UNIT-1

Introduction to Java Basics

History of Java:

- Java is an efficient powerful Object-Oriented Programming language developed in the year of **1991** by **James Gosling** and his team members at Sun micro systems.
- **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Initially Java is called with a name **Oak** is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
- In 1995, Oak was renamed as **"Java"** .
- Java is an island of Indonesia where first coffee was produced (called java coffee).
- Sun micro systems purchased by Oracle Corporation in the year of 2010.
- JDK (Java Development tool Kit) 1.0 released in (January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

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 March, 2014)

Where it is used? 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 irctc.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 type of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used injava for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, 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. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

------------------------------***********------------------------------

Java Characteristics (or) Features of java (or) Java Buzz words:

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Simple

According to Sun, Java language is simple because:

syntax is based on C++ (so easier for programmers to learn it after C++).

removed many confusing and/or rarely-used features e.g., explicit pointers, operatoroverloading etc.

No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

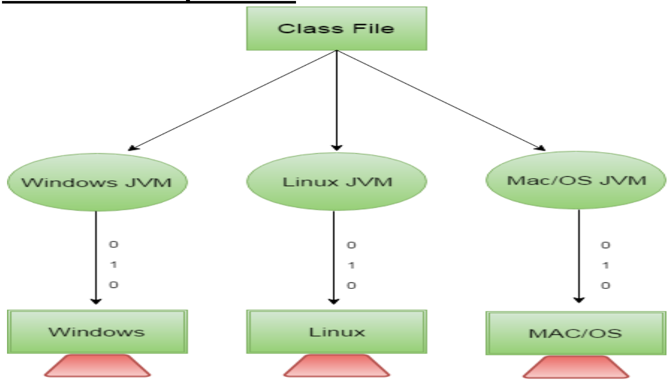
Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules.

Basic concepts of OOPs are:

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

Platform Independent



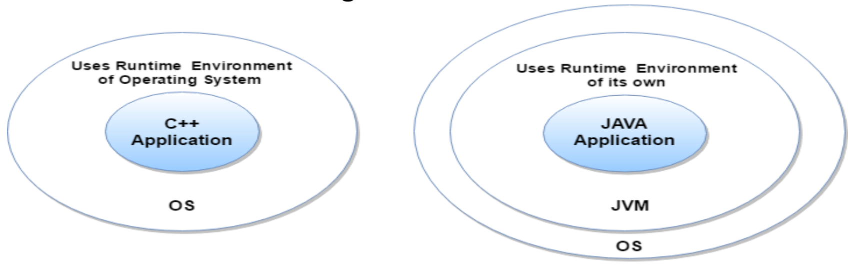
A platform is the hardware or software environment in which a program runs.
There are two types of platforms software-based and hardware-based. Java provides software-based platform.
The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

- Runtime Environment
- API(Application Programming Interface)

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

Secured
Java is secured because:

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


Uses Runtime Environment of Operating System
C++ Application
OS
Uses Runtime Environment of its own
JAVA Application
JVM
OS

Robust
Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral
There are no implementation dependent features e.g. 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. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Portable: We may carry the java bytecode to any platform.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

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.

******..............................******

Java program Compilation and Execution Process:

Sample java program:

```
import java.lang.*; // package import section
class Sample // creating a class
      {
        public static void main(String args[]) // main() execution starts from here
        {
                System.out.println("Hello world"); // writing "Hello world" on console
        }
}
```

- Save the above program with an extension .java i.e Sample.java in a specific location.

Compilation:

- for compilation of java program we use "javac" keyword.

```
javac  Sample.java
```

Execution:

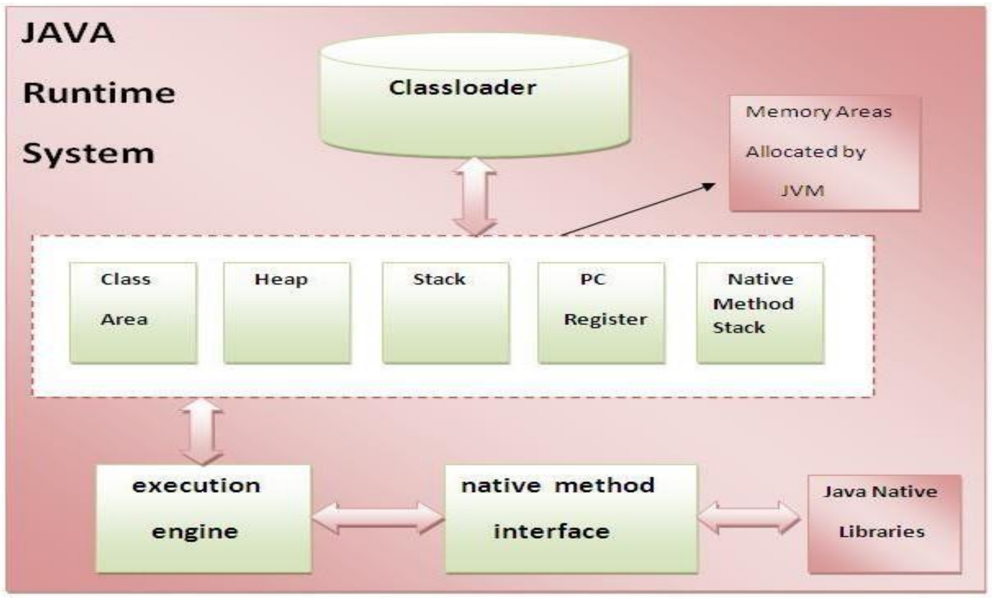
- After successful compilation of java program .class file will be created. This is called bytecode.
- for executing java program we use .class file name with "java" keyword as follows.

$$\begin{tikzcd}\text{Sample.java} \arrow[r] & \text{Compiler} \arrow[r, "\text{Sample.class}"'] & \text{JVM (java virtual Machine)} \arrow[r] & \text{Output}\end{tikzcd}$$

*******__________*******

Java Virtual Machine(JVM):

- JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.
- JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).
- The JVM performs following operations: Loads code,Verifies code, Executes code and Provides runtime environment
- **Internal Archietecture of JVM:**



Classloader: Classloader is a subsystem of JVM that is used to load class files.
Class(Method) Area: Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
Heap: It is the runtime data area in which objects are allocated.

Stack: Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

Program Counter Register: PC (program counter) register. It contains the address of theJava virtual machine instruction currently being executed.
Native Method Stack: It contains all the native methods used in the application.
Execution Engine: It contains

- A virtual processor
- Interpreter: Read byte code stream then execute the instructions.

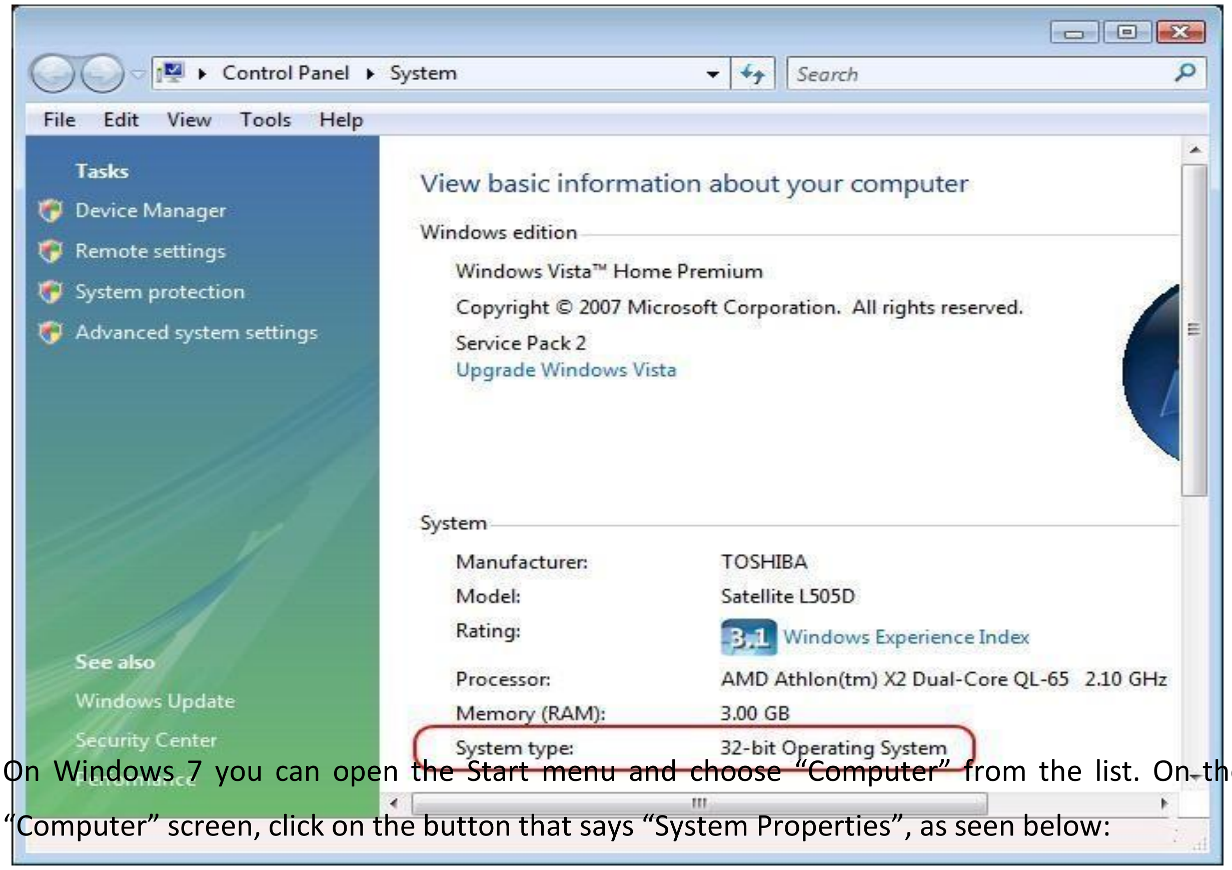
- Just-In-Time(JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and Hence reduces the amount of time needed for compilation. Here the term compiler refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

******_______________******

Installation of JDK:

The JDK software has two different versions: a 32-bit Windows version ar d a 64-bit Windows version. Before you download the software, you will need to know whic version of Windows you are using. If you don't already know what version of Windows you ha e, you can find out fairly quickly!

On older systems find your "Computer" icon from the Start menu or on your desktop. Right-click on this icon and select "Properties". You can then read the "System Type" to determine if you have a 32 or 64-bit operating system



View basic information about your computer

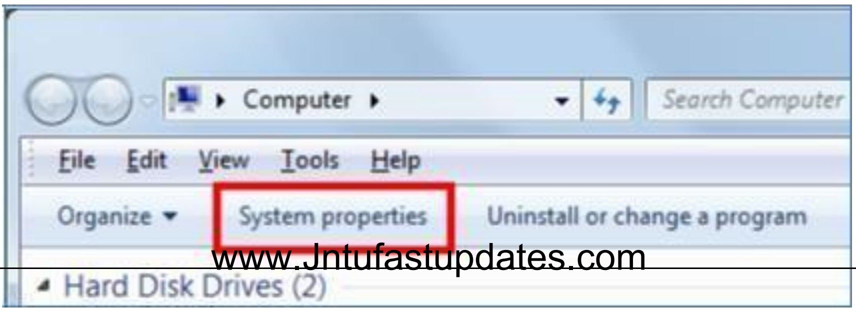
Windows edition

Windows Vista™ Home Premium
Copyright © 2007 Microsoft Corporation. All rights reserved.
Service Pack 2
Upgrade Windows Vista

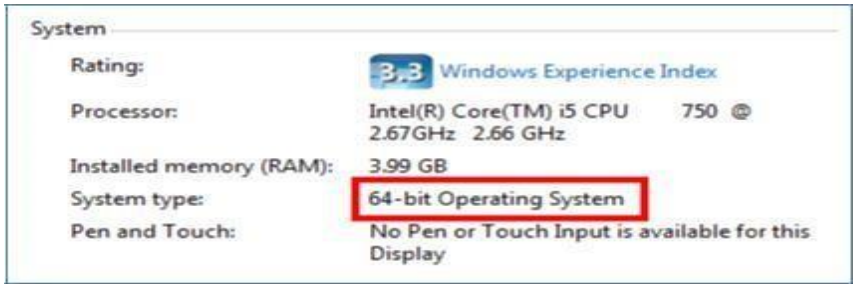
System

Manufacturer:	TOSHIBA
Model:	Satellite L505D
Rating:	3.1 Windows Experience Index
Processor:	AMD Athlon(tm) X2 Dual-Core QL-65 2.10 GHz
Memory (RAM):	3.00 GB
System type:	32-bit Operating System

On Windows 7 you can open the Start menu and choose "Computer" from the list. On the "Computer" screen, click on the button that says "System Properties", as seen below:



This will bring up a screen with information about your computer and your version of Windows. Look for a section titled “System”. This section will easily tell you what version of Windows you have installed next to the “System Type” label:



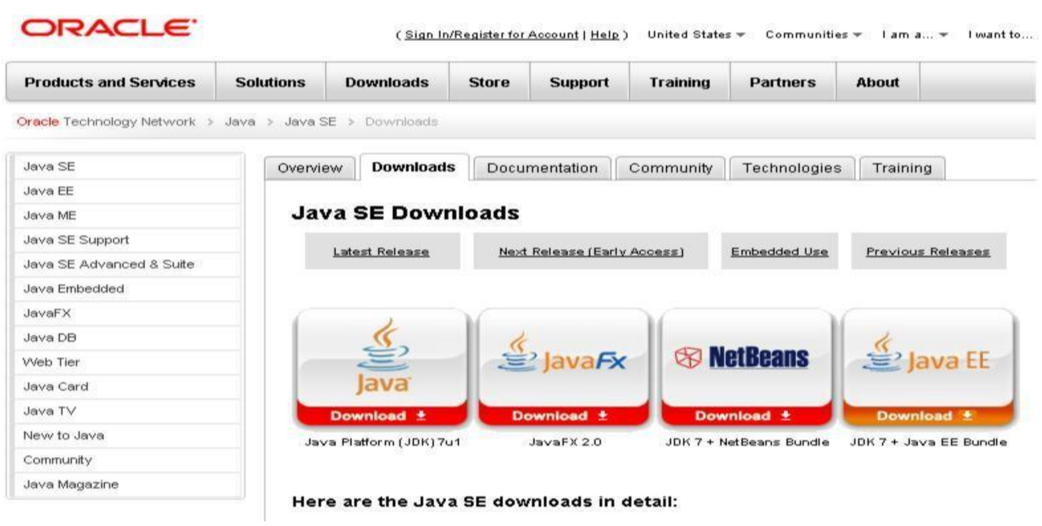
System

Rating:	3.3 Windows Experience Index
Processor:	Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz 2.66 GHz
Installed memory (RAM):	3.99 GB
System type:	64-bit Operating System
Pen and Touch:	No Pen or Touch Input is available for this Display

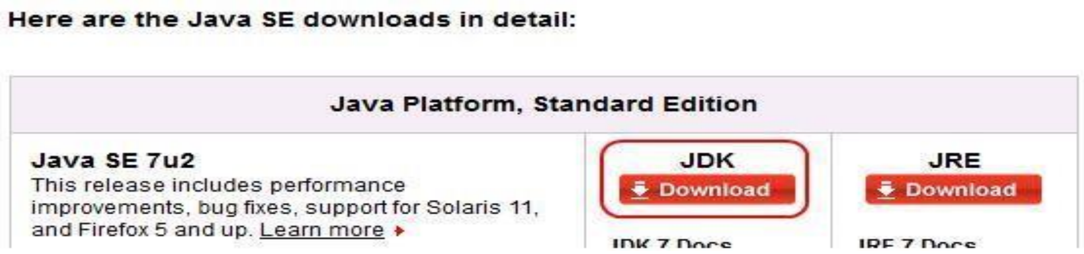
Keep this information handy! We will use it to determine which version of the JDK to download.

Downloading the JDK

To download the *Java Development Kit (JDK)*, launch your web browser (e.g. Internet Explorer) and goto address: http://www.oracle.com/technetwork/java/javase/downloads/index.html. This page shows many download options. The top of the page shows the most common JDK download options:

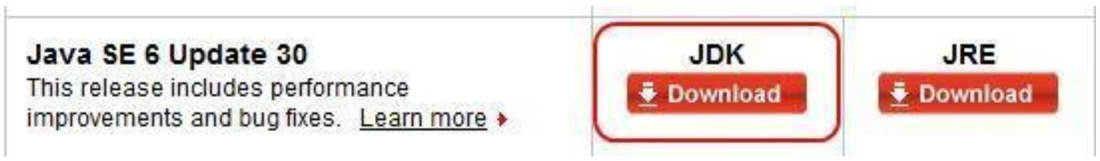


For this course, you can select either JDK version 6 (which we have chosen in the textbook) or JDK version 7 (which is newly released). Select the “JDK” download button for the corresponding version further down on the screen:


Here are the Java SE downloads in detail:
Java Platform, Standard Edition
Java SE 7u2
This release includes performance improvements, bug fixes, support for Solaris 11, and Firefox 5 and up. Learn more
JDK
Download
JRE
Download
JDK 7 Docs
JRE 7 Docs

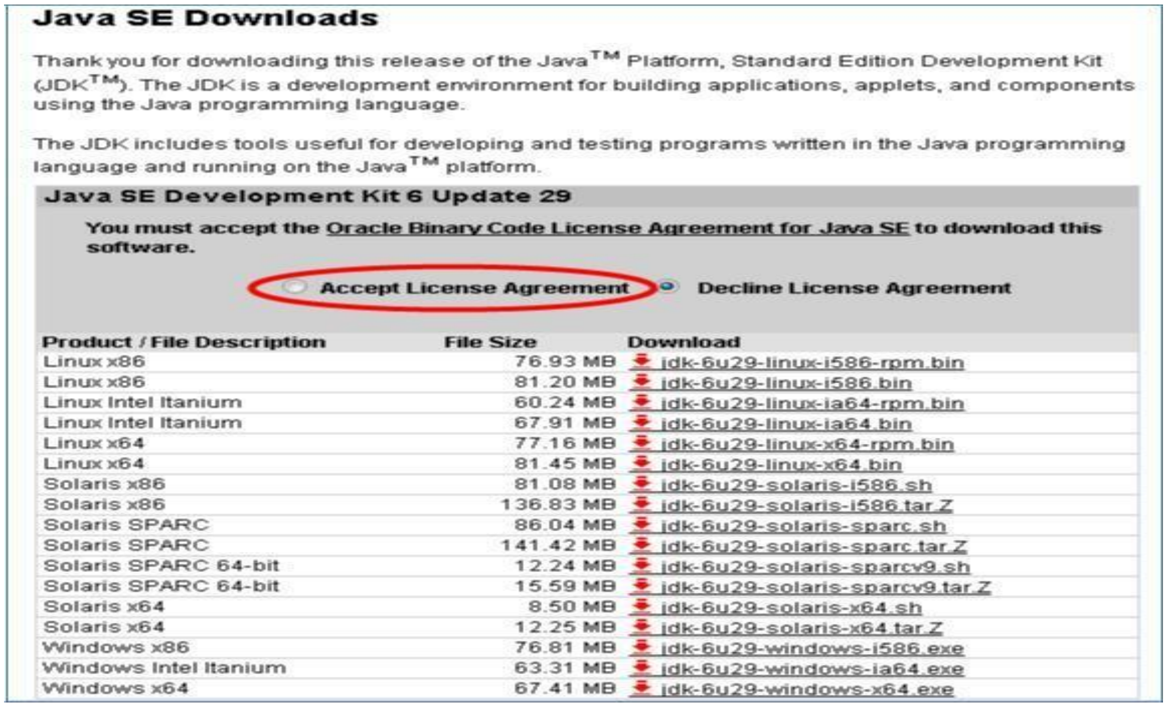
8

Or if you prefer JDK 6.0 (1.6):



Java SE 6 Update 30
This release includes performance improvements and bug fixes. Learn more ›

The Download button will open another page which will list the various JDK installation files.You must choose “Accept License Agreement” in order to continue with the download and installation.



Java SE Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java™ platform.

Java SE Development Kit 6 Update 29

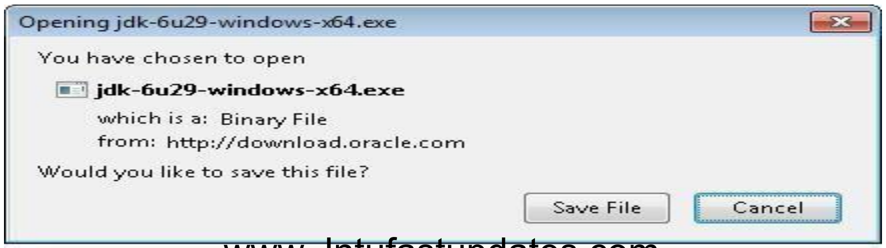
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement [✓ Decline License Agreement]

Product / File Description	File Size	Download
Linux x86	76.93 MB	jdk-6u29-linux-i586-rpm.bin
Linux x86	81.20 MB	jdk-6u29-linux-i586.bin
Linux Intel Itanium	60.24 MB	jdk-6u29-linux-ia64-rpm.bin
Linux Intel Itanium	67.91 MB	jdk-6u29-linux-ia64.bin
Linux x64	77.16 MB	jdk-6u29-linux-x64-rpm.bin
Linux x64	81.45 MB	jdk-6u29-linux-x64.bin
Solaris x86	81.08 MB	jdk-6u29-solaris-i586.sh
Solaris x86	136.83 MB	jdk-6u29-solaris-i586.tar.Z
Solaris SPARC	86.04 MB	jdk-6u29-solaris-sparc.sh
Solaris SPARC	141.42 MB	jdk-6u29-solaris-sparc.tar.Z
Solaris SPARC 64-bit	12.24 MB	jdk-6u29-solaris-sparcv9.sh
Solaris SPARC 64-bit	15.59 MB	jdk-6u29-solaris-sparcv9.tar.Z
Solaris x64	8.50 MB	jdk-6u29-solaris-x64.sh
Solaris x64	12.25 MB	jdk-6u29-solaris-x64.tar.Z
Windows x86	76.81 MB	jdk-6u29-windows-i586.exe
Windows Intel Itanium	63.31 MB	jdk-6u29-windows-ia64.exe
Windows x64	67.41 MB	jdk-6u29-windows-x64.exe

Once you have accepted the license agreement, you will need to choose a file to download. The Windows files are at the bottom of the screen. If you have a 32-bit Windows system, you will choose the “Windows x86” file. If you have a 64-bit Windows system, you will need to choose the “Windows x64” file.

As soon as you click to download the file, a pop-up window will appear asking you to either “Save” or “Run” the program. The look of this window will depend on what version of Windows and which Internet browser you are using to download the file. The following screenshots are from Mozilla Firefox and Internet Explorer:



Opening jdk-6u29-windows-x64.exe

You have chosen to open

jdk-6u29-windows-x64.exe

which is a: Binary File
from: http://download.oracle.com

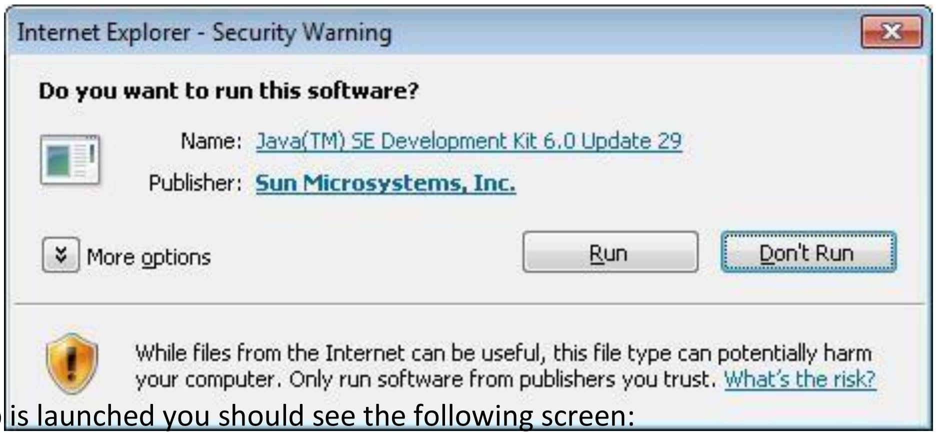
Would you like to save this file?

File Download - Security Warning

Select “Save File” or “Save” to save the file to a location on your local hard drive. You can save it to your Desktop or some other file folder. Remember this location so you can find it later!

Oracle updates the exact version of the JDK frequently. Our examples show JDK version 1.6.29 (6.0.29), but keep in mind that the version available to you at the time of download will likely be different! Or you may choose JDK 1.7.X (7.0.X) if you prefer.

Once the file is saved, use your Windows Explorer to find and run the program by double- clicking in it. Depending on your version of Windows and security settings you may get a security popup as shown below. Click on “Run” to continue.



Internet Explorer - Security Warning

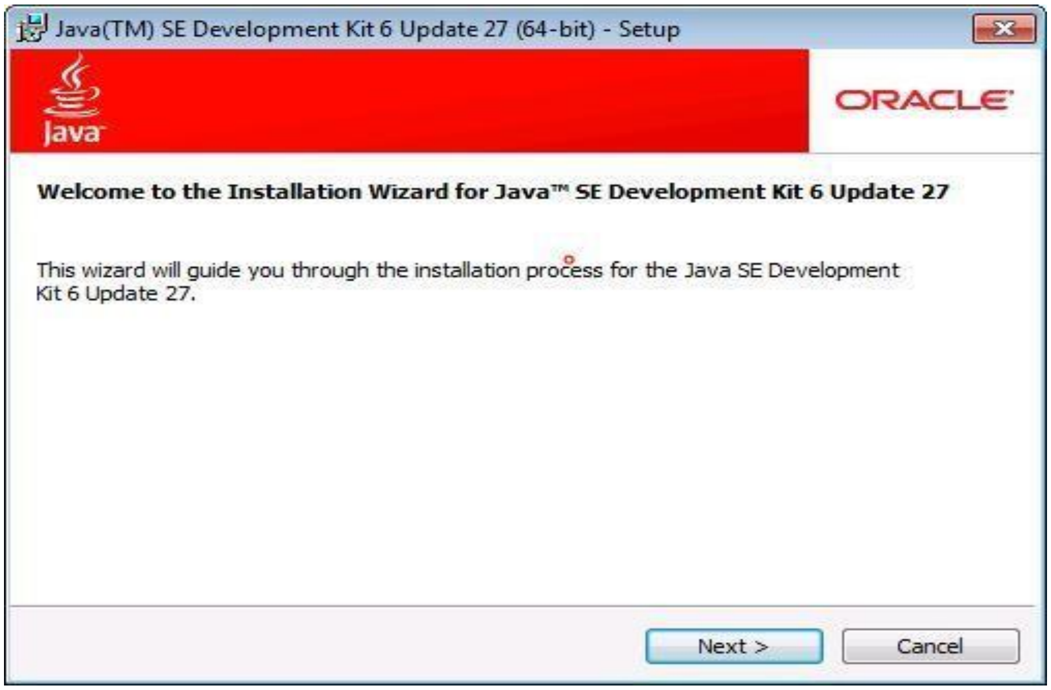
Do you want to run this software?

Name: Java(TM) SE Development Kit 6.0 Update 29

Publisher: Sun Microsystems, Inc.

While files from the Internet can be useful, this file type can potentially harm your computer. Only run software from publishers you trust. What's the risk?

When setup is launched you should see the following screen:



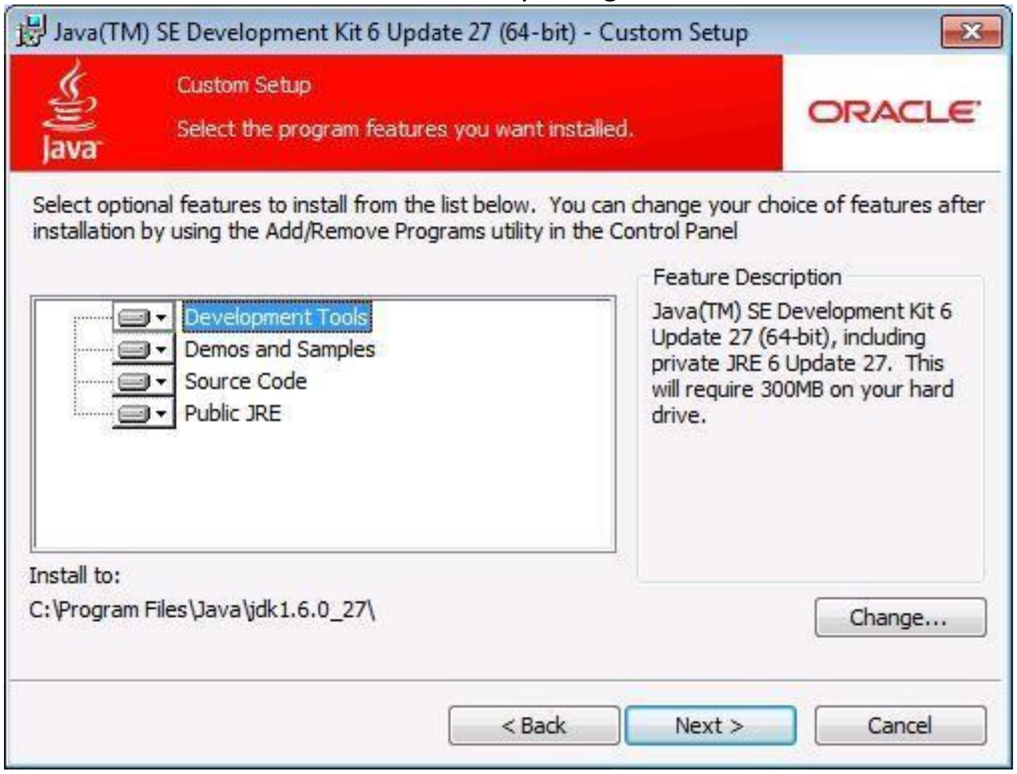
Java(TM) SE Development Kit 6 Update 27 (64-bit) - Setup

Welcome to the Installation Wizard for Java™ SE Development Kit 6 Update 27

This wizard will guide you through the installation process for the Java SE Development Kit 6 Update 27.

This is the first screen in the install process. Click “Next” to continue.

This next screen lists all of the possible JDK options that can be installed. Since we will be covering the basics of Java in this course, you can just accept the defaults and simply click onthe 'Next' button to continue. There is no need to make any changes on this screen.


Java(TM) SE Development Kit 6 Update 27 (64-bit) - Custom Setup
Custom Setup
Select the program features you want installed.
ORACLE

Select optional features to install from the list below. You can change your choice of features after installation by using the Add/Remove Programs utility in the Control Panel

- Development Tools
- Demos and Samples
- Source Code
- Public JRE

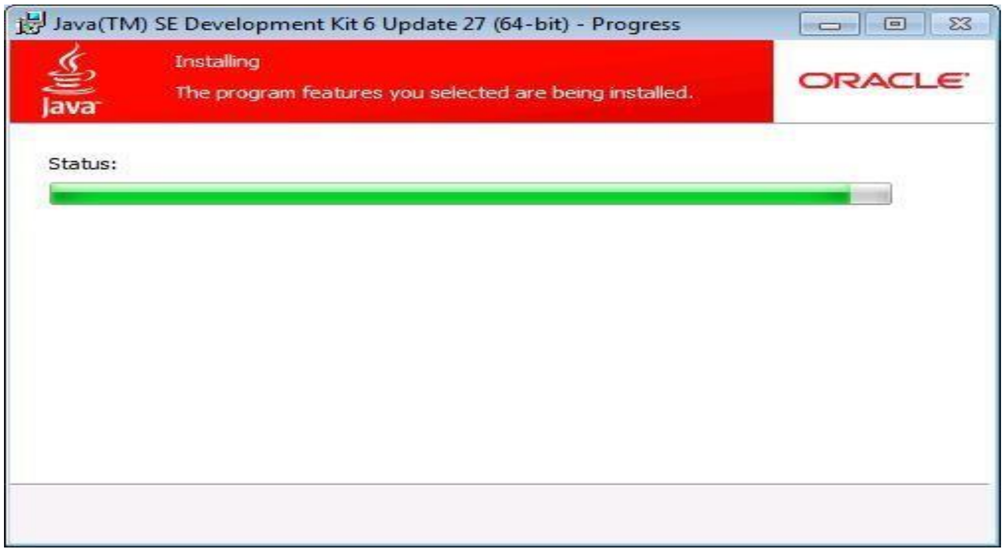
Feature Description

Java(TM) SE Development Kit 6 Update 27 (64-bit), including private JRE 6 Update 27. This will require 300MB on your hard drive.

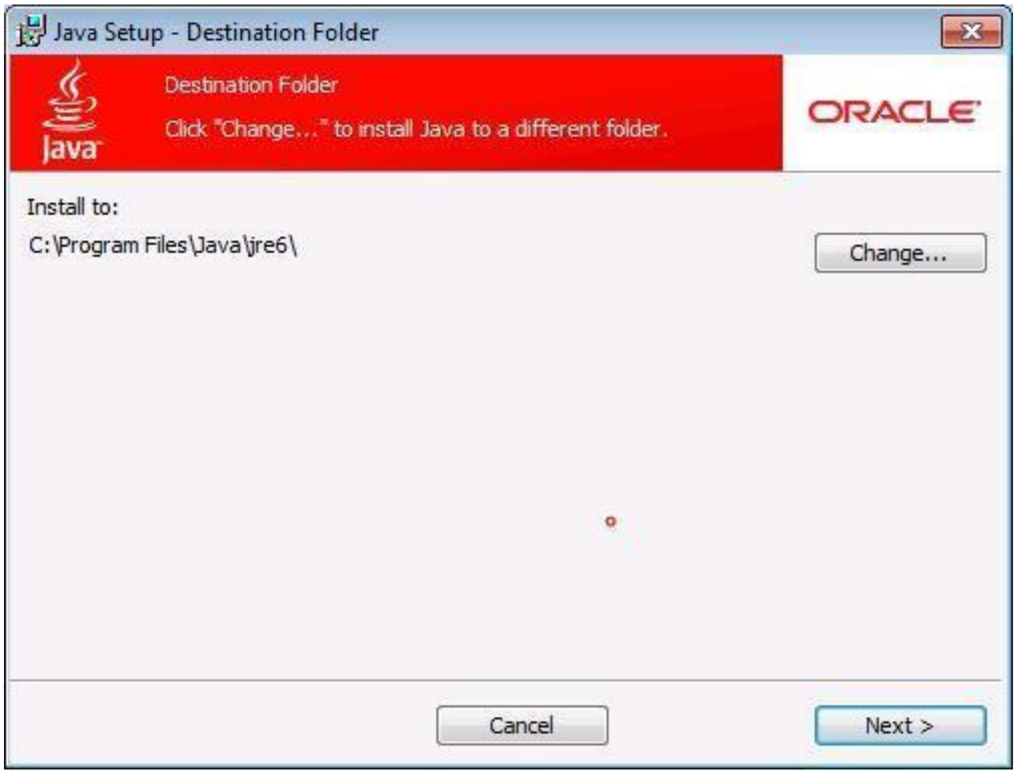
Install to:

C:\Program Files\Java\jdk1.6.0_27\

The next screen will display a simple progress bar while the JDK files are being installed. This process could take anywhere from seconds to minutes, depending on the speed of your computer.


Java(TM) SE Development Kit 6 Update 27 (64-bit) - Progress
Installing
The program features you selected are being installed.
ORACLE
Status:

When the JDK is finished installing, the installation program will install the JRE files. The screen above will allow you to choose the directory where the JRE will be located. We recommend that you allow the files to install in the default directory, as shown below.



Destination Folder

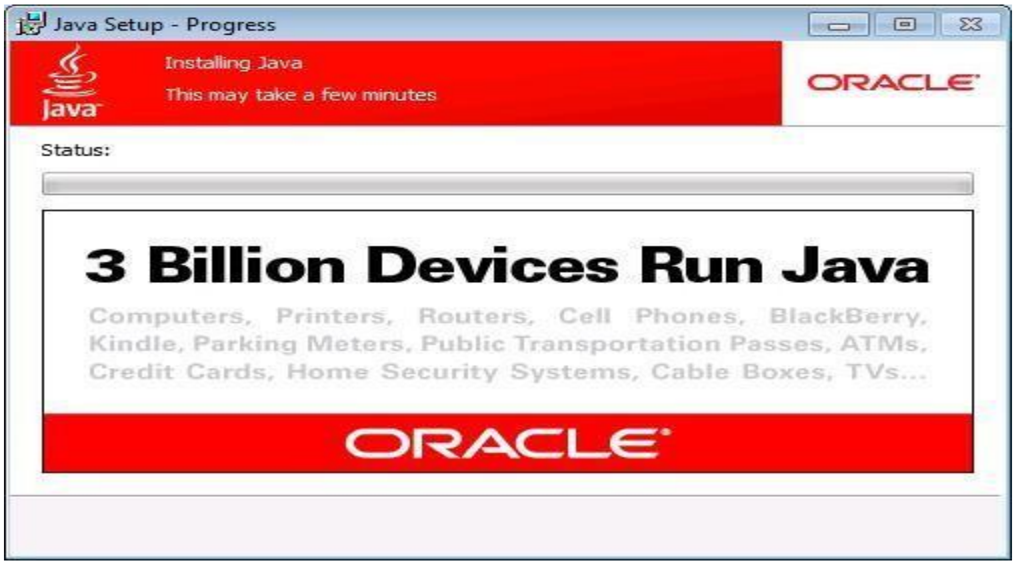
Click "Change..." to install Java to a different folder.

Install to:

C:\Program Files\Java\jre6\

Once you choose the “Next” button, the installation will display another progress bar. This will show the progress of the installation of the JRE files.

The next screen will simply show the progress of your JRE installation. In this first step, the installation program will automatically download additional files from the Oracle website. This is a large program and will take some time!



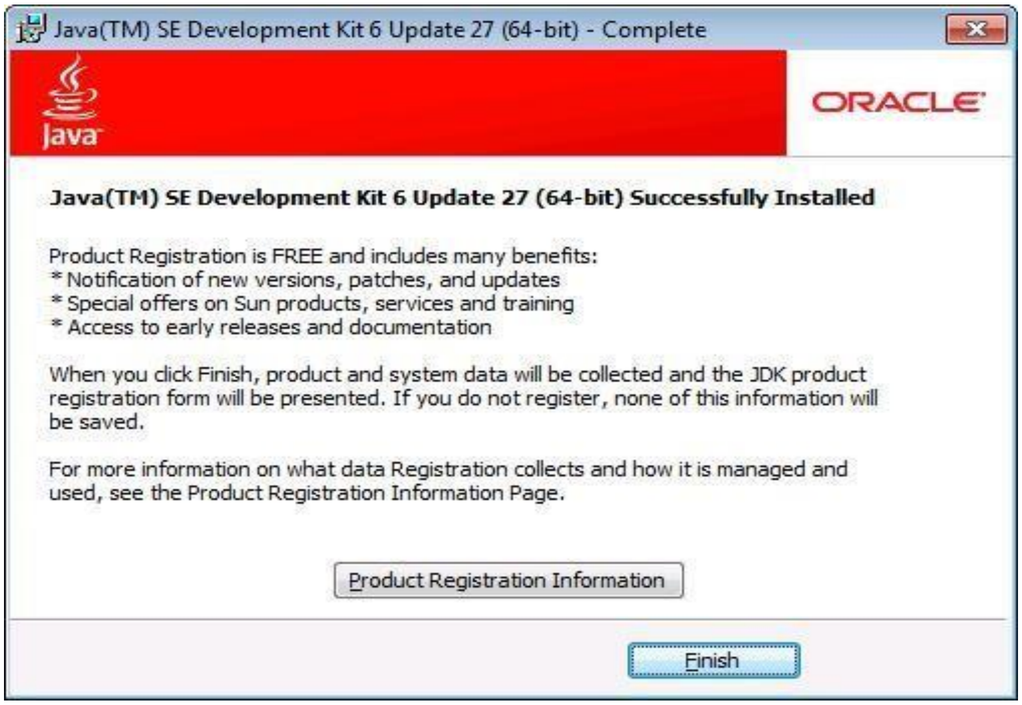
Installing Java

This may take a few minutes

3 Billion Devices Run Java

Computers, Printers, Routers, Cell Phones, BlackBerry, Kindle, Parking Meters, Public Transportation Passes, ATMs, Credit Cards, Home Security Systems, Cable Boxes, TVs...

At this point, the installation of the JDK files is complete. When you click on the “Finish” button on this screen, a browser window will appear, displaying registration information for Java.


Java(TM) SE Development Kit 6 Update 27 (64-bit) - Complete

Java(TM) SE Development Kit 6 Update 27 (64-bit) Successfully Installed

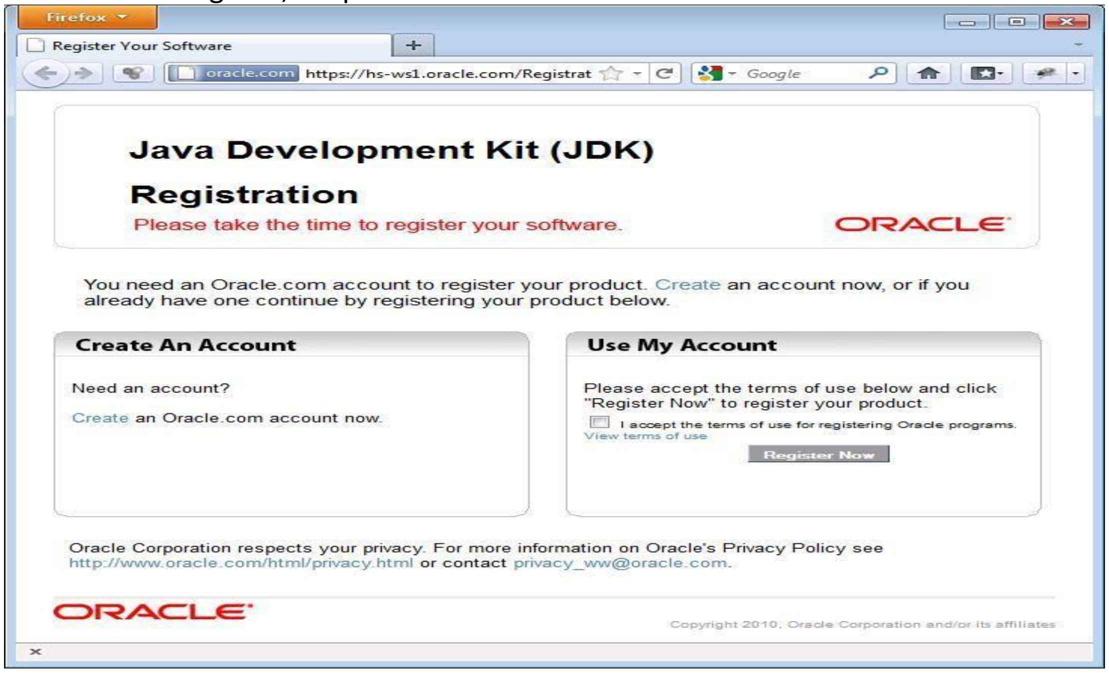
Product Registration is FREE and includes many benefits:
* Notification of new versions, patches, and updates
* Special offers on Sun products, services and training
* Access to early releases and documentation

When you click Finish, product and system data will be collected and the JDK product registration form will be presented. If you do not register, none of this information will be saved.

For more information on what data Registration collects and how it is managed and used, see the Product Registration Information Page.

Registration for the JDK software is optional and is not necessary for the completion of thiscourse.

If you choose not to register, simple close this window.


Register Your Software

https://hs-ws1.oracle.com/Registrat

Java Development Kit (JDK) Registration

Please take the time to register your software.

You need an Oracle.com account to register your product. Create an account now, or if you already have one continue by registering your product below.

Create An Account

Need an account?

Create an Oracle.com account now.

Use My Account

Please accept the terms of use below and click "Register Now" to register your product.

I accept the terms of use for registering Oracle programs.
View terms of use

Oracle Corporation respects your privacy. For more information on Oracle's Privacy Policy see http://www.oracle.com/html/privacy.html or contact privacy_ww@oracle.com.

Copyright 2010, Oracle Corporation and/or its affiliates

Congratulations! You have finished the installation of the JDK and JRE in your Windows computer.

******____________******

Java Basics :

Identifiers:

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of **valid identifiers** are:

  AvgTemp   count   a4 $test   this_is_ok
- **Invalid variable names** include:

  2count   high-temp   Not/ok

Variables:

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optionalinitializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.
- Declaring a Variable:
  In Java, all variables must be declared before they can be used. The basic form ofa variable declaration is shown here:

  type identifier [= *value*], *identifier* [= *value*], ... ;
- Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;              // declares three ints, a, b, and c.
int d = 3, e, f = 5;      // declares three more ints, initializing d and f.
byte z = 22;              // initializes z.
double pi = 3.14159;      // declares an approximation of pi.
char x = 'x';             // the variable x has the value 'x'.
```

- **Types of Variable:**
  There are three types of variables in java:
  - **local variable**: declared inside the method is called local variable.
  - **instance variable**: declared inside the class but outside the method, is called instance variable . It is not declared as static.
  - **static variable**: declared as static is called static variable. It cannot be local.

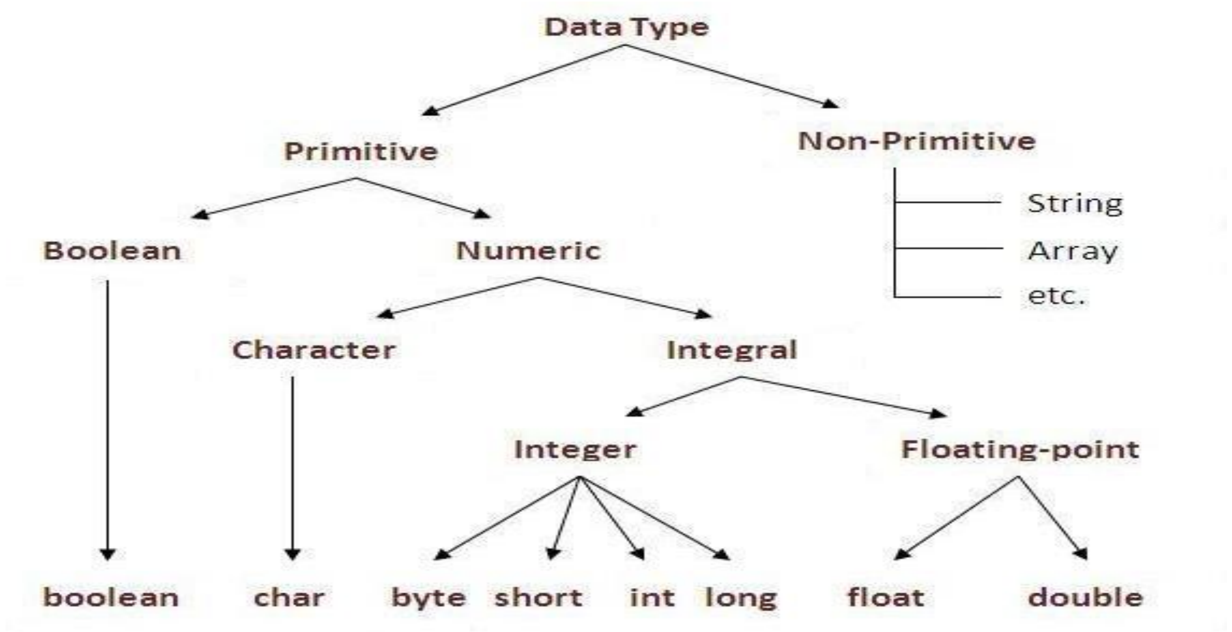
  Example:

```
class A
{
        int   data=50;//instance   variable
        static int m=100;//static variable
        void method()
        {
                int n=90;//local variable
        } }
```

Data types:

Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- Primitive data types
- Non-primitive data types



Data Type	Default Value	Default size	Range
boolean	false	1 bit	-
char	'\u0000'	2 byte	0 to 65,536
byte	0	1 byte	–128 to 127
short	0	2 byte	–32,768 to 32,767
int	0	4 byte	–2,147,483,648 to 2,147,483,647
long	0L	8 byte	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	0.0f	4 byte	1.4e–045 to 3.4e+038
double	0.0d	8 byte	4.9e–324 to 1.8e+308

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system than ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Example on variables:

```
class Simple
{
        public static void main(String[] args)
        {
                int   a=10;
                int   b=10;
                int c=a+b;
                float       f1=10.25f;
                double d=123.4567;
                char ch=’a’;
                boolean  b1=true;
                boolean b2=false;
                System.out.println(“int             c=”+c);
                System.out.println(“float        f1=”+f1);
                System.out.println(“double    d    =“+d);
                System.out.println(“char    ch    =”+ch);
                System.out.println(“boolean b1=”+ b1);
                System.out.println(“boolean b2=”+ b2);
        }
}
```

Literals:

- A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.
- Literals can be assigned to any primitive type variable. For example:

```
        byte  a = 68;
        char a = 'A'
```

- byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.
- Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
        int  decimal = 100;
        int octal = 0144;int
        hexa = 0x64;
```

- String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
        "Hello World"
        "two\nlines"
        "\"This is in quotes\""
```

- String and char types of literals can contain any Unicode characters. For example:

```
        char a = '\u0001';
        String  a = "\u0001";
```

- In java we have some escape sequences.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Keywords:

- There are 49 reserved keywords currently defined in the Java language .
- These keywords, combined with the syntax of the operators and separators, form the definition of the Java language.
- These keywords cannot be used as names for a variable,class, or method.

abstract	continue	goto	package	synchronized
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

- The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use.
- However, the current specification for Java only defines the keywords shown above.
- The **assert** keyword was added by Java 2, version 1.4
- In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

Operators:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Boolean Logical Operators

Arithmetic Operators:

- Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.
- The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.
- Example program:

```
class Test
 {
      public static void main(String args[])
      {
            int a = 10;
            int b = 20;
            int c = 25;
            int d = 25;
            System.out.println("a + b = " + (a + b) );
            System.out.println("a - b = " + (a - b) );
            System.out.println("a * b = " + (a * b) );
            System.out.println("b / a = " + (b / a) );
            System.out.println("b % a = " + (b % a) );
            System.out.println("c % a = " + (c % a) );
            System.out.println("a++ = " + (a++) );
```

```
                System.out.println("b-- = " + (a--) );
                // Check the difference in d++ and ++d
                System.out.println("d++ = " + (d++) );
                System.out.println("++d = " + (++d) );
        }
}
```

<u>This will produce the following result:</u>

```
a + b  = 30
a - b  = -10
a * b  = 200
b / a  = 2
b % a  = 0
c % a  = 5
a++ =  10
b-- =  11
d++ =  25
++d =  27
```

<u>Compound assignment:</u>

- Compound assignment opaerators are +=, -=, *=, /=, %=
- <u>Examples:</u>

int a=10;	(or)	int a=10;	int a=20;	(or)	int a=20;
a=a+10;		a+=10;	a=a/10;		a/=10;

Relational Operators:

- The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering.
- The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.
- Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test, **!=**.

    int a = 4;
    int b = 1;
    boolean c = a < b; // c is false

- <u>Example program:</u>

```
class Test
{
        public static void main(String args[])
        {
                int a = 10;
                int b = 20;
                System.out.println("a == b = " + (a == b) );
                System.out.println("a != b = " + (a != b) );
                System.out.println("a > b = " + (a > b) );
                System.out.println("a < b = " + (a < b) );
                System.out.println("b >= a = " + (b >= a) );
                System.out.println("b <= a = " + (b <= a) );
        }
}
```

Output:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

Bit wise operators:

- Java defines several *bitwise operators* which can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.
- They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment

BITWISE LOGICAL OPERATORS

- The bitwise logical operators are **&**, **|**, **^**, and **~**.
- The bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

- **The Bitwise NOT**
  Also called the *bitwise complement,* the unary NOT operator, **~**, inverts all of the bits ofits operand. For example,
  00101010 becomes 11010101 (after the NOT operator is applied.)
- **The Bitwise AND**
  The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is producedin all other cases. Here is an example:

```
00101010     42
& 00001111  15
--------------
00001010     10
```

- **<u>The Bitwise OR</u>**
  The OR operator, **|**, combines bits such that if either of the bits in the operands is a 1,then the resultant bit is a 1, as shown here:
  ```
  00101010    42
  |  00001111 15
  -------------
  00101111    47
  ```
- **<u>The Bitwise XOR</u>**
  The XOR operator, **^**, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
00101010       42
^  00001111       15
                  ---------------
                  00100101       37
```

<u>Left Shift</u>

- The left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times.
- It has this general form:
  value << num
- Here, *num* specifies the number of positions to left-shift the value in *value.* That is, the **<<** moves all of the bits in the specified value to the left by the number of bit positions specified by *num.*
- For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

```
byte a = 64, b;
int i;
i = a << 1;
b = (byte) (a << 2);
```

After shifting operation i =128 and b=0 (because byte takes only 8-bits)

a<<1 : a is left shifted by one position

	256	128	64	32	16	8	4	2	1
a=(64)			1	0	0	0	0	0	0
i=(a<<1)		1	0	0	0	0	0	0	**0**

a<<2 : a is left shifted by 2 positions

	256	128	64	32	16	8	4	2	1
a=(64)			1	0	0	0	0	0	0
b=(byte)(a<<2)	1	0	0	0	0	0	0	**0**	**0**

Right Shift

- The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times.
- Its general form is shown here:

  value >> num
- Here, *num* specifies the number of positions to right-shift the value in *value.*
- That is, the **>>** moves all of the bits in the specified value to the right the number of bit positions specified by *num.*
- The following code fragment shifts the value 32 to the right by two positions, resulting in **a** being set to 8:

```
int a = 32;
a = a >> 2; // a now contains 8
```

a>>2 : a is right shifted by two positions

		64	32	16	8	4	2	1
a=32			1	0	0	0	0	0
a=a>>2					1	0	0	0

Bitwise Operator Assignments:

- All of the binary bitwise operators have a shorthand form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.
- For example, the following two statements, which shift the value in **a** right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
```

- Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a** OR **b**, are equivalent:

```
a = a | b;
a |= b;
```

- Example program:

```
class OpBitEquals
{
      public static void main(String args[])
      {
            int a = 1;
            int b = 2;
            int c = 3;
            a |= 4;
            b >>= 1;
            c <<= 1;
            a ^= c;
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println("c = " + c);
      }
}
```

output :

```
a = 3
b = 1
c = 6
```

Boolean Logical operators:

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

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

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

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

Short-Circuit Logical Operators(&&, ||):

- Java provides two interesting Boolean operators not found in many other computer languages.
- These are secondary versions of the Boolean AND and OR operators, and are known as *short-circuit* logical operators. As you can see from the preceding table,
- the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator results in **false** when **A** is **false**, no matter what **B** is.
- Example:

```
class Test
{
        public static void main(String args[])
         {
             int a=10,b=30;
             if(a<10 && b<40)  //false
                     System.out.println("Hello");
```

```
                if(a>5 || b>40)  //true
                        System.out.println("Hai");
               }
        }
```
- Output:
  Hai

The ? Operator:

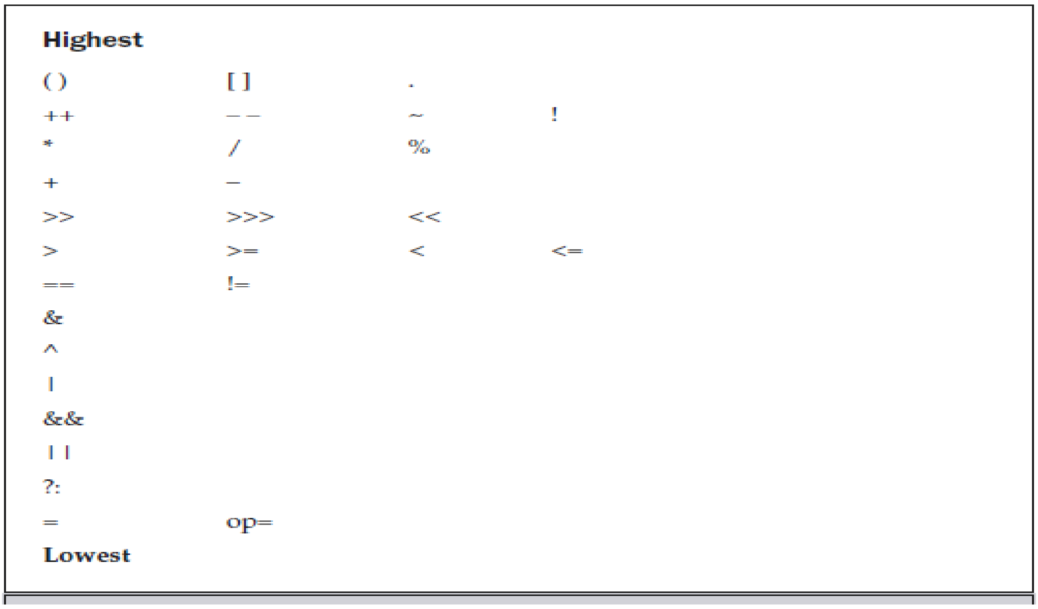
- Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements.
- The **?** has this general form:
  expression1 **?** *expression2* **:** *expression3*
- In the above form expression1 results a Boolean value either true or false. If it is true expression2 evaluated otherwise epression2 evaluated.
- Example:

```
        class Ternary
         {
                public static void main(String args[])
                {
                        int i, k;
                        i = 10;
                        k    =    i    <    0    ?    -i    :    i;
                        System.out.print("Absolute value of ");
                        System.out.println(i + " is " + k);
                }
}
```
- Output :

```
        Absolute value of i is 10
```

Operator Precedence:

- Bellow table shows the order of precedence for Java operators, from highest to lowest.



Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

- This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

  a >> (b + 3)
- However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:
  (a >> b) + 3

- The given expression   a | 4 + c >> b & 7   can be evaluated as follows

  (a | (((4 + c) >> b) & 7))

**********________________***********

Typecasting:(primitive type conversion):

- Typecasting means converting one data type to another data type.
- Typecasting can be done in two ways
  1. Implicit typecasting
  2. Explicit typecasting

Implicit typecasting:

- Compliler perform implicit typecasting automatically when smaller data type value into larger data type.
- In implicit type casting there is no loss of information.
- Implicit typecasting also called as widening or up casting.
- Example:

