ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES: : TIRUPATI

Venkatapuram(Village),Renigunta(Mandal),Tirupati,Chitoor District, Andhra Pradesh-517520.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME OF THE FACULTY: A.MRINALINI

REGULATION: B.TECH II-II SEMESTER

SUBJECT: (15A05403) OBJECT ORIENTED PROGRAMMING USING THOUGH JAVA

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech II - II sem (C.S.E)	T	Tu	C
	3	1	3

(15A05403) OBJECT ORIENTED PROGRAMMING USING THOUGH JAVA

Course Objectives:

- Study the syntax, semantics and features of Java Programming Language
- Learn the method of creating Multi-threaded programs and handle exceptions
- Learn Java features to create GUI applications & perform event handling

Course Outcomes:

- Ability to solve problems using object oriented approach and implement them using Java
- Ability to write Efficient programs with multitasking ability and handle exceptions
- Create user friendly interface

UNIT I:

The History and Evolution of Java:

Java's Lineage, The Creation of java, how java changed the internet, Java's magic: The byte code, Servlets: java on the server side, java Buzzwords, Evolution of java.

An Overview of Java:

Object Oriented Programming, Two control statements, Using blocks of codes, Lexical issues, The java class Libraries.

Data Types, Arrays and Variables:

Primitive Types, Integers, Floating-point Types, Characters, Booleans, literals, variables, Type conversion and casting, Automatic Type Promotion in Expressions, Arrays, strings, Pointers.

UNIT II:

Operators:
Arithmetic Operators, The Bitwise Operators, Relational Operators, Boolean Logic operators, The assignment operator, The ? Operator, Operator Precedence, Using Parentheses.

Control Statements:
Java's selection Statements, Iteration statements, Jump Statements.

Introducing Classes:

Class Fundamentals, Declaring Objects, Assuming Object reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection, The Finalize() method, A Stack class. