

Unit-1

Java Basics:-

- 1. Introduction**
- 2. History of Java**
- 3. Java buzzwords**
- 4. Data types**
- 5. Variables**
- 6. Scope and Life time of variables**
- 7. Arrays**
- 8. Operators**
- 9. Expressions**
- 10. Control statements**
- 11. Type conversion and casting**
- 12. Simple Java programs**
- 13. Concepts of classes**
- 14. Objects**
- 15. Constructors**
- 16. Methods**
- 17. Access control**
- 18. This keyword**
- 19. Garbage collection**
- 20. Overloading methods**
- 21. Parameter passing**
- 22. Recursion**
- 23. Exploring String Class**

1.Introduction:-

1. What is Java?
2. Application
3. Java Platforms / Editions
4. The requirement for Java Hello World Example
5. Creating Hello World Example
6. Parameters used in First Java Program
7. how many ways we can write a Java program?
8. How to set path in Java?
9. OOPS concepts?
10. Memory Allocation?

1.What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

2.Application

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

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irtc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics

8. Games, etc.

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

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.

3.Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](#), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB(Enterprise Java Bean), [JPA](#)(Java Persistence API (Application Programming Interface)), etc.

3) Java ME (Java Micro Edition)

It is a micro platform that is dedicated to mobile applications.

4) JavaFX

It is used to develop rich internet applications. It uses a lightweight user interface API.

4.The requirement for Java Hello World Example

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, [download the JDK](#) and install it.
- Set path of the jdk/bin directory. <http://www.javatpoint.com/how-to-set-path-in-java>
- Create the Java program
- Compile and run the Java program

5.Creating Hello World Example

Let's create the hello java program:

```
1. class Simple{
2.     public static void main(String args[]){
3.         System.out.println("Hello Java");
4.     }
5. }
```

Save the above file as Simple.java.

To compile:

javac Simple.java

To execute:

java Simple

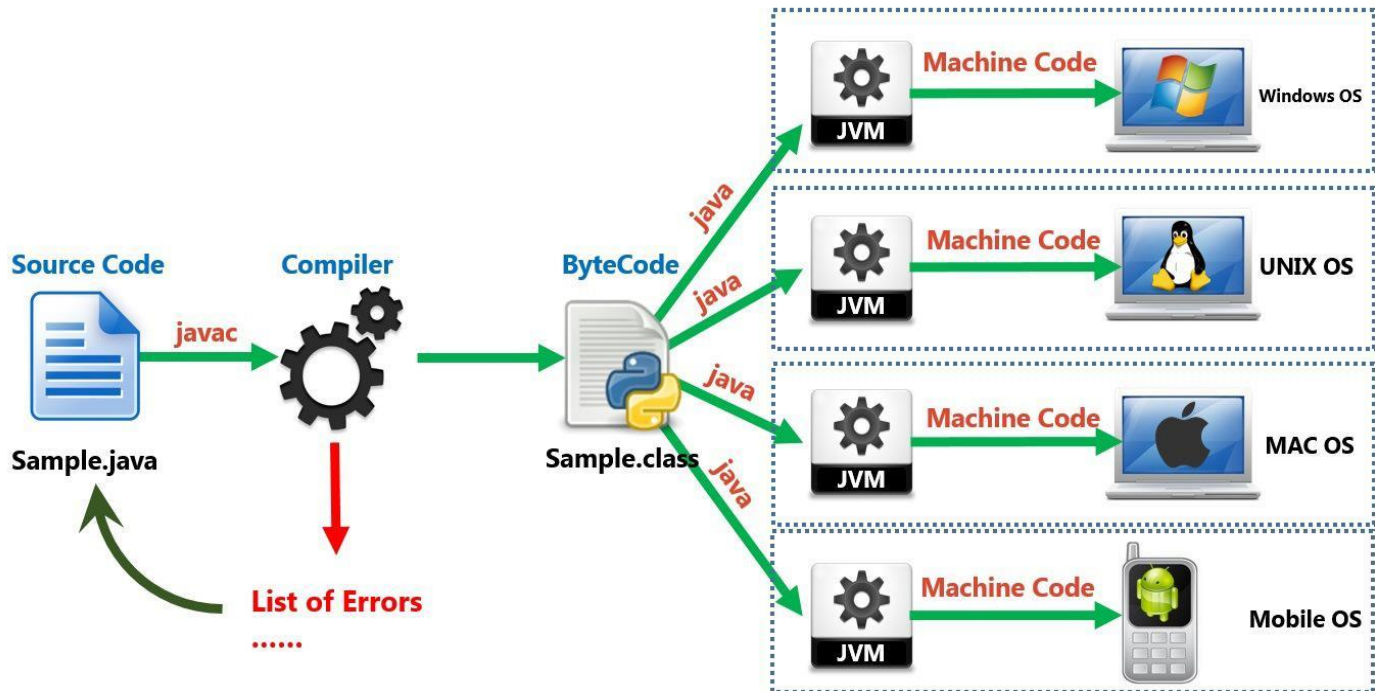
Output:

Hello Java

Execution Process of Java Program

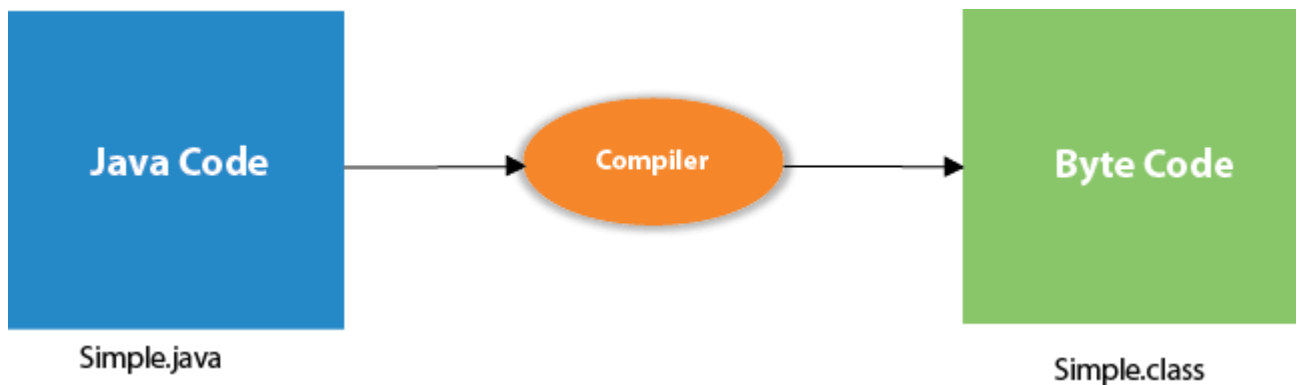
The following three steps are used to create and execute a java program.

- Create a source code (.java file).
- Compile the source code using javac command.
- Run or execute .class file using java command.



Compilation Flow:

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



Can you save a Java source file by another name than the class name?

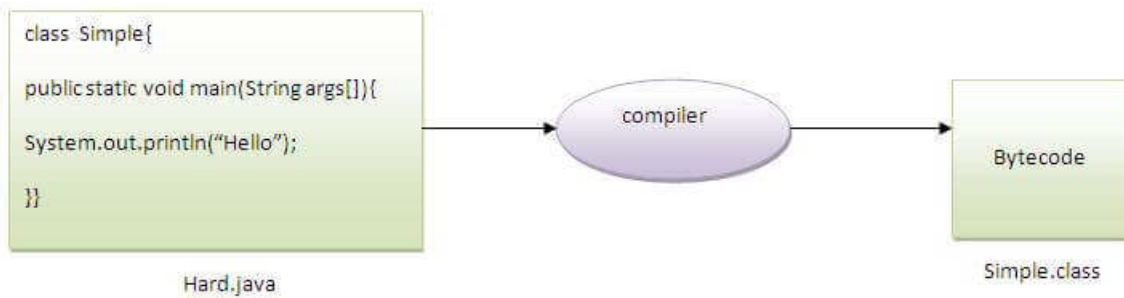
Yes, if the class is not public. It is explained in the figure given below:

Compile time:

Javac Hard.java

Runtime:

Java FirstProgram



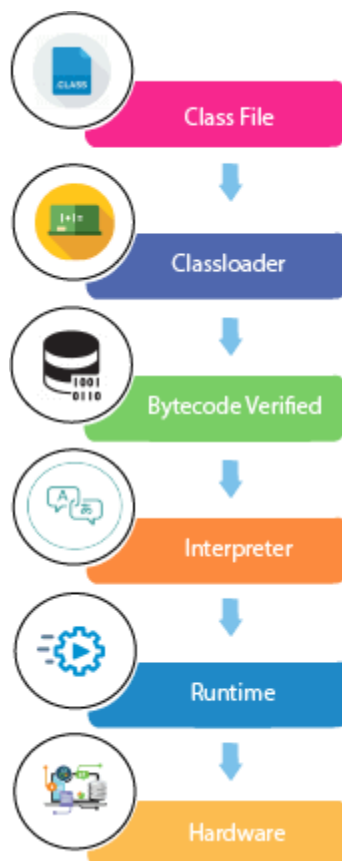
What happens at runtime?

At runtime, the following steps are performed:

Classloader: It is the subsystem of JVM that is used to load class files.

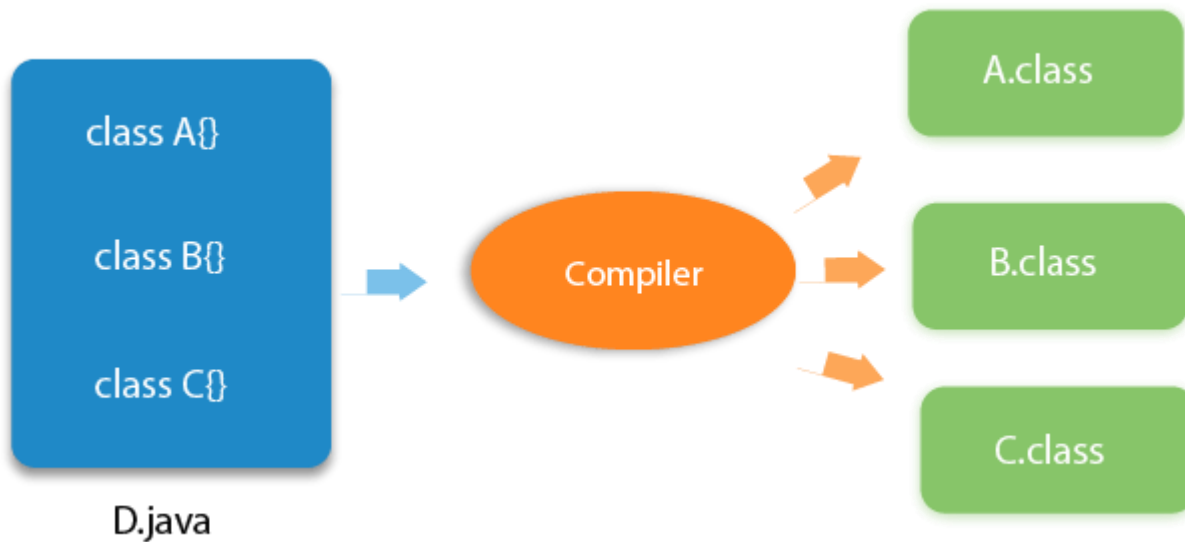
Bytecode Verifier: Checks the code fragments for illegal code that can violate access rights to objects.

Interpreter: Read bytecode stream then execute the instructions.



Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



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*.

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.

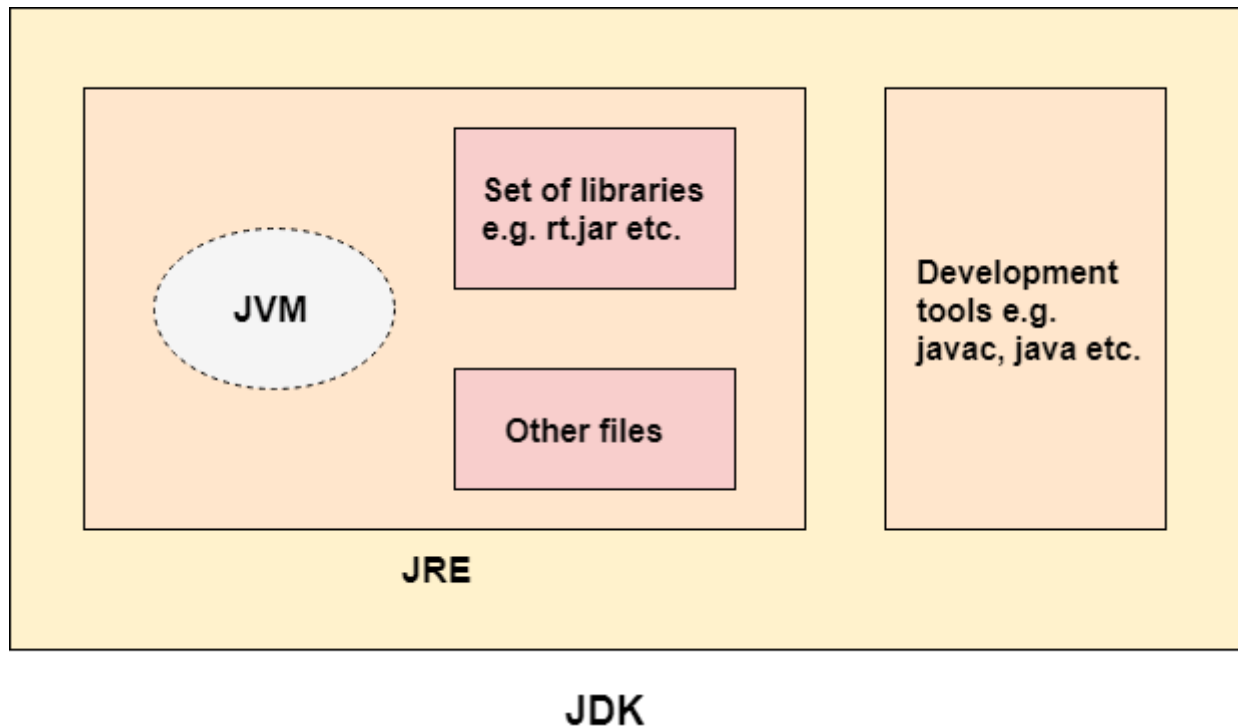
JDK

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:

- o Standard Edition Java Platform
- o Enterprise Edition Java Platform
- o 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.

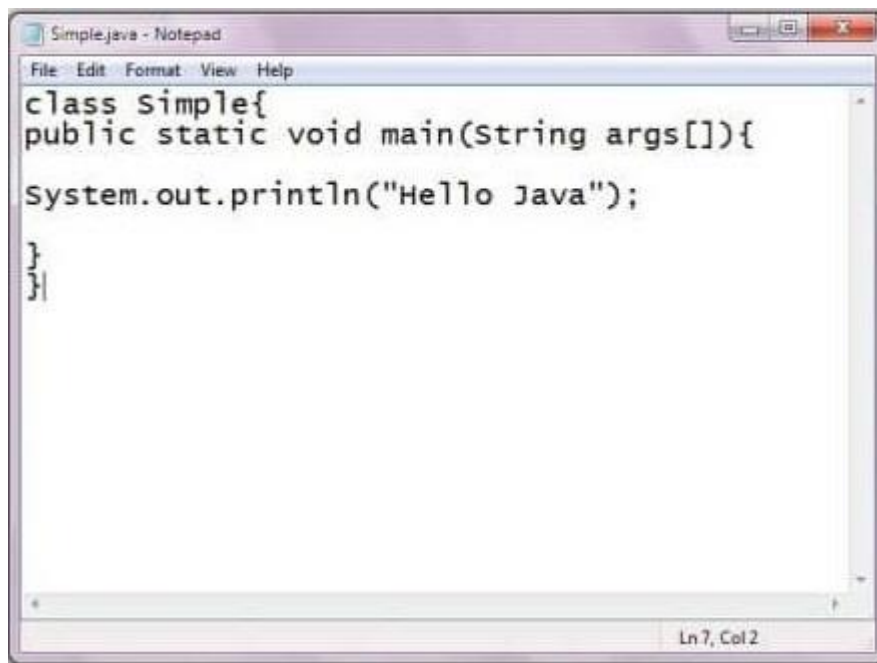


6.Parameters used in First Java Program

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

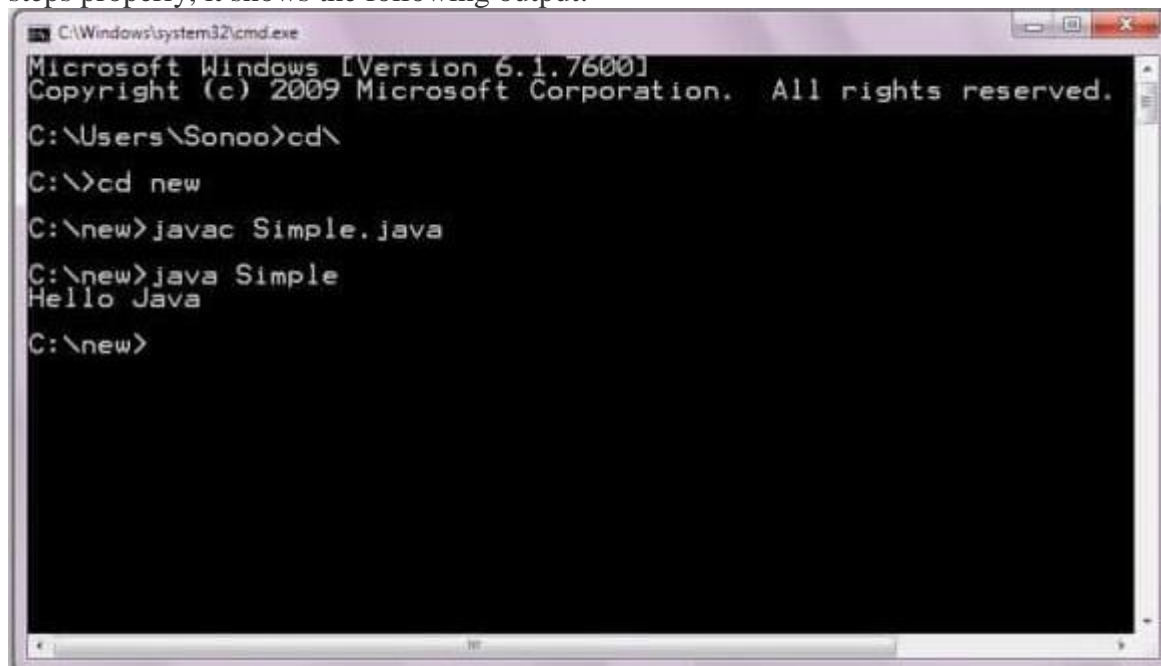
- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that 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](#). We will discuss it in coming section.
- **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. We will discuss the internal working of [System.out.println\(\)](#) statement in the coming section.

To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> Notepad** and write a simple program as we have shown below:



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

As displayed in the above diagram, write the simple program of Java in notepad and saved it as Simple.java. In order to compile and run the above program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**. When we have done with all the steps properly, it shows the following output:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
C:\new>java Simple
Hello Java
C:\new>
```

7.how many ways we can write a Java program?

Valid Java main() method signature

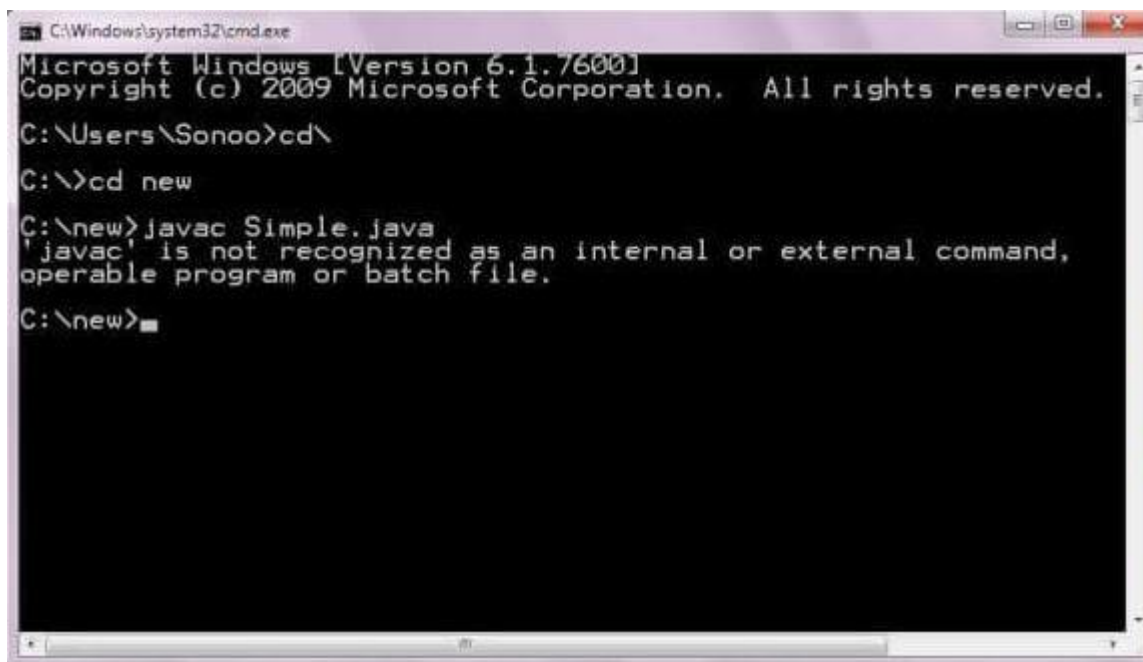
1. **public static void** main(String[] args)
2. **public static void** main(String []args)
3. **public static void** main(String args[])
4. **public static void** main(String... args)
5. **static public void** main(String[] args)
6. **public static final void** main(String[] args)
7. **final public static void** main(String[] args)

Invalid Java main() method signature

1. **public void** main(String[] args)
2. **static void** main(String[] args)
3. **public void static** main(String[] args)
4. **abstract public static void** main(String[] args)

Resolving an error "javac is not recognized as an internal or external command"?

If there occurs a problem like displayed in the below figure, you need to set a path. Since DOS doesn't recognize javac and java as internal or external command. To overcome this problem, we need to set a path. The path is not required in a case where you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path. Click here for [How to set path in java](#).

A screenshot of a Windows command prompt window. The title bar reads 'C:\Windows\system32\cmd.exe'. The window content shows the following text:

```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.
C:\new>
```

8.How to set path in Java

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1. Temporary
2. Permanent

1) How to set the Temporary Path of JDK in Windows

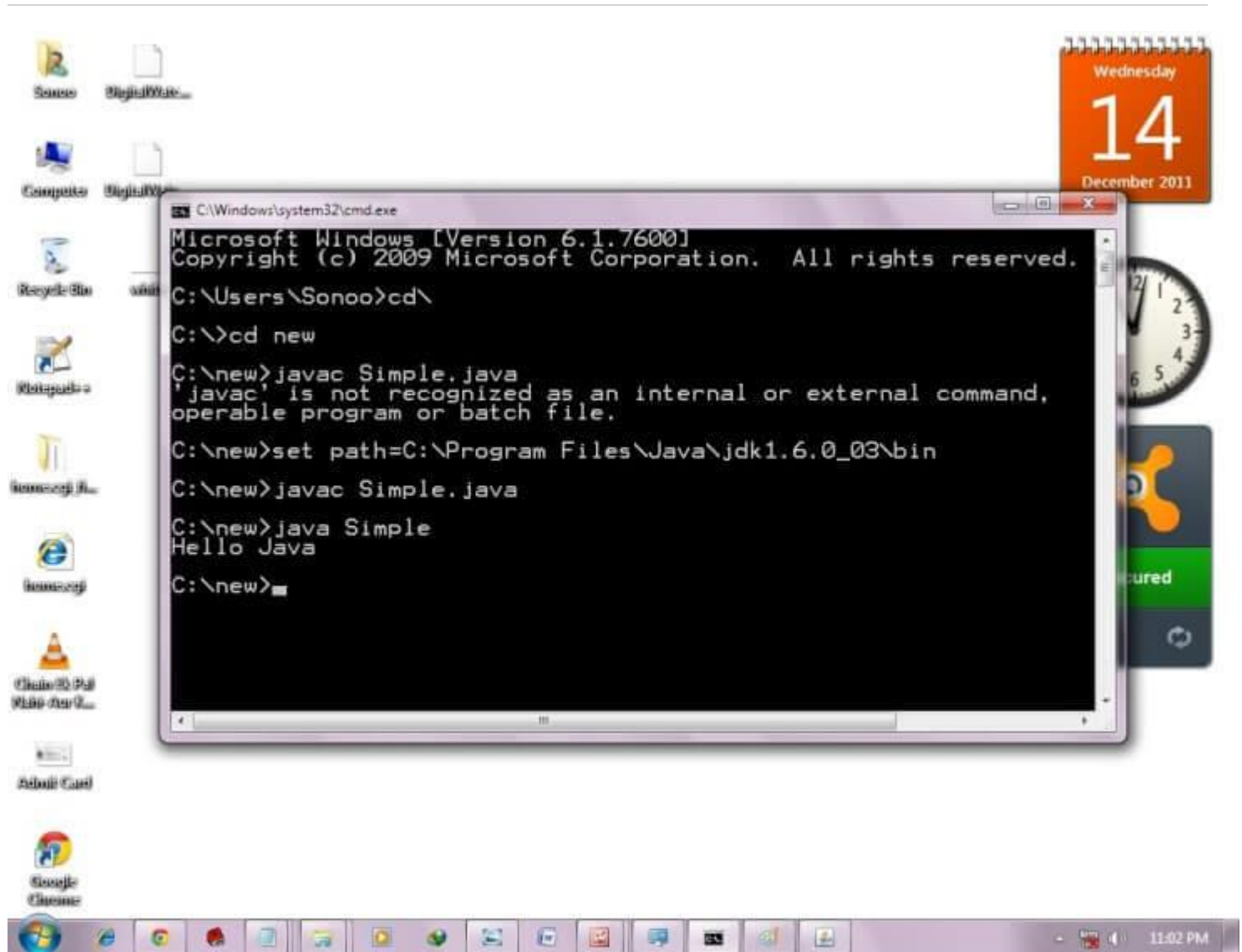
To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

set path=C:\Program Files\Java\jdk1.6.0_23\bin

Let's see it in the figure given below:



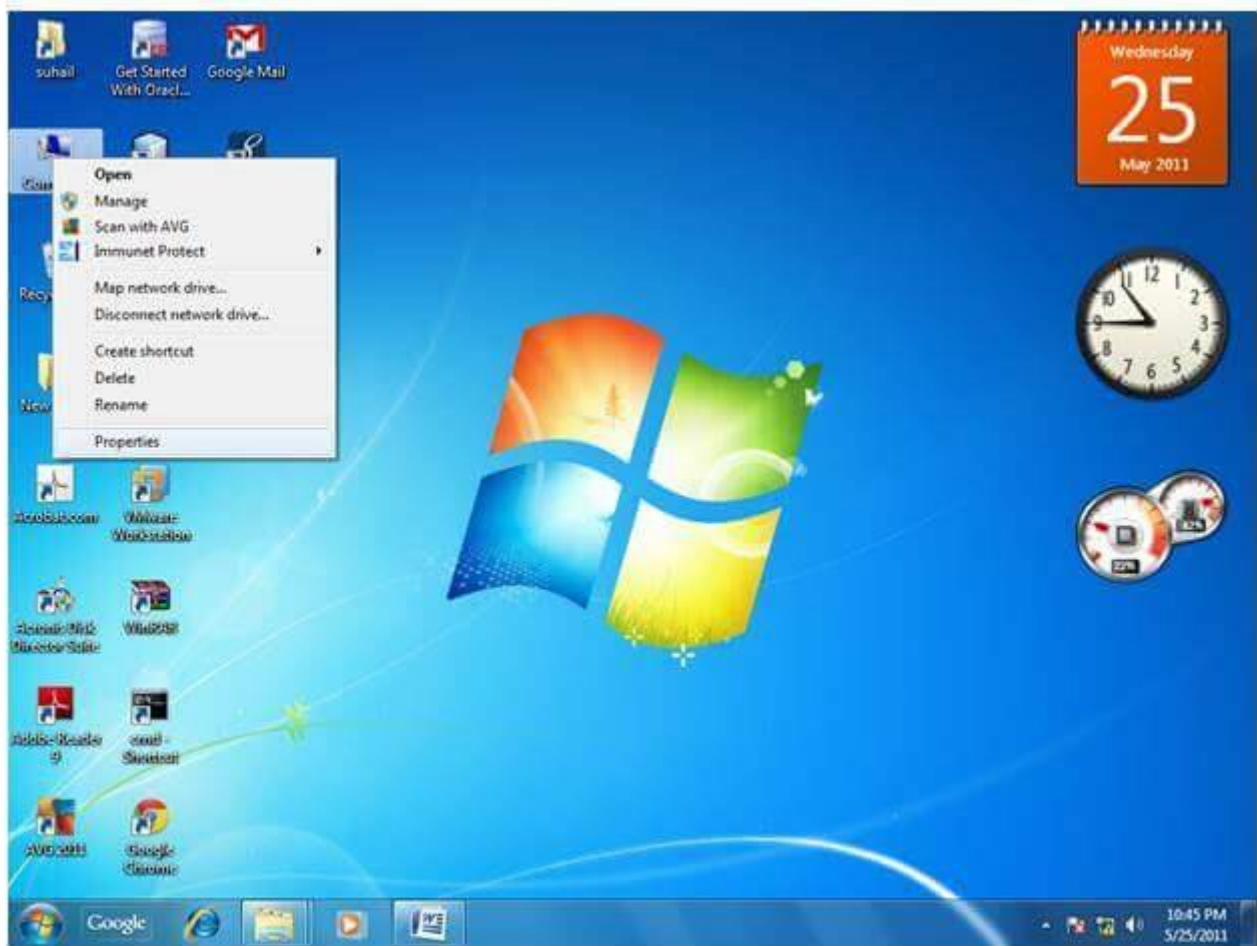
2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

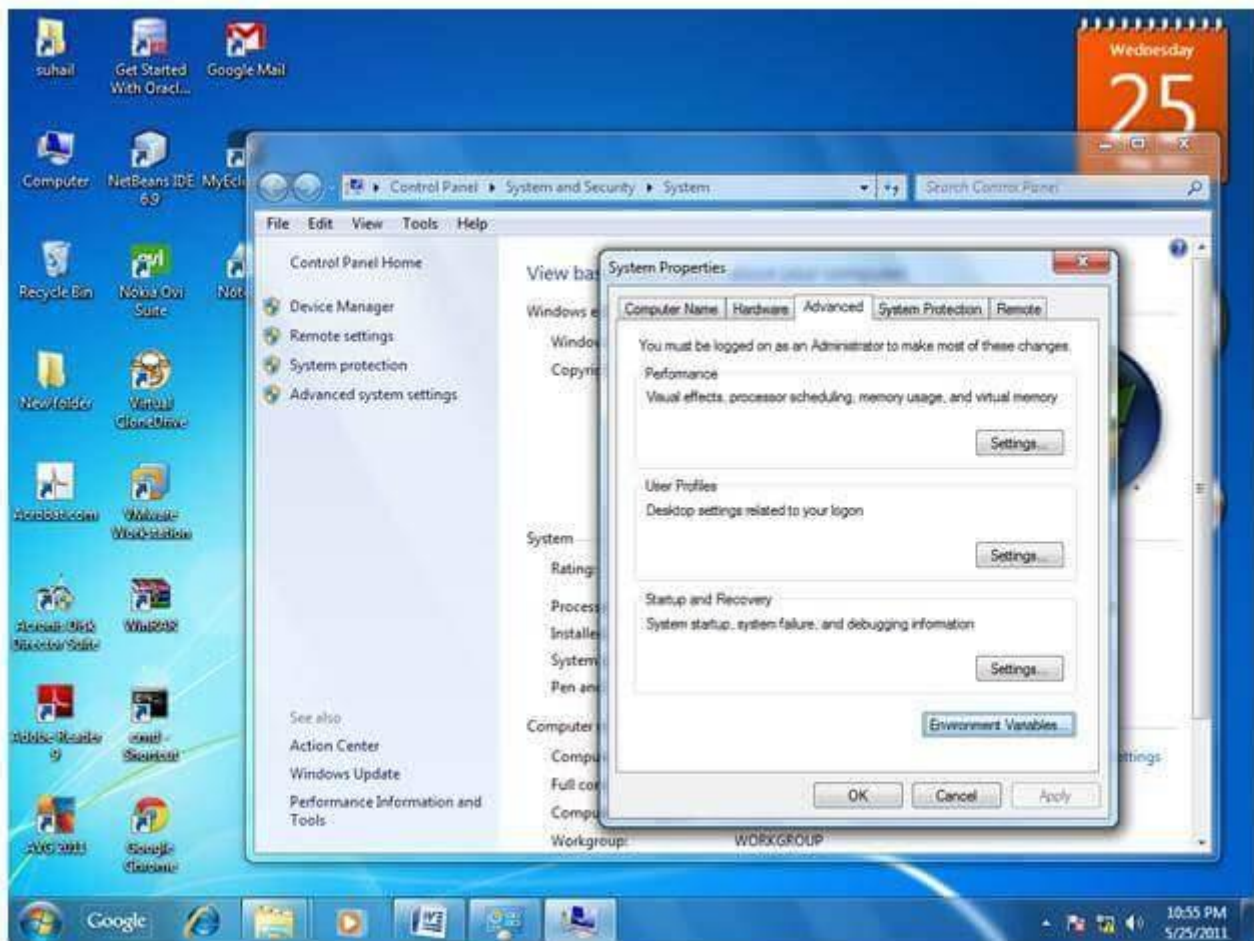
1) Go to MyComputer properties



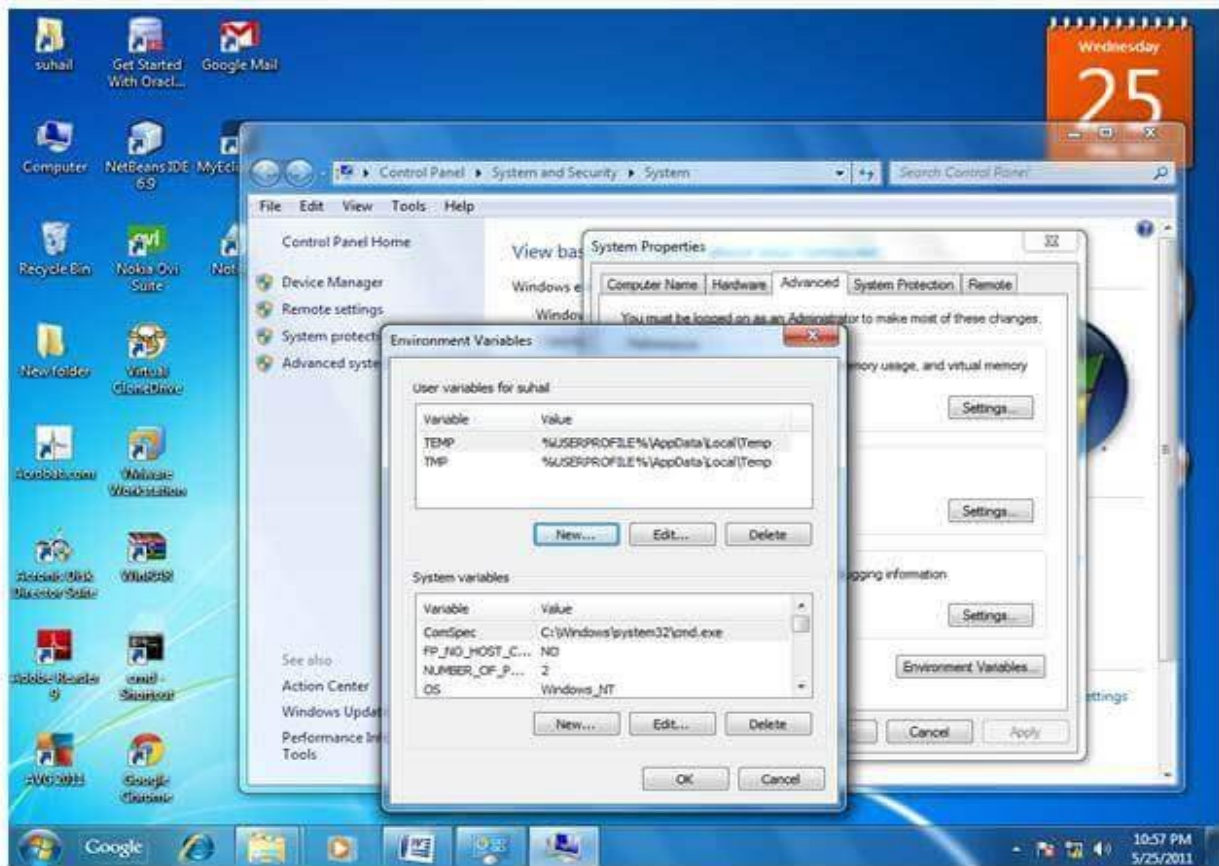
2) Click on the advanced tab



3) Click on environment variables



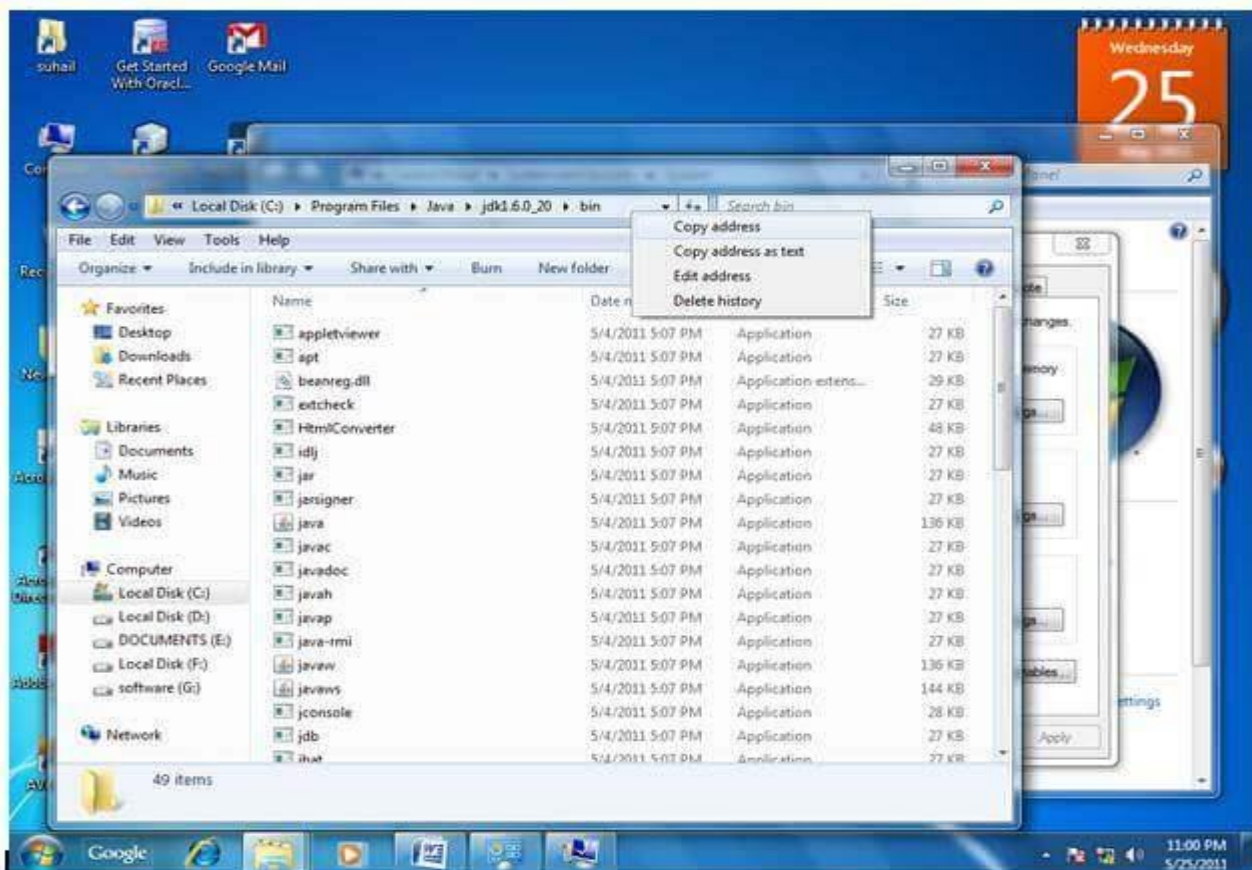
4) Click on the new tab of user variables



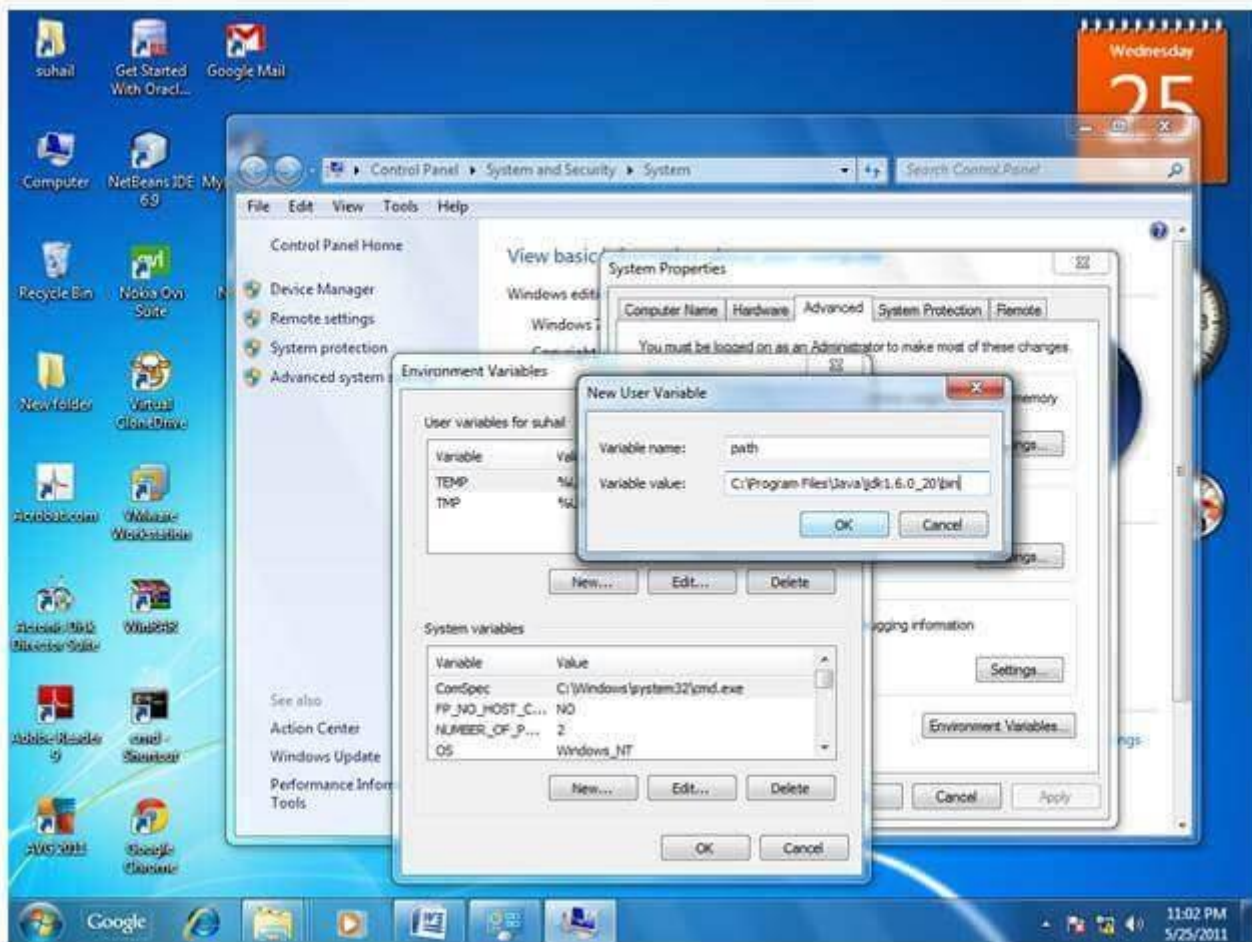
5) Write the path in the variable name



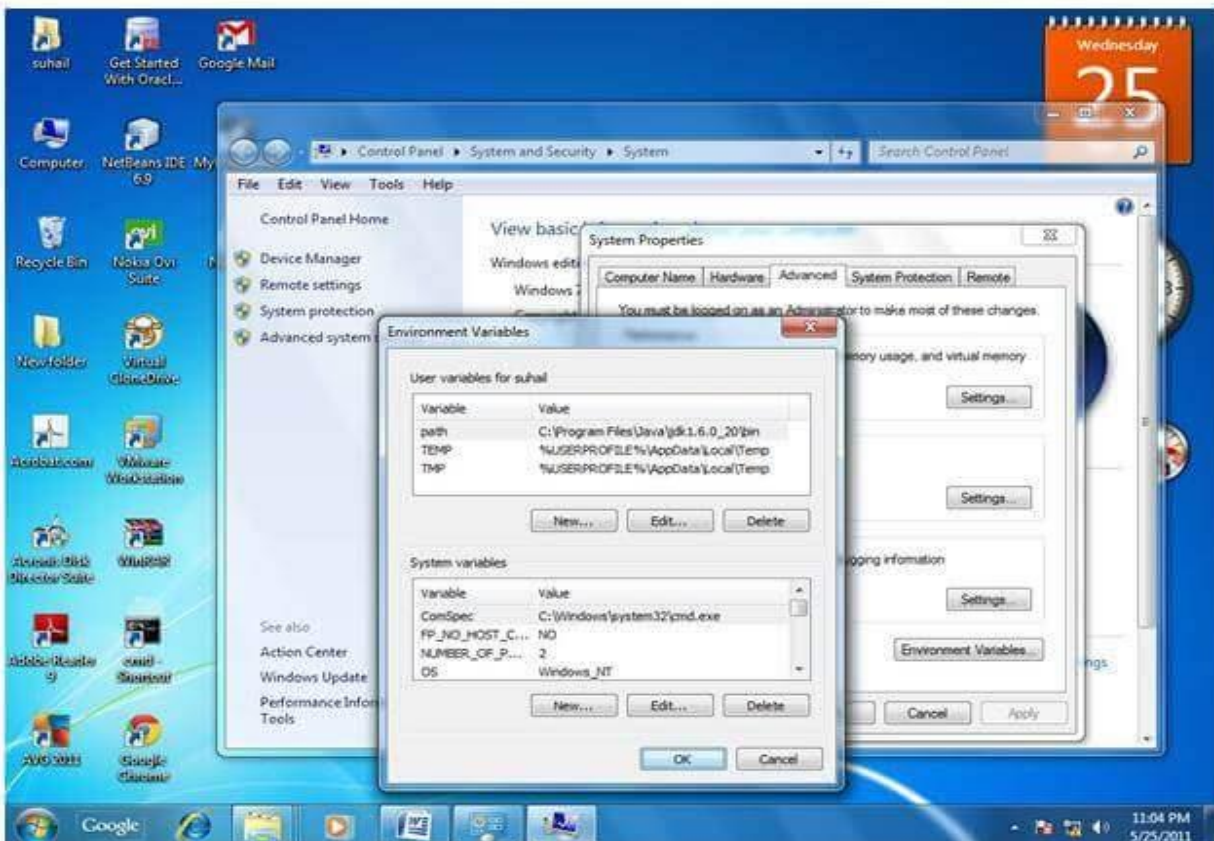
6) Copy the path of bin folder



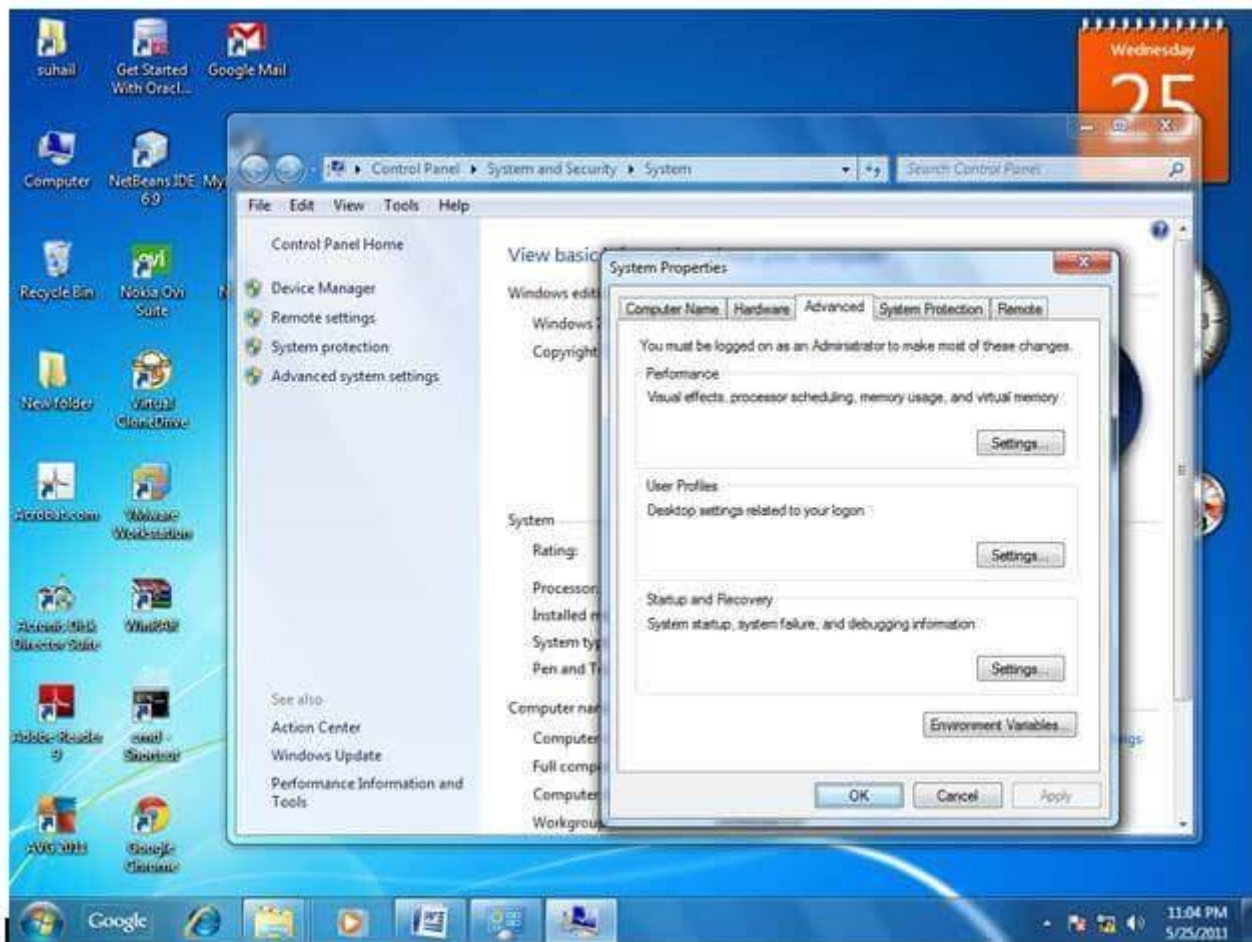
7) Paste path of bin folder in the variable value



8) Click on ok button



9) Click on ok button



Now your permanent path is set. You can now execute any program of java from any drive.

Difference between Pop(Procedure Oriented Programming) and OOP(Object Oriented Programming)

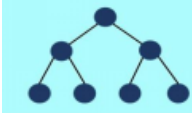

Sr.No.	On the basis of	POP	OOP
1	Definition	In procedure programming, we break down a task into a set of variables and routines after that, it is carried out every step one by one.	Object-oriented programming is a programming paradigm in which every program is follows the concept of object.
2	Problem solving approach	 <p>Top-down Approach</p> <p>The focus is on breaking the bigger problem into smaller one and then repeat the process with each problem.</p>	 <p>Bottom-up Approach</p> <p>The focus is on identifying and resolving smallest problems and then integrating them together to solve the bigger problem.</p>
3	Subdivision	<p>The POP program is divided into small functions.</p> <p>Ex:- #include<stdio.h></p> <pre>Void add() { } int main(){ add(); }</pre>	<p>OOP program are divided into objects.</p> <p>Ex:-class Add{</p> <pre>..... }</pre> <p>Add a=new Add();//object</p>
4	Access Specifiers	No concept of access specifiers	Make use of access specifiers such as public, private, protected etc.,
5	Security	Very less security is available	More Security because using garbage collection.
6	Orientation	Procedural programming is structure/procedure-oriented.	It is object-oriented.
7	Code Reusability	No code reusability in procedural programming.	Due to inheritance, concept of code reusability exists in object-oriented.
8	Virtual Class	In Procedural programming, There is no concept of virtual class.	There exists a concept of virtual classes in object-oriented programming.
9	Overloading	In procedural programming overloading is not possible.	There is a concept of function overloading and operator overloading in object oriented programming.
10	Examples	Examples of procedural programming are fortran, algol, cobol, basic, pascal and c.	Examples of OOP are java, c++, c#, python, JavaScript, etc.,

Table.1.Difference between Pop and OOP

Difference Between c , c++ and java?

C definition:- C is a general-purpose, structured, procedural, and high-level programming language developed by **Dennis Ritchie** in 1972 at **Bell Laboratories**.

C++ definition: - C++ is an object-oriented, general-purpose, programming language developed by **Bjarne Stroustrup** at Bell Labs in 1979.

Java definition: - Java is also an object-oriented, class-based, static, strong, robust, safe, and high-level programming language. It was developed by **James Gosling** in 1995.

S.N.	Basis	C	C++	Java
1	Origin	The C language is based on BCPL.	The C++ language is based on the C language.	The Java programming language is based on both C and C++.
2	Programming Pattern	It is a procedural language.	It is an object-oriented programming language.	It is a pure object-oriented programming language.
3	Approach	It uses the top-down approach.	It uses the bottom-up approach.	It also uses the bottom-up approach.
4	Dynamic or Static	It is a static programming language.	It is also a static programming language.	It is a dynamic programming language.
5	Code Execution	The code is executed directly.	The code is executed directly.	The code is executed by the JVM.
6	Platform Dependency	It is platform dependent.	It is platform dependent.	It is platform-independent because of byte code.
7	Translator	It uses a compiler only to translate the code into machine language.	It also uses a compiler only to translate the code into machine language.	Java uses both compiler and interpreter and it is also known as an interpreted language.
8	File Generation	It generates the .exe, files.	It generates .exe file.	It generates .class file.

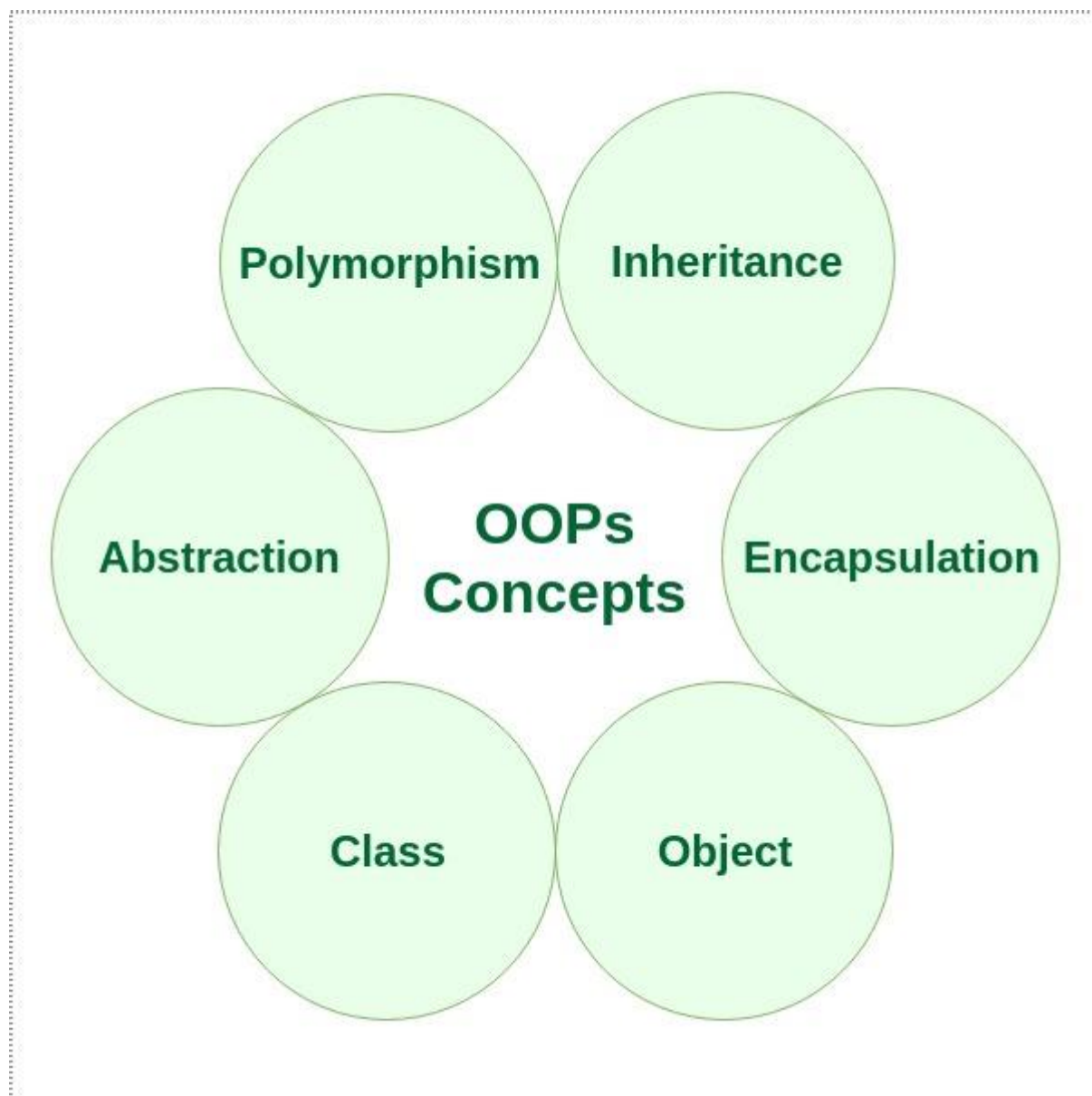
9	Number of Keyword	There are 32 keywords in the C language.	There are 60 keywords in the C++ language.	There are 52 keywords in the Java language.
10	Source File Extension	The source file has a .c extension.	The source file has a .cpp extension.	The source file has a .java extension.
11	Pointer Concept	It supports pointer.	It also supports pointer.	Java does not support the pointer concept because of security.
12	Union and Structure Datatype	It supports union and structure data types.	It also supports union and structure data types.	It does not support union and structure data types.
13	Pre-processor Directives	It uses pre-processor directives such as #include, #define, etc.	It uses pre-processor directives such as #include, #define, #header, etc.	It does not use directives but uses packages.
14	Constructor/ Destructor	It does not support constructor and destructor.	It supports both constructor and destructor.	It supports constructors only.
15	Exception Handling	It does not support exception handling.	It supports exception handling.	It also supports exception handling.
16	Memory Management	It uses the calloc(), malloc(), free(), and realloc() methods to manage the memory.	It uses new and delete operator to manage the memory.	It uses a garbage collector to manage the memory.
17	Overloading	It does not support the overloading concept.	Method and operator overloading can be achieved.	Only method overloading can be achieved.
18	goto Statement	It supports the goto statement.	It also supports the goto statement.	It does not support the goto statements.

19	Used for	It is widely used to develop drivers and operating systems.	It is widely used for operating system, developing browsers.	It is used to develop web applications, mobile applications, and windows applications.
20	Array Size	An array should be declared with size. For example, int num[10].	An array should be declared with size.	An array can be declared without declaring the size. For example, int num[].

8. OOPS Concepts?

OOPS concepts are as follows:

1. [Class](#)
2. [Object](#)
3. [Method](#) and [method passing](#)
4. Pillars of OOPs
 - [Abstraction](#)
 - [Encapsulation](#)
 - [Inheritance](#)
 - [Polymorphism](#)
 - Compile-time polymorphism
 - Runtime polymorphism



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. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

1. **Modifiers:** A class can be public or have default access (Refer to [this](#) for details).
2. **Class name:** The class name should begin with the initial letter capitalized by convention.
3. **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.
4. **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.
5. **Body:** The class body is surrounded by braces, { }.

An **object** is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
2. **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity:** It is a unique name given to an object that enables it to interact with other objects.
4. **Method:** A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping it, which is why they are considered **time savers**. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

class and objects one simple java program :

```

public class GFG {

    static String Employee_name;
    static float Employee_salary;

    static void set(String n, float p) {
        Employee_name = n;
        Employee_salary = p;
    }

    static void get() {
        System.out.println("Employee name is: " +Employee_name );
        System.out.println("Employee CTC is: " + Employee_salary);
    }

    public static void main(String args[]) {
        GFG.set("Rathod Avinash", 10000.0f);
        GFG.get();
    }
}

```

Output

Employee name is: Rathod Avinash

Employee CTC is: 10000.0

Let us now discuss the 4 pillars of OOPs:

Pillar 1: [Abstraction](#)

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the car speed or applying brakes will stop the car, but he does not know how on pressing the accelerator, the speed is actually increasing. He does not know about the inner mechanism of the car or the implementation of the accelerators, brakes etc. in the car. This is what abstraction is.

In Java, abstraction is achieved by [interfaces](#) and [abstract classes](#). We can achieve 100% abstraction using interfaces.

The abstract method contains only method declaration but not implementation.

Demonstration of Abstract class

- **Java**

```

//abstract class
abstract class GFG{
    //abstract methods declaration
    abstract void add();
    abstract void mul();
    abstract void div();
}

```

Pillar 2: [Encapsulation](#)

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.

- In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.
- Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

Demonstration of Encapsulation:

- `Java`

```
//Encapsulation using private modifier

//Employee class contains private data called employee id and employee name
class Employee {
    private int empid;
    private String ename;
}
```

Pillar 3: [Inheritance](#)

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as “**is-a**” relationship.

Let us discuss some frequently used important terminologies:

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).
- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Demonstration of Inheritance :

- `Java`

```
//base class or parent class or super class
class A{
    //parent class methods
    void method1(){ }
    void method2(){ }
}

//derived class or child class or base class
class B extends A{ //Inherits parent class methods
    //child class methods
    void method3(){ }
    void method4(){ }
}
```

Pillar 4: [Polymorphism](#)

It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called **polymorphism**.

E.g.

- `Java`

```
sleep(1000) //millis

sleep(1000,2000) //millis,nanos
```

Polymorphism in java can be classified into two types:

1. Static / Compile-Time Polymorphism
2. Dynamic / Runtime Polymorphism

Note: Polymorphism in Java is mainly of 2 types:

1. [Overloading](#)
2. [Overriding](#)

Example

- Java

```
// Java program to Demonstrate Polymorphism
```

```
// This class will contain  
// 3 methods with same name,  
// yet the program will  
// compile & run successfully
```

```
public class Sum {
```

```
    // Overloaded sum().  
    // This sum takes two int parameters  
    public int sum(int x, int y)  
    {  
        return (x + y);  
    }
```

```
    // Overloaded sum().  
    // This sum takes three int parameters  
    public int sum(int x, int y, int z)  
    {  
        return (x + y + z);  
    }
```

```
    // Overloaded sum().  
    // This sum takes two double parameters  
    public double sum(double x, double y)  
    {  
        return (x + y);  
    }
```

```
    // Driver code  
    public static void main(String args[])  
    {  
        Sum s = new Sum();  
        System.out.println(s.sum(10, 20));  
        System.out.println(s.sum(10, 20, 30));  
        System.out.println(s.sum(10.5, 20.5));  
    }  
}
```

Output

30

60

31.0

10.Java Memory Allocation

What is Stack Memory?

Stack in java is a section of memory which contains methods, local variables, and reference variables. Stack memory is always referenced in Last-In-First-Out order. Local variables are created in the stack.

What is Heap Memory?

Heap is a section of memory which contains Objects and may also contain reference variables. Instance variables

Memory Allocation in Java

Memory Allocation in [Java](#) is the process in which the virtual memory sections are set aside in a program for storing the variables and instances of structures and classes. However, the memory isn't allocated to an object at declaration but only a reference is created. For the memory allocation of the object, new() method is used, so the object is always allocated memory on the heap.

The Java Memory Allocation is divided into following sections :

1. Heap
2. Stack
3. Code
4. [Static](#)

This division of memory is required for its effective management.

- The **code** section contains your **bytecode**.
- The **Stack** section of memory contains **methods, local variables, and reference variables**.
- The **Heap** section contains **Objects** (may also contain reference variables).
- The **Static** section contains **Static data/methods**.

Difference between Local and Instance Variable

Instance variable is declared **inside a class but not inside a method**

```
class Student{  
int num; // num is instance variable  
public void showData{ }
```

Local variable are declared **inside** a **method** including method **arguments**.

```
public void sum(int a){  
  
int x = int a + 3;  
  
// a , x are local variables;  
  
}
```

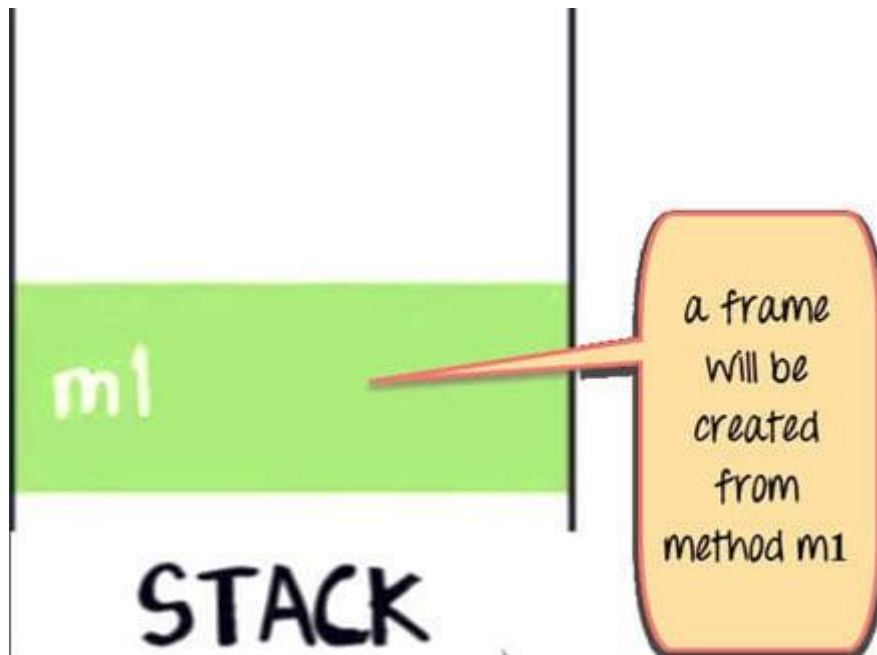
Difference between Stack and Heap

Let's take an example to understand this better.

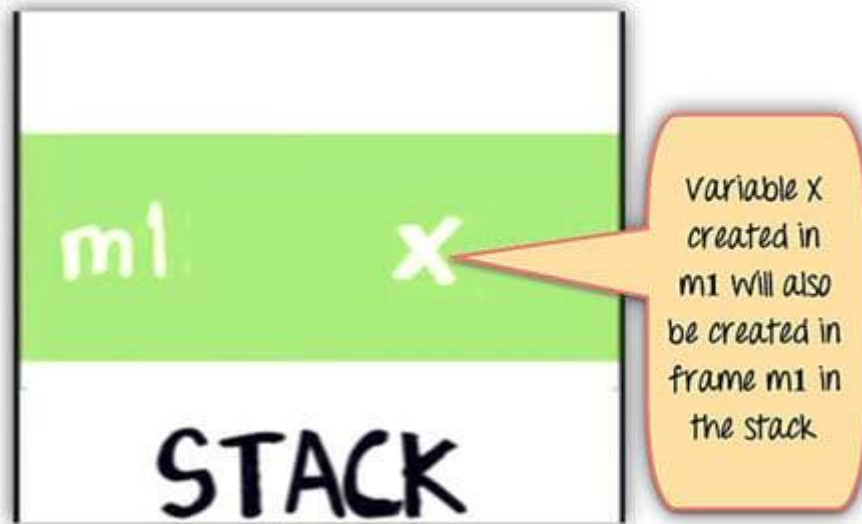
Consider that your main method calling method m1

```
public void m1 {  
int x=20  
}
```

In the stack java, a frame will be created from method m1.



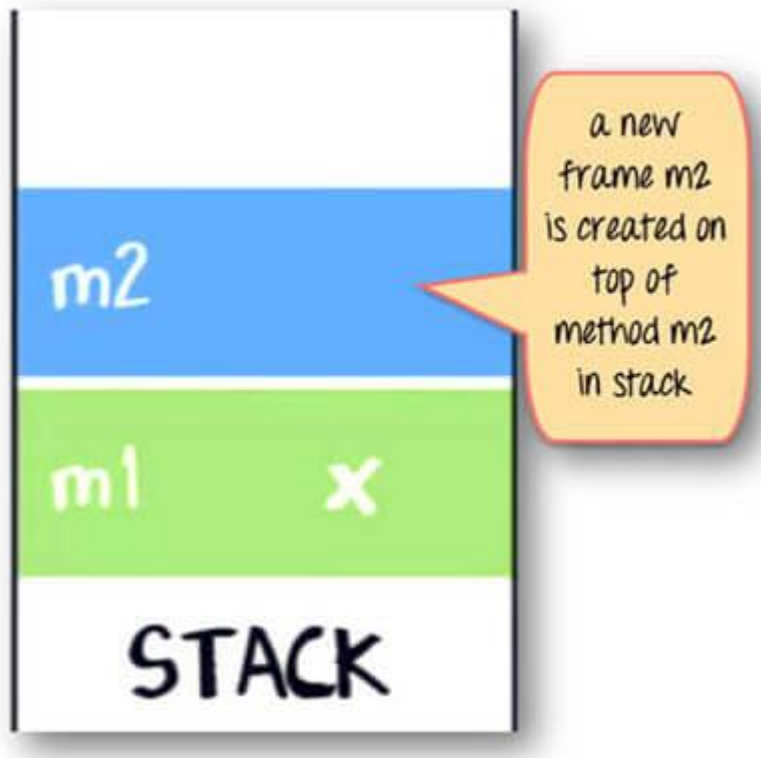
The variable X in m1 will also be created in the frame for m1 in the stack. (See image below).



Method m1 is calling method m2. In the stack java, a new frame is created for m2 on top of the frame m1.

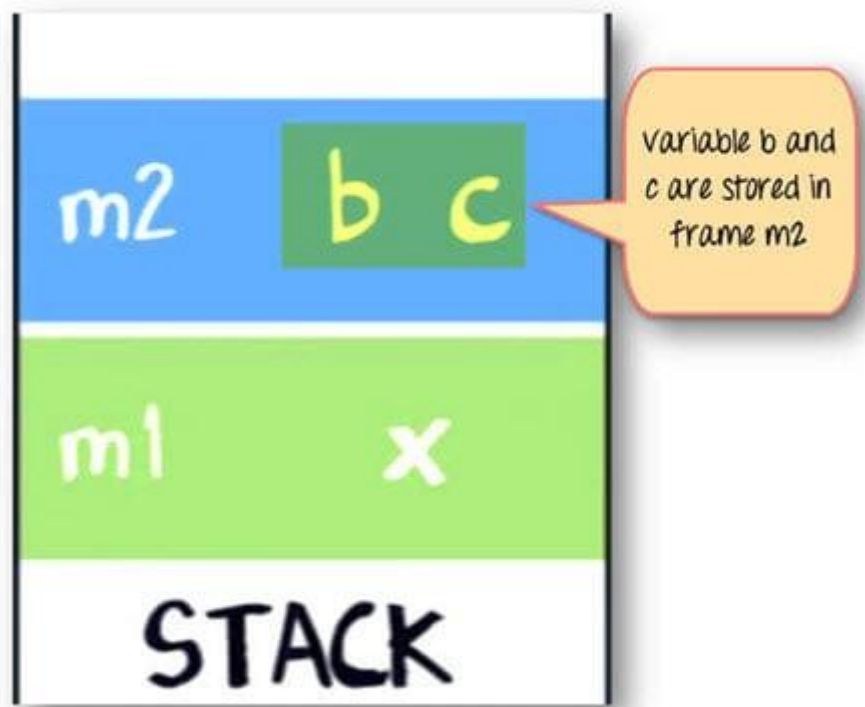
```
public void m1 {  
  int x = 20  
  m2(10)  
}  
public void m2(int b){
```

A speech bubble points from the underlined `m2(10)` to the right, containing the text: 'Method m1 is calling method m2'.



Variable b and c will also be created in a frame m2 in a stack.

```
public void m2(int b){  
    boolean c;  
}
```



Same method m2 is calling method m3. Again a frame m3 is created on the top of the stack (see image below).

```

public void m2(int b){
    boolean c;
    //more code
    m3();
}
public void m3()

```

another method m3 is called by method m2



Now let say our method m3 is creating an object for class “Account,” which has two instances variable int p and int q.

```

Account {
    Int p;
    Int q;
}

```

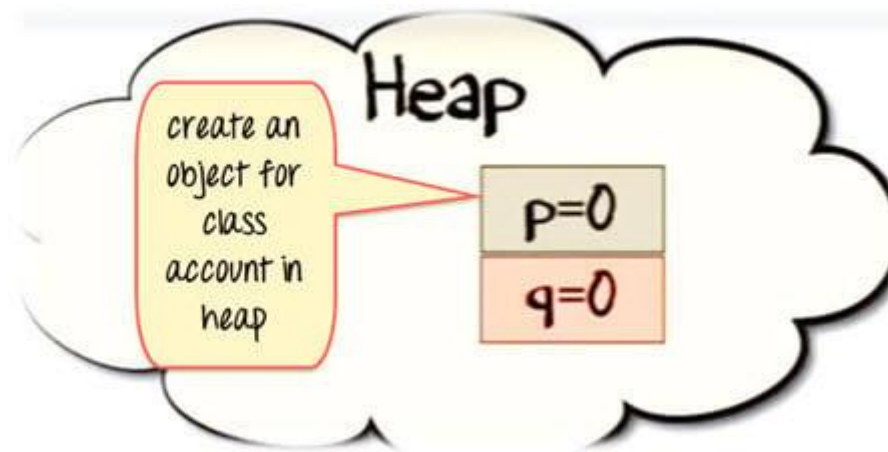
Here is the code for method m3

```

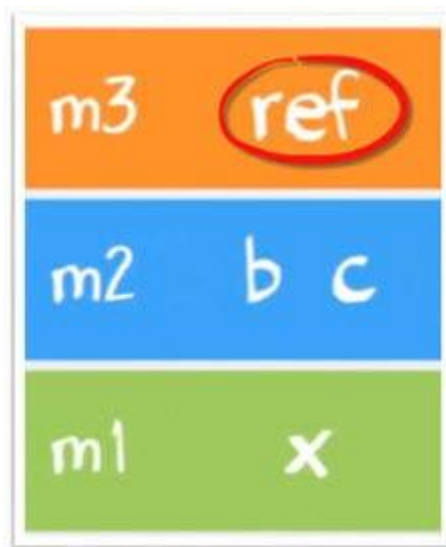
public void m3(){
    Account ref = new Account();
    // more code
}

```

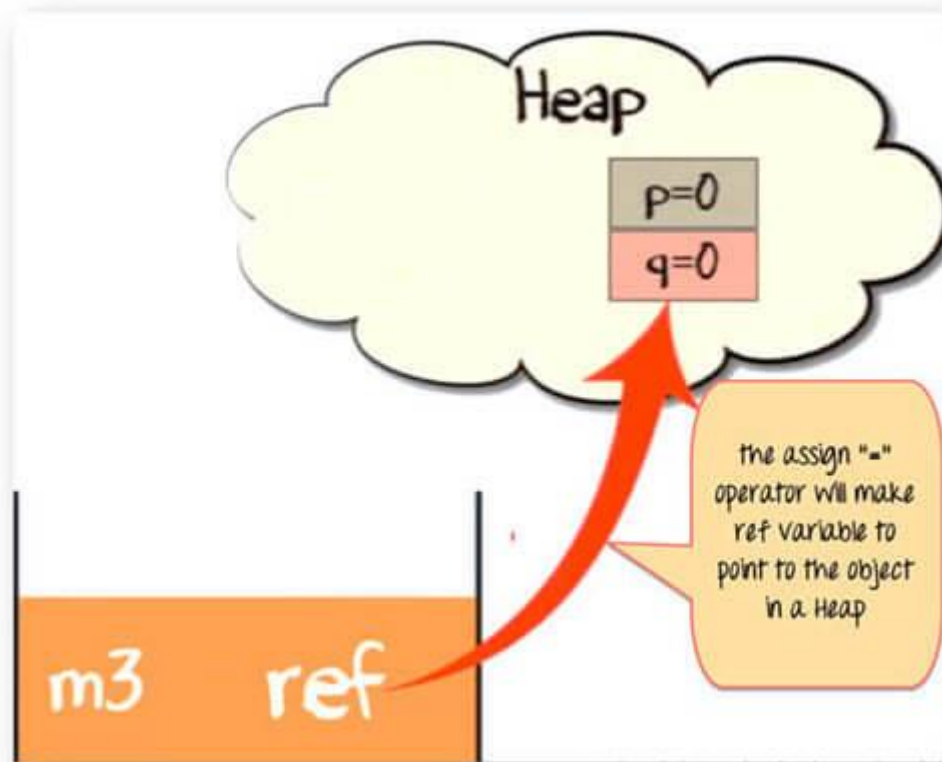
The statement new Account() will create an object of account in heap.



The reference variable “ref” will be created in a stack java.



The assignment “=” operator will make a reference variable to point to the object in the Heap.

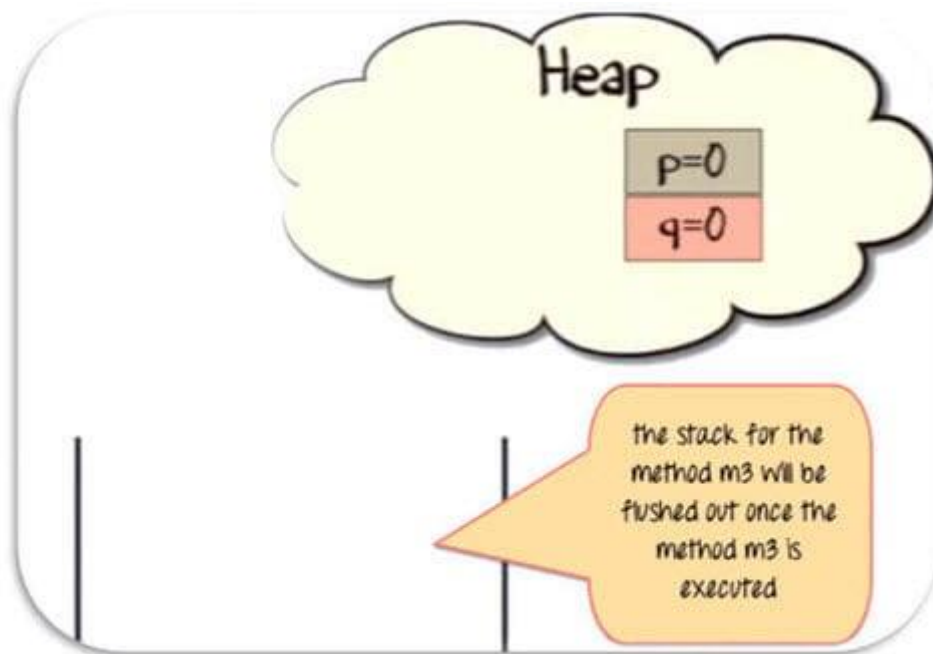


Once the method has completed its execution. The flow of control will go back to the calling method. Which in this case is method m2.

```
public void m2(int b){  
    boolean c;  
    //more code  
    m3();  
}  
public void m3()  
Account ref = new Account();  
//more code
```

Once the method has completed its execution, the flow of control will go back to the calling method

The stack from method m3 will be flushed out.



Since the reference variable will no longer be pointing to the object in the heap, it would be eligible for garbage collection.



Once method m2 has completed its execution. It will be popped out of the stack, and all its variable will be flushed and no longer be available for use.

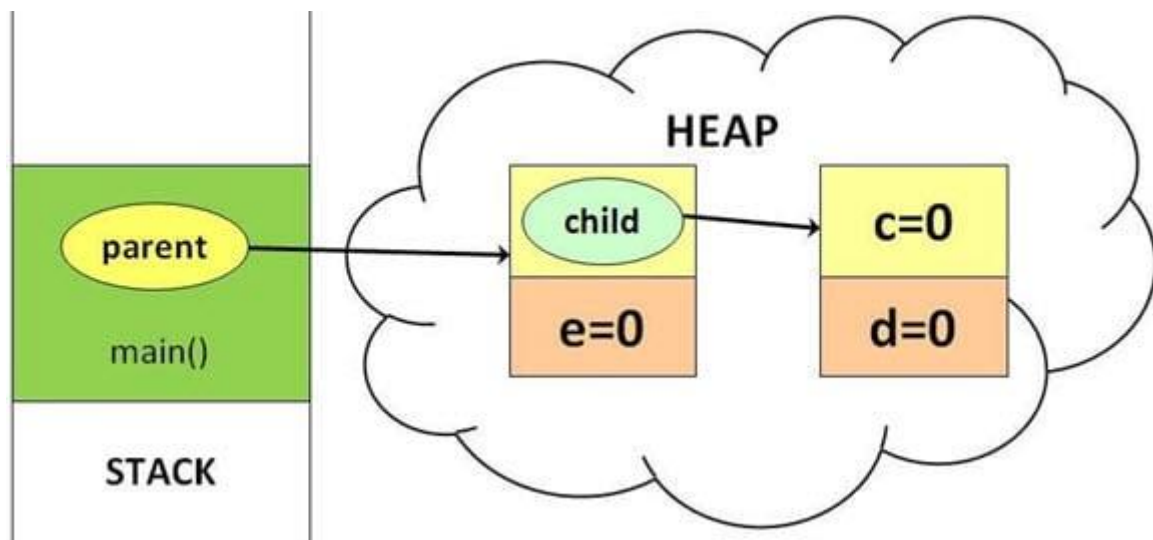
Likewise for method m1.

Eventually, the flow of control will return to the start point of the program. Which usually, is the “main” method.

What if Object has a reference as its instance variable?

```
public static void main(String args[]) {
    A parent = new A(); //more code } class A{ B child = new B(); int e; //more code } class B{ int c; int d; //more code }
```

In this case , the reference variable “child” will be created in heap ,which in turn will be pointing to its object, something like the diagram shown below.



1. 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.

he 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.



- 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 Javacoffee). 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.

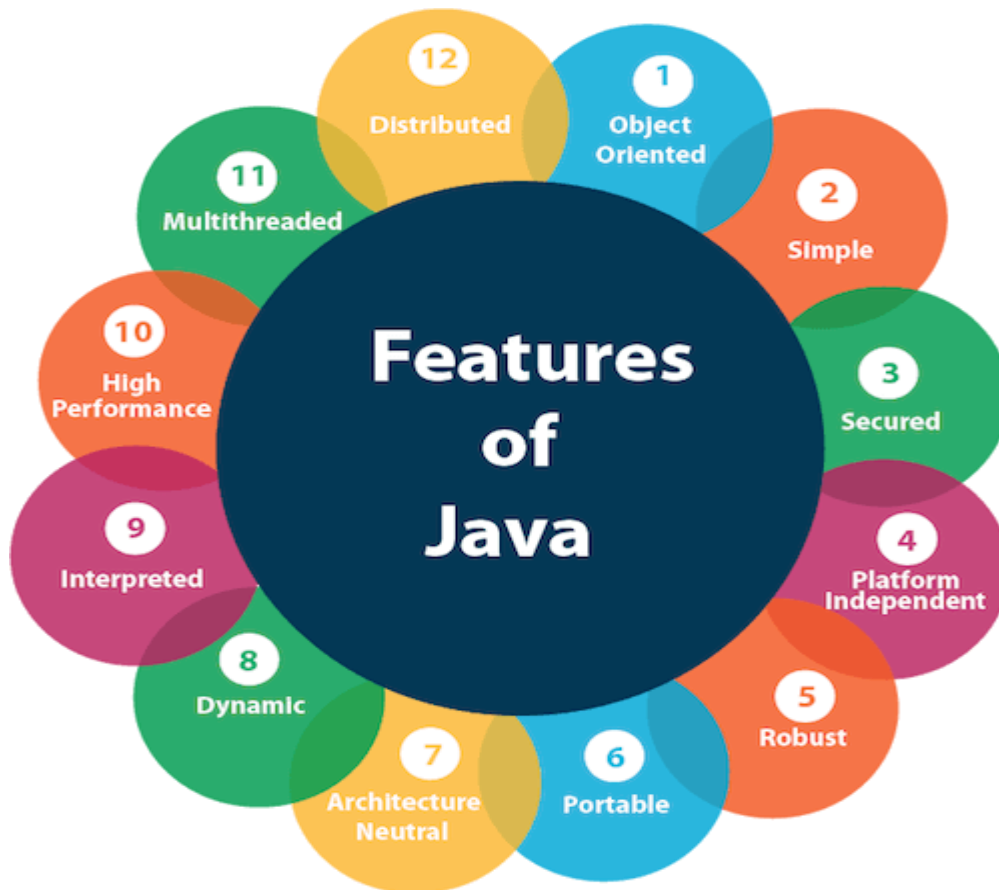
Java Version History

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)

18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (March 2022)
21. Java SE 19 (September 2022)
22. Java SE 20 (to be released by March 2023)

Since Java SE 8 release, the Oracle corporation follows a pattern in which every even version is release in March month and an odd version released in September month.

3. Java buzzwords



A list of the most important features of the Java language is given below.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral

7. Portable
8. High Performance
9. Distributed
10. Multithreaded
11. Dynamic
12. Interpreted

1)Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystems, Java language is a simple programming language because:

- Java syntax is based on C and C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

2)Object-oriented

OOP stands for Object-Oriented Programming. OOP is a programming paradigm in which every program is follows the concept of object. In other words, OOP is a way of writing programs based on the object concept.

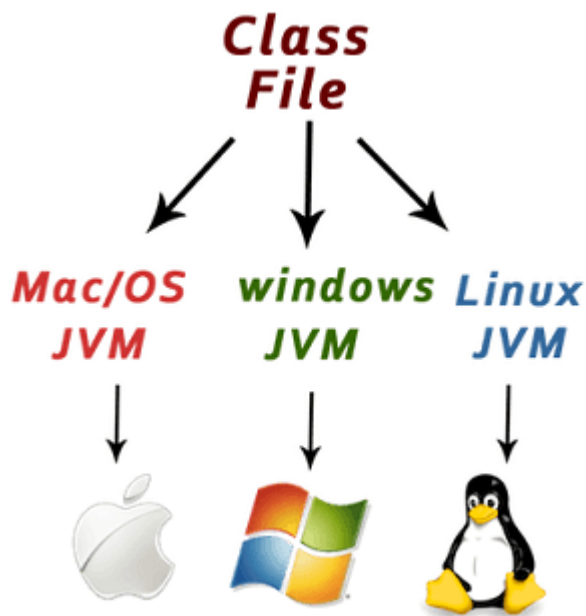
Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

3)Platform Independent

Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

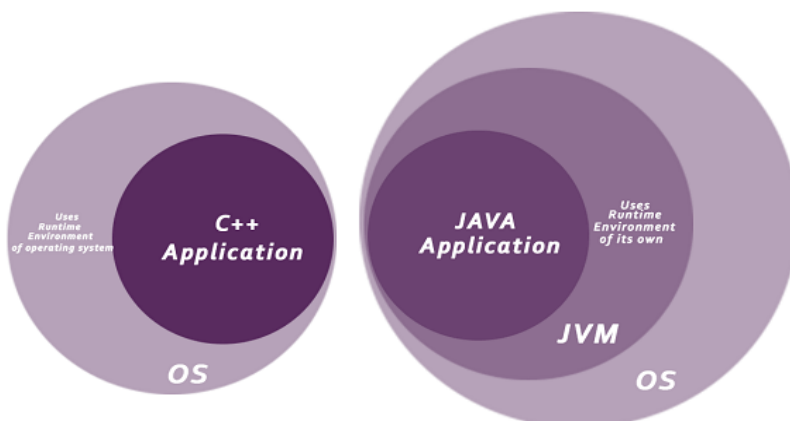
Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into byte code. This byte code is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).



4)Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL(Secure socket layer), JAAS(java Authentication and authorization service), Cryptography, etc.

5)Robust

The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.

- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

6)Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. Java occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

7)Portable

Java is portable because it facilitates you to carry the Java byte code to any platform. It doesn't require any implementation.

8)High-performance

Java is faster than other traditional interpreted programming languages because Java byte code is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

9)Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI(remote method invocation) and EJB(Enterprise java bean) are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

10)Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

11)Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

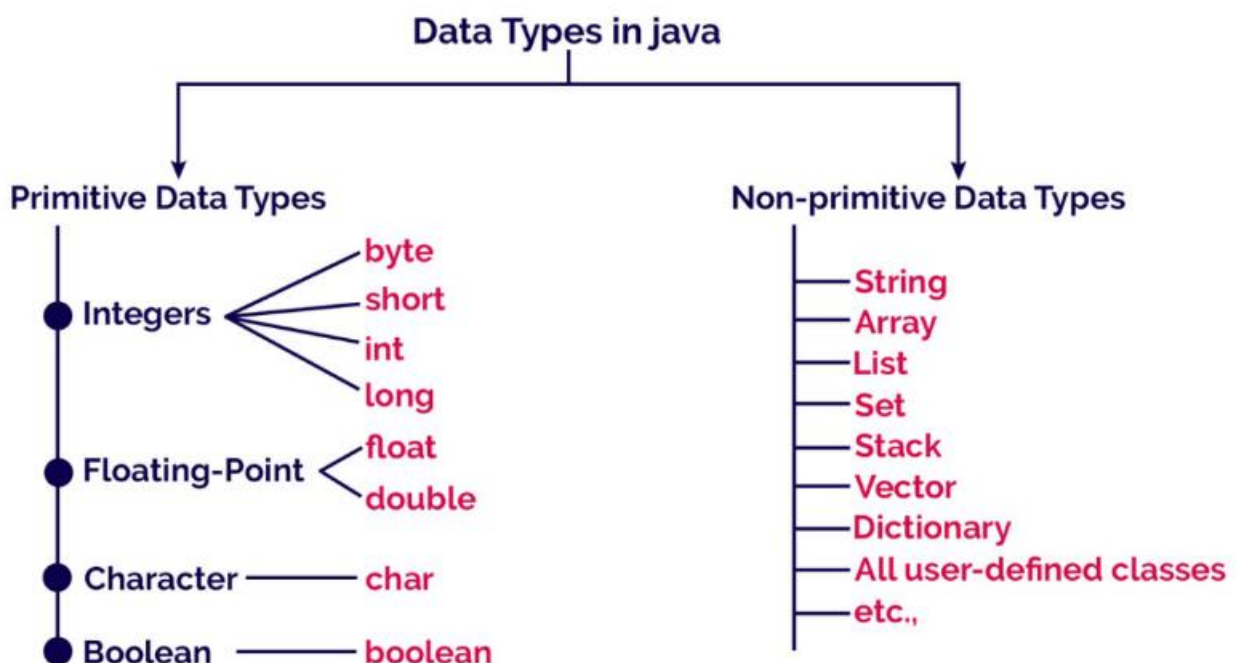
12) Interpreted:-

Java is an interpreter language. It converts byte code to executable code in JVM. It is designed to read the input source code and then translate the executable code instruction by instruction.

3. Data types

Data types specify the different sizes and values that can be stored in the variable. In Java, data types are classified into two types and they are as follows.

- Primitive Data Types
- Non-primitive Data Types



Primitive Data Types

The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size. The primitive data types do not have any additional methods.

In Java, primitive data types include `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

The following table provides more description of each primitive data type.

Data type	Meaning	Memory size	Range	Default Value
-----------	---------	-------------	-------	---------------

byte	Integer numbers	1 byte	-128 to +127	0
short	Integer numbers	2 bytes	-32768 to +32767	0
int	Integer numbers	4 bytes	-2,147,483,648 to +2,147,483,647	0
long	Integer numbers	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	Fractional numbers	4 bytes	-	0.0f
double	Fractional numbers	8 bytes	-	0.0d
char	Single character	2 bytes	0 to 65535	\u0000
boolean	unsigned char	1 bit	0 or 1	0 (false)

Let's look at the following example java program to illustrate primitive data types in java and their default values.

Example

```
public class PrimitiveDataTypes {

    byte i;
    short j;
    int k;
    long l;
    float m;
    double n;
    char ch;
```


boolean p;

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

```
    PrimitiveDataTypes obj = new PrimitiveDataTypes();
```

```
    System.out.println("i = " + obj.i + ", j = " + obj.j + ", k = " + obj.k + ", l = " + obj.l);
```

```
    System.out.println("m = " + obj.m + ", n = " + obj.n);
```

```
    System.out.println("ch = " + obj.ch);
```

```
    System.out.println("p = " + obj.p);
```

```
}
```

```
}
```

When we run the above example code, it produces the following output.

```
eclipse-workspace - DataTypes/src/PrimitiveDataTypes.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
DataTypes
  JRE System Library [JDK1.8.0_101]
  src
    (default package)
      PrimitiveDataTypes.java

PrimitiveDataTypes.java
1 public class PrimitiveDataTypes {
2
3     byte i;
4     short j;
5     int k;
6     long l;
7     float m;
8     double n;
9     char ch;
10    boolean p;
11
12    public static void main(String[] args) {
13
14        PrimitiveDataTypes obj = new PrimitiveDataTypes();
15
16        System.out.println("i = " + obj.i + ", j = " + obj.j + ", k = " + obj.k + ", l = " + obj.l);
17        System.out.println("m = " + obj.m + ", n = " + obj.n);
18        System.out.println("ch = " + obj.ch);
19        System.out.println("p = " + obj.p);
20
21    }
22 }
23

Problems | Javadoc | Declaration | Console
<terminated> PrimitiveDataTypes [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\java.exe (3 Jan 2020, 19:52:56)
i = 0, j = 0, k = 0, l = 0
m = 0.0, n = 0.0
ch =
p = false

PrimitiveDataTypes.java - DataTypes/src
```

Non-primitive Data Types

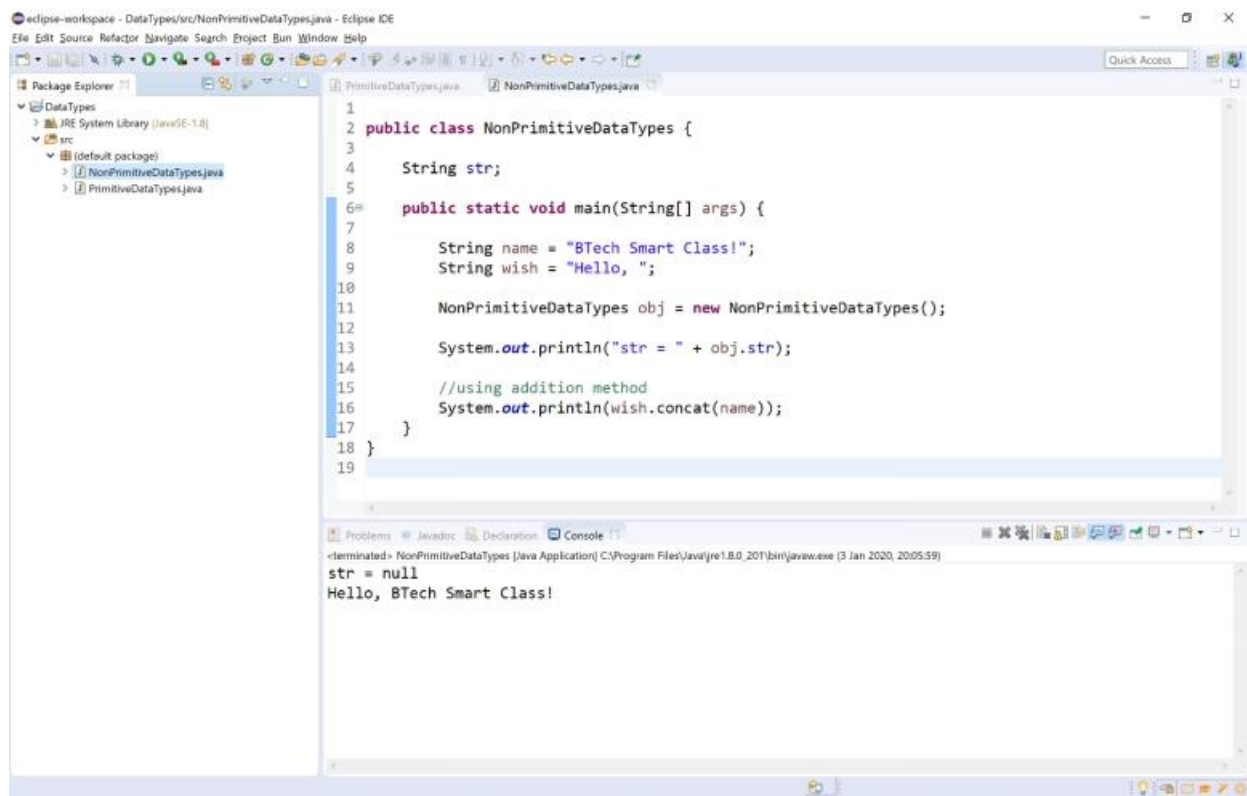
In java, non-primitive data types are the reference data types or user-created data types. All non-primitive data types are implemented using object concepts. Every variable of the non-primitive data type is an object. The non-primitive data types may use additional methods to perform certain operations. The default value of non-primitive data type variable is null.

In java, examples of non-primitive data types are `String`, `Array`, `List`, `Queue`, `Stack`, `Class`, `Interface`, etc.

Example

```
public class NonPrimitiveDataTypes {  
  
    String str;  
  
    public static void main(String[] args) {  
  
        String name = "BTech Smart Class!";  
  
        String wish = "Hello, ";  
  
        NonPrimitiveDataTypes obj = new NonPrimitiveDataTypes();  
  
        System.out.println("str = " + obj.str);  
  
        //using addition method  
        System.out.println(wish.concat(name));  
    }  
}
```

When we run the above example code, it produces the following output.



Primitive Vs Non-primitive Data Types

Primitive Data Type	Non-primitive Data Type
These are built-in data types	These are created by the users
Does not support additional methods	Support additional methods
Always has a value	It can be null
Starts with lower-case letter	Starts with upper-case letter
Size depends on the data type	Same size for all

5. Variables

Defination:-A variable is a named memory location used to store a data value. A variable can be defined as a container that holds a data value. A variable is assigned with a data type.

In java, we use the following syntax to create variables.

Syntax

data_type variable_name;

(or)

data_type variable_name_1, variable_name_2,...;

(or)

data_type variable_name = value;

(or)

data_type variable_name_1 = value, variable_name_2 = value,...;

Variable Naming Rules

A variable can have a short name (like x and y) or a more descriptive name (age, price, carname, etc.).

Go variable naming rules:

- ☐ A variable name must start with a letter or an underscore character (_)
- ☐ A variable name cannot start with a digit
- ☐ A variable name can only contain alpha-numeric characters and underscores (a-z, A-Z, 0-9, and _)
- ☐ Variable names are case-sensitive (age, Age and AGE are three different variables)
- ☐ There is no limit on the length of the variable name
- ☐ A variable name cannot contain spaces
- ☐ The variable name cannot be any Go keywords
- ☐ Beginning with a letter, the dollar sign " \$ ", or the underscore character " _ ".

The convention, however, is to always begin your variable names with a letter, not " \$ " or " _ ".

Multi-Word Variable Names

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

myVariableName = "John"

Pascal Case

Each word starts with a capital letter:

MyVariableName = "John"

Snake Case

Each word is separated by an underscore character:

my_variable_name = "John"

In java programming language variables are classified as follows.

- ☐ **Local variables**
- ☐ **Instance variables or Member variables or Global variables**
- ☐ **Static variables or Class variables**
- ☐ **Final variables**

Local variables

The variables declared inside a method or a block are known as local variables. A local variable is visible within the method in which it is declared. The local variable is created when execution control enters into the method or block and destroyed after the method or block execution completed.

Let's look at the following example java program to illustrate local variable in java.

Ex:-

```
public class LocalVariables {public void show() {
int a = 10;
//static int x = 100;
System.out.println("Inside show method, a = " + a);
}
public void display() {
int b = 20;
System.out.println("Inside display method, b = " + b);
// trying to access variable 'a' - generates an ERROR
```

```
//System.out.println("Inside display method, a = " + a);
}
public static void main(String args[]) {
LocalVariables obj = new LocalVariables();
obj.show();
obj.display();
}
}
```

Output:-

Inside show method, a = 10

Inside display method, b = 20

Instance variables or member variables or global variables

The variables declared inside a class and outside any method, constructor or block are known as instance variables or member variables. These variables are visible to all the methods of the class. The changes made to these variables by method affects all the methods in the class. These variables are created separate copy for every object of that class.

Let's look at the following example java program to illustrate instance variable in java.

```
public class InstanceVariables {int x = 100;// instance variable
public void show() {
System.out.println("Inside show method, x = " + x);
x = x + 100;
}
public void display() {
System.out.println("Inside display method, x = " + x);
}
public static void main(String[] args) {
InstanceVariables obj = new InstanceVariables();
obj.show();
obj.display();
}
}
```

OutPut:-

Inside show method, x = 100

Inside display method, x = 200

Static variables or Class variables

A static variable is a variable that declared using **static** keyword. The instance variables can be static variables but local variables can not. Static variables are initialized only once, at the start of the program execution. The static variable only has one copy per class irrespective of how many objects we create.

The static variable is access by using class name.

Let's look at the following example java program to illustrate static variable in java.

Ex:-

```
public class StaticVariablesExample {int x=10, y=20; // Instance variables
static int z=30; // Static variable
public void show() {
int a; // Local variables
System.out.println("Inside show method,");
System.out.println("x = " + x + ", y = " + y + ", z = " + z);
}
public static void main(String[] args) {
StaticVariablesExample obj = new
StaticVariablesExample();
```

```

System.out.println(obj.x);
System.out.println(obj.y);
System.out.println(StaticVariablesExample.z);
obj.show();
}
}

```

Output:-10

20

30

Inside show method,

x = 10, y = 20, z = 30

Final variables

A final variable is a variable that declared using **final** keyword. The final variable is initialized only once, and does not allow any method to change it's value again. The variable created using **final** keyword acts as constant. All variables like local, instance, and static variables can be final variables.

Let's look at the following example java program to illustrate final variable in java.

Ex:-

```

public class Student
{
final int id=10;
void show()
{
System.out.println(id);
id=20;//final variable can't modify
}
public static void main(String args[]){
Student s1=new Student();
s1.show();
//System.out.println(s1.id);//printing members with a white space }
}

```

Output:-10

6.Scope and Life time of variables

Scope of a Variable

A variable can be declared and defined inside a class, method, or block. It defines the scope of the variable where a variable is available to use. i.e. the visibility or accessibility of a variable. Variable declared inside a block or method are not visible to outside. If we try to do so, we will get a compilation error.

Example:-

```

class DemoScope
{ //class scope
public static void main(String args[])
{ //main method scope
int x;
x=10;

```

```

if(x==10)
{ //condition scope
int y=20;
System.out.println("X and Y:"+x+" "+y);
x=y*2;
}
//y=100;//get compilation error
System.out.println("X is:"+x);}
}

```

Life Time of a Variable:

The lifetime of a variable is the time during which the variable stays in memory and is therefore accessible during program execution. The variables that are local to a method are created the moment the method is activated and are destroyed when the activation of the method terminates.

Ex:-

```

class DemoLifeTime
{
public static void main(String args[])
{
int x;
for(x=0;x<3;x++)
{
int y=-1;
System.out.println("Y is"+y);
y=20;
System.out.println("Y is "+y);
} // After loop completion y is destroyed no longer can use
}
}

```

7.Arrays

Java Arrays

Defination:-array is a collection of similar type of elements which has contiguous memory location.

An array is a collection of similar data values with a single name. An array can also be defined as, a special type of variable that holds multiple values of the same data type at a time.

In java, arrays are objects and they are created dynamically using **new** operator. Every array in java is organized using index values. The index value of an array starts with '0' and ends with 'size-1'. We use the index value to access individual elements of an array.

In java, there are two types of arrays and they are as follows.

- ☐ One Dimensional Array or Single Dimensional Array
- ☐ Multi Dimensional Array

Creating an array

One Dimensional Array:-

In the java programming language, an array must be created using new operator and with a

specific size. The size must be an integer value but not a byte, short, or long. We use the following syntax to create an array.

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arr[]=new datatype[size];

Syntax

data_type array_name[] = new data_type[size];(or)

data_type [] array_name = new data_type[size];

Let's look at the following example program.

Example

```
public class ArrayExample {  
    public static void main(String[] args) {  
        int list[] = new int[5];  
        list[0] = 10;  
        System.out.println("Value at index 0 - " + list[0]);  
        System.out.println("Length of the array - " + list.length);  
    }  
}
```

When we run the above example code, it produces the following output. In java, an array can also be initialized at the time of its declaration. When an array is initialized

at the time of its declaration, it need not specify the size of the array and use of the new operator.

Here, the size is automatically decided based on the number of values that are initialized.

Example

```
int list[] = { 10, 20, 30, 40, 50};
```

NullPointerException with Arrays

In java, an array created without size and initialized to null remains null only. It does not allow us to assign a value. When we try to assign a value it generates a **NullPointerException**.

Look at the following example program.

Example

```
public class ArrayExample {  
    public static void main(String[] args) {  
        short list[] = null; list[0] = 10;
```

```
System.out.println("Value at index 0 - " + list[0]);
}
}
```

When we run the above example code, it produces the following output.

ArrayIndexOutOfBoundsException with Arrays

In java, the JVM (Java Virtual Machine) throws **ArrayIndexOutOfBoundsException** when an array is trying to access with an index value of negative value, value equal to array size, or value more than the array size.

Look at the following example program.

Example

```
public class ArrayExample {public static void main(String[] args) {
short list[] = { 10, 20, 30};
list[4] = 10;
System.out.println("Value at index 0 - " + list[0]);
}
}
```

When we run the above example code, it produces the following output.

Looping through an array

An entire array is accessed using either simple **for statement** or **for-each statement**. Look at the following example program to display sum of all the elements in a list.

Example

```
import java.util.Scanner;
public class ArrayExample {
public static void main(String[] args) {
Scanner read = new Scanner(System.in);
int size, sum = 0;
System.out.print("Enter the size of the list: ");
size = read.nextInt();
short list[] = new short[size];
System.out.println("Enter any " + size + " numbers: ");
for(int i = 0; i < size; i++) // Simple for statement
list[i] = read.nextShort();
for(int i : list) // for-each statement
sum = sum + i;
System.out.println("Sum of all elements: " + sum);
}
}
```

```
}
```

When we run the above example code, it produces the following output.

Multidimensional Array

Two Dimensional Array:-

In java, we can create an array with multiple dimensions. We can create 2-dimensional, 3-dimensional, or any dimensional array.

In Java, multidimensional arrays are arrays of arrays. To create a multidimensional array variable, specify each additional index using another set of square brackets. We use the following syntax to create two-dimensional array.

Syntax to Declare Multidimensional Array in Java

1. `dataType[][] arrayRefVar;` (or)
2. `dataType [][]arrayRefVar;` (or)
3. `dataType arrayRefVar[][];` (or)
4. `dataType []arrayRefVar[];` **Example to instantiate Multidimensional Array in Java**

Syntax

```
data_type array_name[ ][ ] = new data_type[rows][columns];
```

(or)

```
data_type[ ][ ] array_name = new data_type[rows][columns];
```

When we create a two-dimensional array, it created with a separate index for rows and columns.

The individual element is accessed using the respective row index followed by the column index. A multidimensional array can be initialized while it has created using the following syntax.

Syntax

```
data_type array_name[ ][ ] = { { value1, value2}, { value3, value4}, { value5, value6},...};
```

When an array is initialized at the time of declaration, it need not specify the size of the array and use of the new operator. Here, the size is automatically decided based on the number of values that are initialized.

Example

```
int[][] arr=new int[3][3];//3 row and 3 column
```

```
int matrix_a[ ][ ] = { { 1, 2},{3, 4},{5, 6}};
```

The above statement creates a two-dimensional array of three rows and two columns.

Example Program:-

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

```

int arr[][]={ { 1,2,3},{2,4,5},{4,4,5} };
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
System.out.print(arr[i][j]+" ");
}
System.out.println();
}
}

```

}Out Put:-

1 2 3

2 4 5

4 4 5

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.

Syntax:-

Data_type array_name[][]=new data_type[n][];\n n means number of rows

array_name[]=new datatype[n1];\nn1 means number of columns

(or)

Int arr_name[][]={{value1,value2},{value1,value2,value3}};Ex:-

```
int arr[][] = new int[3][];
```

```
arr[0] = new int[3];
```

```
arr[1] = new int[4];
```

```
arr[2] = new int[2];
```

Example Program:-

```
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

}

}

}

```

Out Put:-

```

0 1 2
3 4 5 6
7 8

```

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

//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

Three Dimensional Array(3D Array):-

An array with three indexes (subscripts) is called **three dimensional array in java**.

In other words, a three-dimensional array is a collection of one or more two-dimensional arrays, all of which share a common name.

The general syntax to declare 3D array in java is as follows:

```
data-type[ ][ ][ ] variableName;
```

```
variableName = new data-type[size1][size2][size3];
```

Or,

```
data-type[ ][ ][ ] variableName = new data-type[size1][size2][size3];
```

Example:-

```
int[ ][ ][ ] x = new int[2][3][4];
```

For example:

x[0][0][0] refers to the data in the first table, first row, and first column.

x[1][0][0] refers to the data in the second table, first row, and first column.

x[1][2][3] refers to the data in the second table, third row, and fourth

column.**Example program:-**

```
package arraysProgram;
```

```
public class ThreeDArray {
```

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

```
{
```

```
int[ ][ ][ ] x;
```

```
x = new int[3][3][3];
```

```
for(int i = 0; i < 3; i++)
```

```
{
```

```
for(int j = 0; j < 3; j++)
```

```
for(int k = 0; k < 3; k++)
```

```
x[i][j][k] = i + 1;
```

```
}
```

```
for(int i = 0; i < 3; i++)
```

```
{
```

```
System.out.println("Table-" +(i + 1));
```



```
for(int j = 0; j < 3; j++)  
{  
    for(int k = 0; k < 3; k++)  
        System.out.print(x[i][j][k] + " ");System.out.println();  
    }  
    System.out.println();  
}  
}  
}
```

Out Put:-

Table-1

1 1 1

1 1 1

1 1 1

Table-2

2 2 2

2 2 2

2 2 2

Table-3

3 3 3

3 3 3

3 3 3

8.Operators

An Operator is a symbol that performs an operation. An operator acts on some variables, called operands to get the desired result, For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,

- Arithmetic Operator,
- Bitwise Operator,
- Assignment Operator,
- Relational Operator,
- Logical Operator,
- Ternary Operator and
- Special Operator.

1.Unary Operator: -

The Java unary operators require only one operand. Unary operators are used to perform various operations.

Operator Type	Category	Meaning	Operator
Unary	postfix	<i>Post-Increment Operator</i>	(expr++)
		<i>Post-Decrement Operator</i>	(expr--)
	prefix	Pre-Increment Operator	(++expr)
		Pre-Decrement Operator	(--expr)
	Unary Plus	+expr	
	Unary Minus	-expr	
	Negating an expression Bitwise Not complement	~	
	Logical or boolean	!	

Example:-

```

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

```

```

int a=10,b=-5,d=10;

boolean c=true;

System.out.println(a);//10
System.out.println(a++);//10(11)
System.out.println(++a);//12
System.out.println(a--);//12(11)
System.out.println(--a);//10
System.out.println(~d);//-11
System.out.println(~b);//4
System.out.println(!c);//false
System.out.println(!c);//false
}
}

```

2.Arithmetic Operators:

In java, arithmetic operators are used to performing basic mathematical operations like addition, subtraction, multiplication, division, modulus, increment, decrement, etc.,

Operator	Meaning	Example
+	Addition	$10 + 5 = 15$
-	Subtraction	$10 - 5 = 5$
*	Multiplication	$10 * 5 = 50$
/	Division	$10 / 5 = 2$
%	Modulus - Remainder of the Division	$5 \% 2 = 1$

Example:-

```

public class UnaryOperator{
public static void main(String args[]){
int a=10;

```

```

int b=5;

System.out.println(a+b);//15

System.out.println(a-b);//5

System.out.println(a*b);//50

System.out.println(a/b);//2

System.out.println(a%b);//0

System.out.println(10*10/5+3-1*4/2);

System.out.println(10*10/5+(3-1)*4/2);

}}

```

3.Bitwise Operator:- The bitwise operators are used to perform bit-level operations in the java programming language. When we use the bitwise operators, the operations are performed based on binary values.

Operator Type	Operator meaning	Operator
Bitwise	Bitwise complement	~
	Bitwise And	&
	Bitwise Inclusive OR	
	Bitwise Exclusive or(xor)	^
	Bitwise Left Shift	<<
	Bitwise Right Shift	>>
	Bitwise Zero Fill Right Shift Operator	>>>

Ex:-

```

public class UnaryOperator{

public static void main(String args[]){

int a=5;

int b=6;

int c=2;

int d=-2;

System.out.println(a&b);//4

System.out.println(a|b);//7

System.out.println(a^b);//3

System.out.println(a<<c);//20 5*2^2=5*4=20

```

```
System.out.println(a>>c);//1 5/2^2=5/4=1
```

```
System.out.println(d>>c);-1
```

```
System.out.println(a>>>c);//1
```

```
System.out.println(d>>>c);//1073741823
```

```
}
```

```
}
```

Assignment Operator:-

The operator is used to store some value into a variable .

The assignment operators are used to assign right-hand side value (Rvalue) to the left-hand side variable (Lvalue). The assignment operator is used in different variants along with arithmetic operators. The following table describes all the assignment operators in the java programming language.

Operator	Meaning	Example
=	Assign the right-hand side value to left-hand side variable Assignment operator	A = 15
+=	Add both left and right-hand side values and store the result into left-hand side variable Addition assignment operator	A += 10
-=	Subtract right-hand side value from left-hand side variable value and store the result into left-hand side variable Subtraction assignment operator	A -= B
*=	Multiply right-hand side value with left-hand side variable value and store the result into left-hand side variable Multiplication assignment operator	A *= B
/=	Divide left-hand side variable value with right-hand side variable value and store the result into the left-hand side variable division	A /= B
%=	Divide left-hand side variable value with right-hand side variable value and store the remainder into the left-hand side variable	A %= B

modulus

&=	Logical AND assignment	-
=	Logical OR assignment	-
^=	Logical XOR assignment	-

Ex:-

```
public class UnaryOperator{
public static void main(String args[]){
int a=10,b=10;
System.out.println(a+=b);//20
System.out.println(a-=b);//10
System.out.println(a*=b);//100
System.out.println(a/=b);//10
System.out.println(a%=b);//0
System.out.println(a&=b);//0
System.out.println(a|=b);10
System.out.println(a^=b);0
}
}
```

Relational Operators (<, >, <=, >=, ==, !=)

The relational operators are the symbols that are used to compare two values. That means the relational operators are used to check the relationship between two values. Every relational operator has two possible results either **TRUE** or **FALSE**. In simple words, the relational operators are used to define conditions in a program. The following table provides information about relational operators.

Operator	Meaning	Example
<	Returns TRUE if the first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Greater than operator	
>	Returns TRUE if the first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE

Less than operator

<=	Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE
	Less than or equal to	
>=	Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
	Greater than or equal to	
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
	Equal to operator	
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE
	Not equal to operator	

```
Ex:- public class UnaryOperator{
public static void main(String args[]){
int a=10,b=10;
System.out.println(a>b);//false
System.out.println(a<b);//false
System.out.println(a>=b);//true
System.out.println(a<=b);//true
System.out.println(a==b);//true
System.out.println(a!=b);//false

}
}
```

Logical Operators

The logical operators are the symbols that are used to combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
----------	---------	---------

&	Logical AND - Returns TRUE if all conditions are TRUE otherwise returns FALSE	false & true => false
	Logical OR - Returns FALSE if all conditions are FALSE otherwise returns TRUE	false true => true
^	Logical XOR - Returns FALSE if all conditions are same otherwise returns TRUE	true ^ true => false
!	Logical NOT - Returns TRUE if condition is FALSE and returns FALSE if it is TRUE	!false => true
&&	short-circuit AND - Similar to Logical AND (&), but once a decision is finalized it does not evaluate remaining.	false & true => false
	short-circuit OR - Similar to Logical OR (), but once a decision is finalized it does not evaluate remaining.	false true => true

???

The operators &, |, and ^ can be used with both boolean and integer data type values. When they are used with integers, performs bitwise operations and with boolean, performs logical operations.

???

Logical operators and Short-circuit operators both are similar, but in case of short-circuit operators once the decision is finalized it does not evaluate remaining expressions.

Ex:-

```
class Test4
{
public static void main(String args[])
{
int a=10,b=20;
System.out.println(a>b & a<b);//false
System.out.println(a>b && a<b);//false
System.out.println(a>b | a<b);//true
System.out.println(a>b || a<b);//true
System.out.println(a>b ^ a<b);//true
System.out.println(!(a>b));//true
}
```

}

Conditional or Ternary Operator:-

The Ternary operator is a conditional operator that decreases the length of code while performing comparisons and conditionals. This method is an alternative for using if-else and nested if-else statements. The order of execution for this operator is from left to right.

Syntax:(Condition) ? (Statement1) : (Statement2);

Conditional or Ternary Operator (?:) in Java



Syntax:

```
variable = Expression1 ? Expression2 : Expression3
```

Ex: exp1?exp2:exp3

```
a=10;b=20;
```

```
a>b ? S.O.Pln("a is big") : S.O.Pln("b is big");
```

Ex:-

```
class Test4
{
    public static void main(String args[])
    {
        int a=10,b=20;
        int max=a>b?a:b;
        System.out.println(max);
    }
}
```

Special Operators:-

- 1.instanceof operator
- 2.Member or Dot Operator
- 3.Cast Operator

1.Instanceof: This operator allows us to determine that the object belongs to a particular class or not.Ex: Person instanceof Student (Object instanceof Class)**2. . (Dot) Operator:** This operator is used to access the instance variables and the methods of class objectsEx: Person.age; Person.getdata();

3.Cast Operator:- Cast operator is used to convert one data type to other.

```
class Test4
{
double b=20;
int a=(int)b;
public static void main(String args[])
{
Test4 t=new Test4();
System.out.println(t instanceof Test4);
System.out.println(t.b);
System.out.println(t.a);
}
}
```

9.Expressions

In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.In the java programming language, an expression is defined as follows.

An expression is a collection of operators and operands that represents a specific value.

In the above definition, an **operator** is a symbol that performs tasks like arithmetic operations, logical operations, and conditional operations, etc.

Operands are the values on which the operators perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

In the java programming language, expressions are divided into THREE types. They are as follows.

Infix Expression

Postfix Expression

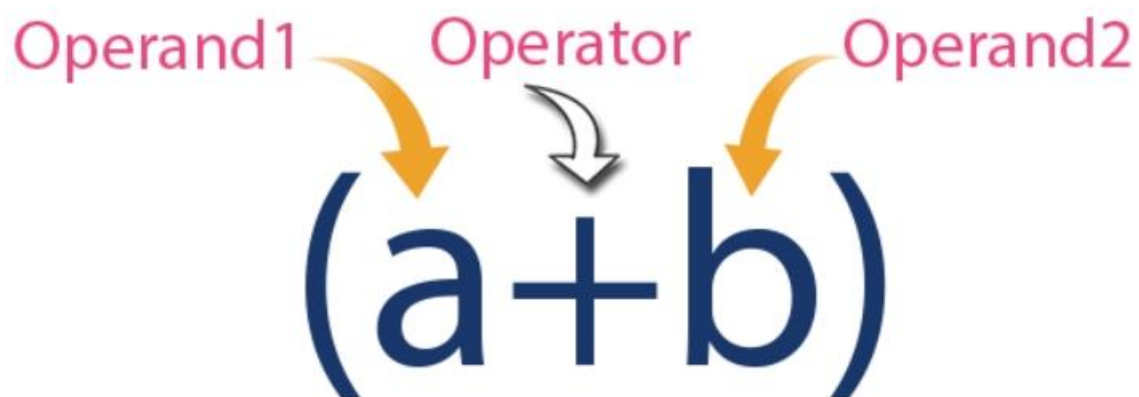
Prefix Expression

The above classification is based on the operator position in the expression.

Infix Expression

The expression in which the operator is used between operands is called infix expression. The infix expression has the following general structure.

Example



Postfix Expression

The expression in which the operator is used after operands is called postfix expression. The postfix expression has the following general structure.

Example



Prefix Expression

The expression in which the operator is used before operands is called a prefix expression. The prefix expression has the following general structure.

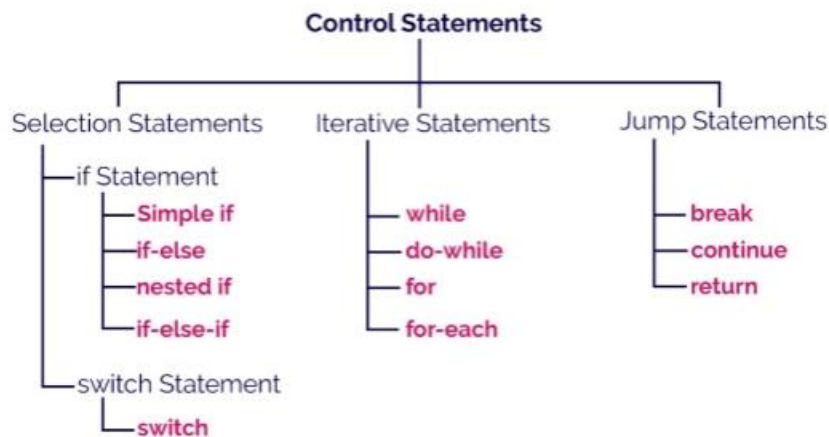
Example



10. Control statements

Java Control Statements

In java, the default execution flow of a program is a sequential order. But the sequential order of execution flow may not be suitable for all situations. Sometimes, we may want to jump from line to another line, we may want to skip a part of the program, or sometimes we may want to execute a part of the program again and again. To solve this problem, java provides control statements.



www.btechsmartclass.com

In java, the control statements are the statements which will tell us that in which order the instructions are getting executed. The control statements are used to control the order of execution according to our requirements. Java provides several control statements, and they are classified as follows.

Types of Control Statements

In java, the control statements are classified as follows.

- Selection Control Statements (Decision Making Statements)
- Iterative Control Statements (Looping Statements)
- Jump Statements

Let's look at each type of control statements in java.

Selection Control Statements

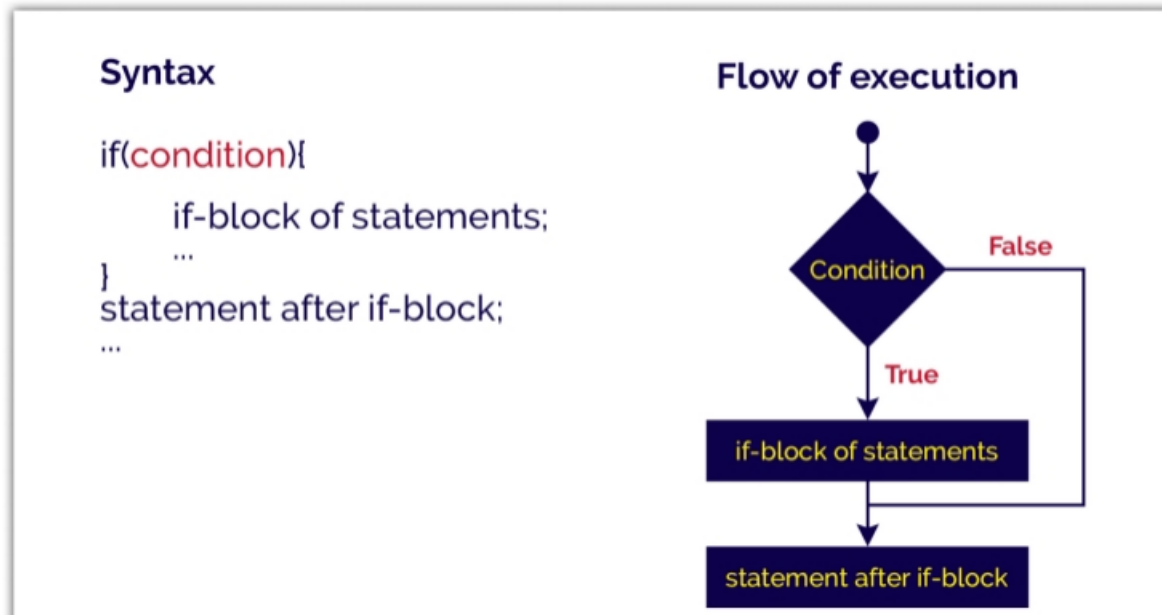
In java, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition. Java provides the following selection statements.

- if statement
- if-else statement
- if-else-if statement

- nested if statement
- switch statement

if statement in java

In java, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result. The if statement checks, the given condition then decides the execution of a block of statements. If the condition is True, then the block of statements is executed and if it is False, then the block of statements is ignored. The syntax and execution flow of if the statement is as follows.



Let's look at the following example java code.

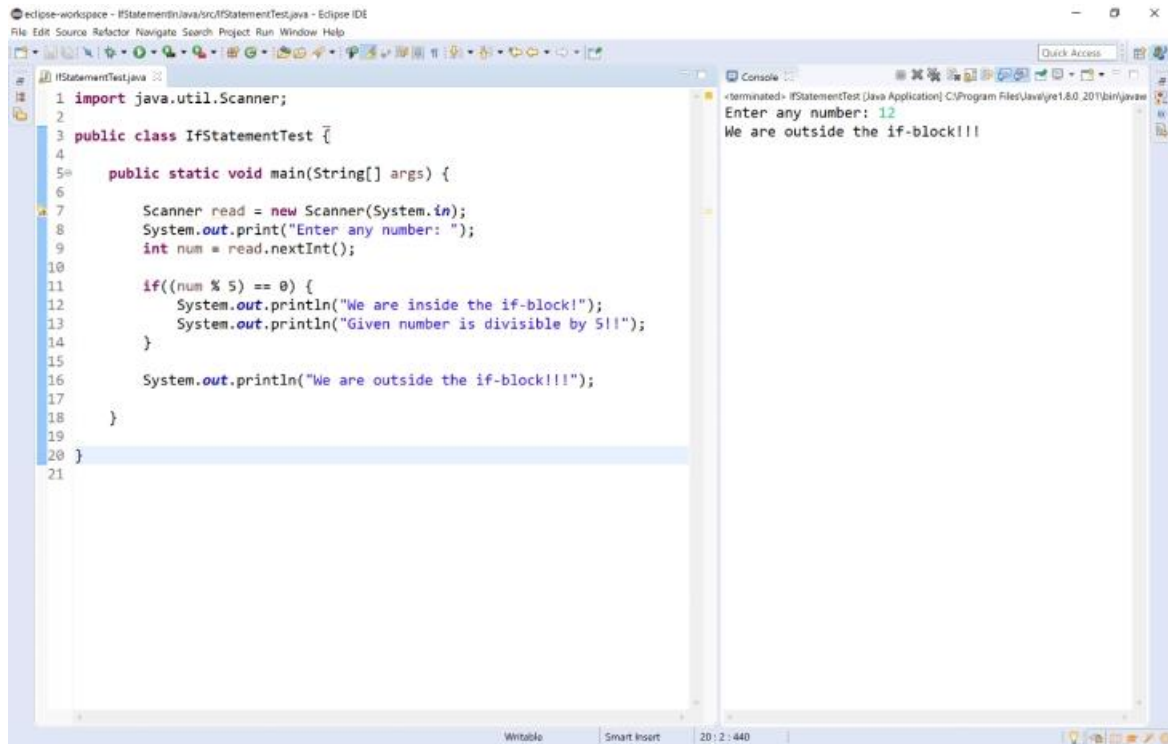
Java Program

```
import java.util.Scanner;  
  
public class IfStatementTest {  
  
    public static void main(String[] args) {  
  
        Scanner read = new Scanner(System.in);  
        System.out.print("Enter any number: ");  
        int num = read.nextInt();  
  
        if((num % 5) == 0) {  
            System.out.println("We are inside the if-block!");  
            System.out.println("Given number is divisible by 5!!!");  
        }  
  
        System.out.println("We are outside the if-block!!!");  
    }  
}
```

```
}
```

```
}
```

When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE with the file `IfStatementTest.java` open. The code is as follows:

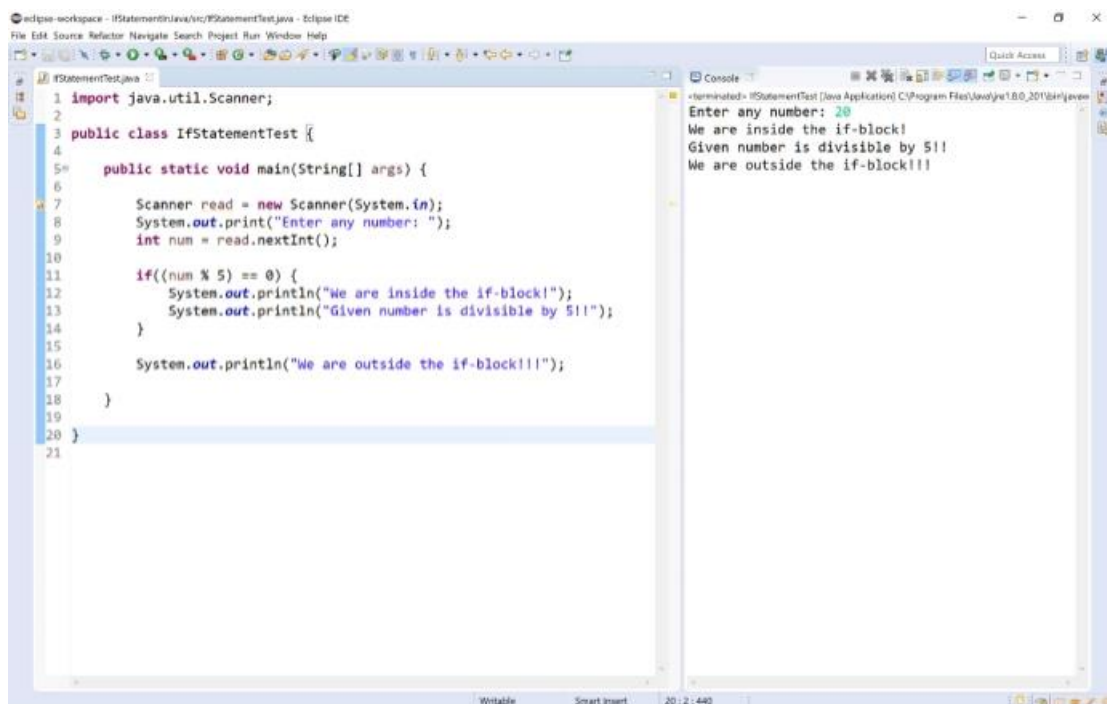
```
1 import java.util.Scanner;
2
3 public class IfStatementTest {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.print("Enter any number: ");
9         int num = read.nextInt();
10
11         if((num % 5) == 0) {
12             System.out.println("We are inside the if-block!");
13             System.out.println("Given number is divisible by 5!!");
14         }
15
16         System.out.println("We are outside the if-block!!!");
17     }
18 }
19
20
21
```

The console output shows the program execution with the input 12:

```
<terminated> IfStatementTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\java.exe
Enter any number: 12
We are outside the if-block!!!
```

In the above execution, the number 12 is not divisible by 5. So, the condition becomes False and the condition is evaluated to False. Then the if statement ignores the execution of its block of statements.

When we enter a number which is divisible by 5, then it produces the output as follows.



The screenshot shows the Eclipse IDE with the file `IfStatementTest.java` open. The code is the same as in the previous screenshot:

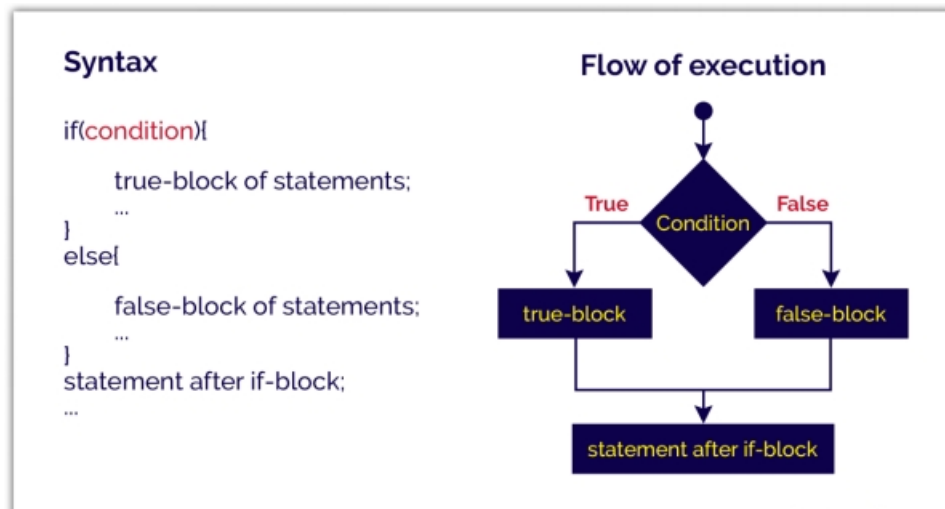
```
1 import java.util.Scanner;
2
3 public class IfStatementTest {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.print("Enter any number: ");
9         int num = read.nextInt();
10
11         if((num % 5) == 0) {
12             System.out.println("We are inside the if-block!");
13             System.out.println("Given number is divisible by 5!!");
14         }
15
16         System.out.println("We are outside the if-block!!!");
17     }
18 }
19
20
21
```

The console output shows the program execution with the input 20:

```
<terminated> IfStatementTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\java.exe
Enter any number: 20
We are inside the if-block!
Given number is divisible by 5!!
We are outside the if-block!!!
```

if-else statement in java

In java, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result. The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result. If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed. The syntax and execution flow of if-else statement is as follows.



Let's look at the following example java code.

Java Program

```
import java.util.Scanner;

public class IfElseStatementTest {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.print("Enter any number: ");
        int num = read.nextInt();

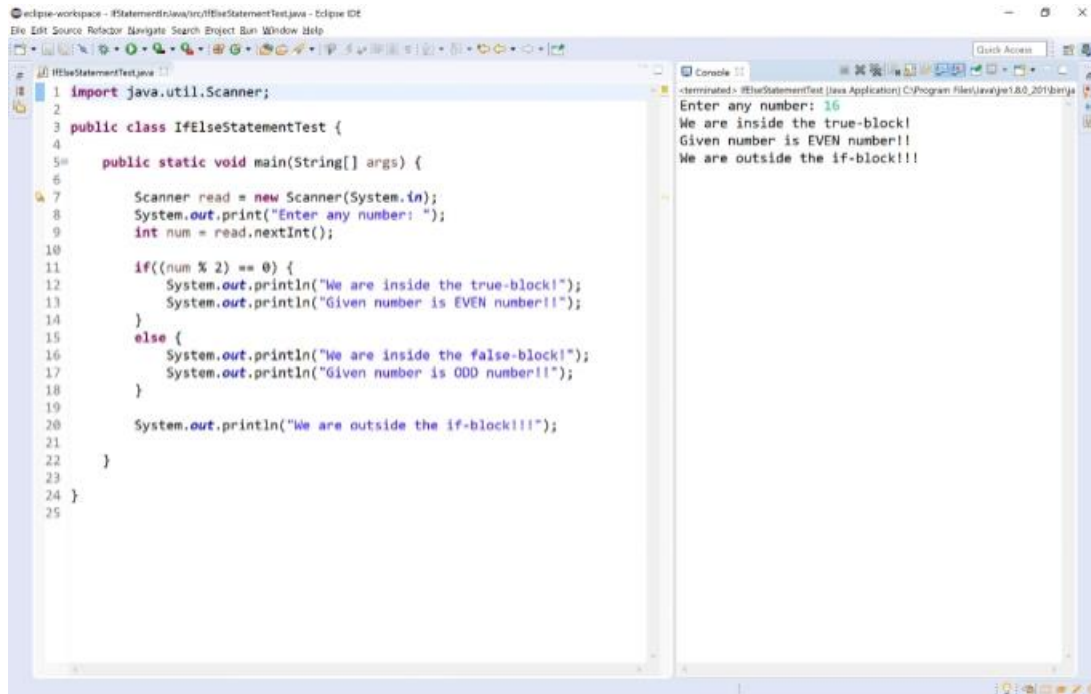
        if((num % 2) == 0) {
            System.out.println("We are inside the true-block!");
            System.out.println("Given number is EVEN number!!");
        }
        else {
            System.out.println("We are inside the false-block!");
            System.out.println("Given number is ODD number!!");
        }

        System.out.println("We are outside the if-block!!!");
    }
}
```

```
}
```

```
}
```

When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE with a Java file named `IfElseStatementTest.java`. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class IfElseStatementTest {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.print("Enter any number: ");
9         int num = read.nextInt();
10
11         if((num % 2) == 0) {
12             System.out.println("We are inside the true-block!");
13             System.out.println("Given number is EVEN number!!");
14         }
15         else {
16             System.out.println("We are inside the false-block!");
17             System.out.println("Given number is ODD number!!");
18         }
19
20         System.out.println("We are outside the if-block!!!");
21     }
22 }
23
24
25
```

The console output shows the program running with the input `16`:

```
<terminated> IfElseStatementTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\java
Enter any number: 16
We are inside the true-block!
Given number is EVEN number!!
We are outside the if-block!!!
```

Nested if statement in java

Writing an if statement inside another if-statement is called nested if statement. The general syntax of the nested if-statement is as follows.

Syntax

```
if(condition_1){
    if(condition_2){
        inner if-block of statements;
        ...
    }
    ...
}
```

Let's look at the following example java code.

Java Program

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

```
public class NestedIfStatementTest {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.print("Enter any number: ");
        int num = read.nextInt();

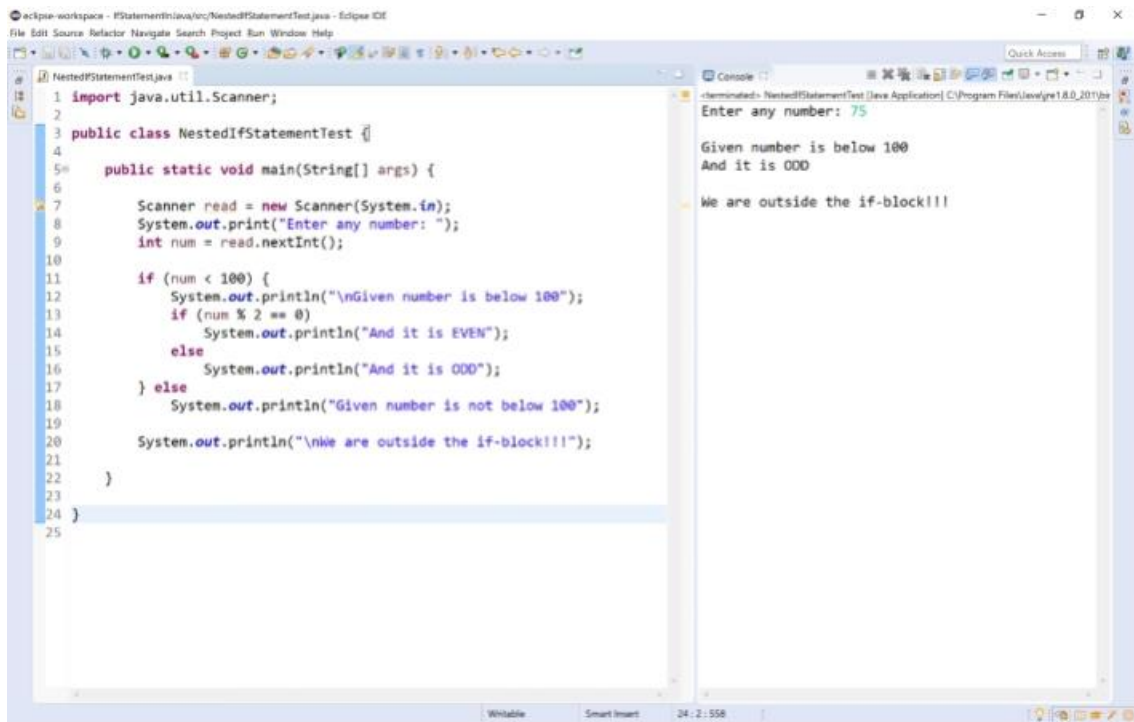
        if (num < 100) {
            System.out.println("\nGiven number is below 100");
            if (num % 2 == 0)
                System.out.println("And it is EVEN");
            else
                System.out.println("And it is ODD");
        } else
            System.out.println("Given number is not below 100");

        System.out.println("\nWe are outside the if-block!!!");

    }

}
```

When we run this code, it produce the following output.



if-else if statement in java

Writing an if-statement inside else of an if statement is called if-else-if statement. The general syntax of the an if-else-if statement is as follows.

Syntax

```

if(condition_1){
    condition_1 true-block;
    ...
}
else if(condition_2){
    condition_2 true-block;
    condition_1 false-block too;
    ...
}

```

Let's look at the following example java code.

Java Program

```

import java.util.Scanner;

public class IfElseIfStatementTest {

    public static void main(String[] args) {

```

```
int num1, num2, num3;
Scanner read = new Scanner(System.in);
System.out.print("Enter any three numbers: ");
num1 = read.nextInt();//10
num2 = read.nextInt();//20
num3 = read.nextInt();//30

if( num1>=num2 && num1>=num3)
    System.out.println("\nThe largest number is " + num1) ;

else if (num2>=num1 && num2>=num3)
    System.out.println("\nThe largest number is " + num2) ;

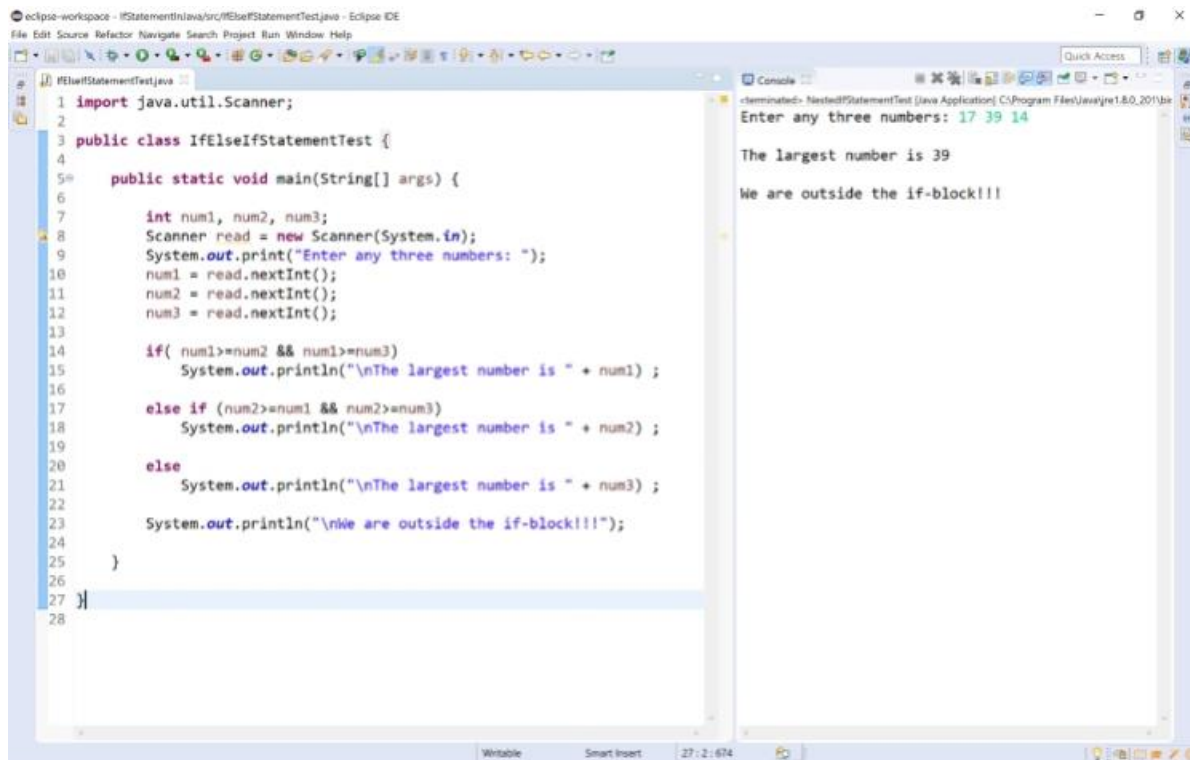
else
    System.out.println("\nThe largest number is " + num3) ;

    System.out.println("\nWe are outside the if-block!!!");

}

}
```

When we run this code, it produce the following output.



switch statement in java

Using the switch statement, one can select only one option from more number of options very easily. In the switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches the value associated with an option, the execution starts from that option. In the switch statement, every option is defined as a **case**.

The switch statement has the following syntax and execution flow diagram.

The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched.

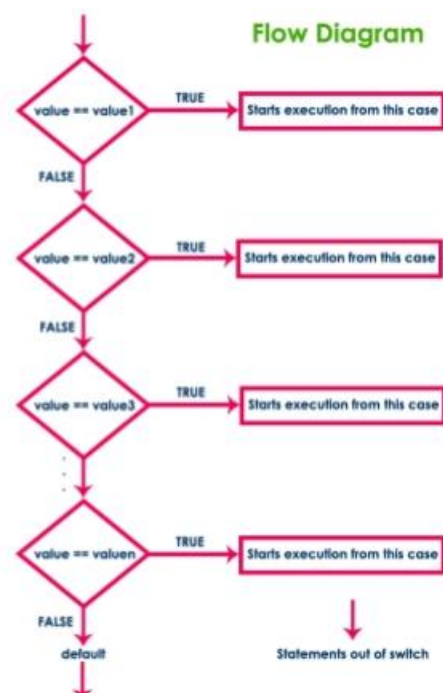
Syntax

```

switch ( expression or value )
{
    case value1: set of statements;
        ....
    case value2: set of statements;
        ....
    case value3: set of statements;
        ....
    case value4: set of statements;
        ....
    case value5: set of statements;
        ....
    .
    .
    default: set of statements;
}

```

Flow Diagram



Let's look at the following example java code.

Java Program

```
import java.util.Scanner;

public class SwitchStatementTest {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.print("Press any digit: ");

        int value = read.nextInt();

        switch( value )
        {
            case 0: System.out.println("ZERO"); break ;
            case 1: System.out.println("ONE"); break ;
            case 2: System.out.println("TWO"); break ;
            case 3: System.out.println("THREE"); break ;
            case 4: System.out.println("FOUR"); break ;
            case 5: System.out.println("FIVE"); break ;
            case 6: System.out.println("SIX"); break ;
            case 7: System.out.println("SEVEN"); break ;
            case 8: System.out.println("EIGHT"); break ;
            case 9: System.out.println("NINE"); break ;
            default: System.out.println("Not a Digit");
        }

    }

}
```

When we run this code, it produce the following output.

The screenshot shows the Eclipse IDE with a Java file named `SwitchStatementTest.java`. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class SwitchStatementTest {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.println("Press any digit: ");
9
10        int value = read.nextInt();
11
12        switch( value )
13        {
14            case 0: System.out.println("ZERO"); break ;
15            case 1: System.out.println("ONE"); break ;
16            case 2: System.out.println("TWO"); break ;
17            case 3: System.out.println("THREE"); break ;
18            case 4: System.out.println("FOUR"); break ;
19            case 5: System.out.println("FIVE"); break ;
20            case 6: System.out.println("SIX"); break ;
21            case 7: System.out.println("SEVEN"); break ;
22            case 8: System.out.println("EIGHT"); break ;
23            case 9: System.out.println("NINE"); break ;
24            default: System.out.println("Not a Digit") ;
25        }
26    }
27 }
28
29 }
30
```

The console window on the right shows the output of the program:

```
<terminated> SwitchStatementTest (Java Application) C:\Program Files\Java\jre1.8.0_201\bin\
Press any digit: 8
EIGHT
```

??? In java, the case value of a switch statement can be a **String** value.

Iterative Control Statements

In java, the iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as the given condition is True. Using iterative statements reduces the size of the code, reduces the code complexity, makes it more efficient, and increases the execution speed. Java provides the following iterative statements.

- while statement
- do-while statement
- for statement
- for-each statement

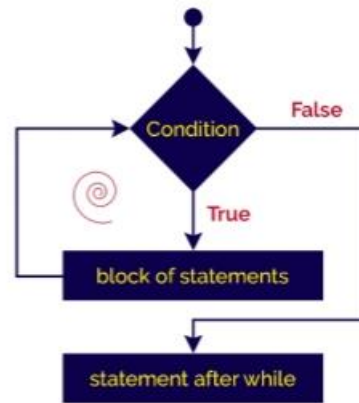
while statement in java

The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement. The syntax and execution flow of while statement is as follows.

Syntax

```
while(boolean-expression){  
    block of statements;  
    ...  
}  
statement after while;  
...
```

Flow of execution

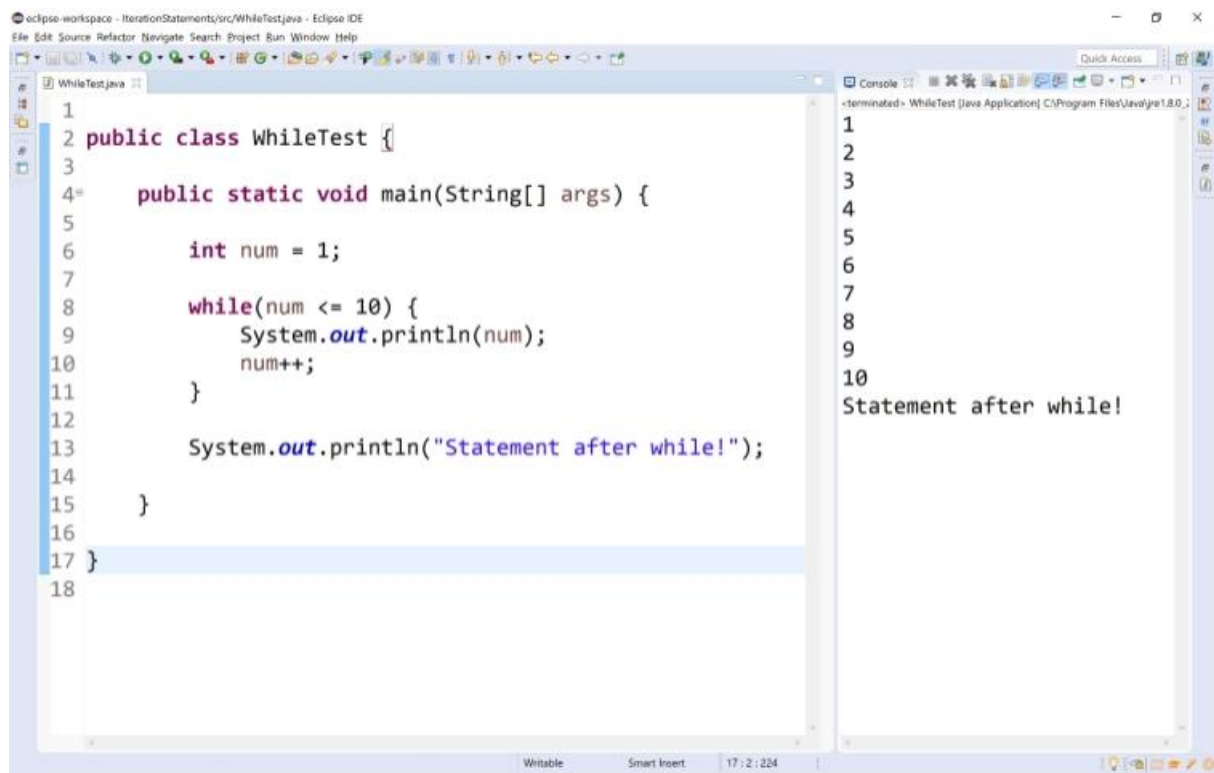


Let's look at the following example java code.

Java Program

```
public class WhileTest {  
  
    public static void main(String[] args) {  
  
        int num = 1;  
  
        while(num <= 10) {  
            System.out.println(num);  
            num++;  
        }  
  
        System.out.println("Statement after while!");  
  
    }  
  
}
```

When we run this code, it produce the following output.



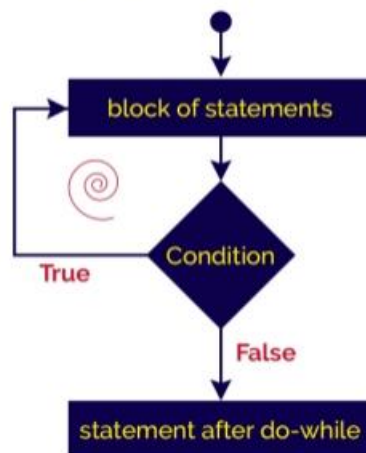
do-while statement in java

The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as the **Exit control looping statement**. The do-while statement has the following syntax.

Syntax

```
do{
    block of statements;
}while(boolean-expression);
statement after do-while;
...
```

Flow of execution



Let's look at the following example java code.

Java Program

```
public class DoWhileTest {

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

```

int num = 1;

do {
    System.out.println(num);
    num++;
} while(num <= 10);

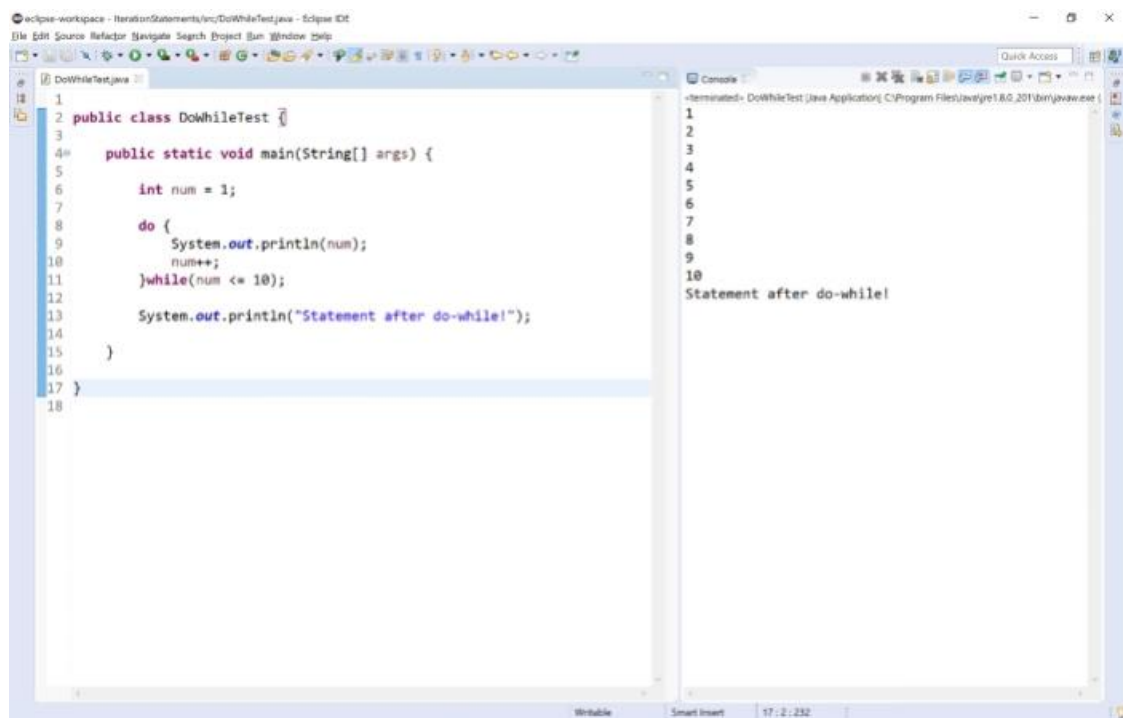
System.out.println("Statement after do-while!");

}

}

```

When we run this code, it produce the following output.



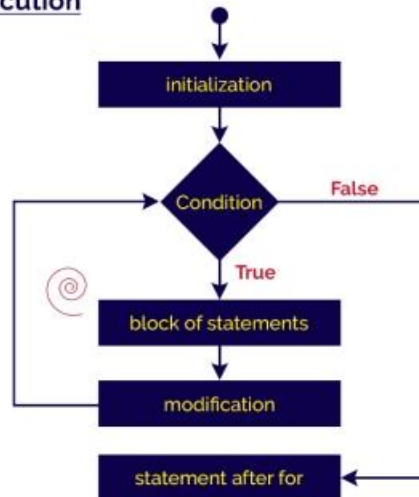
for statement in java

The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE. The for statement has the following syntax and execution flow diagram.