```
int  n='a';
System.out.println(n); //97
```

$$\begin{tikzcd} \text{byte} & \text{short} \arrow[r] & {} & {} \arrow[rd] & & & \\ & & & \text{int} & {} & \text{long} & \text{float} & \text{double} \\ \text{char} & & & {} \arrow[ru] & & & \end{tikzcd}$$

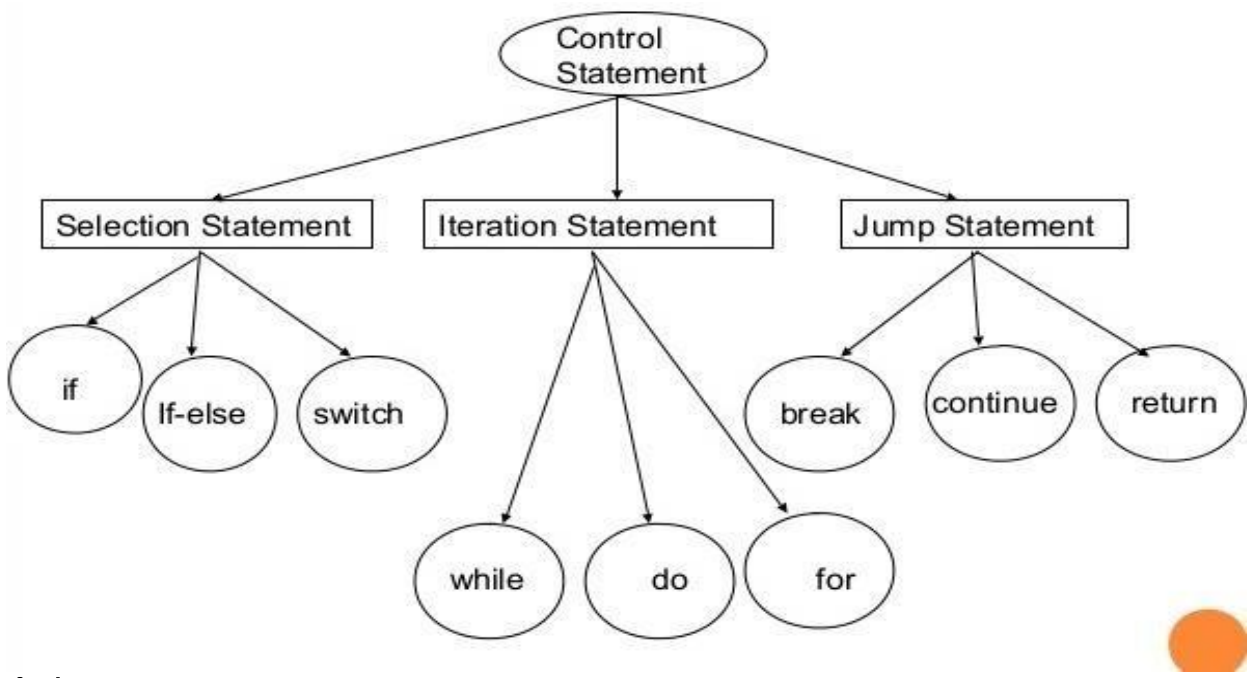
Explicit typecasting:

- Programmer performs explicit typecasting when larger data type value into smaller data type.
- In explicit type casting there may be chance of loss of information.
- Implicit typecasting also called as narrowing or down casting.
- Example:

```
double d=10.5;
int  n=(int)d; // n becomes 10
```

$$\begin{tikzcd}
\text{byte} & \text{short} \arrow[l] & \\
 & & \text{int} \arrow[ul] \arrow[dl] & \text{long} \arrow[l] & \text{float} \arrow[l] & \text{double} \arrow[l] \\
\text{char} & {} &
\end{tikzcd}$$

Decision making and Looping statements:



if-else:

- General form of if-else is

```
if ( condition )
{
   // true action
}
else
{
   // false action
}
```

- In the above general form condition must be Boolean type either true or false.

  Example:

```
int  a=10,b=20;
```

```
      if (a<b)
            System.out.println(“ a is smaller than b”); // executes
     else
           System.out.println(“ a is larger than b”);
```

- **Nested if- else:**

```
int a=10,b=20,c=30;
      if (a<b && a<c)
            System.out.println(“ a is largest”); // executes
      else if( b<a && b<c )
            System.out.println(“ b is largest”);
     else
            System.out.println(“ c is largest”);
```

switch:

- The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

  Syntax:

```
        switch(expression)
        {
case value1:     //code to be executed;
break;  //optional
case value2:    //code to be executed;
break;  //optional
                    ......

                    default:       //code to be executed if all cases are not matched;
}
```

Example:

```
int choice=2;
switch(choice)
{
        case 1: System.out.println(“this is case 1”);
                break
        case 2: System.out.println(“this is case 2”);
                break;
        case 3: System.out.println(“this is case 3”);
                break;

        default: System.out.println(“this is default case”);
}
```

Output: this is case 2

for loop:

- The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
- There are three types of for loop in java.
  - Simple For Loop
  - For-each or Enhanced For Loop
  - Labeled For Loop

Simple for Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

```
for(initialization;condition;incr/decr)
{
        //code to be executed
}
```

Example:

```
class ForExample

{

        public static void main(String[] args)
        {
                 for(int i=1;i<=10;i++)
                {
                        System.out.print(i+" ");
                }
        }
}
```

for-each Loop

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

Syntax:

```
for(Type var:array)
{
        //code to be executed
}
```

Example:

```
class ForEachExample
{
    public static void main(String[] args)
    {
       int arr[]={12,23,44,56,78};

       for(int i:arr)
           {
```

```
                    System.out.println(i+" ");
                }
            }
        }
```

Labeled for Loop

- We can have name of each for loop. To do so, we use label before the for loop. It isuseful if we have nested for loop so that we can break/continue specific for loop.
- Normally, break and continue keywords breaks/continues the inner most for loop only.

Syntax:

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

Example:

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

Output:

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

While Loop:

- The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

  Syntax:

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

  Example:

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

  Output:

```
1 2 3 4 5 6 7 8 9 10
```

do-while Loop:

- The Java *do-while loop* is used to iterate a part of the program several times.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.
- The Java *do-while loop* is executed at least once because condition is checked after loop body.

  Syntax:

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

  Example:

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

  Output:

```
1 2 3 4 5 6 7 8 9 10
```

break Statement:

- The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Syntax:

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

Example:

```
class BreakExample
   {
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
                {
                         if(i==5)
                        {
break;
}
                        System.out.print(i);
                }
        }
}
```

Output:

```
1 2 3 4
```

continue Statement:

- The Java *continue statement* is used to continue loop.
- It continues the current flow of the program and skips the remaining code at specified condition.
- In case of inner loop, it continues only inner loop.

Syntax:

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

Example:

```
class ContinueExample
 {
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
                {
                        if(i==5)
                        {
continue;
}
                        System.out.print(i);
                }
}
}
```

Output:

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

Methods:

- In java, a method is like function i.e. used to expose behavior of an object.
- The advantages of methods are code reusability and code optimization
- This is the general form of a method:

```
type name(parameter-list)
{
        // body of method
}
```

- Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas.
- Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

  return *value*;

  Here, *value* is the value returned.

- **Example program for usage of method:**

```
class Box
 {
        double    width;
        double    height;
        double depth;
        void volume() // method
        {
                double          vol=width*height*depth;
                System.out.println("Volume is " + vol);
        }
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box  mybox  =  new  Box();
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth  =  15;
                mybox.volume();
        }
}
```

Output: Volume is 3000.0

- **Example program for return a value from method :**

```
class Box
 {
        double      width;
        double     height;
        double depth;
        double volume() // method return type is double
        {
                return width*height*depth;
        }
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box    mybox    =    new    Box();
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                double vol= mybox.volume();
                System.out.printlln(“volume is” + vol);
        }
}
```

Output: Volume is 3000.0

Example program for method with parameters:

```
class Values
 {
        void add(int a,int b)
        {
                int c=a+b;
                System.out.printlln(“addition of a, b  is ”+c);
        }
 }
class Addition
{
        public static void main(String args[])
        {
                Addition      ad=new      Addition();
                ad.add(10,20);
        }
}
```

Output: addition of a, b is 30

- **Returning values from a method:**

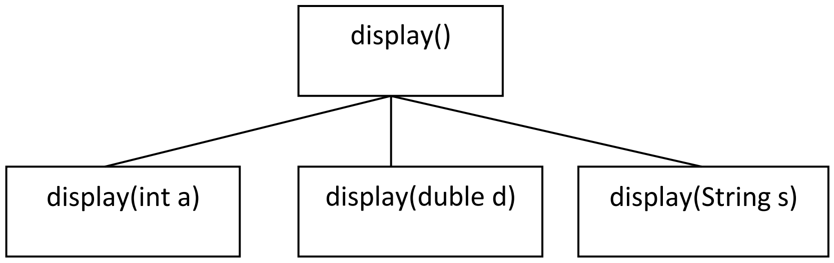
```
class Values
 {
        void add(int a,int b)
        {
                c=a+b;
                return c;
        }
 }
class Addition
{
        public static void main(String args[])
        {
                int res;
                Addition      ad=new      Addition();      res=ad.add(10,20);
                System.out.printlln("addition of a, b  is "+res);
        }
}
```

Output: addition of a, b is 30

******_________*******

Method Overloading:

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- These methods are said to be *overloaded,* and the process is referred to as *method overloading.*
- Method overloading is one of the ways that Java implements polymorphism.
- **Method overloading** is also called as " **Static binding** or **Compile time polymorphism** or **Early binding**".



- **Example for method overloading**

```
class Overload
 {
        void dispaly()
        {
                System.out.println("No parameters");
        }
        void display(int a)
        {
                System.out.println("integer a: " + a);
        }
        void display(double d)
        {
                System.out.println("double d: " + d);
        }
        void display(String s)
        {
                System.out.println("String s:" + s);
        }
}  class OverloadDemo
{
        public static void main(String args[])
        {
                OverloadDemo   ob   =   new   OverloadDemo();
                ob.display();
                ob. display(10);
                ob. display(10.325);
                ob. display("hello");
        }
}
```

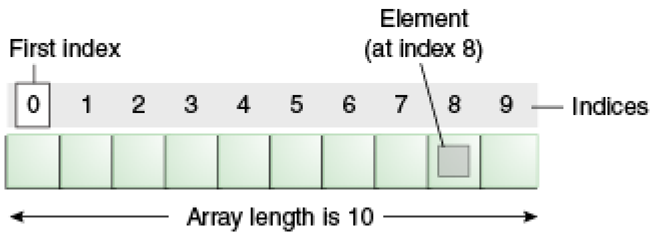
Output:

```
No      parameters
integer a: 10
double    d:    10.325
```

```
String s: hello
```

Arrays:

- An *array* is a group of similar data elements that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.



- The advantage of an array is code optimization and random access.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array

- Syntax for declare an array

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

- Instantiation of an Array

```
arr=new datatype[size];
```

- Example

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

                int     a[]=new     int[3];
                a[0]=10;
                a[1]=20;
                a[2]=70;
                for(int i=0;i<a.length;i++) System.out.println(a[i]);
        }
}
```

Output: 10
20
70

- **Declaration, Instantiation and Initialization of an Array:**

```
datatype arr[]={val1,val2,val3,…};//declaration, instantiation and initialization
```

Example:

```
class Testarray1
{
        public static void main(String args[])
        {
                int a[]={33,3,4,5};
                for(int                 i=0;i<a.length;i++)
                        System.out.print(a[i]+" ");
        }
}
```

Output:33 3 4 5

Multidimensional array

- data is stored in row and column based index (also known as matrix form).
- Syntax to Declare Multidimensional Array

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

- To instantiate Multidimensional Array

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

- To initialize Multidimensional Array

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
…………………arr[2][2]=9;
```

- Example:

```
    class Testarray3
    {
                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();
                        }
                  }
}
```

Output:1 2 3
2 4 5
4 4 5

******______******

UNIT- 1 - PART-B

Classes, Objects and Inheritance

Syllabus:

Classes and Objects- classes, Creating Objects, Methods, constructors, overloading methods and constructors, garbage collection, static keyword, this keyword, parameter passing, recursion, Arrays, Strings, Command line arguments.
Inheritance: Types of Inheritance, Deriving classes using extends keyword, concept of overriding, super keyword, final keyword.

Class:

- A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.
- Simple classes may contain only code or only data, most real-world classes contain both.
- A class is declared by use of the **class** keyword.
- The general form of a class is

```
class  classname
{
        type  instance-variable1;
        type  instance-variable2;
        // …….
        type  instance-variableN;
        type  methodname1(parameter-list)
        {
                // body of method
        }
        type  methodname2(parameter-list)
         {
                // body of method
        }
        // ...
        type  methodnameN(parameter-list)
        {
                // body of method
        }
}
```

- The data, or variables, defined within a **class** are called ***instance variables.***
- The code is contained within *methods.* Collectively, the methods and variables defined within a class are called ***members*** of the class.
- Simple Class like as bellow

```
class Box
{
        double width;
        double height;
        double depth;
}
```

- In the above, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**.
- a **class** declaration only creates a template; it does not create an actual object.
- For creating a **Box** object we use,

  Box mybox = new Box(); // create a Box object called mybox
- Here **mybox** will be an instance of **Box**.
- To acces variables, methods inside a class we have to use .(dot) operator.
- The dot operator links the name of the object with the name of an instance variable.
- to assign the **width** variable of **mybox** the value 100, you would use the following statement:

  mybox.width=100;
- **Example:**

