * <b>Note:</b> Comments make the code readable and
 * easy to understand.
 *
 * @author Anurati
 * @version 16.0
 * @since 2021-07-06
 */

 public class Calculate{
    /**
     * This method calculates the summation of two integers.
     * @param input1 This is the first parameter to sum() method
     * @param input2 This is the second parameter to the sum() method.
     * @return int This returns the addition of input1 and input2
     */
    public int sum(int input1, int input2){
        return input1 + input2;
    }
    /**
    * This is the main method uses of sum() method.
    * @param args Unused
```

```
    * @see IOException
    */
    public static void main(String[] args) {
        Calculate obj = new Calculate();
        int result = obj.sum(40, 20);
        System.out.println("Addition of numbers: " + result);
    }
 }
```

Compile it by javac tool:

Create Document

```
C:\Users\Anurati\Desktop\abcDemo>javac Calculate.java

C:\Users\Anurati\Desktop\abcDemo>java Calculate
Addition of numbers: 60
```

Create documentation API by **javadoc** tool:

```
C:\Users\Anurati\Desktop\abcDemo>javadoc Calculate.java
Loading source file Calculate.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_161
Building tree for all the packages and classes...
Generating .\Calculate.html...
Generating .\package-frame.html...
Generating .\package-summary.html...
Generating .\package-tree.html...
Generating .\constant-values.html...
Building index for all the packages and classes...
Generating .\overview-tree.html...
Generating .\index-all.html...
Generating .\deprecated-list.html...
Building index for all classes...
Generating .\allclasses-frame.html...
Generating .\allclasses-noframe.html...
Generating .\index.html...
Generating .\help-doc.html...
```

Now, the HTML files are created for the **Calculate** class in the current directory, i.e., **abcDemo**. Open the HTML files, and we can see the explanation of Calculate class provided through the documentation comment.

Are Java comments executable? As we know, Java comments are not executed by the compiler or interpreter, however, before the lexical transformation of code in compiler, contents of the code are encoded into ASCII in order to make the processing easy.

Test.java

```
public class Test{
    public static void main(String[] args) {
        //the below comment will be executed
// \u000d System.out.println("Java comment is executed!!");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac Test.java

C:\Users\Anurati\Desktop\abcDemo>java Test
Java comment is executed!!
```

The above code generate the output because the compiler parses the Unicode character **\u000d** as a **new line** before the lexical transformation, and thus the code is transformed as shown below:

Test.java

```
public class Test{
    public static void main(String[] args) {
        //the below comment will be executed
//
System.out.println("Java comment is executed!!");
    }
}
```

Thus, the Unicode character shifts the print statement to next line and it is executed as a normal Java code.

Java String

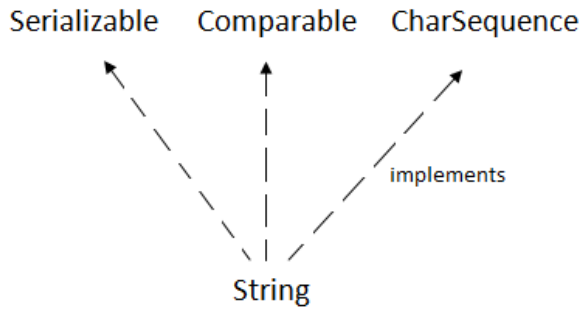
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

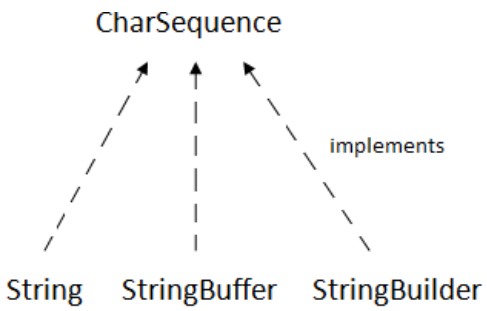
is same as: `String s="javatpoint";`

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.

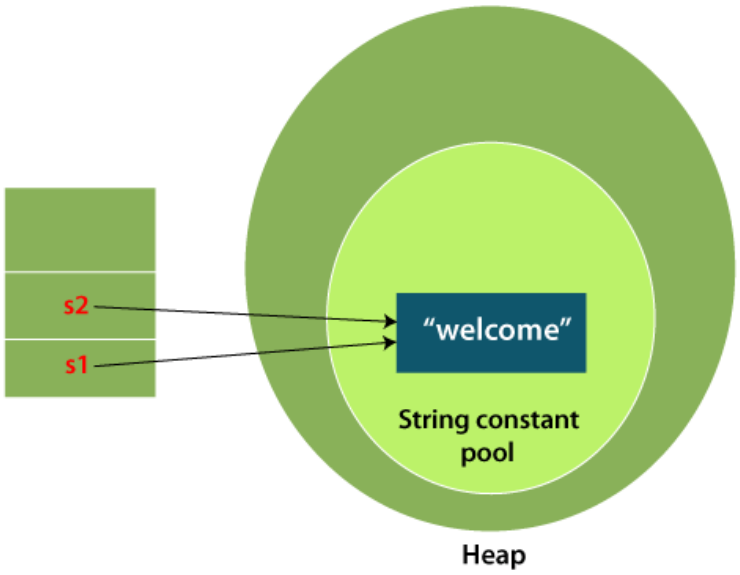


CharSequence Interface - The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

What is String in Java? Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.



How to create a string object? There are two ways to create String object:

1) String Literal - Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal? To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

StringExample.java

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

Constructors

1. **String(byte[] byte_arr)** – Construct a new String by decoding the *byte array*. It uses the platform's default character set for decoding**. Example:**

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s_byte =new String(b_arr); //Geeks
```

2. **String(byte[] byte_arr, Charset char_set)** – Construct a new String by decoding the *byte array*. It uses the *char_set* for decoding.
Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
Charset cs = Charset.defaultCharset();
String s_byte_char = new String(b_arr, cs); //Geeks
```

3. **String(byte[] byte_arr, String char_set_name)** – Construct a new String by decoding the *byte array*. It uses the *char_set_name* for decoding. It looks similar to the above constructs and they appear before similar functions but it takes the *String(which contains char_set_name)* as parameter while the above constructor takes *CharSet.* **Example:**

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s = new String(b_arr, "US-ASCII"); //Geeks
```

4. **String(byte[] byte_arr, int start_index, int length)** – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location).* **Example:**

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s = new String(b_arr, 1, 3); // eek
```

5. **String(byte[] byte_arr, int start_index, int length, Charset char_set)** – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.Uses *char_set* for decoding. **Example:**

```
byte[] b_arr = {71, 101, 101, 107, 115};
Charset cs = Charset.defaultCharset();
String s = new String(b_arr, 1, 3, cs); // eek
```

6. **String(byte[] byte_arr, int start_index, int length, String char_set_name)** – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.Uses *char_set_name* for decoding. **Example:**

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s = new String(b_arr, 1, 4, "US-ASCII"); // eeks
```

7. **String(char[] char_arr)** – Allocates a new String from the given *Character array* **Example:**

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
String s = new String(char_arr); //Geeks
```

8. **String(char[] char_array, int start_index, int count)** – Allocates a String from a given *character array* but choose *count* characters from the *start_index*. **Example:**

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
String s = new String(char_arr , 1, 3); //eek
```

9. **String(int[] uni_code_points, int offset, int count)** – Allocates a String from a *uni_code_array* but choose *count* characters from the *start_index*. **Example:**

```
int[] uni_code = {71, 101, 101, 107, 115};
String s = new String(uni_code, 1, 3); //eek
```

10. **String(StringBuffer s_buffer)** – Allocates a new string from the string in *s_buffer* **Example:**

```
StringBuffer s_buffer = new StringBuffer("Geeks");
String s = new String(s_buffer); //Geeks
```

11. **String(StringBuilder s_builder)** – Allocates a new string from the string in *s_builder* **Example:**

```
StringBuilder s_builder = new StringBuilder("Geeks");
String s = new String(s_builder); //Geeks
```

String Methods

1. **int length():** Returns the number of characters in the String.

```
"GeeksforGeeks".length();  // returns 13
```

2. **Char charAt(int i):** Returns the character at i^{th} index.

```
"GeeksforGeeks".charAt(3); // returns  'k'
```

3. **String substring (int i):** Return the substring from the i^{th} index character to end.

```
"GeeksforGeeks".substring(3); // returns "ksforGeeks"
```

4. **String substring (int i, int j):** Returns the substring from i to j-1 index.

```
 "GeeksforGeeks".substring(2, 5); // returns "eks"
```

5. **String concat(String str):** Concatenates specified string to the end of this string.

```
String s1 = "Geeks";
String s2 = "forGeeks";
String output = s1.concat(s2); // returns "GeeksforGeeks"
```

6. **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.

```
String s = "Learn Share Learn";
int output = s.indexOf("Share"); // returns 6
```

7. **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "Learn Share Learn";
int output = s.indexOf("ea",3);// returns 13
```

8. **Int lastIndexOf(String s):** Returns the index within the string of the last occurrence of the specified string.

```
String s = "Learn Share Learn";
int output = s.lastIndexOf("a"); // returns 14
```

9. **boolean equals(Object otherObj):** Compares this string to the specified object.

```
Boolean out = "Geeks".equals("Geeks"); // returns true
Boolean out = "Geeks".equals("geeks"); // returns false
```

10. **boolean equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.

```
Boolean out= "Geeks".equalsIgnoreCase("Geeks"); // returns true
Boolean out = "Geeks".equalsIgnoreCase("geeks"); // returns true
```

11. **int compareTo(String anotherString):** Compares two string lexicographically.

```
int out = s1.compareTo(s2);  // where s1 ans s2 are
                             // strings to be compared
```

```
 This returns difference s1-s2. If :
 out < 0  // s1 comes before s2
 out = 0  // s1 and s2 are equal.
 out > 0   // s1 comes after s2.
```

12. int compareToIgnoreCase(String anotherString): Compares two string lexicographically, ignoring case considerations.