Syntax

```
for(initialization; boolean-expression; modification){  
    block of statements;  
    ...  
}  
statement after for;  
...
```

Flow of execution



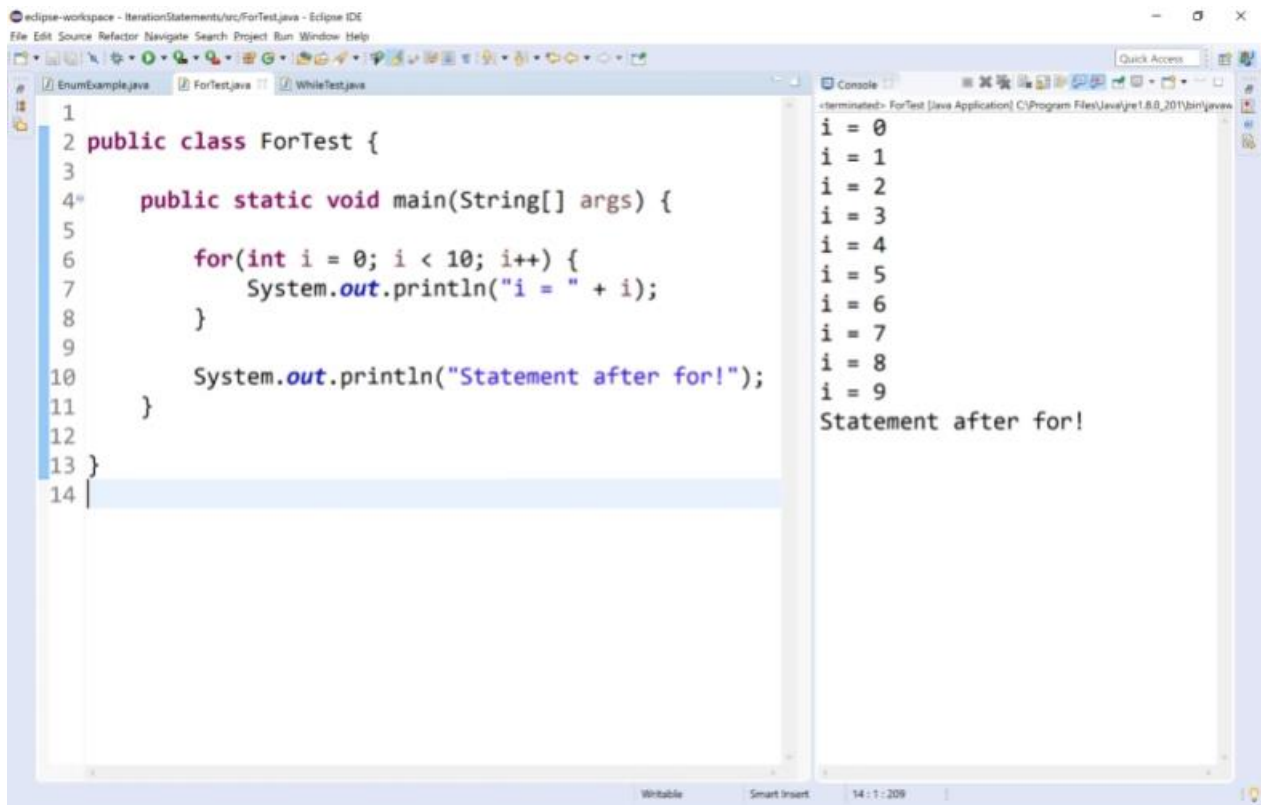
In for-statement, the execution begins with the **initialization** statement. After the initialization statement, it executes **Condition**. If the condition is evaluated to true, then the block of statements executed otherwise it terminates the for-statement. After the block of statements execution, the **modification** statement gets executed, followed by condition again.

Let's look at the following example java code.

Java Program

```
public class ForTest {  
  
    public static void main(String[] args) {  
  
        for(int i = 0; i < 10; i++) {  
            System.out.println("i = " + i);  
        }  
  
        System.out.println("Statement after for!");  
    }  
  
}
```

When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE with a workspace named 'IterationStatements'. The editor displays a file 'ForTest.java' with the following code:

```
1
2 public class ForTest {
3
4     public static void main(String[] args) {
5
6         for(int i = 0; i < 10; i++) {
7             System.out.println("i = " + i);
8         }
9
10        System.out.println("Statement after for!");
11    }
12
13 }
14
```

The console on the right shows the output of the program:

```
<terminated> ForTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\java.exe
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
Statement after for!
```

for-each statement in java

The Java for-each statement was introduced since Java 5.0 version. It provides an approach to traverse through an array or collection in Java. The for-each statement also known as **enhanced for** statement. The for-each statement executes the block of statements for each element of the given array or collection.

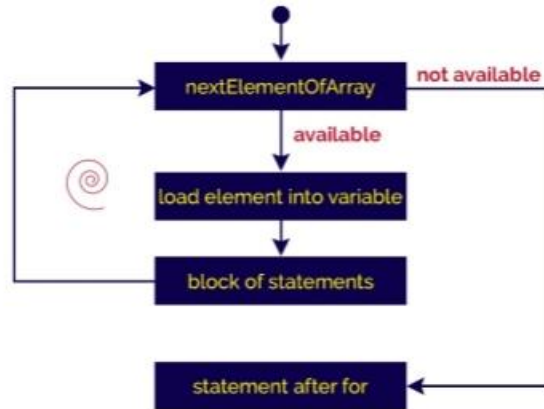
??? In for-each statement, we can not skip any element of given array or collection.

The for-each statement has the following syntax and execution flow diagram.

Syntax

```
for( dataType variableName : Array ){  
    block of statements;  
    ...  
}  
statement after for;  
...
```

Flow of execution



Let's look at the following example java code.

Java Program

```
public class ForEachTest {  
  
    public static void main(String[] args) {  
  
        int[] arrayList = { 10, 20, 30, 40, 50 };  
  
        for(int i : arrayList) {  
            System.out.println("i = " + i);  
        }  
  
        System.out.println("Statement after for-each!");  
    }  
  
}
```

When we run this code, it produce the following output.

The screenshot shows the Eclipse IDE with a Java project named 'IterationStatements'. The main editor displays the source code for 'ForEachTest.java'. The code defines a public class 'ForEachTest' with a 'main' method. Inside the 'main' method, an integer array 'arrayList' is initialized with values {10, 20, 30, 40, 50}. A 'for-each' loop iterates over this array, printing each element 'i' followed by 'i = ' and the value. After the loop, a message 'Statement after for-each!' is printed. The console on the right shows the execution output: 'i = 10', 'i = 20', 'i = 30', 'i = 40', 'i = 50', and 'Statement after for-each!'. The status bar at the bottom indicates 'Writable', 'Smart Insert', and the time '15:2:258'.

```
1 public class ForEachTest {
2
3
4     public static void main(String[] args) {
5
6         int[] arrayList = {10, 20, 30, 40, 50};
7
8         for(int i : arrayList) {
9             System.out.println("i = " + i);
10        }
11
12        System.out.println("Statement after for-each!");
13    }
14
15 }
```

Console Output:

```
<terminated> ForEachTest [Java Application] C:\Program Files\Java\jre1.8
i = 10
i = 20
i = 30
i = 40
i = 50
Statement after for-each!
```

Jump Statements

In java, the jump statements are used to terminate a block or take the execution control to the next iteration. Java provides the following jump statements.

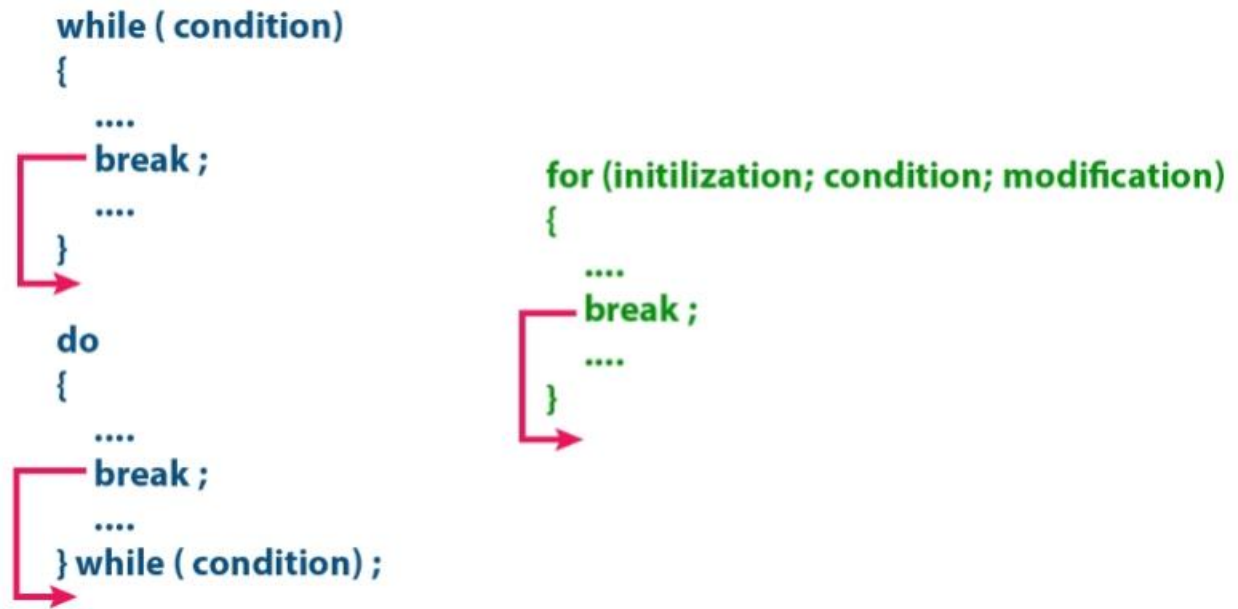
- break
- continue
- return

break statement in java

The break statement in java is used to terminate a switch or looping statement. That means the break statement is used to come out of a switch statement and a looping statement like while, do-while, for, and for-each.

??? Using the break statement outside the switch or loop statement is not allowed.

The flowing picture depicts the execution flow of the break statement.



Let's look at the following example java code.

Java Program

```
public class JavaBreakStatement {  
  
    public static void main(String[] args) {  
  
        int list[] = { 10, 20, 30, 40, 50 };  
  
        for(int i : list) {  
            if(i == 30)  
                break;  
            System.out.println(i);  
        }  
  
    }  
  
}
```

When we run this code, it produce the following output.

The screenshot shows the Eclipse IDE with a Java file named `JavaBreakStatement.java`. The code defines a public class `JavaBreakStatement` with a `main` method. Inside `main`, an array `list` is initialized with values `{10, 20, 30, 40, 50}`. A `for` loop iterates over each element `i` in `list`. If `i` is equal to 30, the `break` statement is executed, terminating the loop. Otherwise, `System.out.println(i)` is called. The console output shows the values 10 and 20, indicating that the loop was terminated when it reached the value 30.

```
1 public class JavaBreakStatement {
2
3     public static void main(String[] args) {
4
5         int list[] = {10, 20, 30, 40, 50};
6
7         for(int i : list) {
8             if(i == 30)
9                 break;
10            System.out.println(i);
11        }
12    }
13 }
14
15
16
17
```

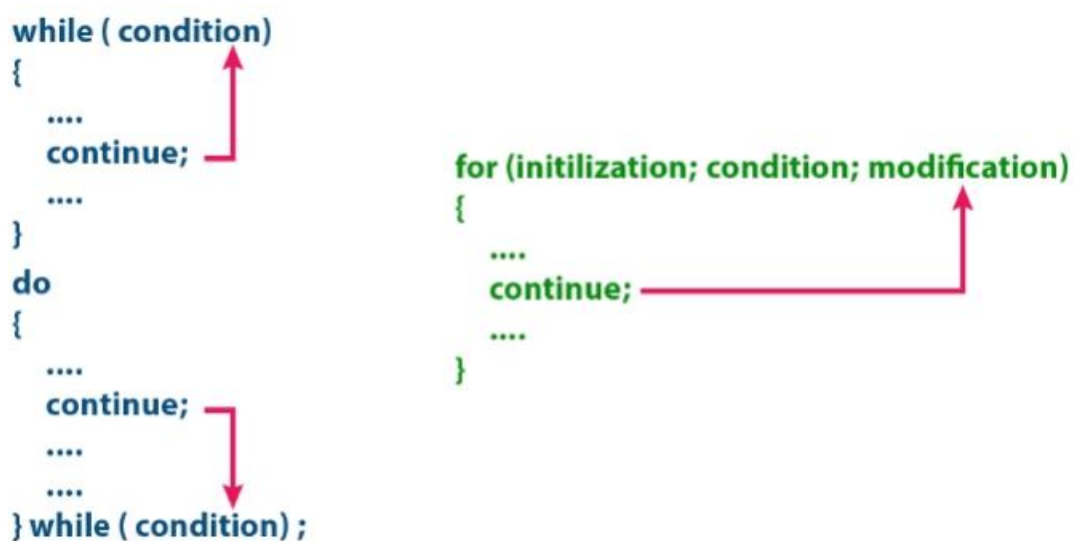
Console Output:

```
<terminated> JavaBreakStatement [Java Application] C:\Program Files\Ja
10
20
```

continue statement in java

The continue statement is used to move the execution control to the beginning of the looping statement. When the continue statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The continue statement can be used with looping statements like while, do-while, for, and for-each.

When we use continue statement with while and do-while statements, the execution control directly jumps to the condition. When we use continue statement with for statement the execution control directly jumps to the modification portion (increment/decrement/any modification) of the for loop. The continue statement flow of execution is as shown in the following figure.



Let's look at the following example java code.

Java Program

```

public class JavaContinueStatement {

    public static void main(String[] args) {

        int list[] = {10, 20, 30, 40, 50};

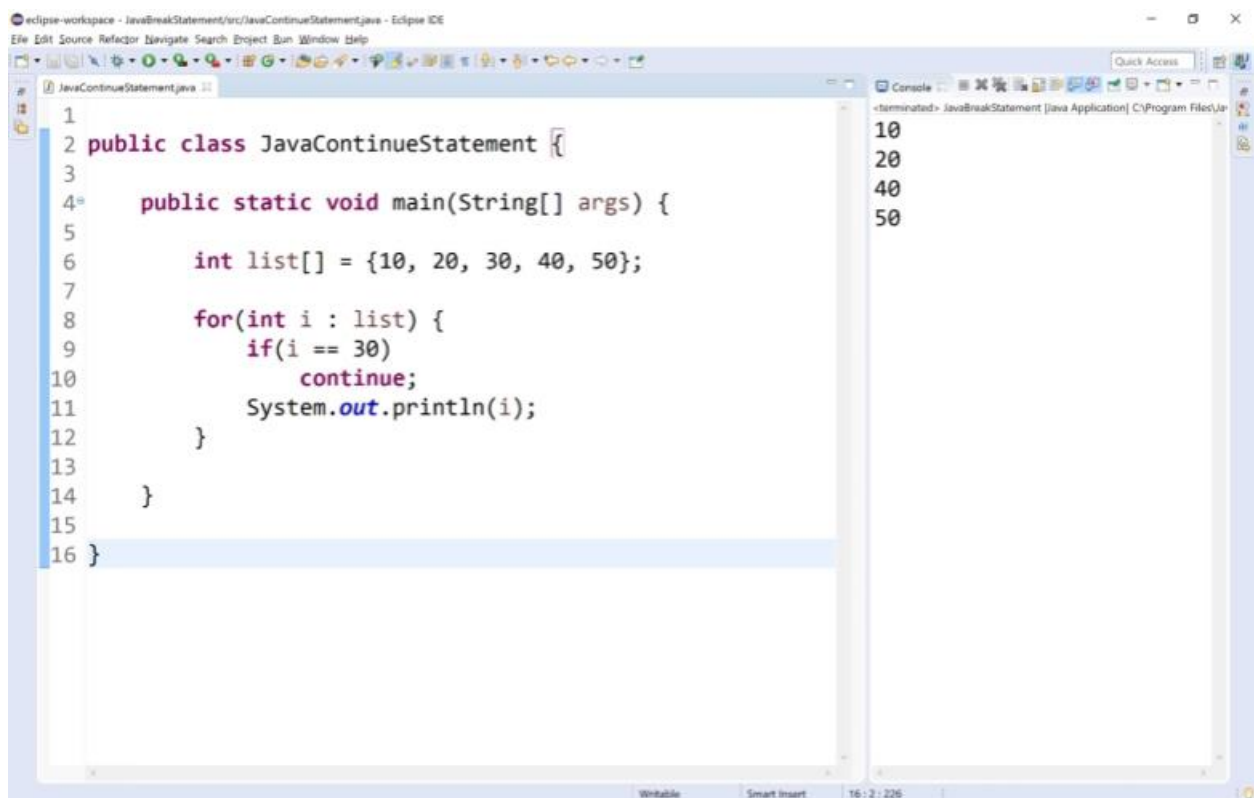
        for(int i : list) {
            if(i == 30)
                continue;
            System.out.println(i);
        }

    }

}

```

When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE interface. The main editor window displays the source code for `JavaContinueStatement.java`, which is identical to the code block above. The code is as follows:

```

1 public class JavaContinueStatement {
2
3     public static void main(String[] args) {
4
5         int list[] = {10, 20, 30, 40, 50};
6
7         for(int i : list) {
8             if(i == 30)
9                 continue;
10            System.out.println(i);
11        }
12    }
13
14 }
15
16 }

```

The right-hand side of the IDE shows the 'Console' window. It displays the output of the program, which consists of the numbers 10, 20, 40, and 50, each on a new line. The number 30 is not present in the output, demonstrating the effect of the `continue` statement. The console title bar indicates the application is terminated.

Labelled break and continue statement in java

The java programming language does not support `goto` statement, alternatively, the break and continue statements can be used with label.

The labelled break statement terminates the block with specified label. The labeled continue statement takes the execution control to the beginning of a loop with specified label.

Let's look at the following example java code.

Java Program

```
import java.util.Scanner;

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

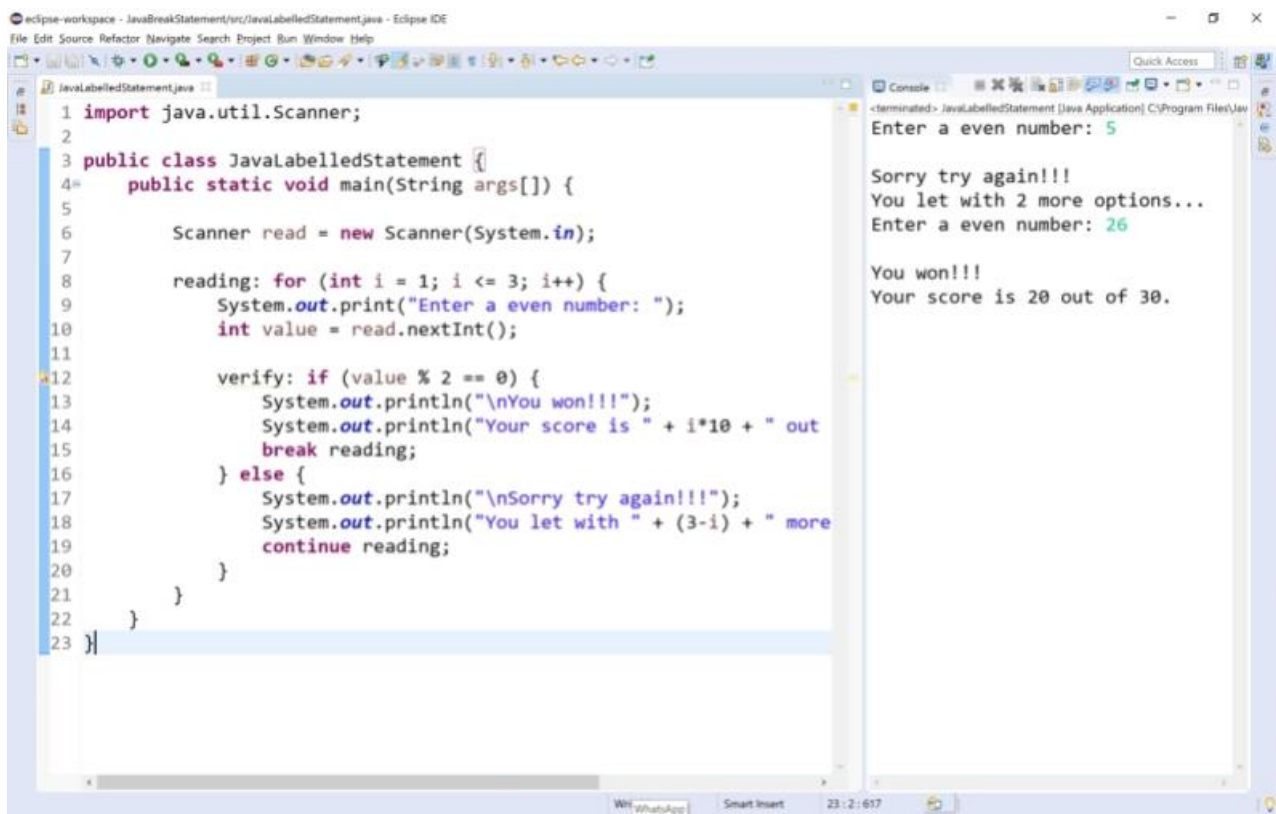
        Scanner read = new Scanner(System.in);

        reading: for (int i = 1; i <= 3; i++) {
            System.out.print("Enter a even number: ");
            int value = read.nextInt();

            verify: if (value % 2 == 0) {
                System.out.println("\nYou won!!!");
                System.out.println("Your score is " + i*10 + " out of 30.");
                break reading;
            } else {
                System.out.println("\nSorry try again!!!");
                System.out.println("You let with " + (3-i) + " more options...");

                continue reading;
            }
        }
    }
}
```

When we run this code, it produce the following output.



return statement in java

In java, the return statement is used to terminate a method with or without a value. The return statement takes the execution control to the calling function. That means the return statement transfers the execution control from the called function to the calling function by carrying a value.

??? Java allows the use of return-statement with both, with and without return type methods.

In java, the return statement is used with both methods with and without return type. In the case of a method with the return type, the return statement is mandatory, and it is optional for a method without return type.

When a return statement is used with a return type, it carries a value of return type. But, when it is used without a return type, it does not carry any value. Instead, it simply transfers the execution control.

Let's look at the following example java code.

Java Program

```
import java.util.Scanner;

public class JavaReturnStatementExample {

    int value;

    int readValue() {

        Scanner read = new Scanner(System.in);

        System.out.print("Enter any number: ");
```

```

        return this.value=read.nextInt();
    }

    void showValue(int value) {
        for(int i = 0; i <= value; i++) {
            if(i == 5)
                return;
            System.out.println(i);
        }
    }

    public static void main(String[] args) {

        JavaReturnStatementExample obj = new JavaReturnStatementExample();

        obj.showValue(obj.readValue());

    }
}

```

When we run this code, it produce the following output.

The screenshot shows the Eclipse IDE with the file `JavaReturnStatementExample.java` open. The code is as follows:

```

1 import java.util.Scanner;
2 public class JavaReturnStatementExample {
3
4     int value;
5
6     int readValue() {
7         Scanner read = new Scanner(System.in);
8         System.out.print("Enter any number: ");
9         return this.value=read.nextInt();
10    }
11
12    void showValue(int value) {
13        for(int i = 0; i <= value; i++) {
14            if(i == 5)
15                return;
16            System.out.println(i);
17        }
18    }
19
20    public static void main(String[] args) {
21
22        JavaReturnStatementExample obj = new JavaReturnStatementExample();
23
24        obj.showValue(obj.readValue());
25
26    }
27 }
28

```

The Console window on the right shows the output of the program:

```

<terminated> JavaReturnStatementExample [Java Applicatio
Enter any number: 10
0
1
2
3
4

```

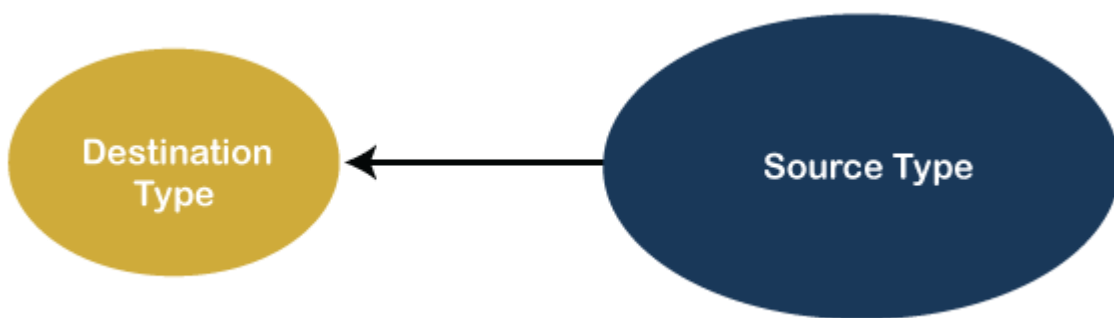
11.Type conversion and casting

The two terms **type casting** and the **type conversion** are used in a program to convert one data type to another data type. The conversion of data type is possible only by the compiler when they are compatible with each other. Let's discuss the difference between type casting and type conversion in any programming language.

What is a type casting?(or) Narrowing conversion

When a data type is converted into another data type by a programmer or user while writing a program code of any programming language, the mechanism is known as **type casting**. The programmer manually uses it to convert one data type into another. It is used if we want to change the target data type to another data type. Remember that the destination data type must be smaller than the source data type. Hence it is also called a narrowing conversion.

Double->float->long->int->char->short->byte



Syntax:

1. `Destination_datatype = (target_datatype) variable;`
2. `(data_type)` it is known as casting operator

Target_datatype: It is the data type in which we want to convert the destination data type. The variable defines a value that is to be converted in the target_data type. Let's understand the concept of type casting with an example.

Suppose, we want to convert the **float** data type into **int** data type. Here, the target data type is smaller than the source data because the size of **int** is 4 bytes, and the size of the **float** data type is 8 bytes. And when we change it, the value of the float variable is truncated and convert into an integer variable. Casting can be done with a compatible and non-compatible data type.

1. `float b = 3.0;`
2. `int a = (int) b; // converting a float value into integer`

Let's understand the type casting through a java program.

AreaOfRectangle.java

`()` is Cast operator

Datatype variable=(target-type)variable;

Java Variable example:Narrowing (typecasting)

Class Simple

```
{  
Public static void main(String args[])  
{  
float f=10.5f;  
Int a=(int)f;  
System.out.println(a);  
System.out.println(f);  
}  
}
```

o/p:-

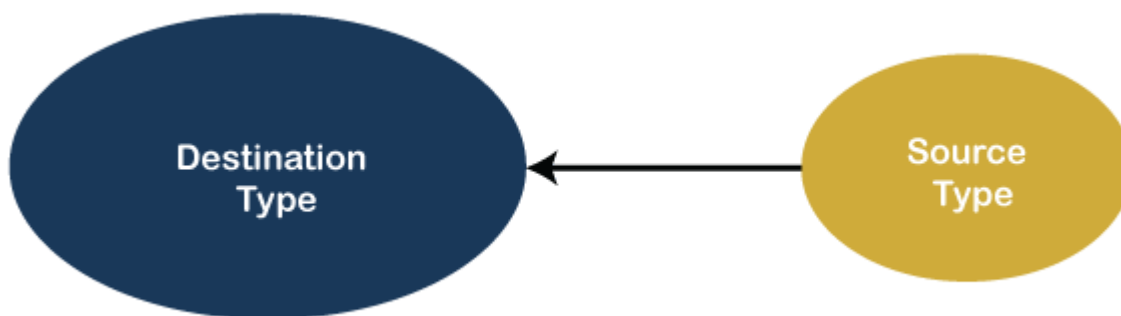
10.5

10

What is type conversion?(or) widening conversion

If a data type is automatically converted into another data type at compile time is known as type conversion. The conversion is performed by the compiler if both data types are compatible with each other. Remember that the destination data type should not be smaller than the source type. It is also known as **widening** conversion of the data type.

Byte->short->char->int->long->float->double



Let's understand the type conversion with an example.

Suppose, we have an **int** data type and want to convert it into a **float** data type. These are data types compatible with each other because their types are numeric, and the size of int is 4 bytes which is smaller than float data type. Hence, the compiler automatically converts the data types without losing or truncating the values.

- 1. `int a = 20;`
- 2. `Float b;`
- 3. `b = a; // Now the value of variable b is 20.000 /* It defines the conversion of int data type to float data type without losing the information. */`

In the above example, the int data type is converted into the float, which has a larger size than int, and hence it widens the source data type.

Let's understand type conversion through a java program.

```
Class Simple
{
Public static void main(String args[])
{
Int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}
}
```

o/p:

10
10.0

Note:

- 1.For widening conversions,the numeric types,including integer and floating-point types are compatible with each other.
- 2.No automatic conversions from the numeric types to char or Boolean. also, char and Boolean are not compatible with each other.

Difference Between Type Casting and Type Conversion

S.N.	Type Casting	Type Conversion
1	Type casting is a mechanism in which one data type is converted to another data type using a casting () operator by a programmer.	Type conversion allows a compiler to convert one data type to another data type at the compile time of a program or code.
2	It can be used both compatible data type and incompatible data type.	Type conversion is only used with compatible data types, and hence it does not require any casting operator.

3	It requires a programmer to manually casting one data into another type.	It does not require any programmer intervention to convert one data type to another because the compiler automatically compiles it at the run time of a program.
4	It is used while designing a program by the programmer.	It is used or take place at the compile time of a program.
5	When casting one data type to another, the destination data type must be smaller than the source data.	When converting one data type to another, the destination type should be greater than the source data type.
6	It is also known as narrowing conversion because one larger data type converts to a smaller data type.	It is also known as widening conversion because one smaller data type converts to a larger data type.
7	It is more reliable and efficient.	It is less efficient and less reliable.
8	There is a possibility of data or information being lost in type casting.	In type conversion, data is unlikely to be lost when converting from a small to a large data type.
8	float b = 3.0; int a = (int) b	int x = 5, y = 2, c; float q = 12.5, p; p = q/x;

12.Simple Java programs

1.Fibonacci series in Java

In fibonacci series, *next number is the sum of previous two numbers* for example 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 etc. The first two numbers of fibonacci series are 0 and 1.

Example:-

```

1. class FibonacciExample1{
2. public static void main(String args[])
3. {
4. int n1=0,n2=1,n3,i,count=10;
5. System.out.print(n1+" "+n2);//printing 0 and 1
6.
7. for(i=2;i<count;++i)//loop starts from 2 because 0 and 1 are already printed
8. {
9. n3=n1+n2;
10. System.out.print(" "+n3);
11. n1=n2;
```

```

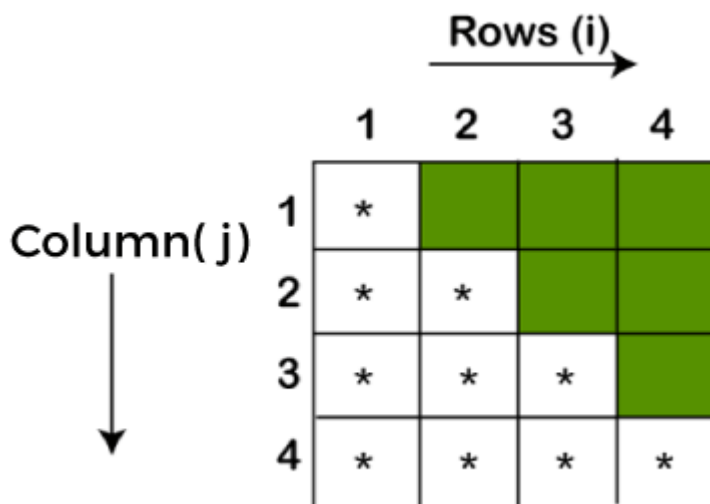
12. n2=n3;
13. }
14.
15. }}

```

Output:-

0 1 1 2 3 5 8 13 21 34

2.Right Triangle Star Pattern



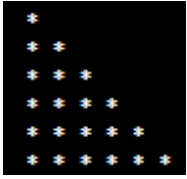
```

1. public class RightTrianglePattern
2. {
3.     public static void main(String args[])
4.     {
5.         //i for rows and j for columns
6.         //row denotes the number of rows you want to print
7.         int i, j, row=6;
8.         //outer loop for rows
9.         for(i=0; i<row; i++)
10.        {
11.            //inner loop for columns
12.            for(j=0; j<=i; j++)
13.            {
14.                //prints stars
15.                System.out.print("* ");
16.            }
17.            //throws the cursor in a new line after printing each line
18.            System.out.println();

```

19. }
20. }
21. }

Output:-



3.Program to print the duplicate elements of an array

In this program, we need to print the duplicate elements present in the array. This can be done through two loops. The first loop will select an element and the second loop will iteration through the array by comparing the selected element with other elements. If a match is found, print the duplicate element.