```
class Box
 {
        double width;
        double height;
        double depth;
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox = new Box();
                double vol;
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                vol = mybox.width * mybox.height * mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

- The above program must save with **BoxDemo.java** (because BoxDemo contains main())
- After compiling the program two class files are created one is **Box.class** and other one is **BoxDemo.class**
- At the time of running BoxDemo.class file will be loaded by JVM
- **For compilation:** javac BoxDemo.java
- **For Running:** java BoxDemo
- **Output:**

```
Volume is 3000.0
```

- **We can create multiple objects for the same class**
- Example

```
class Box
 {
        double width;
        double height;
        double depth;
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box  mybox1 = new Box();
                Box  mybox1 = new Box();
                double vol1,vol2;
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                vol1 = mybox1.width * mybox1.height * mybox1.depth;
                mybox1.width = 10;
                mybox1.height = 15;
                mybox1.depth = 5;
                vol2= mybox2.width * mybox2.height * mybox2.depth;
                System.out.println("Volume of mybox1  is " + vol1);
                System.out.println("Volume of mybox2  is " + vol2);

        }
}
```

- Output

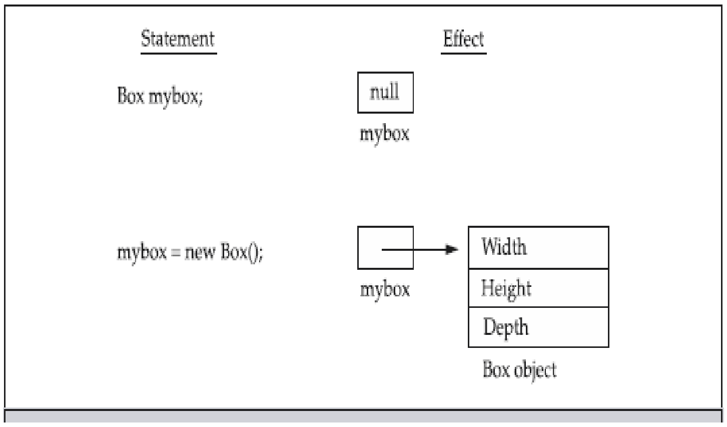
```
Volume of mybox1  is 3000.0
Volume of mybox2  is 750.0
```

Objects:

- Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.
- for creating objects we have to use following code.

Box mybox;
// declare reference to object

mybox = new Box();
// allocate a Box object
Here new keyword is allocating memory of Box object.



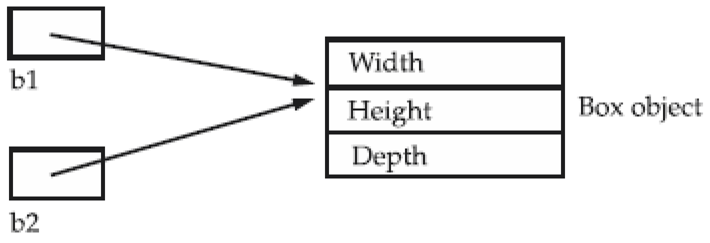
3

Assigning Object Reference Variables:

- For example see the following code

```
Box b1 = new Box();
Box b2 = b1;
```

- **In the above code b1** and **b2** will both refer to the *same* object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object.
- It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.



******----------*******

Methods:

- In java, a method is like function i.e. used to expose behavior of an object.
- The advantages of methods are code reusability and code optimization
- This is the general form of a method:

  type name(*parameter-list*)
  {
  // body of method
  }
- Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas.
- Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

  return *value*;

  Here, *value* is the value returned.

- **Example program for usage of method:**

```
class Box
 {
        double width;
        double height;
        double depth;
        void volume() // method
        {
                double vol=width*height*depth;
                System.out.println("Volume is " + vol);
        }
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox = new Box();
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                mybox.volume();
        }
}
```

Output: Volume is 3000.0

- **Example program for return a value from method :**

```
class Box
 {
        double width;
        double height;
        double depth;
        double volume() // method return type is double
        {
                return width*height*depth;
        }
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox = new Box();
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                double vol= mybox.volume();
                System.out.printlln("volume is" + vol);
        }
}
```

Output: Volume is 3000.0

- **Example program for method with parameters:**

```
class Values
 {
        void add(int a,int b)
        {
                int c=a+b;
                System.out.printlln("addition of a, b  is "+c);
        }
 }
class Addition
{
        public static void main(String args[])
        {
                Addition ad=new Addition();
                ad.add(10,20);
        }
}
```

Output: addition of a, b is 30

- **Returning values from a method:**

```
class Values
 {
        void add(int a,int b)
        {
                c=a+b;
                return c;
        }
 }
class Addition
{
        public static void main(String args[])
        {
                int res;
                Addition ad=new Addition();
                res=ad.add(10,20);
                System.out.printlln("addition of a, b  is "+res);
        }
}
```

Output: addition of a, b is 30

Constructors:

- A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors do not any return type, not even **void.**
- This is because the implicit return type of a class' constructor is the class type itself.
- Example:

```
class Addition
{
        Addition()
        {
                System.out.println("This is Default constructor");
        }
}
class ConstructorDemo
{
        public static void main(String args[])
        {
                Addition ad=new Addition();
        }
}
```

Output: This is Default constructor

- Now we can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Addition ad=new Addition();
```

new Addition( ) is calling the **Addition( )** constructor.

- **When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class.**

Parameterized Constructors:

- We can pass parameters to the constructor

```
class Addition
{
        Addition(int a,int b)
        {
                int c;
                c=a+b;
                System.out.println("Tha addition of a, b is ");
                System.out.println(c);
        }
}
```

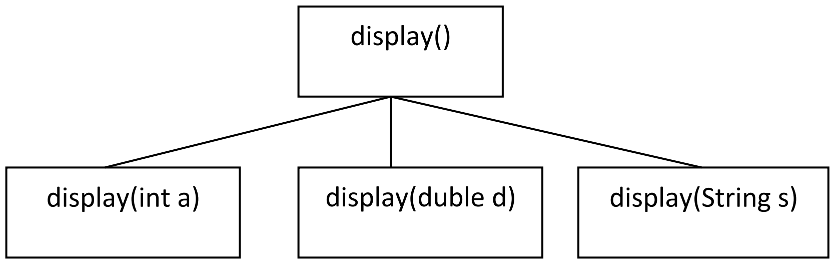
```
class ParameterConstructor
{
        public static void main(String args[])
        {
                Addition ad=new Addition(10,20);
        }
}
```

Output: Tha addition of a, b is 30

******_________*******

Method Overloading:

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- These methods are said to be *overloaded,* and the process is referred to as *method overloading.*
- Method overloading is one of the ways that Java implements polymorphism.
- **Method overloading** is also called as " **Static binding** or **Compile time polymorphism** or **Early binding**".



- **Example for method overloading**

```
class Overload
 {
        void dispaly()
        {
                System.out.println("No parameters");
        }
        void display(int a)
        {
                System.out.println("integer a: " + a);
        }
        void display(double d)
        {
                System.out.println("double d: " + d);
        }
        void display(String s)
        {
                System.out.println("String s:" + s);
        }
}
```

```
class OverloadDemo
{
        public static void main(String args[])
        {
                OverloadDemo ob = new OverloadDemo();
                ob.display();
                ob. display(10);
                ob. display(10.325);
                ob. display(“hello”);
        }
}
```

Output:

```
No parameters
integer a: 10
double d: 10.325
String s: hello
```

Constructor overloading:

- In addition to overloading normal methods, you can also overload constructor methods.
- In Java it is possible to define two or more constructors within the same class that share the same name, as long as their parameter declarations are different.
- Example

```
class Display
{
        Dispaly()
        {
                System.out.println("No parameters");
        }
        Dispaly(int a)
        {
                System.out.println("integer a: " + a);
        }
        Dispaly(double d)
        {
                System.out.println("double d: " + d);
        }
        Display(String s)
        {
                System.out.println("String s:" + s);
        }
}
class OverloadDemo
{
        public static void main(String args[])
        {
                Display ds=new Display();
                Display ds1=new Display(10);
                Display ds2=new Display(10.325);
                Display ds3=new Display(“hello”);
        }
}
```

Output:

```
No parameters
integer a: 10
double d: 10.325
String s: hello
```

Java 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 do so, we were using 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 can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By annonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By annonymous object:

```
new Employee();
```

static keyword:

- It is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static**.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.
- Instance variables declared as **static** are, essentially, global variables.
- Methods declared as **static** have several restrictions:
    - They can only call other **static** methods.
    - They must only access **static** data.
    - They cannot refer to **this** or **super** in any way.
- We can declare a **static** block which gets executed exactly once, when the class is first loaded.
- Example

```
class UseStatic
{
        static int a = 3;
        static int b;
        static void display(String s)
        {
                System.out.println(“Static method invoked String s= “+s);
        }
        static
        {
                System.out.println("Static block initialized.");
                b = a * 4;
        }
}
class  StaticDemo
{
        public static void main(String args[])
        {
                System.out.println(“static variable a= ”+ UseStatic.a);
                System.out.println(“static variable b= “+UseStatic.b);
                UseStatic. Display(“JAVA”);
        }
}
```

<u>Ouput:</u>

```
Static block initialized.
static variable a= 3
static variable b= 12
static method invoked String s= JAVA
```

this keyword:

- In java, this is a **reference variable** that refers to the current object.

Usage of this keyword

- Here some usages of this keyword.
    1. To refer current class instance variable.
    2. To Invoke current class constructor.

To refer current class instance variable:

- If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

```
class Student
{
         int id;
        String name;
        Student10(int id,String name)
        {
                this.id = id;
                this.name = name;
        }
        void display()
        {
                System.out.println(id+" "+name);
        }
        public static void main(String args[])
        {
                Student  s1 = new Student10(111,"Karan");
                Student  s2 = new Student10(321,"Aryan");
                s1.display();
                s2.display();
        }
}
```

Output:

```
111 Karan
321 Aryan
```

To invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor (constructor chaining).
- This approach is better if you have many constructors in the class and want to reuse that constructor.
- Example:

```
class Student1
{
        int id;
        String name;
        Student1()
        {
                System.out.println("default constructor is invoked");
        }

        Student1(int id,String name)
        {
                this ();//it is used to invoked current class constructor.
                this.id = id;
```

```
                    this.name = name;
          }
          void display()
          {
                   System.out.println(id+" "+name);
          }

          public static void main(String args[])
          {
                    Student1 e1 = new Student13(111,"karan");
                    Student1 e2 = new Student13(222,"Aryan");
                    e1.display();
                    e2.display();
          }
}
```

Output:

```
default constructor is invoked
default constructor is invoked
111 Karan
222 Aryan
           **********------------*********
```

parameter passing techniques:

- call by value (or) pass by value
- call by reference (or) pass by reference

Call by value:

- In call by value we are passing only value to the method.
- If we perform any changes inside method that changes will not reflected to main method.
- Example:

```
class CallByVal
{

          public static void main(String args[])
          {
                   int x=20;
                   System.out.println("Before method calling value= "+x);
                   display(x);
                   System.out.println("After method calling value= "+x);
          }

          public static void display(int y)
          {
                   y=y+1;
                   System.out.println("Inside method value= "+y);
          }

}
```

Output:

```
Before method calling value= 20
Inside method value= 21
After method calling value= 20
```

Call by reference:

- In call by reference we are passing reference i.e an object to the method.
- In this case, If we perform any changes inside method that changes will not reflected to main method.
- Example:

```
class CallByRef
{
        int x;
        public static void main(String args[])
        {
                CalByRef c=new CalByRef();
                c.x=20;
                System.out.println("Before method calling value= "+c.x);
                display(c);
                System.out.println("After method calling value= "+c.x);
         }
        public static void display(CalByRef m)
        {
                m.x=m.x+1;
                System.out.println("Inside method value= "+m.x);
        }
}
```

Output:

```
Before method calling value= 20
Inside method value= 21
After method calling value= 21
```

******______******

Recursion:

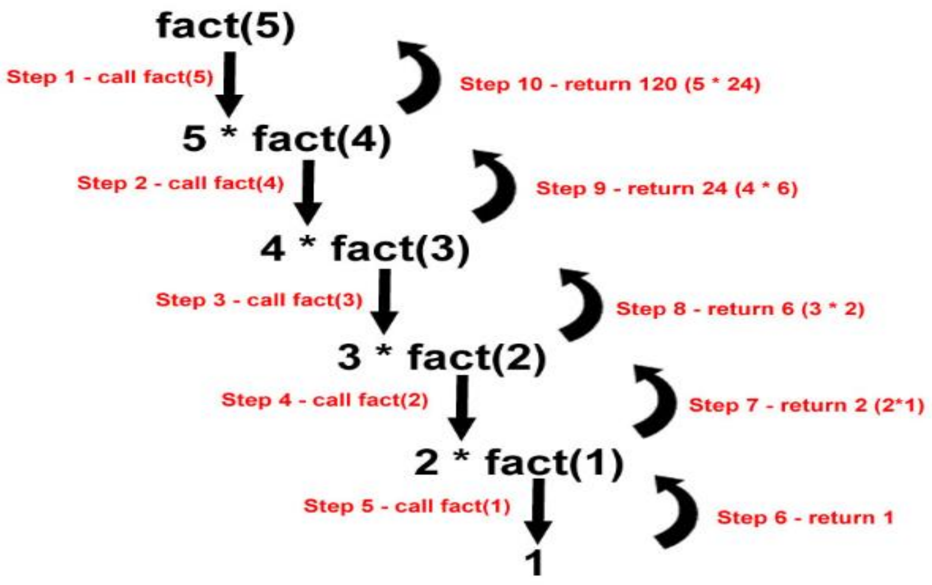
- Recursion is the process of method calling itself.
- A method calling itself is called as recursive method.
- The classic example of recursion is the computation of the factorial of a number.