```
 int out = s1.compareToIgnoreCase(s2);
// where s1 ans s2 are
// strings to be compared

 This returns difference s1-s2. If :
 out < 0  // s1 comes before s2
 out = 0   // s1 and s2 are equal.
 out > 0   // s1 comes after s2.
```

Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

13. String toLowerCase(): Converts all the characters in the String to lower case.

```
String word1 = “HeLLo”;
String word3 = word1.toLowerCase(); // returns “hello"
```

14. String toUpperCase(): Converts all the characters in the String to upper case.

```
String word1 = “HeLLo”;
String word2 = word1.toUpperCase(); // returns “HELLO”
```

15. String trim(): Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = “ Learn Share Learn “;
String word2 = word1.trim(); // returns “Learn Share Learn”
```

16. String replace (char oldChar, char newChar): Returns new string by replacing all occurrences of *oldChar* with *newChar.*

```
String s1 = “feeksforfeeks“;
String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”
```

Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks

```
// Java code to illustrate different constructors and methods
// String class.
import java.io.*;
import java.util.*;
class Test
{
    public static void main (String[] args)
    {
        String s= "GeeksforGeeks";
        // or String s= new String ("GeeksforGeeks");

        // Returns the number of characters in the String.
        System.out.println("String length = " + s.length());

        // Returns the character at ith index.
        System.out.println("Character at 3rd position = " + s.charAt(3));

        // Return the substring from the ith  index character to end of string
        System.out.println("Substring " + s.substring(3));

        // Returns the substring from i to j-1 index.
        System.out.println("Substring  = " + s.substring(2,5));

        // Concatenates string2 to the end of string1.
```

```
        String s1 = "Geeks";
        String s2 = "forGeeks";
        System.out.println("Concatenated string  = " + s1.concat(s2));

        // Returns the index within the string
        // of the first occurrence of the specified string.
        String s4 = "Learn Share Learn";
        System.out.println("Index of Share " + s4.indexOf("Share"));

        // Returns the index within the string of the
        // first occurrence of the specified string,
        // starting at the specified index.
        System.out.println("Index of a  = " + s4.indexOf('a',3));

        // Checking equality of Strings
        Boolean out = "Geeks".equals("geeks");
        System.out.println("Checking Equality  " + out);
        out = "Geeks".equals("Geeks");
        System.out.println("Checking Equality  " + out);

        out = "Geeks".equalsIgnoreCase("gEeks ");
        System.out.println("Checking Equality " + out);

        //If ASCII difference is zero then the two strings are similar
        int out1 = s1.compareTo(s2);
        System.out.println("the difference between ASCII value is="+out1);
        // Converting cases
        String word1 = "GeeKyMe";
        System.out.println("Changing to lower Case " + word1.toLowerCase());

        // Converting cases
        String word2 = "GeekyME";
        System.out.println("Changing to UPPER Case " + word2.toUpperCase());

        // Trimming the word
        String word4 = " Learn Share Learn ";
        System.out.println("Trim the word " + word4.trim());

        // Replacing characters
        String str1 = "feeksforfeeks";
        System.out.println("Original String " + str1);
        String str2 = "feeksforfeeks".replace('f' ,'g') ;
        System.out.println("Replaced f with g -> " + str2);
    }
}
```

Output :

```
String length = 13
Character at 3rd position = k
Substring ksforGeeks
Substring = eks
Concatenated string = GeeksforGeeks
Index of Share 6
Index of a = 8
Checking Equality false
Checking Equality true
Checking Equality false
the difference between ASCII value is=-31
Changing to lower Case geekyme
Changing to UPPER Case GEEKYME
Trim the word Learn Share Learn
Original String feeksforfeeks
Replaced f with g -> geeksgorgeeks
```

Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

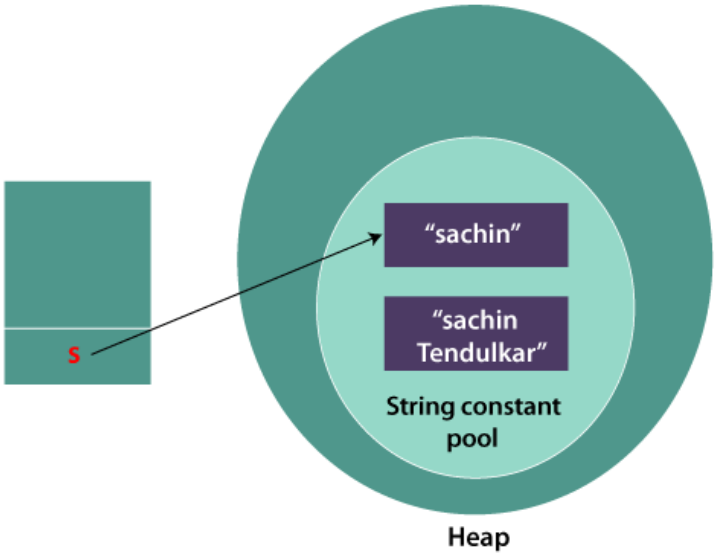
Once String object is created its data or state can't be changed but a new String object is created. Let's try to understand the concept of immutability by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Output: Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.

As you can see in the above figure that two objects are created but ***s*** reference variable still refers to "Sachin" not to "Sachin Tendulkar". But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.




```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

Output: Sachin Tendulkar

In such a case, s points to the "Sachin Tendulkar". Please notice that still Sachin object is not modified.

Why String objects are immutable in Java? As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

1. ClassLoader: A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different. To avoid this kind of misinterpretation, String is immutable.

2. Thread Safe: As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

3. Security: As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

4. Heap Space: The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

Why String class is Final in Java? The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

Java String compare

We can compare String in Java on the basis of content and reference. It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

1) By Using equals() Method - The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   String s4="Saurav";
   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
 }
}
```

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

```
class Teststringcomparison2{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="SACHIN";
   System.out.println(s1.equals(s2));//false
   System.out.println(s1.equalsIgnoreCase(s2));//true
 }
}
```

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings.

2) By Using == operator - The == operator compares references not values.

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);//true (because both refer to same instance)
   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
 }
}
```

3) String compare by compareTo() method - The above code, demonstrates the use of **==** operator used for comparing two ***String*** objects.

By Using compareTo() method - The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

- **s1 == s2**: The method returns 0.
- **s1 > s2**: The method returns a positive value.
- **s1 < s2**: The method returns a negative value.

```
class Teststringcomparison4{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(because s1>s3)
   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 }
}
```

String Concatenation in Java

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1) String Concatenation by + (String concatenation) operator - Java String concatenation operator (+) is used to add strings. For Example:

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Sachin"+" Tendulkar";
   System.out.println(s); //Sachin Tendulkar
 }
}
```

The **Java compiler transforms** above code to this:

```
String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();
```

In Java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and it's append method. String concatenation operator produces a new String by appending the second operand onto the end of the first operand. The String concatenation operator can concatenate not only String but primitive values also. For Example:

```
class TestStringConcatenation2{
 public static void main(String args[]){
   String s=50+30+"Sachin"+40+40;
   System.out.println(s); //80Sachin4040
 }
}
```

Note: After a string literal, all the + will be treated as string concatenation operator.

2) String Concatenation by concat() method - The String concat() method concatenates the specified string to the end of current string. Syntax:

```
public String concat(String another)
```

Let's see the example of String concat() method.

```
class TestStringConcatenation3{
 public static void main(String args[]){
   String s1="Sachin ";
   String s2="Tendulkar";
   String s3=s1.concat(s2);
   System.out.println(s3);//Sachin Tendulkar
  }
```

```
}
```

The above Java program, concatenates two String objects ***s1*** and ***s2*** using ***concat()*** method and stores the result into ***s3*** object.

There are some other possible ways to concatenate Strings in Java,

1. String concatenation using StringBuilder class - StringBuilder is class provides append() method to perform concatenation operation. The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double. StringBuilder is the most popular and fastest way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.

```
public class StrBuilder
{
    /* Driver Code */
    public static void main(String args[])
    {
        StringBuilder s1 = new StringBuilder("Hello");    //String 1
        StringBuilder s2 = new StringBuilder(" World");    //String 2
        StringBuilder s = s1.append(s2);   //String 3 to store the result
            System.out.println(s.toString());  //Displays result: Hello World
    }
}
```

In the above code snippet, **s1, s2** and **s** are declared as objects of **StringBuilder** class. **s** stores the result of concatenation of **s1** and **s2** using **append**() method.

2. String concatenation using format() method - String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.

```
public class StrFormat
{
    /* Driver Code */
    public static void main(String args[])
    {
        String s1 = new String("Hello");    //String 1
        String s2 = new String(" World");    //String 2
        String s = String.format("%s%s",s1,s2);   //String 3 to store the result
        System.out.println(s.toString());  //Displays result :  Hello World
    }
}
```

Here, the String objects **s** is assigned the concatenated result of Strings **s1** and **s2** using **String.format()** method. format() accepts parameters as format specifier followed by String objects or values.

3. String concatenation using String.join() method (Java Version 8+) - The String.join() method is available in Java version 8 and all the above versions. String.join() method accepts arguments first a separator and an array of String objects.

```
public class StrJoin
{
    /* Driver Code */
    public static void main(String args[])
    {
        String s1 = new String("Hello");    //String 1
        String s2 = new String(" World");    //String 2
        String s = String.join("",s1,s2);   //String 3 to store the result
            System.out.println(s.toString());  //Displays result
    }
}
```

Output: Hello World

In the above code snippet, the String object **s** stores the result of **String.join("",s1,s2)** method. A separator is specified inside quotation marks followed by the String objects or array of String objects.

4. String concatenation using StringJoiner class (Java Version 8+) - StringJoiner class has all the functionalities of String.join() method. In advance its constructor can also accept optional arguments, prefix and suffix.

```
public class StrJoiner
{
    /* Driver Code */
    public static void main(String args[])
    {
        StringJoiner s = new StringJoiner(", ");   //StringeJoiner object
        s.add("Hello");    //String 1
        s.add("World");    //String 2
        System.out.println(s.toString());  //Displays result
    }
}
```

Output: Hello, World

In the above code snippet, the StringJoiner object **s** is declared and the constructor StringJoiner() accepts a separator value. A separator is specified inside quotation marks. The add() method appends Strings passed as arguments.

5. String concatenation using Collectors.joining() method (Java (Java Version 8+)

The Collectors class in Java 8 offers joining() method that concatenates the input elements in a similar order as they occur.

```
import java.util.*;
import java.util.stream.Collectors;
public class ColJoining
{
    /* Driver Code */
    public static void main(String args[])
    {
        List<String> liststr = Arrays.asList("abc", "pqr", "xyz"); //List of String array
    String str = liststr.stream().collect(Collectors.joining(", ")); //performs joining operation
        System.out.println(str.toString());  //Displays result
    }
}
```

Output: abc, pqr, xyz

Here, a list of String array is declared. And a String object **str** stores the result of **Collectors.joining()** method.

Substring in Java

A part of String is called **substring**. In other words, substring is a subset of another String. Java String class provides the built-in substring() method that extract a substring from the given string by using the index values passed as an argument. In case of substring() method startIndex is inclusive and endIndex is exclusive. Suppose the string is "**computer**", then the substring will be com, compu, ter, etc.

Note: Index starts from 0.

You can get substring from the given String object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive). The method throws an IndexOutOfBoundException when the startIndex is larger than the length of String or less than zero.

2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex. The method throws an IndexOutOfBoundException when the startIndex is less than zero or startIndex is greater than endIndex or endIndex is greater than length of String.

In case of String:

- startIndex: inclusive
- **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

```
String s="hello";
System.out.println(s.substring(0,2)); //returns he  as a substring
```

In the above substring, 0 points the first letter and 2 points the second letter i.e., e (because end index is exclusive).

TestSubstring.java

```
public class TestSubstring{
 public static void main(String args[]){
 String s="SachinTendulkar";
 System.out.println("Original String: " + s);
 System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar
 System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin
 }
}
```

The above Java programs, demonstrates variants of the ***substring()*** method of ***String*** class. The startindex is inclusive and endindex is exclusive.

Using String.split() method: The split() method of String class can be used to extract a substring from a sentence. It accepts arguments in the form of a regular expression.

```
import java.util.*;
public class TestSubstring2
{
    /* Driver Code */
    public static void main(String args[])
    {
        String text= new String("Hello, My name is Sachin");
        /* Splits the sentence by the delimeter passed as an argument */
        String[] sentences = text.split("\\.");
        System.out.println(Arrays.toString(sentences));
    }
}
```

Output: [Hello, My name is Sachin]

In the above program, we have used the split() method. It accepts an argument \\. that checks a in the sentence and splits the string into another string. It is stored in an array of String objects sentences.

Java String Class Methods

The **java.lang.String** class provides a lot of built-in methods that are used to manipulate **string in Java**. By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc. Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application. Let's use some important methods of String class.

Java String toUpperCase() and toLowerCase() method - The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

```
public class Stringoperation1{
public static void main(String ar[]){
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
```

```
}
}
```

Java String trim() method - The String class trim() method eliminates white spaces before and after the String.

```
public class Stringoperation2 {
public static void main(String ar[]){
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
}
}
```

Java String startsWith() and endsWith() method - The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

```
public class Stringoperation3{
public static void main(String ar[]){
String s="Sachin";
 System.out.println(s.startsWith("Sa"));//true
 System.out.println(s.endsWith("n"));//true
}
}
```

Java String charAt() Method - The String class charAt() method returns a character at specified index.

```
public class Stringoperation4{
public static void main(String ar[]){
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
}
}
```

Java String length() Method - The String class length() method returns length of the specified String.

```
public class Stringoperation5{
public static void main(String ar[]){
String s="Sachin";
System.out.println(s.length());//6
}
}
```

Java String intern() Method - A pool of strings, initially empty, is maintained privately by the class String.

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.

```
public class Stringoperation6{
public static void main(String ar[]){
String s=new String("Sachin");
String s2=s.intern();
System.out.println(s2);//Sachin
}
}
```

Java String valueOf() Method - The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

```
public class Stringoperation7{
public static void main(String ar[]){
int a=10;
String s=String.valueOf(a);
```

```
System.out.println(s+10);  // 1010
}
}
```

Java String replace() Method - The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
public class Stringoperation8{
public static void main(String ar[]){
String s1="Java is a programming language. Java is a platform. Java is an Island.";
String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
System.out.println(replaceString);
}
}
```

Output: Kava is a programming language. Kava is a platform. Kava is an Island.

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed. Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.

public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String? A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

1) StringBuffer Class append() Method - The append() method concatenates the given argument with this String.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

2) StringBuffer insert() Method - The insert() method inserts the given String with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

3) StringBuffer replace() Method - The replace() method replaces the given String from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

4) StringBuffer delete() Method - The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

5) StringBuffer reverse() Method - The reverse() method of the StringBuilder class reverses the current String.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

6) StringBuffer capacity() Method - The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

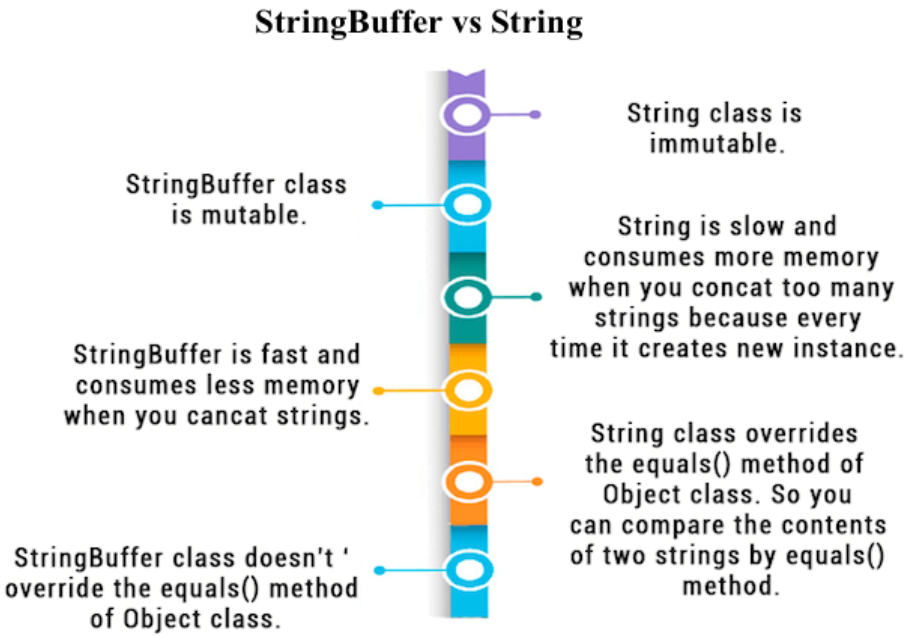
7) StringBuffer ensureCapacity() method - The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBufferExample7{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10);//now no change
System.out.println(sb.capacity());//now 34
sb.ensureCapacity(50);//now (34*2)+2
System.out.println(sb.capacity());//now 70
}
}
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory



Performance Test of String and StringBuffer

```
public class ConcatTest{
    public static String concatWithString()    {
        String t = "Java";
        for (int i=0; i<10000; i++){
            t = t + "Tpoint";
        }
        return t;
    }
    public static String concatWithStringBuffer(){
        StringBuffer sb = new StringBuffer("Java");
        for (int i=0; i<10000; i++){
            sb.append("Tpoint");
        }
        return sb.toString();
    }
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        concatWithString();
        System.out.println("Time taken by Concating with String: "+(System.currentTimeMillis()-
startTime)+"ms");
        startTime = System.currentTimeMillis();
        concatWithStringBuffer();
        System.out.println("Time taken by Concating with  StringBuffer: "+(System.currentTimeMillis()
-startTime)+"ms");
    }
}
```

Output:

```
Time taken by Concating with String: 578ms
Time taken by Concating with  StringBuffer: 0ms
```

The above code, calculates the time required for concatenating a string using the String class and StringBuffer class.

String and StringBuffer HashCode Test - As we can see in the program given below, String returns new hashcode while performing concatenation but the StringBuffer class returns same hashcode.

```
public class InstanceTest{
    public static void main(String args[]){
        System.out.println("Hashcode test of String:");
        String str="java";
        System.out.println(str.hashCode());
        str=str+"tpoint";
        System.out.println(str.hashCode());

        System.out.println("Hashcode test of StringBuffer:");
        StringBuffer sb=new StringBuffer("java");
        System.out.println(sb.hashCode());
        sb.append("tpoint");
        System.out.println(sb.hashCode());
    }
}
```

Output:

```
Hashcode test of String:
3254818
229541438
Hashcode test of StringBuffer:
118352462
118352462
```

Difference between StringBuffer and StringBuilder - Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

A list of differences between StringBuffer and StringBuilder is given below:

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

StringBuffer Example

```
//Java Program to demonstrate the use of StringBuffer class.
public class BufferTest{
    public static void main(String[] args){
        StringBuffer buffer=new StringBuffer("hello");
        buffer.append("java");
        System.out.println(buffer);
    }
}
```

Output: hellojava

StringBuilder Example

```
//Java Program to demonstrate the use of StringBuilder class.
public class BuilderTest{
    public static void main(String[] args){
        StringBuilder builder=new StringBuilder("hello");
        builder.append("java");
        System.out.println(builder);
    }
}
```

Output: hellojava

Performance Test of StringBuffer and StringBuilder - Let's see the code to check the performance of StringBuffer and StringBuilder classes.