1	2	3	4	2	7	8	8	3
---	---	---	---	---	---	---	---	---

In the above array, the first duplicate will be found at the index 4 which is the duplicate of the element (2) present at index 1. So, duplicate elements in the above array are 2, 3 and 8.

Output:-

```
1. public class DuplicateElement {  
2.     public static void main(String[] args) {  
3.         //Initialize array  
4.         int [] arr = new int [] {1, 2, 3, 4, 2, 7, 8, 8, 3};  
5.         System.out.println("Duplicate elements in given array: ");  
6.         //Searches for duplicate element  
7.         for(int i = 0; i < arr.length; i++) {  
8.             for(int j = i + 1; j < arr.length; j++) {  
9.                 if(arr[i] == arr[j])  
10.                    System.out.println(arr[j]);  
11.             }  
12.         }  
13.     }  
14. }
```

Output:-

Duplicate elements in given array:

2
3
8

4. Java Program to count the total number of vowels and consonants in a string

In this program, our task is to count the total number of vowels and consonants present in the given string.

As we know that, the characters a, e, i, o, u are known as vowels in the English alphabet. Any character other than that is known as the consonant.

To solve this problem, First of all, we need to convert every upper-case character in the string to lower-case so that the comparisons can be done with the lower-case vowels only not upper-case vowels, i.e.(A, E, I, O, U). Then, we have to traverse the string using a for or while loop and match each character with all the vowels, i.e., a, e, i, o, u. If the match is found, increase the value of count by 1 otherwise continue with the normal flow of the program.

Example:-

```
1. public class CountVowelConsonant {
2.     public static void main(String[] args) {
3.
4.         //Counter variable to store the count of vowels and consonant
5.         int vCount = 0, cCount = 0;
6.
7.         //Declare a string
8.         String str = "This is a really simple sentence";
9.
10.        //Converting entire string to lower case to reduce the comparisons
11.        str = str.toLowerCase();
12.
13.        for(int i = 0; i < str.length(); i++) {
14.            //Checks whether a character is a vowel
15.            if(str.charAt(i) == 'a' || str.charAt(i) == 'e' || str.charAt(i) == 'i' || str.charAt(i) == 'o' || str.charAt(i) == 'u')
16.            {
17.                //Increments the vowel counter
18.                vCount++;
19.            }
20.            //Checks whether a character is a consonant
21.            else if(str.charAt(i) >= 'a' && str.charAt(i) <= 'z') {
22.                //Increments the consonant counter
23.                cCount++;
24.            }
25.        }
26.        System.out.println("Number of vowels: " + vCount);
27.        System.out.println("Number of consonants: " + cCount);
28.    }
```

Output:-

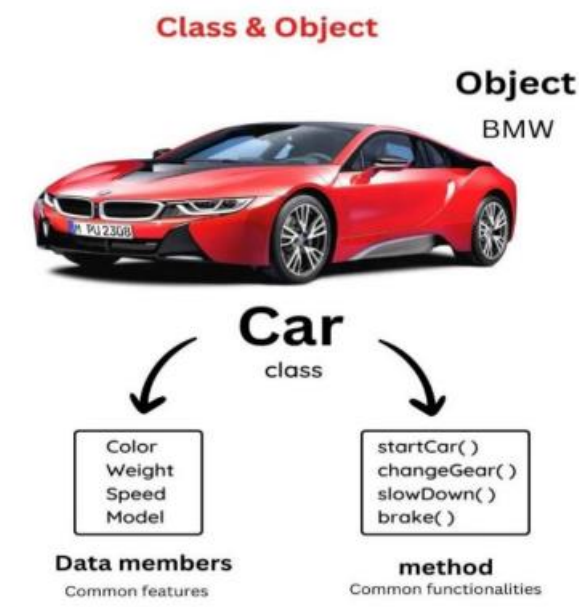
Number of vowels: 10

Number of consonants: 17

13. Concepts of Classes:-

Definition: - A class is a template or blueprint from which you can create an individual object. Class doesn't consume any space. It is logical entity.

Ex:-Car



A class in java can contain:

- i. Fields or variables
- ii. Methods
- iii. Constructors
- iv. Blocks
- v. Nested class and interface

Syntax to declare a class:-

```
access_modifier class<class_name>
```

```
{
```

```
    data member;
```

```
    method;
```

```
    constructor;
```

```
nested class;  
  
interface;  
  
}
```

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:

1. **Modifiers:** A class can be public or has default access.
2. **Class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **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.
5. **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.
6. **Body:** The class body is surrounded by braces, { }.

Example Program:-

```
public class Car  
{  
    String color;  
    int weight;  
    int speed;  
    String model;  
    void startCar()  
    {  
        System.out.println("car Started");  
    }  
    void changeGear()  
    {  
        System.out.println("Gear Changed");  
    }  
    void slowDown()  
    {  
        System.out.println("Going slowly");  
    }  
    void brake()  
    {  
        System.out.println("Sudden Brake");  
    }  
}
```

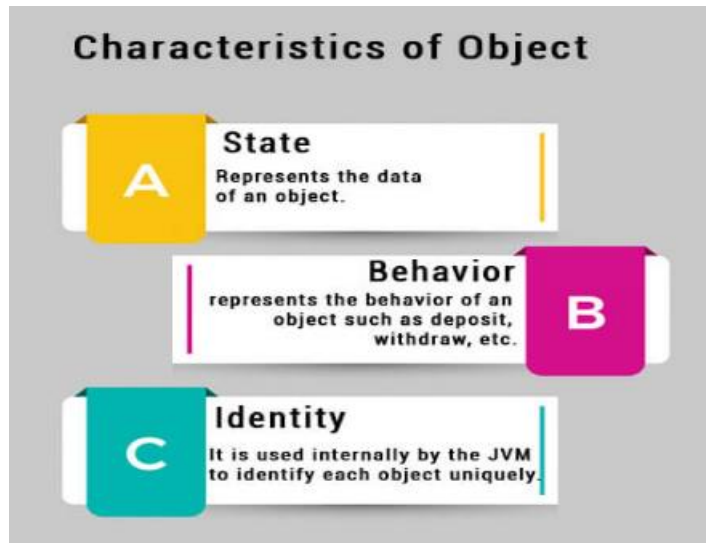
14.Object: -

Definition: - Object is a real world entity that we can touch physically is referred as object such as a pen, chair, table, computer, watch, etc., Object is an instance of a class. Each entity will be having its own state and

behavior and it is referring as an object. An object contains an address and takes up some space in memory.

Example:-

Objects of the class car can be BMW, Mercedes, Ferrari, etc.



New:-

The new keyword is used to allocate memory at runtime. All objects get memory in heap memory area.

Syntax:-

```
<class_name> <object_name>=new <class_name>();
```

(Or)

```
<class_name> <object_name>;
```

```
<object_name>=new <class_name>();
```

Example Creating Object for Car:-

```
Car BMW=new BMW();
```

```
Car Ferrari=new Ferrari();
```

(or)

```
Car BMW; //creating BMW reference variable for class Car
```

```
BMW=new BMW(); //now BMW object is created in heap memory
```

Example Program creating Objects and Class:-

```
class Car
{
String color="red";
int weight=50;
int speed=100;
String model="1";
void startCar()
{
    System.out.println("car Started");
}
```

```

}
void changeGear()
{
    System.out.println("Gear Changed");
}
void slowDown()
{
    System.out.println("Going slowly");
}
void brake()
{
    System.out.println("Sudden Brake");
}
public static void main(String args[])// objects are always create in main method because JVM is
executed main method directly
{
    Car bmw=new Car();
    Car ferrari;
    ferrari=new Car();
    System.out.println(bmw.color);//calling variable
    ferrari.color="block";//initializing a value through object
    System.out.println(ferrari.color);
    bmw.startCar();//calling method
    ferrari.startCar();
}
}

```

Output:-

```

red
block
car Started
car Started

```

15. Constructors

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:

1. `<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.

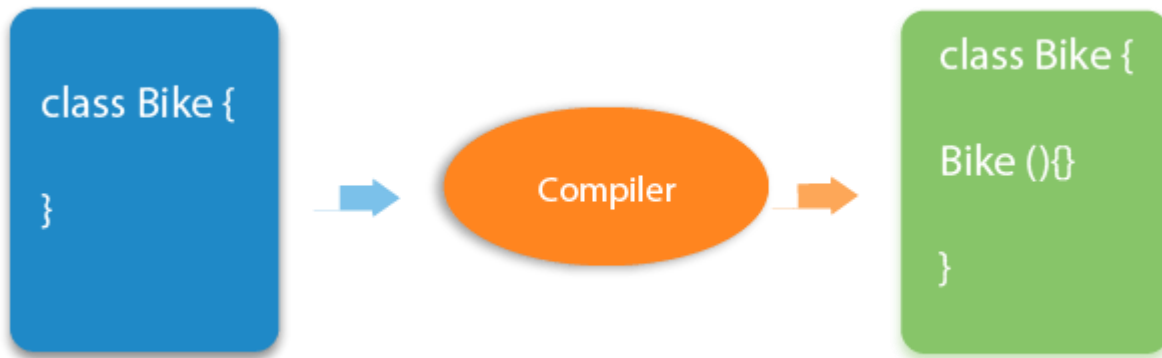
1. `//Java Program to create and call a default constructor`
2. `class Bike1{`
3. `//creating a default constructor`
4. `Bike1(){System.out.println("Bike is created");}`
5. `//main method`
6. `public static void main(String args[]){`
7. `//calling a default constructor`
8. `Bike1 b=new Bike1();`
9. `}`
10. `}`

[Test it Now](#)

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

```
1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4.     int id;
5.     String name;
6.     //method to display the value of id and name
7.     void display(){System.out.println(id+" "+name);}
8.
9.     public static void main(String args[]){
10.        //creating objects
11.        Student3 s1=new Student3();
12.        Student3 s2=new Student3();
13.        //displaying values of the object
14.        s1.display();
15.        s2.display();
16.    }
17. }
```

Test it Now

Output:

```
0 null
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.

```
1. //Java Program to demonstrate the use of the parameterized constructor.
2. class Student4{
3.     int id;
4.     String name;
5.     //creating a parameterized constructor
6.     Student4(int i,String n){
7.         id = i;
8.         name = n;
9.     }
10.    //method to display the values
11.    void display(){System.out.println(id+" "+name);}
12.
13.    public static void main(String args[]){
14.        //creating objects and passing values
15.        Student4 s1 = new Student4(111,"Karan");
16.        Student4 s2 = new Student4(222,"Aryan");
17.        //calling method to display the values of object
18.        s1.display();
19.        s2.display();
20.    }
21. }
```

[Test it Now](#)

Output:

```
111 Karan
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.

Example of Constructor Overloading

```
1. //Java program to overload constructors
2. class Student5{
3.     int id;
4.     String name;
5.     int age;
6.     //creating two arg constructor
7.     Student5(int i,String n){
8.         id = i;
9.         name = n;
10.    }
11.    //creating three arg constructor
12.    Student5(int i,String n,int a){
13.        id = i;
14.        name = n;
15.        age=a;
16.    }
17.    void display(){System.out.println(id+" "+name+" "+age);}
18.
19.    public static void main(String args[]){
20.        Student5 s1 = new Student5(111,"Karan");
21.        Student5 s2 = new Student5(222,"Aryan",25);
22.        s1.display();
23.        s2.display();
24.    }
25. }
```

[Test it Now](#)

Output:

```
111 Karan 0
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.

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.

16.Methods

What is a method in Java?

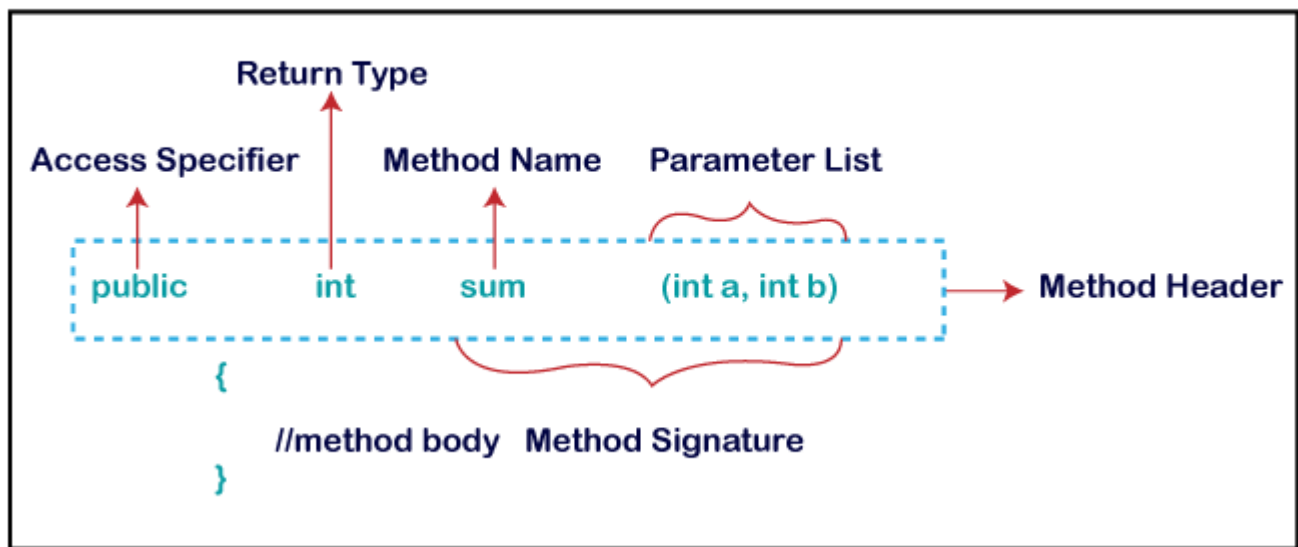
A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main()** method.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.

Method Declaration



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparison()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

● Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length()**, **equals()**, **compareTo()**, **sqrt()**, etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

```
1. public class Demo
2. {
3.     public static void main(String[] args)
4.     {
5.         // using the max() method of Math class
6.         System.out.print("The maximum number is: " + Math.max(9,7));
7.     }
8. }
```

Output:

In the above example, we have used three predefined methods **main()**, **print()**, and **max()**. We have used these methods directly without declaration because they are predefined. The **print()** method is a method of **PrintStream** class that prints the result on the console. The **max()** method is a method of the **Math** class that returns the greater of two numbers.

max
<pre>public static int max(int a, int b)</pre>
Returns the greater of two int values. same value.
Parameters: a - an argument. b - another argument.
Returns: the larger of a and b.

In the above method signature, we see that the method signature has access specifier **public**, non-access modifier **static**, return type **int**, method name **max()**, parameter list (**int a, int b**). In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.

Similarly, we can also see the method signature of the **print()** method.

• User-defined Method

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

1. `//user defined method`
2. `public static void findEvenOdd(int num)`
3. `{`
4. `//method body`
5. `if(num%2==0)`
6. `System.out.println(num+" is even");`
7. `else`
8. `System.out.println(num+" is odd");`
9. `}`

We have defined the above method named **findevenodd()**. It has a parameter **num** of type **int**. The method does not return any value that's why we have used **void**. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

How to Call or Invoke a User-defined Method

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
1. import java.util.Scanner;
2. public class EvenOdd
3. {
4.     public static void main (String args[])
5.     {
6.         //creating Scanner class object
7.         Scanner scan=new Scanner(System.in);
8.         System.out.print("Enter the number: ");
9.         //reading value from the user
10.        int num=scan.nextInt();
11.        //method calling
12.        findEvenOdd(num);
13.    }
```

In the above code snippet, as soon as the compiler reaches at line **findEvenOdd(num)**, the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

EvenOdd.java

```
1. import java.util.Scanner;
2. public class EvenOdd
3. {
4.     public static void main (String args[])
5.     {
6.         //creating Scanner class object
7.         Scanner scan=new Scanner(System.in);
8.         System.out.print("Enter the number: ");
9.         //reading value from user
10.        int num=scan.nextInt();
11.        //method calling
12.        findEvenOdd(num);
13.    }
14.    //user defined method
15.    public static void findEvenOdd(int num)
16.    {
17.        //method body
18.        if(num%2==0)
19.            System.out.println(num+" is even");
```

```
20. else
21. System.out.println(num+" is odd");
22. }
23. }
```

Output 1:

```
Enter the number: 12
12 is even
```

Output 2:

```
Enter the number: 99
99 is odd
```

Let's see another program that return a value to the calling method.

In the following program, we have defined a method named **add()** that sum up the two numbers. It has two parameters n1 and n2 of integer type. The values of n1 and n2 correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

Addition.java

```
1. public class Addition
2. {
3.     public static void main(String[] args)
4.     {
5.         int a = 19;
6.         int b = 5;
7.         //method calling
8.         int c = add(a, b); //a and b are actual parameters
9.         System.out.println("The sum of a and b is= " + c);
10.    }
11.    //user defined method
12.    public static int add(int n1, int n2) //n1 and n2 are formal parameters
13.    {
14.        int s;
15.        s=n1+n2;
16.        return s; //returning the sum
17.    }
18. }
```

Output:

```
The sum of a and b is= 24
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Example of static method

Display.java

```
1. public class Display
2. {
3.     public static void main(String[] args)
4.     {
5.         show();
6.     }
7.     static void show()
8.     {
9.         System.out.println("It is an example of static method.");
10.    }
11. }
```

Output:

```
It is an example of a static method.
```

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

InstanceMethodExample.java

```
1. public class InstanceMethodExample
2. {
3.     public static void main(String [] args)
4.     {
5.         //Creating an object of the class
6.         InstanceMethodExample obj = new InstanceMethodExample();
7.         //invoking instance method
8.         System.out.println("The sum is: "+obj.add(12, 13));
9.     }
10.    int s;
11.    //user-defined method because we have not used static keyword
```

```
12. public int add(int a, int b)
13. {
14. s = a+b;
15. //returning the sum
16. return s;
17. }
18. }
```

Output:

```
The sum is: 25
```

There are two types of instance method:

- **Accessor Method**
- **Mutator Method**

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. It is also known as **getters**. It returns the value of the private field. It is used to get the value of the private field.

Example

```
1. public int getId()
2. {
3. return Id;
4. }
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word **set**. It is also known as **setters** or **modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
1. public void setRoll(int roll)
2. {
3. this.roll = roll;
4. }
```

Example of accessor and mutator method

Student.java

```
1. public class Student
2. {
3. private int roll;
4. private String name;
5. public int getRoll() //accessor method
```

```

6. {
7.     return roll;
8. }
9. public void setRoll(int roll) //mutator method
10. {
11.     this.roll = roll;
12. }
13. public String getName()
14. {
15.     return name;
16. }
17. public void setName(String name)
18. {
19.     this.name = name;
20. }
21. public void display()
22. {
23.     System.out.println("Roll no.: "+roll);
24.     System.out.println("Student name: "+name);
25. }
26. }

```

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

Syntax

```

1. abstract void method_name();

```

Example of abstract method

Demo.java

```

1. abstract class Demo //abstract class
2. {
3.     //abstract method declaration
4.     abstract void display();
5. }
6. public class MyClass extends Demo
7. {
8.     //method impelmentation
9.     void display()
10. {

```

```

11. System.out.println("Abstract method?");
12. }
13. public static void main(String args[])
14. {
15. //creating object of abstract class
16. Demo obj = new MyClass();
17. //invoking abstract method
18. obj.display();
19. }
20. }

```

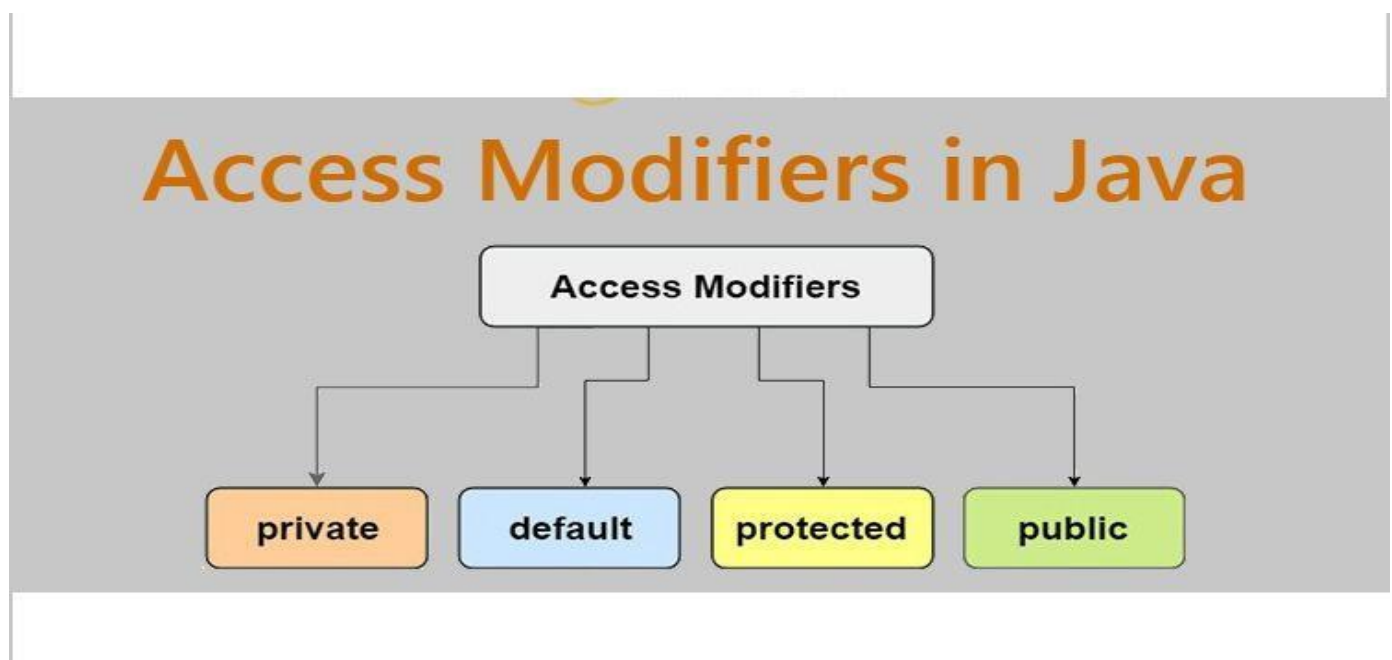
Output:

Abstract method...

17.Access control

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:



1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

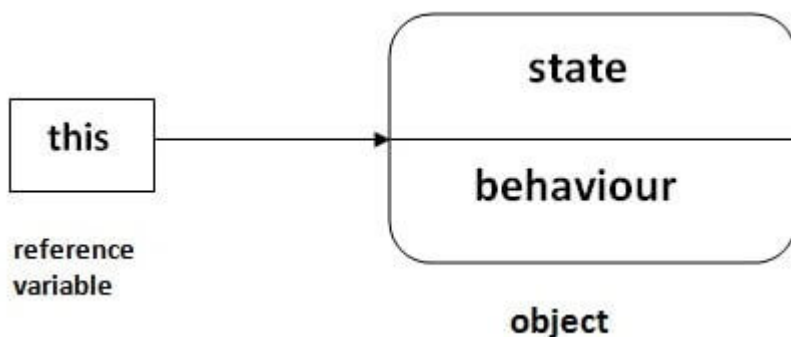
Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

18.This keyword

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.

Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01

this can be used to refer current class instance variable.

04

this can be passed as an argument in the method call.

02

this can be used to invoke current class method (implicitly).

05

this can be passed as argument in the constructor call.

03

this() can be used to invoke current class Constructor.

06

this can be used to return the current class instance from the method

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:

```
1. class Student{
2.     int rollno;
3.     String name;
4.     float fee;
5.     Student(int rollno,String name,float fee){
6.         rollno=rollno;
7.         name=name;
8.         fee=fee;
9.     }
10.    void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. class TestThis1{
13.     public static void main(String args[]){
14.         Student s1=new Student(111,"ankit",5000f);
15.         Student s2=new Student(112,"sumit",6000f);
```



```
16. s1.display();
17. s2.display();
18. }}
```

Test it Now

Output:

```
0 null 0.0
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

```
1. class Student{
2.     int rollno;
3.     String name;
4.     float fee;
5.     Student(int rollno,String name,float fee){
6.         this.rollno=rollno;
7.         this.name=name;
8.         this.fee=fee;
9.     }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis2{
14.     public static void main(String args[]){
15.         Student s1=new Student(111,"ankit",5000f);
16.         Student s2=new Student(112,"sumit",6000f);
17.         s1.display();
18.         s2.display();
19.     }}
```

Test it Now

Output:

```
111 ankit 5000.0
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

```
1. class Student{
2.     int rollno;
3.     String name;
4.     float fee;
```

```

5. Student(int r,String n,float f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis3{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

```

[Test it Now](#)

Output:

```

111 ankit 5000.0
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.

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.

```

1. class A5{
2. void m(){
3. System.out.println(this);//prints same reference ID
4. }
5. public static void main(String args[]){
6. A5 obj=new A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }

```

[Test it Now](#)

Output:

```

A5@22b3ea59
A5@22b3ea59

```

19. Garbage collection

- In java, garbage means unreferenced objects
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
To achieve this we can use
- free() function in C language and Delete() in C++
But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

- **How to unreferenced object?**

There are 3 ways to unreferenced object

- 1.By nulling the reference**

Ex:- Animal a=new Animal();
a=null;

- 2. By assigning a reference to another**

Ex:- Animal a1=new Animal();
Animal a2=new Animal();
a1=a2;

- 3.By anonymous object**

Ex:- new Animal();

- **Finalize() method**

It is invoked each time before the object is garbage collected.

This method can be used to perform cleanup processing.

This method is defined in Object class as

Protected void finalize(){}

- **gc() method**

It is used to invoke the garbage collector to perform cleanup processing.

The gc() is found in System and Runtime classes.

It is static method which is used to call finalize before destroying the object.

public static void gc(){}

- **Example Program:-**

```
class Animal
{
    Animal(){
        System.out.println("Object is created");
    }
    protected void finalize()
    {
        System.out.println("Object is destroyed");
    }
}
```

```

class AnimalDemo
{
public static void main(String args[])
{
Animal a=new Animal();
a=null;//1.By assigning a null
Animal a1=new Animal();
Animal a2=new Animal();
a1=a2;//2. By assigning reference to another
new Animal();//By anonymous object
System.gc();
}
}

```

Output:-

Object is created
Object is created
Object is created
Object is created
Object is destroyed
Object is destroyed
Object is destroyed

20.Overloading methods

Defination:-

If a [class](#) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

Advantage of method overloading

Method overloading *increases the readability of the program.*

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Note:- In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating [static methods](#) so that we don't need to create instance for calling methods.

```

1. class Adder{
2.     static int add(int a,int b){return a+b;}
3.     static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6.     public static void main(String[] args){
7.         System.out.println(Adder.add(11,11));
8.         System.out.println(Adder.add(11,11,11));
9.     }}

```

[Test it Now](#)

Output:

```

22
33

```

Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in [data type](#). The first add method receives two integer arguments and second add method receives two double arguments.

```

1. class Adder{
2.     static int add(int a, int b){return a+b;}
3.     static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6.     public static void main(String[] args){
7.         System.out.println(Adder.add(11,11));
8.         System.out.println(Adder.add(12.3,12.6));
9.     }}

```

[Test it Now](#)

Output:

```

22
24.9

```

21.Parameter passing

There are different ways in which parameter data can be passed into and out of [methods and functions](#). Let us assume that a function $B()$ is called from another function $A()$. In this case A is called the “*caller function*” and B is called the “*called function or callee function*”. Also, the arguments which A sends to B are called *actual arguments* and the parameters of B are called *formal arguments*.

Types of parameters:

Formal Parameter: A variable and its type as they appear in the prototype of the function or method.

Syntax:

function_name(datatype variable_name)

Actual Parameter: The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Syntax:

func_name(variable name(s));

Two ways passing parameters into java

1.Pass by value

2.Pass by reference

1.Pass by Value (or) call by value: In the pass by value concept, the method is called by passing a value. So, it is called pass by value. It does not affect the original parameter.

In case of call by value original value is not changed.

Example:-

```
1. class Operation{
2.     int data=50;
3.
4.     void change(int data){
5.         data=data+100;//changes will be in the local variable only
6.     }
7.
8.     public static void main(String args[]){
9.         Operation op=new Operation();
10.
11.         System.out.println("before change "+op.data);
12.         op.change(500);
13.         System.out.println("after change "+op.data);
14.
15.     }
16. }
```

Output:

before change 50
after change 50

2.Pass by Reference (or) call by reference: In the pass by reference concept, the method is called using an alias or reference of the actual parameter. So, it is called pass by reference. It forwards the unique identifier of the object to the method. If we made changes to the parameter's instance member, it would affect the original value.

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value.

Let's take a simple example:

```
1. class Operation2{
2.     int data=50;
3.
4.     void change(Operation2 op){
5.         op.data=op.data+100;//changes will be in the instance variable
6.     }
7.
8.
9.     public static void main(String args[]){
10.        Operation2 op=new Operation2();
11.
12.        System.out.println("before change "+op.data);
13.        op.change(op);//passing object
14.        System.out.println("after change "+op.data);
15.
16.    }
17. }
```

Output:

```
before change 50
after change 150
```

Why [Java](#) does not support pass by reference concept?

Java does not support call by reference because in call by reference we need to pass the address and address are stored in pointers n java does not support pointers and it is because pointers breaks the security. Java is always pass-by-value.

22.Recursion

a [method](#) that calls itself is known as a recursive method. And, this process is known as recursion. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

How Recursion works?

```

public static void main(String[] args) {
    ... ..
    recurse() .....
    ... ..
}

static void recurse() {
    ... ..
    recurse() .....
    ... ..
}

```

The diagram illustrates two types of method calls. A 'Normal Method Call' is shown as a horizontal dashed arrow from the `recurse()` line inside the `main` method to the `recurse()` line inside the `recurse` method. A 'Recursive Call' is shown as a horizontal dashed arrow from the `recurse()` line inside the `recurse` method to another `recurse()` line further down within the same `recurse` method block.

In the above example, we have called the `recurse()` method from inside the `main` method. (normal method call). And, inside the `recurse()` method, we are again calling the same `recurse` method. This is a recursive call.

In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.

Hence, we use the [if...else statement](#) (or similar approach) to terminate the recursive call inside the method.

Example: Factorial of a Number Using Recursion

```

class Factorial {

    static int factorial( int n ) {
        if (n != 0) // termination condition
            return n * factorial(n-1); // recursive call
        else
            return 1;
    }

    public static void main(String[] args) {
        int number = 4, result;
        result = factorial(number);
        System.out.println(number + " factorial = " + result);
    }
}

```

Run Code

Output:

```
4 factorial = 24
```

In the above example, we have a method named `factorial()`. The `factorial()` is called from the `main()` method. with the `number` variable passed as an argument.