```
class Factorial
{
        int fact(int n)
        {
                if(n==1)
                        return 1;
                else
                        n*fact(n-1) ;
        }
}
class Recursion
{
        public static void main(String args[])
        {
                Factorial f = new Factorial();
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

Output:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

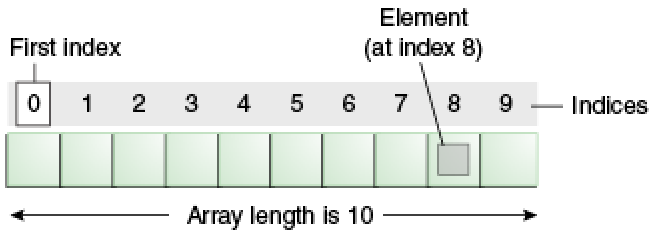
- Recursion process of fact(5) is given bellow.



******______******

Arrays:

- An *array* is a group of similar data elements that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.



- The advantage of an array is code optimization and random access.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array

- Syntax for declare an array

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

- Instantiation of an Array

```
arr=new datatype[size];
```

- Example

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

                int a[]=new int[3];
                a[0]=10;
                a[1]=20;
                a[2]=70;
                for(int i=0;i<a.length;i++)
                        System.out.println(a[i]);
        }
}

Output: 10
        20
        70
```

- **Declaration, Instantiation and Initialization of an Array:**

  datatype arr[]={val1,val2,val3,…};//declaration, instantiation and initialization

  Example:

```
class Testarray1
{
        public static void main(String args[])
        {
                int a[]={33,3,4,5};
                for(int i=0;i<a.length;i++)
                        System.out.print(a[i]+" ");
        }
}
```

  Output:33 3 4 5

Multidimensional array

- data is stored in row and column based index (also known as matrix form).
- Syntax to Declare Multidimensional Array

```
dataType[][]  arrayRefVar; (or)
dataType  [][]arrayRefVar; (or)
dataType  arrayRefVar[][]; (or)
dataType  []arrayRefVar[];
```

- To instantiate Multidimensional Array

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

- To initialize Multidimensional Array

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
…………………arr[2][2]=9;
```

- Example:

```
class Testarray3
{
              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();
                        }
                }
}
```

Output:

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

******-------******

Strings:

- Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters.
- The java.lang.String class is used to create string object.
- Strings are immutable in nature, i.e once an object created that object doesn't allow any changes to it.
- String is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
        char[] ch={'j','a','v','a'};
        String s=new String(ch);
is same as:
        String s="javA";
```

- There are two ways to create String object:
    1. By string literal
    2. By new keyword

String Literal

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

  String s="welcome";

- The advantage of string literal usage is memory efficiency.

By new keyword

String s=new String("Welcome");

Example:

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

Output

```
java
strings
example
```

String handling methods:

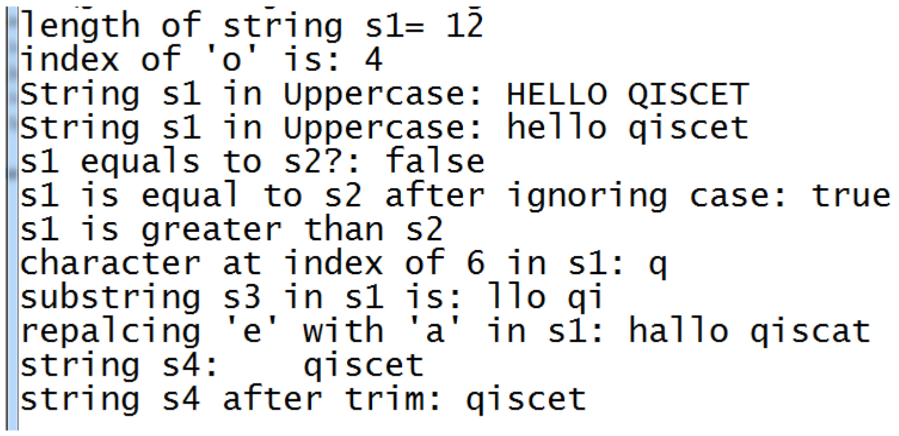
S.N	Method name	General form	Example
1	length()	int length( )	String s = new String("hello"); System.out.println(s.length());
2	charAt( )	char charAt(int *where*)	char ch; ch = "abc".charAt(1);
3	getChars( )	void getChars (int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)	String s = "This is a demo of the getChars method."; int start = 10; int end = 14; char buf[] = new char[10]; s.getChars(start, end, buf, 0); System.out.println(buf);
4	equals( )	boolean equals(Object *str*)	
5	equalsIgnoreCase( )	boolean equalsIgnoreCase (String *str*)	String s1 = "Hello"; String s2 = "hello"; System.out.println("s1 and s2 are equal " +s1.equals(s2)); System.out.println("s1 and s2 are equal" +s1.equalsIgnoreCase(s4));
6	substring( )	String substring(int *startIndex*) String substring(int *startIndex*, int *endIndex*)	String s1="hello java"; System.out.println(substring(1,4));
7	concat( )	String concat(String *str*)	String s1 = "one"; String s2 = s1.concat("two");
8	replace( )	String replace(char *original*, char *replacement*)	String s = "Hello".replace('l', 'w');
9	trim( )	String trim( )	String s = " Hello World "; System.out.println(s.trim());
10	toLowerCase( ) toUpperCase( )	String toLowerCase( ) String toUpperCase( )	String s1 = " HELLO"; String s2="hello"; System.out.println(s1.toLowerCase( )); System.out.println(s2.toUpperCase( ));

Example program String handling methods:

```
class StringDemo1
{
  public static void main(String args[])
  {
        String s1="hello qiscet";
        String s2="HELLO QISCET";
        System.out.println("length of string s1= "+s1.length());
        System.out.println("index of 'o' is: "+s1.indexOf('o'));
        System.out.println("String s1 in Uppercase: "+s1.toUpperCase());
        System.out.println("String s1 in Uppercase: "+s2.toLowerCase());
        System.out.println("s1 equals to s2?: "+s1.equals(s2));
        System.out.println("s1 is equal to s2 after ignoring case:"
                                        +s1.equalsIgnoreCase(s2));
        int result=s1.compareTo(s2);
        if(result==0)
```

```
                System.out.println("s1 is equals to s2");
        else if(result>0)
                System.out.println("s1 is greater than s2");
        else
                System.out.println("s1 is smaller than s2");
        System.out.println("character at index of 6 in s1: "+s1.charAt(6));
        String s3=s1.substring(2,8);
        System.out.println("substring s3 in s1 is: "+s3);
        System.out.println("repalcing 'e' with 'a' in s1: "+s1.replace('e','a'));
        String s4="   qiscet   ";
        System.out.println("string s4: "+s4);
        System.out.println("string s4 after trim: "+s4.trim());
  }
}
```

Output:



```
length of string s1= 12
index of 'o' is: 4
String s1 in Uppercase: HELLO QISCET
String s1 in Uppercase: hello qiscet
s1 equals to s2?: false
s1 is equal to s2 after ignoring case: true
s1 is greater than s2
character at index of 6 in s1: q
substring s3 in s1 is: llo qi
repalcing 'e' with 'a' in s1: hallo qiscat
string s4:    qiscet
string s4 after trim: qiscet
```

*******________*******

StringBuffer:

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.
- General form of StringBuffer

StringBuffer sb=new StringBuffer(“Hello World”);

- StringBuffer class exists in “java.lang” package.
- StringBuffer is immutable that means, any changes performed to the StringBuffer object that changes will reflect to object.

- **StringBuffer Methods:**

S. N	Method name	General form	Example

1	length( )	int length( )	StringBuffer sb = newStringBuffer("Hello");
			System.out.println("buffer = " + sb);
			System.out.println("length="+sb.length());
			System.out.println("capacity="+ sb.capacity());

2	capacity( )	int capacity( )	
3	ensure Capacity( )	void ensureCapacity (int *capacity*)	sb.ensureCapacity(30));

4	setLength( )	void setLength(int *len*)	StringBuffer sb = new StringBuffer("Hello");
			System.out.println("buffer before = " + sb);
			System.out.println("charAt(1) before = " + sb.charAt(1));
			sb.setCharAt(1, 'i');
			sb.setLength(2);
			System.out.println("buffer after = " + sb);
			System.out.println("charAt(1) after = " + sb.charAt(1));

5	charAt( )	char charAt(int *where*)	
6	setCharAt( )	void setCharAt(int *where*, char *ch*)	

7	getChars( )	void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)	StringBuffer sb=new StringBuffer("This is a demo of the getChars method.");
			int start = 10;
			int end = 14;
			char buf[] = new char[10];
			sb.getChars(start, end, buf, 0);
			System.out.println(buf);

8	append()	StringBuffer append(String *str*)
		StringBuffer append(int *num*)
			StringBuffer sb = new StringBuffer(40);
			s = sb.append("is a number");
			System.out.println(s);

9	insert( )	StringBuffer insert(int *index*, String *str*)
		StringBuffer insert(int *index*, char *ch*)
			StringBuffer sb = new StringBuffer("I Java!");
			sb.insert(2, "like ");
			System.out.println(sb);

10	reverse( )	StringBuffer reverse( )	StringBuffer s = new StringBuffer("abcdef");
			System.out.println(s);
			s.reverse();
			System.out.println(s);

11	delete( ) and deleteCharAt( )	StringBuffer delete(int *startIndex*, int *endIndex*)
		StringBuffer deleteCharAt(int *loc*)
			StringBuffer sb = new StringBuffer("This is a test.");
			sb.delete(4, 7);
			System.out.println("After delete: " + sb);
			sb.deleteCharAt(0);
			System.out.println("After deleteCharAt: " + sb);

12	replace( )	StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)	StringBuffer sb = new StringBuffer("This is a test.");
			sb.replace(5, 7, "was");
			System.out.println("After replace: " + sb);

13	substring( )	String substring(int *startIndex*)
		String substring(int *startIndex*, int *endIndex*)
			StringBuffer sb = new StringBuffer(”hello java”);
			String s= substring(1,4);
			System.out.println(s);

******_______*******

Command line arguments:

- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to **main( )**.
- Forexample,
-

```
class CommandLine
{
        public static void main(String args[])
         {
                for(int i=0; i<args.length; i++)
                        System.out.println("args[" + i + "]: " +args[i]);
        }
}
```

Compilation and Execution:

```
javac  CommandLine.java
java CommandLine this is a test 100 -1
```

Output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

*******------------********