```
//Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
public class ConcatTest{
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        StringBuffer sb = new StringBuffer("Java");
        for (int i=0; i<10000; i++){
            sb.append("Tpoint");
        }
        System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() -
 startTime) + "ms");
        startTime = System.currentTimeMillis();
        StringBuilder sb2 = new StringBuilder("Java");
        for (int i=0; i<10000; i++){
            sb2.append("Tpoint");
        }
        System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() -
 startTime) + "ms");
    }
}
```

Output:

```
Time taken by StringBuffer: 16ms
Time taken by StringBuilder: 0ms
```

How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below: In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method.

ImmutableDemo.java

```
public final class Employee
{
final String pancardNumber;
public Employee(String pancardNumber)
{
this.pancardNumber=pancardNumber;
}
public String getPancardNumber(){
return pancardNumber;
}
}
public class ImmutableDemo
{
public static void main(String ar[])
{
Employee e = new Employee("ABC123");
String s1 = e.getPancardNumber();
System.out.println("Pancard Number: " + s1);
}
}
```

Output: Pancard Number: ABC123

The above class is immutable because:

- The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
- The class is final so we cannot create the subclass.
- There is no setter methods i.e. we have no option to change the value of the instance variable.

Java toString() Method

If you want to represent any object as a string, **toString() method** comes into existence. The toString() method returns the String representation of the object. If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

Advantage of Java toString() method - By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

Understanding problem without toString() method - Let's see the simple code that prints reference.

```
class Student{
 int rollno;
 String name;
 String city;
 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");
   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```

Output:

```
Student@1fee6fc
Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since Java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

Example of Java toString() method

```
class Student{
 int rollno;
 String name;
 String city;
 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");
   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
```

```
 }
}
```

Output:

```
101 Raj lucknow
102 Vijay ghaziabad
```

In the above program, Java compiler internally calls ***toString()*** method, overriding this method will return the specified values of ***s1*** and ***s2*** objects of Student class.

Java Arrays

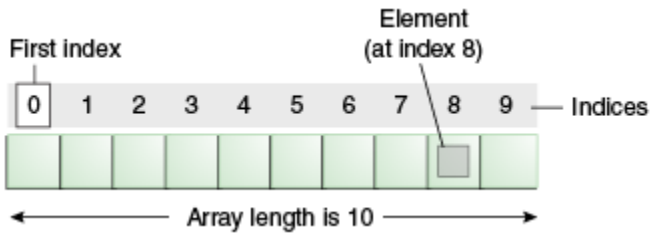
Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, and 2nd element is stored on 1st index and so on. Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized; we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array - Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array - We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization

//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

```
for(data_type variable:array){
//body of the loop
}

//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
```

```
}}
```

Output:

```
33
3
4
5
```

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array. Let's see the simple example to get the minimum number of an array using a method.

```
//Java Program to demonstrate the way of passing an array
//to method.
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
System.out.println(min);
}

public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

Output: 3

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
//Java Program to demonstrate the way of passing an anonymous array
//to method.
public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}

public static void main(String args[]){
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}}
```

Output:

```
10
22
44
66
```

Returning Array from the Method

We can also return an array from the method in Java.

```
//Java Program to return an array from the method
class TestReturnArray{
//creating method which returns an array
```

```
static int[] get(){
return new int[]{10,30,50,90,60};
}

public static void main(String args[]){
//calling method which returns an array
int arr[]=get();
//printing the values of an array
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}}
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.
public class TestArrayException{
public static void main(String args[]){
int arr[]={50,60,70,80};
for(int i=0;i<=arr.length;i++){
System.out.println(arr[i]);
}
}}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
       at TestArrayException.main(TestArrayException.java:5)
50
60
70
80
```

Multidimensional Array in Java

In Java, multidimensional arrays are actually arrays of arrays. In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array - Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

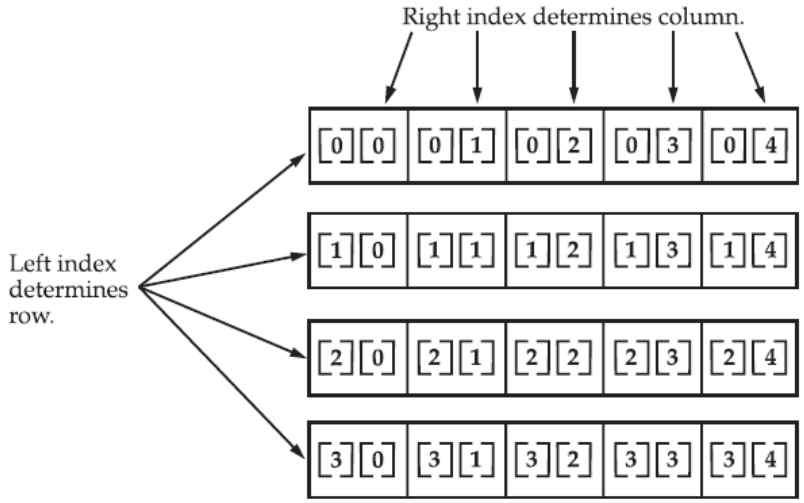
```
1 2 3
2 4 5
4 4 5
```

// Demonstrate a two-dimensional array.

```
public class TwoDArray {
    public static void main(String args[]){
        int twoD[][] = new int[4][5];
        int i, j, k = 0;

        for(i = 0; i < 4; i++){
            for(j = 0; j < 5; j++){
                twoD[i][j] = k;
                k++;
            }
        }

        for(i = 0; i < 4; i++){
            for(j = 0; j < 5; j++){
                System.out.print(twoD[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```



Given: int twoD [ ] [ ] = new int [4] [5] ;

Let's look at one more example that uses a multidimensional array. The following program creates a 3 by 4 by 5, three-dimensional array. It then loads each element with the product of its indexes. Finally, it displays these products.

```
public class ThreeDMatrix {
    public static void main(String args[]){
        int threeD[][][] = new int[3][4][5];
        int i, j, k;

        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++){
                for(k = 0; k < 5; k++){
                    threeD[i][j][k] = i * j * k;
                }
            }
        }

        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++){
                for(k = 0; k < 5; k++){
                    System.out.print(threeD[i][j][k] + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
```

```
0 6 12 18 24
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;

        //printing the data of a jagged array
        for (int i=0; i<arr.length; i++){
            for (int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}
```

Output:

```
0 1 2
3 4 5 6
7 8
```

What is the class name of Java array? In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
//Java Program to get the class name of array in Java
class Testarray4{
public static void main(String args[]){
//declaration and initialization of array
int arr[]={4,4,5};
//getting the class name of Java array
Class c=arr.getClass();
String name=c.getName();
//printing the class name of Java array
System.out.println(name);

}}
```

Output: I

Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

Syntax of arraycopy method

```
public static void arraycopy(
Object src, int srcPos,Object dest, int destPos, int length
)
```

Example of Copying an Array in Java

```
//Java Program to copy a source array into a destination array in Java
class TestArrayCopyDemo {
    public static void main(String[] args) {
        //declaring a source array
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
        //declaring a destination array
        char[] copyTo = new char[7];
        //copying array using System.arraycopy() method
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        //printing the destination array
        System.out.println(String.valueOf(copyTo));
    }
}
```

Output: caffein

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

```
//Java Program to clone the array
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
System.out.println("Printing original array:");
for(int i:arr)
System.out.println(i);

System.out.println("Printing clone of the array:");
int carr[]=arr.clone();
for(int i:carr)
System.out.println(i);
System.out.println("Are both equal?");
System.out.println(arr==carr);

}}
```

Output:

```
Printing original array:
33
3
4
5
Printing clone of the array:
33
3
4
5
Are both equal?
false
```

Addition of 2 Matrices in Java

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
```

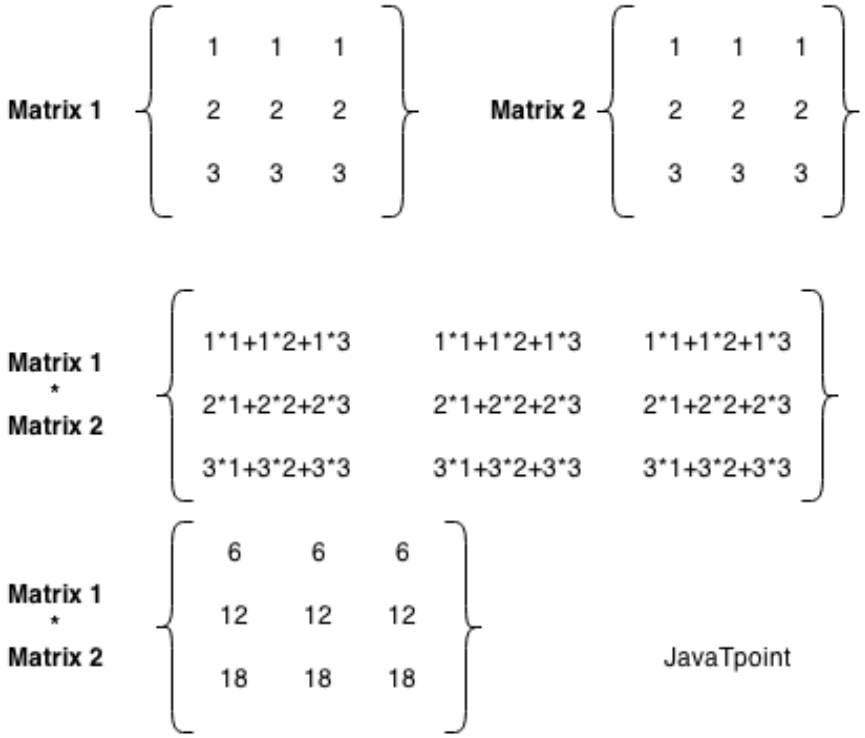
```
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```

Output:

```
2 6 8
6 8 10
```

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

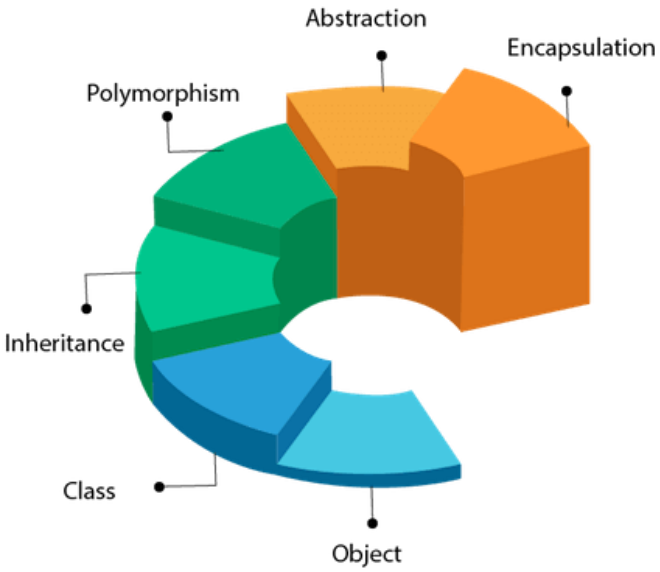
```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};
//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3];  //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
```

```
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}}
```

Output:

```
6 6 6
12 12 12
18 18 18
```

Java OOPs Concepts



In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language. **Smalltalk** is considered the first truly object-oriented programming language. The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

Object - Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class - *Collection of objects* is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance - *When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism - If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism. Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction - *Hiding internal details and showing functionality* is known as abstraction. For example phone call, we don't know the internal processing. In Java, we use abstract class and interface to achieve abstraction.

Encapsulation - Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.


Capsule

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling - Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion - Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association - Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation - Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition - The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

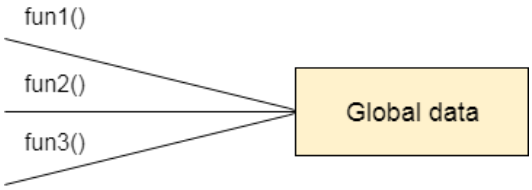


Figure: Data Representation in Procedure-Oriented Programming

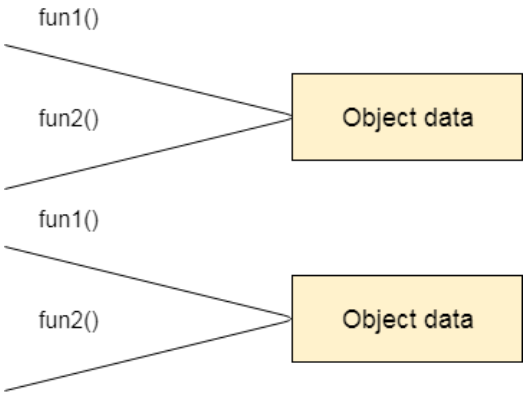


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

What is the difference between an object-oriented programming language and object-based programming language? Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

Java Naming Convention

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc. But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of Naming Conventions in Java - By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Naming Conventions of the Different Identifiers - The following table shows the popular conventions used for the different identifiers.

Identifiers Type	Naming Rules	Examples

Class	It should start with the uppercase letter.
	It should be a noun such as Color, Button, System, Thread, etc.
	Use appropriate words, instead of acronyms.
		`public class Employee`
		`{`
		`//code snippet`
		`}`

Interface	It should start with the uppercase letter.
	It should be an adjective such as Runnable, Remote, ActionListener.
	Use appropriate words, instead of acronyms.
		`interface Printable`
		`{`
		`//code snippet`
		`}`

Method	It should start with lowercase letter.
	It should be a verb such as main(), print(), println().
	If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
		`class Employee`
		`{`
		`// method`
		`void draw()`
		`{`
		`//code snippet`
		`}`
		`}`

Variable	It should start with a lowercase letter such as id, name.
	It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
	Avoid using one-character variables such as x, y, z.
		`class Employee`
		`{`
		`// variable`
		`int id;`
		`//code snippet`
		`}`

Package	It should be a lowercase letter such as java, lang.
	If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
		`//package`
		`package com.javatpoint;`
		`class Employee`
		`{`
		`//code snippet`
		`}`

Constant	It should be in uppercase letters such as RED, YELLOW.
	If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
	It may contain digits but not as the first letter.
		`class Employee`
		`{`
		`//constant`
		`static final`
		`int MIN_AGE = 18;`
		`//code snippet`
		`}`

CamelCase in Java naming conventions - Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

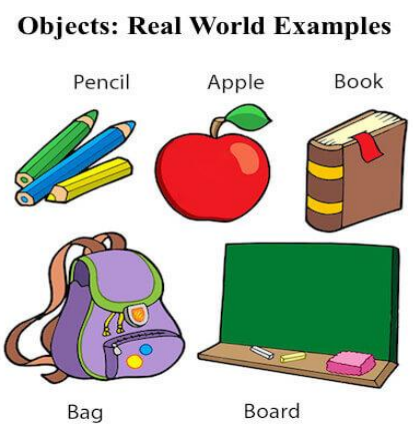
Objects and Classes in Java

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java? An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.



- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.

- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

Object Definitions:

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behavior.
- The object is an instance of a class.

What is a class in Java? A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical. A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

Instance variable in Java - A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class - In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value. Here, we are creating a main() method inside the class.

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
```

```
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable  // 0
  System.out.println(s1.name);  // null
 }
}
```

Object and Class Example: main outside the class - In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);  // 0
  System.out.println(s1.name);  // null
 }
}
```

3 Ways to initialize object

1) Object and Class Example: Initialization through reference - Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space  // 101 Sonoo
 }
}
```

We can also create multiple objects and store information in it through reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
```

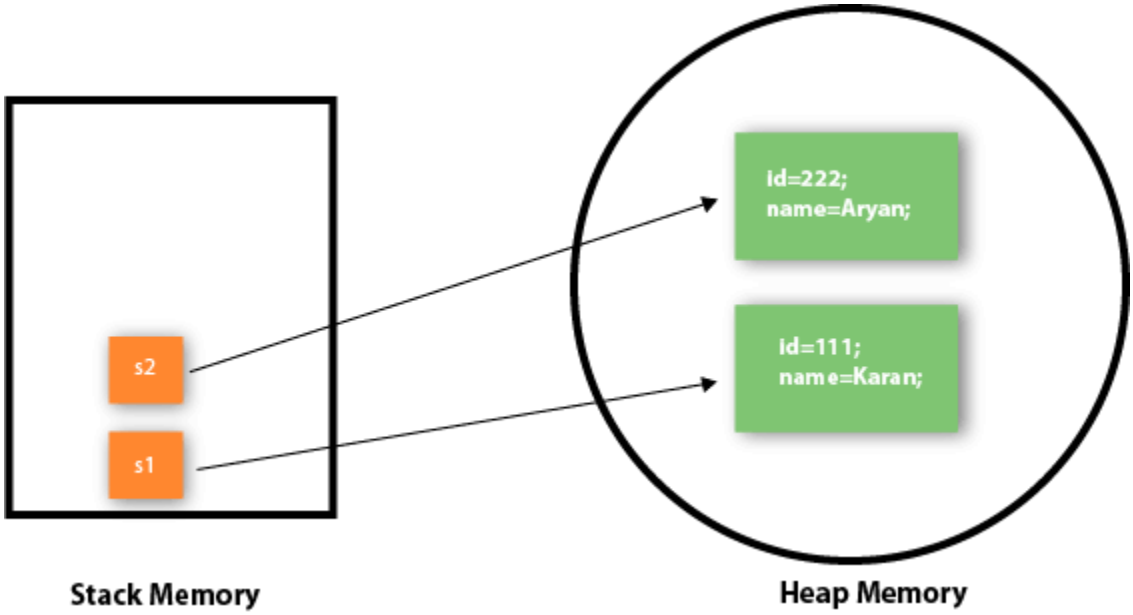
```
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method - In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();  // 111 Karan
  s2.displayInformation();  // 222 Aryan
 }
}
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

Object and Class Example: Employee - Let's see an example where we are maintaining records of employees.

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle - There is given another example that maintains the records of Rectangle class.

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();  // 55
  r2.calculateArea();  // 45
}
}
```

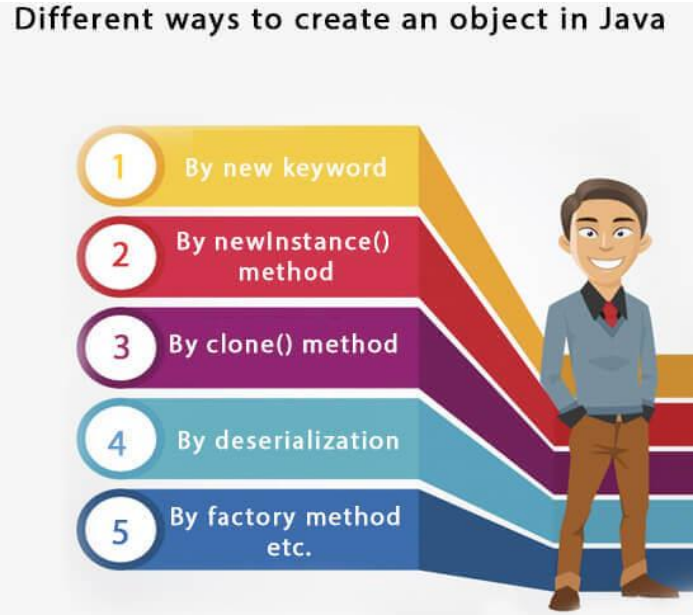
What are the different ways to create an object in Java? There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only. If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation();//anonymous object
```



Calling method through a reference:

```
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){

 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output: Factorial is 120

Creating multiple objects by one type only - We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of reference variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects

//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
 public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea(); // 55
  r2.calculateArea();  // 45
}
}
```

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
class Account{
int acc_no;
String name;
float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Example: Java Class and Objects

```
class Lamp {
  // stores the value for light
  // true if light is on
  // false if light is off
  boolean isOn;

  // method to turn on the light
```

```
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);

  }

  // method to turn off the light
  void turnOff() {
    isOn = false;
    System.out.println("Light on? " + isOn);
  }
}

class Main {
  public static void main(String[] args) {
    // create objects led and halogen
    Lamp led = new Lamp();
    Lamp halogen = new Lamp();
    // turn on the light by
    // calling method turnOn()
    led.turnOn();
    // turn off the light by
    // calling method turnOff()
    halogen.turnOff();
  }
}
```

Output:

```
Light on? true
Light on? false
```

In the above program, we have created a class named Lamp. It contains a variable: isOn and two methods: turnOn() and turnOff(). Inside the Main class, we have created two objects: led and halogen of the Lamp class. We then used the objects to call the methods of the class.

- **led.turnOn()** - It sets the isOn variable to true and prints the output.
- **halogen.turnOff()** - It sets the isOn variable to false and prints the output.