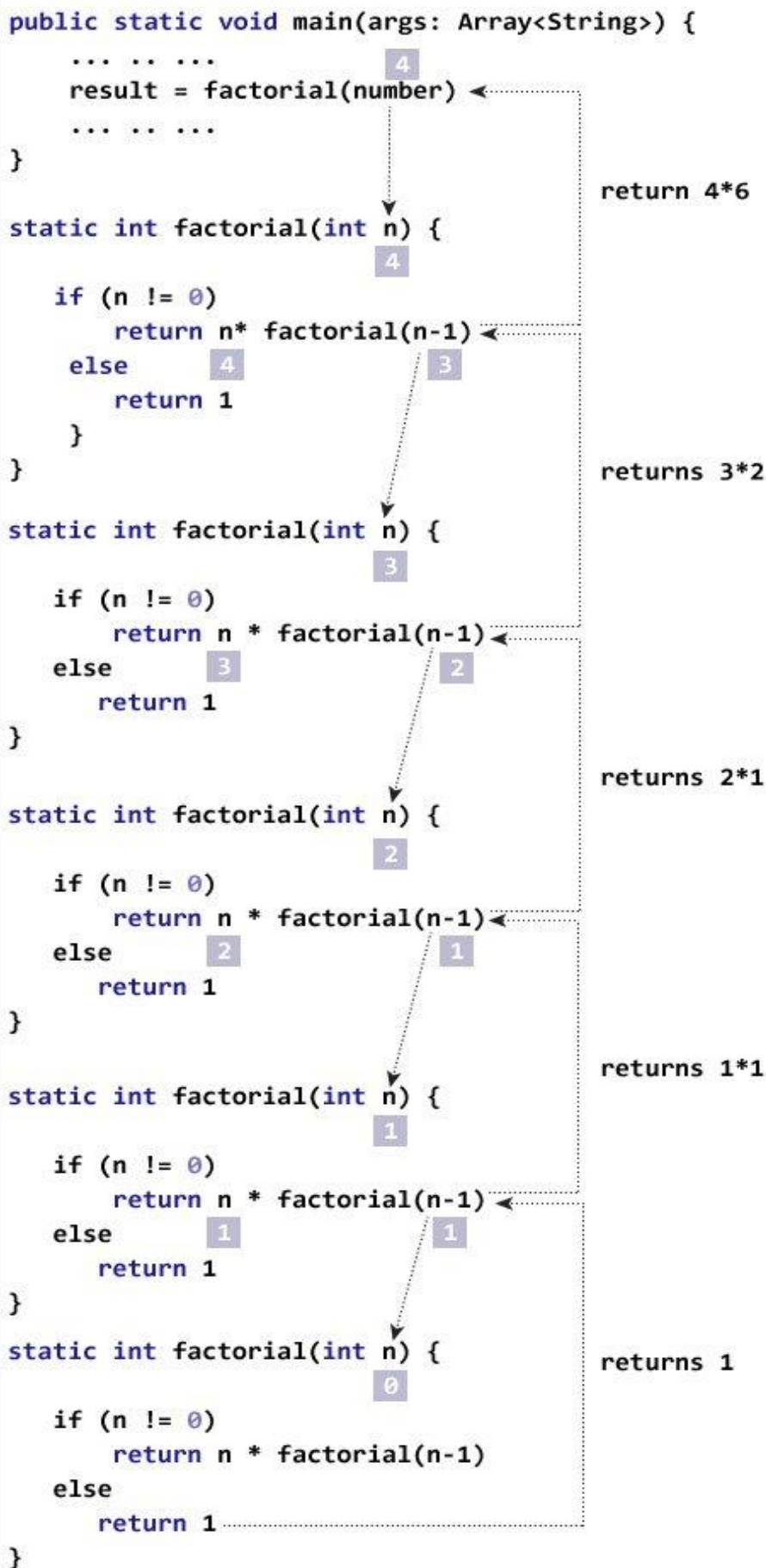
Here, notice the statement,


```
return n * factorial(n-1);
```

The `factorial()` method is calling itself. Initially, the value of `n` is 4 inside `factorial()`. During the next recursive call, 3 is passed to the `factorial()` method. This process continues until `n` is equal to 0. When `n` is equal to 0, the `if` statement returns false hence 1 is returned. Finally, the accumulated result is passed to the `main()` method.

Working of Factorial Program

The image below will give you a better idea of how the factorial program is executed using recursion.

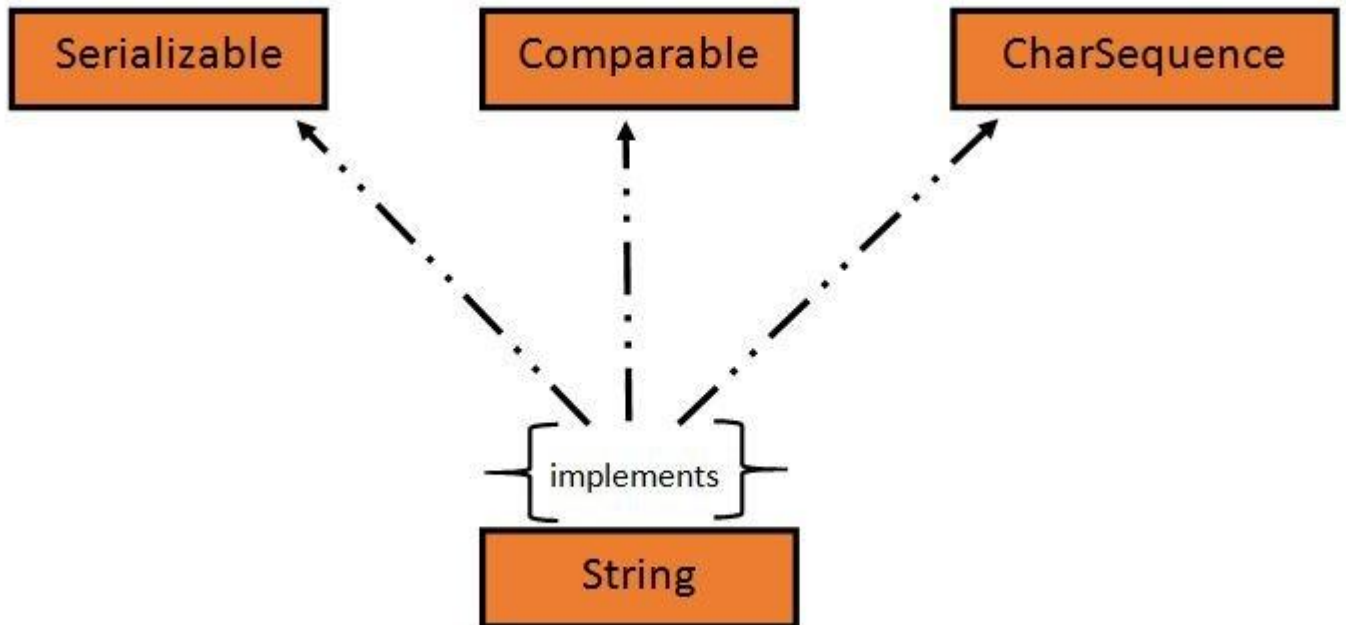


Factorial Program using Recursion

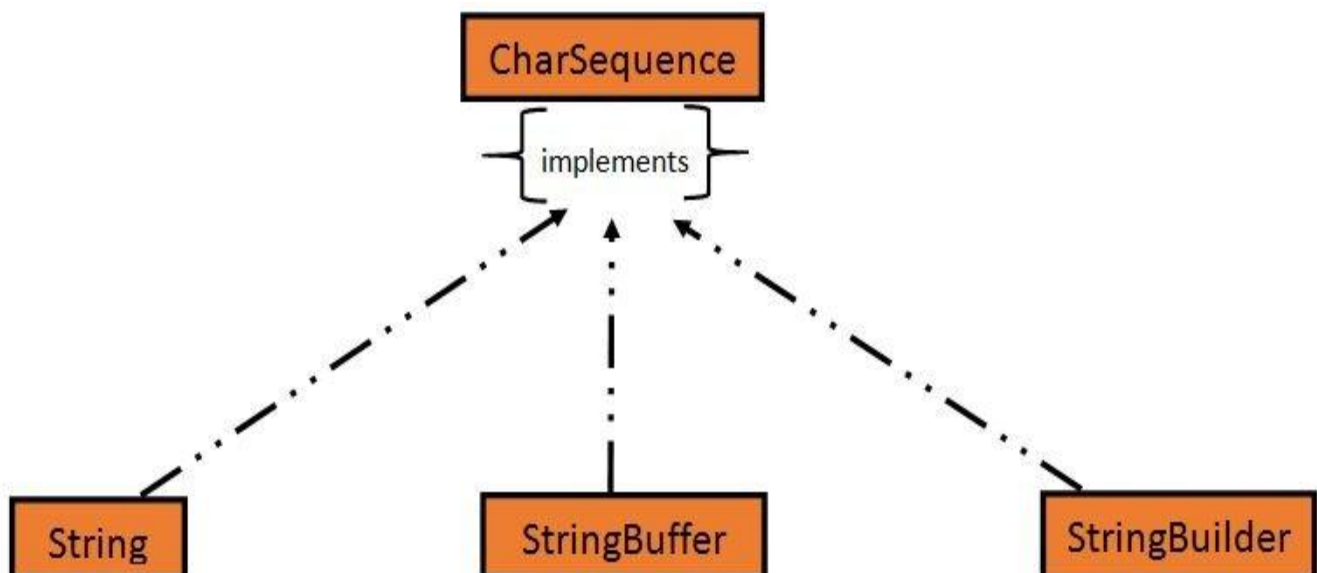
23.Exploring String Class

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created. String is represented by String class which is located into `java.lang` package.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.



In Java, **CharSequence** Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.



What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

- **Creating a String object**

String can be created in number of ways, here are a few ways of creating string object.

1) Using a String literal

String literal is a simple string enclosed in double quotes `" "`. A string literal is treated as a String object.

```
public class Demo{  
  
    public static void main(String[] args) {  
  
        String s1 = "Hello Java";  
  
        System.out.println(s1);  
  
    }  
  
}
```

Hello Java

2) Using new Keyword

We can create a new string object by using **new** operator that allocates memory for the object.

```
public class Demo{  
  
    public static void main(String[] args) {  
  
        String s1 = new String("Hello Java");  
  
        System.out.println(s1);  
  
    }  
  
}
```

```
}
```

```
}
```

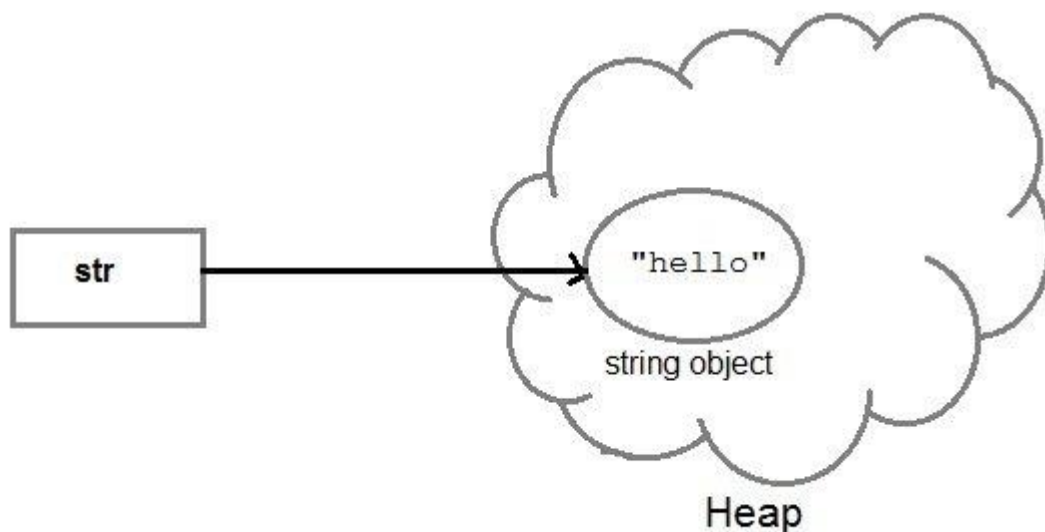
Hello Java

Each time we create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as **string constant pool** inside the heap memory.

- **String object and How they are stored**

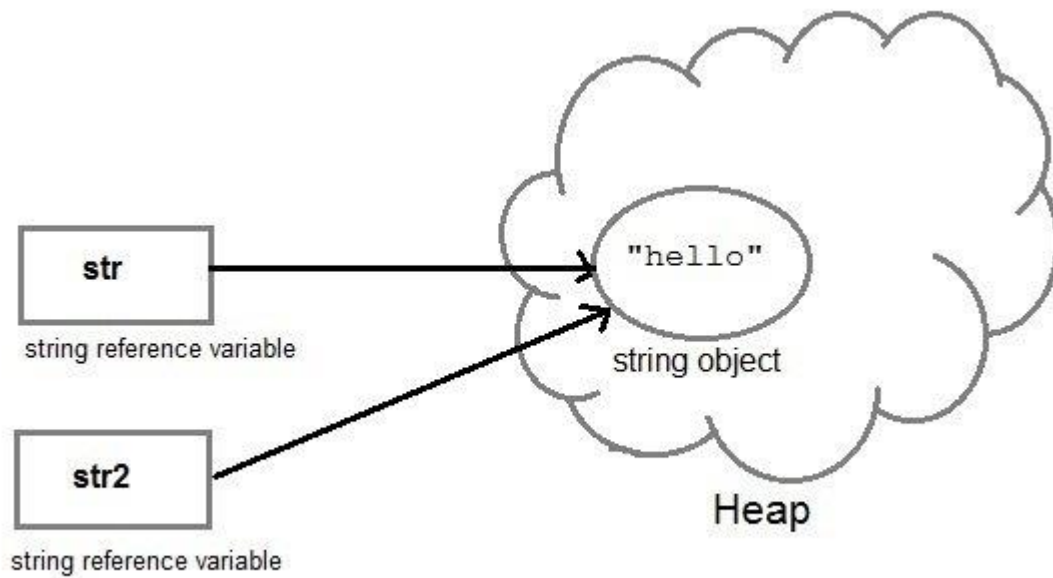
When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already.

```
String str= "Hello";
```



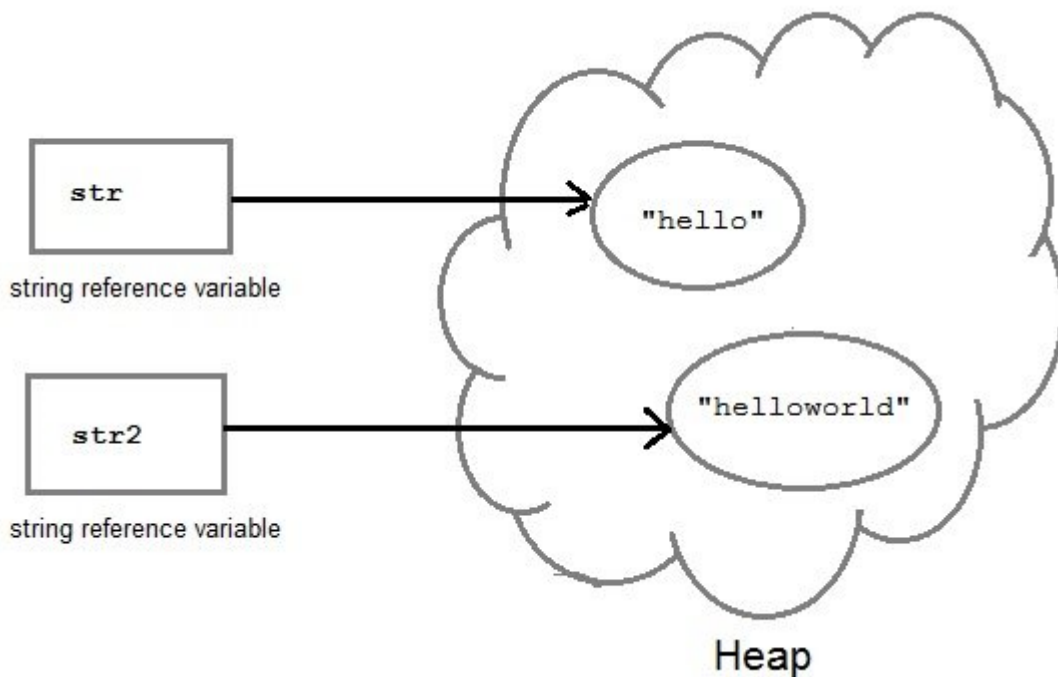
And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

```
String str2 = str;
```



But if we change the new string, its reference gets modified.

```
str2=str2.concat("world");
```



- **Concatenating String**

There are 2 methods to concatenate two or more string.

1. Using **concat()** method
2. Using **+** operator

1) Using concat() method

Concat() method is used to add two or more string into a single string object. It is string class method and returns a string object.

```
public class Demo{

    public static void main(String[] args) {

        String s = "Hello";

        String str = "Java";

        String str1 = s.concat(str);

        System.out.println(str1);

    }

}
```

Copy

HelloJava

2) Using + operator

Java uses "+" operator to concatenate two string objects into single one. It can also concatenate numeric value with string object. See the below example.

```
public class Demo{

    public static void main(String[] args) {

        String s = "Hello";

        String str = "Java";
```

```
String str1 = s+str;

String str2 = "Java"+11;

System.out.println(str1);

System.out.println(str2);

}

}
```

Copy

HelloJava

Java11

- **String Comparison**

To compare string objects, Java provides methods and operators both. So we can compare string in following three ways.

1. Using `equals()` method
2. Using `==` operator
3. By `CompareTo()` method

1.Using `equals()` method

`equals()` method compares two strings for equality. Its general syntax is,

```
boolean equals (Object str)
```

Copy

Example

It compares the content of the strings. It will return **true** if string matches, else returns **false**.

```
public class Demo{
```



```
public static void main(String[] args) {  
  
    String s = "Hell";  
  
    String s1 = "Hello";  
  
    String s2 = "Hello";  
  
    boolean b = s1.equals(s2);    //true  
  
    System.out.println(b);  
  
    b =    s.equals(s1) ;    //false  
  
    System.out.println(b);  
  
}  
  
}
```

Copy

true

false

2.Using == operator

The double **equal (==)** operator compares two object references to check whether they refer to same instance. This also, will return **true** on successful match else returns false.

```
public class Demo{  
  
    public static void main(String[] args) {  
  
        String s1 = "Java";  
  
        String s2 = "Java";  
  
        String s3 = new String ("Java");  
  
        boolean b = (s1 == s2);    //true
```

```

    System.out.println(b);

    b =    (s1 == s3);    //false

    System.out.println(b);

}

}

```

Copy

```

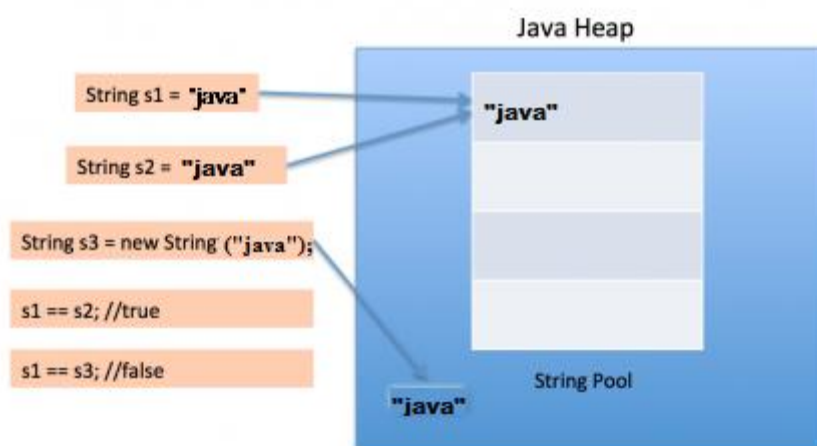
true
false

```

Explanation

We are creating a new object using new operator, and thus it gets created in a non-pool memory area of the heap. s1 is pointing to the String in string pool while s3 is pointing to the String in heap and hence, when we compare s1 and s3, the answer is false.

The following image will explain it more clearly.



3.By **compareTo()** method

String **compareTo()** method compares values and returns an integer value which tells if the string compared is less than, equal to or greater than the other string. It compares the String based on natural ordering i.e alphabetically. Its general syntax is.

Syntax:

```
int compareTo(String str)
```

Example:

```
public class HelloWorld{  
  
    public static void main(String[] args) {  
  
        String s1 = "Abhi";  
  
        String s2 = "Viraaj";  
  
        String s3 = "Abhi";  
  
        int a = s1.compareTo(s2);    //return -21 because s1 < s2  
  
        System.out.println(a);  
  
        a = s1.compareTo(s3);    //return 0 because s1 == s3  
  
        System.out.println(a);  
  
        a = s2.compareTo(s1);    //return 21 because s2 > s1  
  
        System.out.println(a);  
  
    }  
}
```

-21

0

21

• String class functions

The methods specified below are some of the most commonly used methods of the **String** class in Java. We will learn about each method with help of small code examples for better understanding.

1.charAt() method

String **charAt()** function returns the character located at the specified index.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "studytonight";  
  
        System.out.println(str.charAt(2));  
  
    }  
  
}
```

Output: u

NOTE: Index of a String starts from 0, hence **str.charAt(2)** means third character of the String str.

2.equalsIgnoreCase() method

String **equalsIgnoreCase()** determines the equality of two Strings, ignoring their case (upper or lower case doesn't matter with this method).

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "java";  
  
        System.out.println(str.equalsIgnoreCase("JAVA"));  
  
    }  
  
}
```

```
}
```

```
}
```

```
true
```

3.indexOf() method

String `indexOf()` method returns the index of first occurrence of a substring or a character. `indexOf()` method has four override methods:

- `int indexOf(String str)`: It returns the index within this string of the first occurrence of the specified substring.
- `int indexOf(int ch, int fromIndex)`: It returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
- `int indexOf(int ch)`: It returns the index within this string of the first occurrence of the specified character.
- `int indexOf(String str, int fromIndex)`: It returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

Example:

```
public class StudyTonight {  
  
    public static void main(String[] args) {  
  
        String str="StudyTonight";  
  
        System.out.println(str.indexOf('u')); //3rd form  
  
        System.out.println(str.indexOf('t', 3)); //2nd form  
  
        String subString="Ton";  
  
        System.out.println(str.indexOf(subString)); //1st form  
  
        System.out.println(str.indexOf(subString,7)); //4th form  
  
    }  
}
```

```
}
```

```
2
```

```
11
```

```
5
```

```
-1
```

NOTE: -1 indicates that the substring/Character is not found in the given String.

4.length() method

String **length()** function returns the number of characters in a String.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "Count me";  
  
        System.out.println(str.length());  
  
    }  
  
}
```

```
8
```

5.replace() method

String **replace()** method replaces occurrences of character with a specified new character.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "Change me";  
  
        System.out.println(str.replace('m','M'));
```

```
}  
  
}  
  
;
```

Change Me

6.substring() method

String **substring()** method returns a part of the string. **substring()** method has two override methods.

1. public String substring(int begin);
2. public String substring(int begin, int end);

The first argument represents the starting point of the substring. If the **substring()** method is called with only one argument, the substring returns characters from specified starting point to the end of original string.

If method is called with two arguments, the second argument specifies the end point of substring.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "0123456789";  
  
        System.out.println(str.substring(4));  
  
        System.out.println(str.substring(4,7));  
  
    }  
  
}
```

7.toLowerCase() method

String `toLowerCase()` method returns string with all uppercase characters converted to lowercase.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "ABCDEF";  
  
        System.out.println(str.toLowerCase());  
  
    }  
  
}
```

abcdef

8.toUpperCase() method

This method returns string with all lowercase character changed to uppercase.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String str = "abcdef";  
  
        System.out.println(str.toUpperCase());  
  
    }  
  
}
```

ABCDEF

9.valueOf() method

String class uses overloaded version of **valueOf()** method for all primitive data types and for type Object.

NOTE: **valueOf()** function is used to convert **primitive data types** into Strings.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int num = 35;  
  
        String s1 = String.valueOf(num);    //converting int to String  
  
        System.out.println(s1);  
  
        System.out.println("type of num is: "+s1.getClass().getName());  
  
    }  
  
}
```

35

type of num is: java.lang.String

10.toString() method

String **toString()** method returns the string representation of an object. It is declared in the Object class, hence can be overridden by any java class. (Object class is super class of all java classes).

```
public class Car {  
  
    public static void main(String args[])  
  
    {  
  
        Car c = new Car();  
  
        System.out.println(c);  
  
    }  
  
}
```

```
}

public String toString()

{

    return "This is my car object";

}

}
```

This is my car object

Whenever we will try to print any object of class Car, its `toString()` function will be called.

NOTE: If we don't override the `toString()` method and directly print the object, then it would print the object id that contains some hashcode.

11.trim() method

This method returns a string from which any leading and trailing whitespaces has been removed.

```
public class Demo {

    public static void main(String[] args) {

        String str = "  hello  ";

        System.out.println(str.trim());

    }

}
```

hello

12.contains()Method

String **contains()** method is used to check the sequence of characters in the given string. It returns true if a sequence of string is found else it returns false.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String a = "Hello welcome to studytonight.com";  
  
        boolean b = a.contains("studytonight.com");  
  
        System.out.println(b);  
  
        System.out.println(a.contains("javatpoint"));  
  
    }  
}
```

true

false

13.endsWith() Method

String **endsWith()** method is used to check whether the string ends with the given suffix or not. It returns true when suffix matches the string else it returns false.

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String a="Hello welcome to studytonight.com";
```

```
System.out.println(a.endsWith("m"));

System.out.println(a.endsWith("com"));

}

}
```

true

true

14.format() Method

String `format()` is a string method. It is used to the format of the given string.

Following are the format specifier and their datatype:

Format Specifier	Data Type
%a	floating point
%b	Any type
%c	character
%d	integer
%e	floating point

%f	floating point
%g	floating point
%h	any type
%n	none
%o	integer
%s	any type
%t	Date/Time
%x	integer

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        String a1 = String.format("%d", 125);  
  
        String a2 = String.format("%s", "studytonight");  
  
        String a3 = String.format("%f", 125.00);  
  
        String a4 = String.format("%x", 125);  
  
        String a5 = String.format("%c", 'a');
```

```
System.out.println("Integer Value: "+a1);

System.out.println("String Value: "+a2);

System.out.println("Float Value: "+a3);

System.out.println("Hexadecimal Value: "+a4);

System.out.println("Char Value: "+a5);

}

}
```

Integer Value: 125

String Value: studytonight

Float Value: 125.000000

Hexadecimal Value: 7d

Char Value: a

15.getBytes() Method

String **getBytes()** method is used to get byte array of the specified string.

```
public class Demo {

    public static void main(String[] args) {

        String a="studytonight";

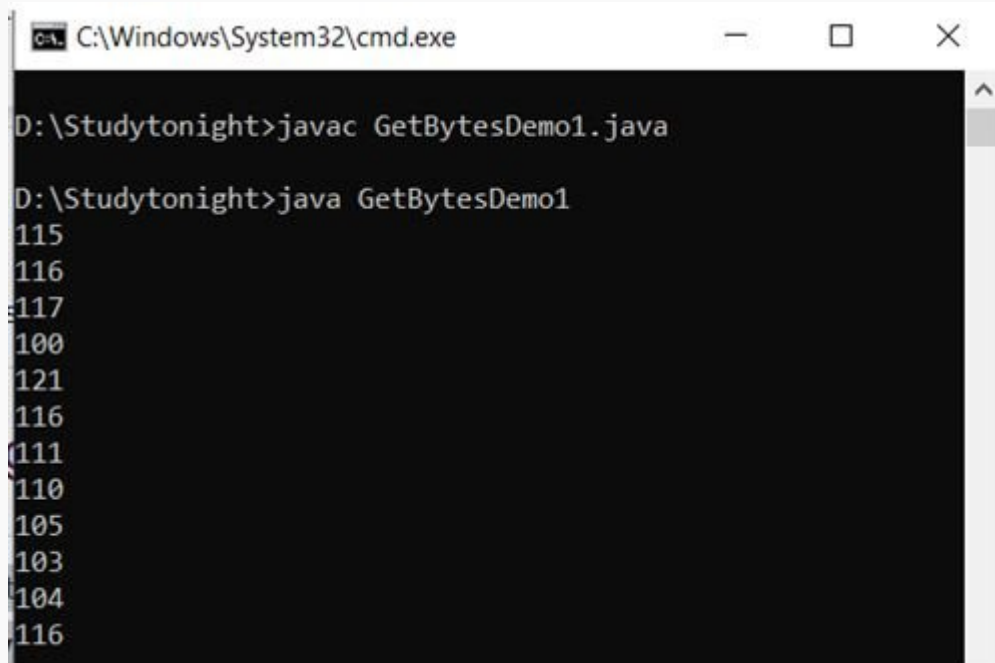
        byte[] b=a.getBytes();

        for(int i=0;i<b.length;i++)
```

```

{
    System.out.println(b[i]);
}
}
}
}

```



A screenshot of a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The window shows the following commands and output:

```

D:\Studytonight>javac GetBytesDemo1.java
D:\Studytonight>java GetBytesDemo1
115
116
117
100
121
116
111
110
105
103
104
116

```

16.getChars() Method

String `getChars()` method is used to copy the content of the string into a char array.

```

public class Demo {

    public static void main(String[] args) {

        String a= new String("Hello Welcome to studytonight.com");

        char[] ch = new char[16];
    }
}

```

```
try

{

    a.getChars(6, 12, ch, 0);

    System.out.println(ch);

}

catch(Exception ex)

{

    System.out.println(ex);

}

}

}
```

Welcom

17.isEmpty() Method

String **isEmpty()** method is used to check whether the string is empty or not. It returns true when length string is zero else it returns false.

```
public class IsEmptyDemo1

{

    public static void main(String args[])

    {
```



```
String a="";

String b="studytonight";

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

System.out.println(b.isEmpty());

}

}
```

true

false

18.join() Method

String **join()** method is used to join strings with the given delimiter. The given delimiter is copied with each element

```
public class JoinDemo1

{

    public static void main(String[] args)

    {

        String s = String.join("*","Welcome to studytonight.com");

        System.out.println(s);

        String date1 = String.join("/", "23", "01", "2020");

        System.out.println("Date: "+date1);

    }

}
```

```
String time1 = String.join(":", "2", "39", "10");

System.out.println("Time: "+time1);

}

}
```

Welcome to studytonight.com

Date: 23/01/2020

Time: 2:39:10

19.startsWith() Method

String **startsWith()** is a string method in java. It is used to check whether the given string starts with given prefix or not. It returns true when prefix matches the string else it returns false.

```
public class Demo {

    public static void main(String[] args) {

        String str = "studytonight";

        System.out.println(str.startsWith("s"));

        System.out.println(str.startsWith("t"));

        System.out.println(str.startsWith("study",1));

    }

}
```

true
false
false

String Methods List

Method	Description
char charAt(int index)	It returns char value for the particular index
int length()	It returns string length
static String format(String format, Object... args)	It returns a formatted string.
static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
String substring(int beginIndex)	It returns substring for given begin index.
String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.

static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
boolean equals(Object another)	It checks the equality of string with the given object.
boolean isEmpty()	It checks if string is empty.
String concat(String str)	It concatenates the specified string.
String replace(char old, char new)	It replaces all occurrences of the specified char value.
String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
String[] split(String regex)	It returns a split string matching regex.
String[] split(String regex, int limit)	It returns a split string matching regex and limit.

String intern()	It returns an interned string.
int indexOf(int ch)	It returns the specified char value index.
int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
int indexOf(String substring)	It returns the specified substring index.
int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
String toLowerCase()	It returns a string in lowercase.
String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
String toUpperCase()	It returns a string in uppercase.
String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
String trim()	It removes beginning and ending spaces of this string.

static String valueOf(int value)	It converts given type into string. It is an overloaded method.
----------------------------------	-----------------------------------------------------------------

• 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.

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.

StringBufferExample.java

```

1. class StringBufferExample{
2.     public static void main(String args[]){
3.         StringBuffer sb=new StringBuffer("Hello ");
4.         sb.append("Java");//now original string is changed
5.         System.out.println(sb);//prints Hello Java
6.     }
7. }
```

Output:

• StringBuilder Class

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Important Constructors of StringBuilder class

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

StringBuilder Examples

.

StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this String.

StringBuilderExample.java

```
1. class StringBuilderExample{
2. public static void main(String args[]){
3.     StringBuilder sb=new StringBuilder("Hello ");
4.     sb.append("Java");//now original string is changed
5.     System.out.println(sb);//prints Hello Java
6. }
7. }
```

Output:

```
Hello Java
```

Difference between String and StringBuffer

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

Difference between StringBuffer and StringBuilder

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