The variable isOn defined inside the class is also called an instance variable. It is because when we create an object of the class, it is called an instance of the class. And, each instance will have its own copy of the variable.

That is, led and halogen objects will have their own copy of the isOn variable.

Java Methods

A method is a block of code that performs a specific task. Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

- a method to draw the circle
- a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

In Java, there are two types of methods:

- **User-defined Methods**: We can create our own method based on our requirements.
- **Standard Library Methods**: These are built-in methods in Java that are available to use.

Let's first learn about user-defined methods.

Declaring a Java Method

```
returnType methodName() {
  // method body
}
```

Here,

- **returnType** - It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value. If the method does not return a value, its return type is void.
- **methodName** - It is an identifier that is used to refer to the particular method in a program.
- **method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.

For example,

```
int addNumbers() {
// code
}
```

In the above example, the name of the method is adddNumbers(). And, the return type is int. We will learn more about return types later in this tutorial.

This is the simple syntax of declaring a method. However, the complete syntax of declaring a method is

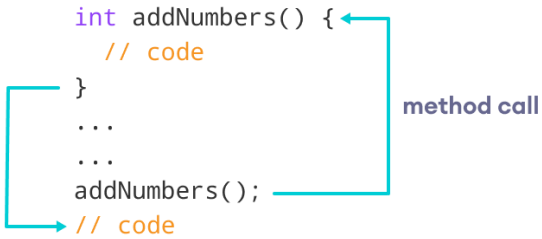
```
modifier static returnType nameOfMethod (parameter1, parameter2, ...) {
  // method body
}
```

Here,

- **modifier** - It defines access types whether the method is public, private, and so on.
- **static** - If we use the static keyword, it can be accessed without creating objects. For example, the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.
- **parameter1/parameter2** - These are values passed to a method. We can pass any number of arguments to a method.

Calling a Method in Java - In the above example, we have declared a method named addNumbers(). Now, to use the method, we need to call it. Here's is how we can call the addNumbers() method.

```
// calls the method
addNumbers();
```



Working of Java Method Call

Example 1: Java Methods

```
class Main {
  // create a method
  public int addNumbers(int a, int b) {
    int sum = a + b;
    // return value
    return sum;
  }
```

```
  public static void main(String[] args) {
    int num1 = 25;
    int num2 = 15;
    // create an object of Main
    Main obj = new Main();
    // calling method
    int result = obj.addNumbers(num1, num2);
    System.out.println("Sum is: " + result);
  }
}
```

Output - Sum is: 40

In the above example, we have created a method named addNumbers(). The method takes two parameters a and b. Notice the line, `int result = obj.addNumbers(num1, num2);`

Here, we have called the method by passing two arguments num1 and num2. Since the method is returning some value, we have stored the value in the result variable.

Note: The method is not static. Hence, we are calling the method using the object of the class.

Java Method Return Type - A Java method may or may not return a value to the function call. We use the **return statement** to return any value. For example,

```
int addNumbers() {
...
return sum;
}
```

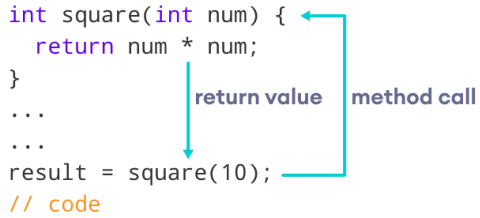
Here, we are returning the variable sum. Since the return type of the function is int. The sum variable should be of int type. Otherwise, it will generate an error.

Example 2: Method Return Type

```
class Main {
// create a method
  public static int square(int num) {
    // return statement
    return num * num;
  }

  public static void main(String[] args) {
    int result;
    // call the method
    // store returned value to result
    result = square(10);
    System.out.println("Squared value of 10 is: " + result);
  }
}
```

Output: Squared value of 10 is: 100

In the above program, we have created a method named square(). The method takes a number as its parameter and returns the square of the number. Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.



Representation of the Java method returning a value

Note: If the method does not return any value, we use the void keyword as the return type of the method. For example,

```
public void square(int a) {
  int square = a * a;
  System.out.println("Square is: " + a);
}
```

Method Parameters in Java - A method parameter is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,

```
// method with two parameters
int addNumbers(int a, int b) {
  // code
}

// method with no parameter
int addNumbers(){
  // code
}
```

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,

```
// calling the method with two parameters
addNumbers(25, 15);
// calling the method with no parameters
addNumbers()
```

Example 3: Method Parameters

```
class Main {
  // method with no parameter
  public void display1() {
    System.out.println("Method without parameter");
  }

  // method with single parameter
  public void display2(int a) {
    System.out.println("Method with a single parameter: " + a);
  }

  public static void main(String[] args) {
    // create an object of Main
    Main obj = new Main();
    // calling method with no parameter
    obj.display1();
    // calling method with the single parameter
    obj.display2(24);
  }
}
```

Output

```
Method without parameter
```

```
Method with a single parameter: 24
```

Here, the parameter of the method is int. Hence, if we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

Note: The argument 24 passed to the display2() method during the method call is called the actual argument.

The parameter num accepted by the method definition is known as a formal argument. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

Standard Library Methods - The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

For example,

- print() is a method of java.io.PrintSteam. The print("...") method prints the string inside quotation marks.
- sqrt() is a method of Math class. It returns the square root of a number.

Here's a working example:

Example 4: Java Standard Library Method

```
public class Main {
  public static void main(String[] args) {
    // using the sqrt() method
    System.out.print("Square root of 4 is: " + Math.sqrt(4)); // Square root of 4 is: 2.0
  }
}
```

What are the advantages of using methods?

1. The main advantage is **code reusability**. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".

Example 5: Java Method for Code Reusability

```
public class Main {
  // method defined
  private static int getSquare(int x){
    return x * x;
  }

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      // method call
      int result = getSquare(i);
      System.out.println("Square of " + i + " is: " + result);
    }
  }
}
```

Output:

```
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25
```

In the above program, we have created the method named getSquare() to calculate the square of a number. Here, the method is used to calculate the square of numbers less than **6**.

Hence, the same method is used again and again.

2. Methods make code more **readable and easier** to debug. Here, the getSquare() method keeps the code to compute the square in a block. Hence, makes it more readable.

Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default. There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor - There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor: `<class_name>(){}`

Example of default constructor - In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output: Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

Q) What is the purpose of a default constructor? The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type. Example of default constructor that displays the default values

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
int id;
String name;
//method to display the value of id and name
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();  // 0 null
s2.display();  // 0 null
}
}
```

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor? The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor - In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();  // 111 Karan
    s2.display();  // 222 Aryan
```

```
   }
}
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();  // 111 Karan 0
    s2.display();  // 222 Aryan 25
   }
}
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Java Copy Constructor - There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++. There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
//Java program to initialize the values from one object to another object.
class Student6{
```

```
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();  // 111 Karan
    s2.display();  // 111 Karan
   }
}
```

Copying values without constructor - We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{
    int id;
    String name;
    Student7(int i,String n){
    id = i;
    name = n;
    }
    Student7(){}
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student7 s1 = new Student7(111,"Karan");
    Student7 s2 = new Student7();
    s2.id=s1.id;
    s2.name=s1.name;
    s1.display();  // 111 Karan
    s2.display();  // 111 Karan
   }
}
```

Q) Does constructor return any value? Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization? Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java? Yes.

What is the purpose of Constructor class? Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

Java static keyword

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class. The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block

4. Nested class

1) Java static variable - If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable - It makes your program **memory efficient** (i.e., it saves memory).

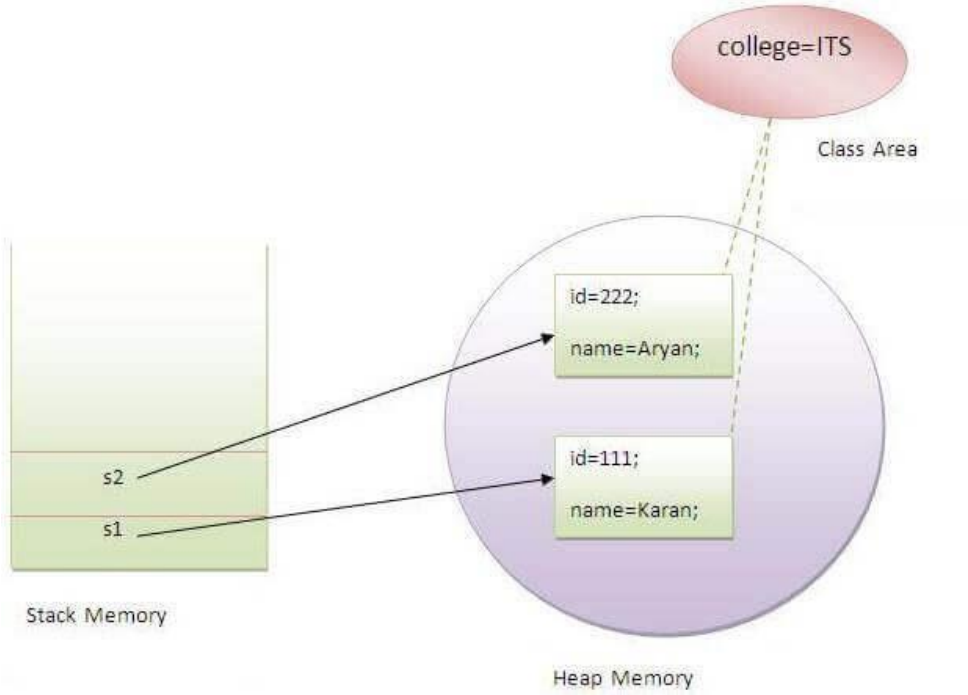
Understanding the problem without static variable

```
class Student{
     int rollno;
     String name;
     String college="ITS";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();  // 111 Karan ITS
 s2.display();  // 222 Aryan ITS
 }
}
```



Program of the counter without static variable - In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.
class Counter{
int count=0;//will get memory each time when the instance is created
Counter(){
count++;//incrementing value
System.out.println(count);
}

public static void main(String args[]){
//Creating objects
Counter c1=new Counter();  // 1
Counter c2=new Counter();  // 1
Counter c3=new Counter();  // 1
}
}
```

Program of counter by static variable - As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();  // 1
Counter2 c2=new Counter2();  // 2
```

```
Counter2 c3=new Counter2();  // 3
}
}
```

2) Java static method - If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
//Java Program to demonstrate the use of a static method.
class Student{
     int rollno;
     String name;
     static String college = "ITS";
     //static method to change the value of static variable
     static void change(){
     college = "BBDIT";
     }
     //constructor to initialize the variable
     Student(int r, String n){
     rollno = r;
     name = n;
     }
     //method to display values
     void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
    public static void main(String args[]){
    Student.change();//calling change method
    //creating objects
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    Student s3 = new Student(333,"Sonoo");
    //calling display method
    s1.display();  // 111 Karan BBDIT
    s2.display();  // 222 Aryan BBDIT
    s3.display();  // 333 Sonoo BBDIT
    }
}
```

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method

class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);  // 125
  }
}
```

Restrictions for the static method - There are two main restrictions for the static method. They are:

1. The static method cannot use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{
 int a=40;//non static
```

```
 public static void main(String args[]){
  System.out.println(a); //  Compile Time Error
 }
}
```

Q) Why is the Java main method static? It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output:

```
static block is invoked
Hello main
```

Q) Can we execute a program without main() method? No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

```
class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

Output:

```
static block is invoked
Since JDK 1.7 and above, output would be:
Error: Main method not found in class A3, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Java Access Modifiers

What are Access Modifiers? In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example,

```
class Animal {
    public void method1() {...}
   private void method2() {...}
}
```

In the above example, we have declared 2 methods: method1() and method2(). Here,

- method1 is public - This means it can be accessed by other classes.
- method2 is private - This means it cannot be accessed by other classes.

Note the keyword public and private. These are access modifiers in Java. They are also known as visibility modifiers.
Note: You cannot set the access modifier of getters methods.

Types of Access Modifier

Before you learn about types of access modifiers, make sure you know about Java Packages. There are four access modifiers keywords in Java and they are:

Modifier	Description
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Default Access Modifier - If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered. For example,

```
package defaultPackage;
class Logger {
    void message(){
        System.out.println("This is a message");
    }
}
```

Here, the Logger class has the default access modifier. And the class is visible to all the classes that belong to the defaultPackage package. However, if we try to use the Logger class in another class outside of defaultPackage, we will get a compilation error.

Private Access Modifier - When variables and methods are declared private, they cannot be accessed outside of the class. For example,

```
class Data {
    // private variable
    private String name;
}

public class Main {
    public static void main(String[] main){
        // create an object of Data
        Data d = new Data();
        // access private variable and field from another class
        d.name = "Programiz";
    }
}
```

In the above example, we have declared a private variable named name. When we run the program, we will get the following error:

```
Main.java:18: error: name has private access in Data
        d.name = "Programiz";
         ^
```

The error is generated because we are trying to access the private variable of the Data class from the Main class.

You might be wondering what if we need to access those private variables. In this case, we can use the getters and setters method. For example,

```
class Data {
    private String name;
    // getter method
    public String getName() {
        return this.name;
    }
    // setter method
    public void setName(String name) {
        this.name= name;
    }
```

```
}
public class Main {
    public static void main(String[] main){
        Data d = new Data();
        // access the private variable using the getter and setter
        d.setName("Programiz");
        System.out.println(d.getName());
    }
}
```

Output: The name is Programiz

In the above example, we have a private variable named name. In order to access the variable from the outer class, we have used methods: getName() and setName(). These methods are called getter and setter in Java.

Here, we have used the setter method (setName()) to assign value to the variable and the getter method (getName()) to access the variable.

We have used this keyword inside the setName() to refer to the variable of the class.

Note: We cannot declare classes and interfaces private in Java. However, the nested classes can be declared private. To learn more, visit Java Nested and Inner Class.

Protected Access Modifier - When methods and data members are declared protected, we can access them within the same package as well as from subclasses. For example,

```
class Animal {
    // protected method
    protected void display() {
        System.out.println("I am an animal");
    }
}

class Dog extends Animal {
    public static void main(String[] args) {
        // create an object of Dog class
        Dog dog = new Dog();
         // access protected method
        dog.display();
    }
}
```

Output: I am an animal

In the above example, we have a protected method named display() inside the Animal class. The Animal class is inherited by the Dog class. We then created an object dog of the Dog class. Using the object we tried to access the protected method of the parent class.

Since protected methods can be accessed from the child classes, we are able to access the method of Animal class from the Dog class.

Note: We cannot declare classes or interfaces protected in Java.

Public Access Modifier - When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction. For example,

```
// Animal.java file
// public class
public class Animal {
    // public variable
    public int legCount;
    // public method
    public void display() {
```

```
        System.out.println("I am an animal.");
        System.out.println("I have " + legCount + " legs.");
    }
}

// Main.java
public class Main {
    public static void main( String[] args ) {
        // accessing the public class
        Animal animal = new Animal();
        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }
}
```

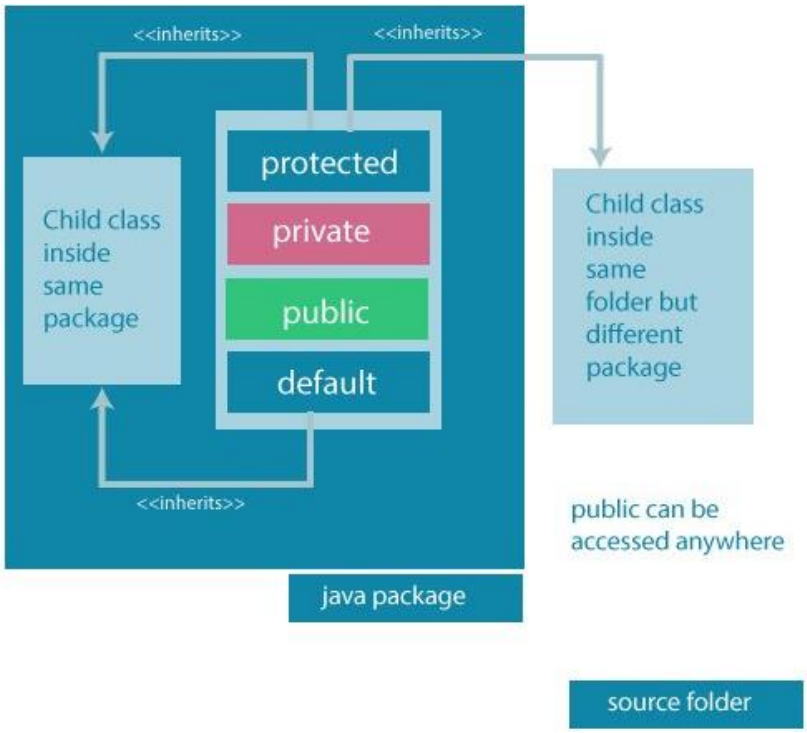
Output:

```
I am an animal.
I have 4 legs.
```

Here,

- The public class Animal is accessed from the Main class.
- The public variable legCount is accessed from the Main class.
- The public method display() is accessed from the Main class.

Access Modifiers Summarized in one figure

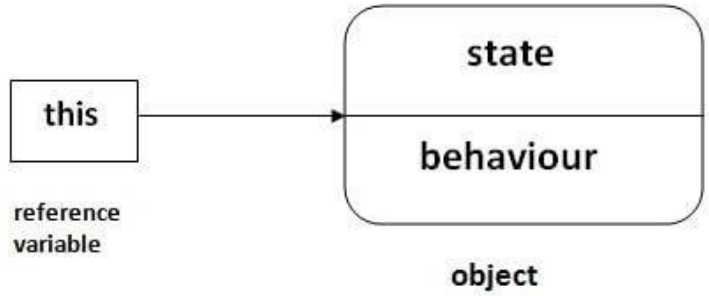


	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Accessibility of all Access Modifiers in Java - Access modifiers are mainly used for encapsulation. It can help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.

this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this keyword

Here is given the 6 usage of java this keyword.

Suggestion: If you are beginner to java, lookup only three usages of this keyword.

1) this: to refer current class instance variable - The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword - Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
```

```
Student s2=new Student(112,"sumit",6000f);
s1.display();  // 0 null 0.0
s2.display();  // 0 null 0.0
}}
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();  // 111 ankit 5000.0
s2.display();  // 112 sumit 6000.0
}}
```

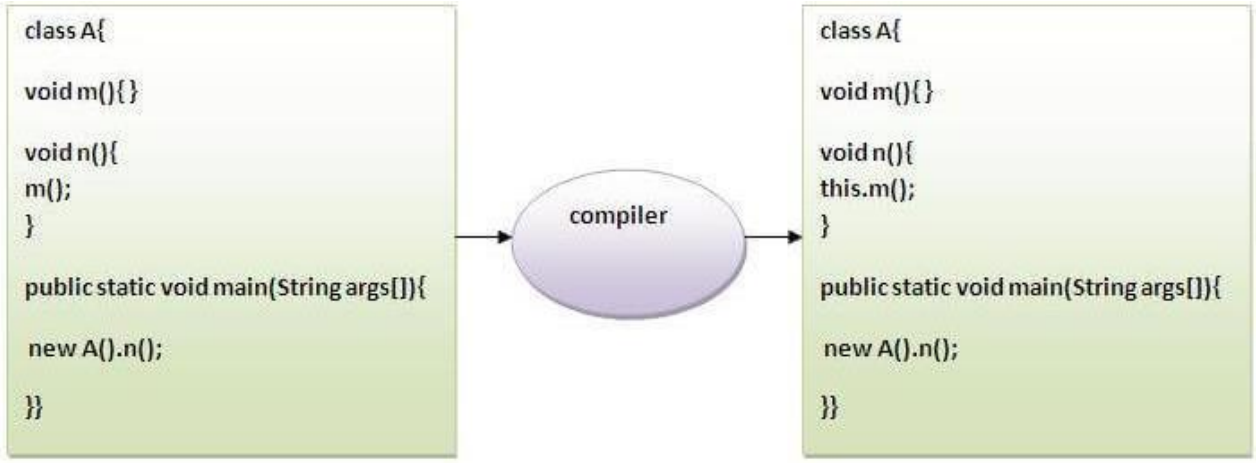
If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();  // 111 ankit 5000.0
s2.display();  // 112 sumit 6000.0
}}
```

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

2) this: to invoke current class method - You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example




```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor - The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
```

```
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

Real usage of this() constructor call - The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit java 0.0
112 sumit java 6000.0
```

Rule: Call to this() must be the first statement in constructor.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);//C.T.Error
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis8{
```

```
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output: Compile Time Error: Call to this must be first statement in constructor

4) this: to pass as an argument in the method - The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

Output: method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call - We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}

class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

Output:10

6) this keyword can be used to return current class instance - We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name(){
```

```
return this;
}
```

Example of this keyword that you return as a statement from the method

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

Output:

```
Hello java
Proving this keyword
```

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
class A5{
void m(){
System.out.println(this);//prints same reference ID
}
public static void main(String args[]){
A5 obj=new A5();
System.out.println(obj);//prints the reference ID
obj.m();
}
}
```

Output:

```
A5@22b3ea59
A5@22b3ea59
```

A Stack Class

While the Box class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP), one of OOP's most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a "data engine."

No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.

Here is a class called **Stack** that implements a stack for integers:

```
// this class defines an integer stack that can hold 10 values
class Stack {
    int stck[] = new int[10];
    int tos;

    // initialize top-of-stack
    Stack(){
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item){
        if(tos == 9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop(){
        if(tos < 0){
            System.out.println("Stack underflow.");
            return 0;
        } else
            return stck[tos--];
    }
}

class StackTest{
    public static void main(String args[]){
        Stack myStack1 = new Stack();
        Stack myStack2 = new Stack();
        // push some numbesr onto the stack
        for(int i = 0; i < 10; i++) myStack1.push(i);
        for(int i = 10; i < 20; i++) myStack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1: ");
        for(int i = 0; i < 10; i++)
            System.out.println(myStack1.pop());

        System.out.println("Stack in mystack2: ");
        for(int i = 0; i < 10; i++)
            System.out.println(myStack2.pop());
    }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
```

```
14
13
12
11
10
```

Arrays Revisited - Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array.

```
// improved stack class that uses the length array member.
class Stack{
    private int stck[];
    private int tos;

    // allocate and initialize stack
    Stack(int size){
        stck = new int[size];
        tos = -1;
    }

    // push an item onto the stack
    void push(int item){
        if(tos == stck.length - 1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // pop an item from the stack
    int pop(){
        if(tos < 0){
            System.out.println("Stack underflow.");
            return 0;
        } else
            return stck[tos--];
    }
}

public class StackTest2 {
    public static void main(String args[]){
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);
        // push some numbers onto the stack
        for(int i = 0; i < 5; i++) mystack1.push(i);
        for(int i = 0; i < 8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1: ");
        for(int i = 0; i < 5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2: ");
        for(int i = 0; i < 8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

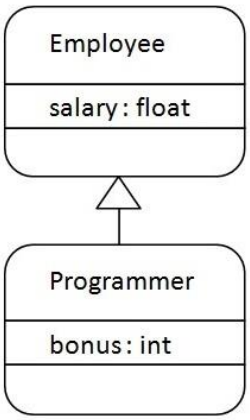
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Inheritance Example - In this example, we have a base class Teacher and a sub class PhysicsTeacher. Since class PhysicsTeacher extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class. Here we have collegeName, designation and work() method which are

common to all the teachers so we have declared them in the base class, this way the child classes like MathTeacher, MusicTeacher and PhysicsTeacher do not need to write this code and can be used directly from base class.

```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
        System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
        PhysicsTeacher obj = new PhysicsTeacher();
        System.out.println(obj.collegeName);
        System.out.println(obj.designation);
        System.out.println(obj.mainSubject);
        obj.does();
   }
}
```

Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Based on the above example we can say that PhysicsTeacher **IS-A** Teacher. This means that a child class has IS-A relationship with the parent class. This is inheritance is known as **IS-A relationship** between child and parent class

Note: The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

```
class Teacher {
   private String designation = "Teacher";
   private String collegeName = "Beginnersbook";
   public String getDesignation() {
        return designation;
   }
   protected void setDesignation(String designation) {
        this.designation = designation;
   }
   protected String getCollegeName() {
        return collegeName;
   }
   protected void setCollegeName(String collegeName) {
        this.collegeName = collegeName;
   }
   void does(){
        System.out.println("Teaching");
   }
}

public class JavaExample extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
        JavaExample obj = new JavaExample();
        /* Note: we are not accessing the data members
         * directly we are using public getter method
         * to access the private members of parent class
```

```
         */
        System.out.println(obj.getCollegeName());
        System.out.println(obj.getDesignation());
        System.out.println(obj.mainSubject);
        obj.does();
    }
}
```

The output is:

```
Beginnersbook
Teacher
Physics
Teaching
```

The important point to note in the above example is that the child class is able to access the private members of parent class through **protected methods** of parent class. When we make a instance variable(data member) or method **protected**, this means that they are accessible only in the class itself and in child class. These public, protected, private etc. are all access specifiers and we will discuss them in the coming tutorials.

A More Practical Example - Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the Box class developed in the preceding chapter will be extended to include a fourth component called weight. Thus, the new class will contain a box's width, height, depth, and weight.

```
class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob){ // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d){
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box(){
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len){
        width = height = depth = len;
    }

    // compute and return volume
    double volume(){
        return width * height * depth;
    }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box{
    double weight; // weight of box
```

```
    // constructor for BoxWeight
    BoxWeight(double w, double h, double d, double m){
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}

class DemoBoxWeight{
    public static void main(String args[]){
        BoxWeight box1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight box2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;

        vol = box1.volume();
        System.out.println("Volume of box1 is " + vol);
        System.out.println("Weight of box2 is " + box1.weight);

        vol = box2.volume();
        System.out.println("Volume of box2 is " + vol);
        System.out.println("Weight of box2 is " + box2.weight);
    }
}
```

The output from this program is shown here:

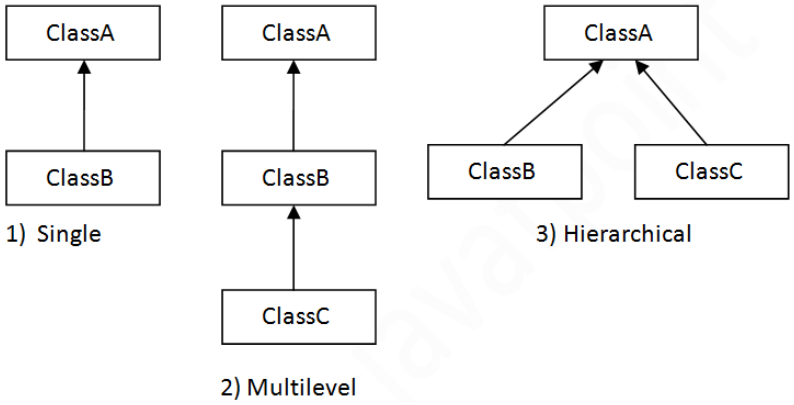
```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

BoxWeight inherits all of the characteristics of Box and adds to them the weight component. It is not necessary for BoxWeight to re-create all of the features found in Box. It can simply extend Box to meet its own purposes.

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

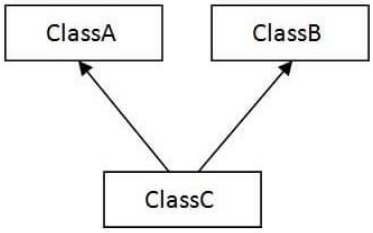
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

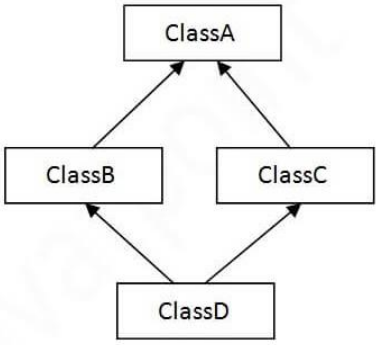


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

Single Inheritance Example - When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example - When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example - When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Practical Example of Creating a Multilevel Hierarchy

```
class Box {
    private double width;
    private double height;
    private double depth;

    Box(Box ob){
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    Box(double w, double h, double d){
        width = w;
        height = h;
        depth = d;
    }

    Box(){
        width = -1;
        height = -1;
        depth = -1;
    }

    Box(double len){
        width = height = depth = len;
    }

    double volume(){
        return width * height * depth;
    }
}
```

```
class BoxWeight extends Box{
    double weight;
    BoxWeight(BoxWeight ob){
        super(ob);
        weight = ob.weight;
    }

    BoxWeight(double w, double h, double d, double m){
        super(w, h, d);
        weight = m;
    }

    BoxWeight(){
        super();
        weight = -1;
    }

    BoxWeight(double len, double m){
        super(len);
        weight = m;
    }
}

class Shipment extends BoxWeight{
    double cost;
    // construct clone of an object
    Shipment(Shipment ob){
        super(ob);
        cost = ob.cost;
    }
    // constructor when all parameters are specified
    Shipment(double w, double h, double d, double m , double c){
        super(w, h, d, m);
        cost = c;
    }
    Shipment(){
        super();
        cost = -1;
    }
    // constructor used when cube is created
    Shipment(double len, double m ,double c){
        super(len , m);
        cost = c;
    }
}

class DemoShipment{
    public static void main(String args[]){
        Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
        double vol;

        vol = shipment1.volume();
        System.out.println("Volume of shipment1 is " + vol);
        System.out.println("Weight of shipment1 is " + shipment1.weight);
        System.out.println("Shipping cost: $" + shipment1.cost);
        vol = shipment2.volume();
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is " + shipment2.weight);
        System.out.println("Shipping cost: $" + shipment2.cost);
    }
}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

Q) Why multiple inheritance is not supported in java? To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

```
Compile Time Error
```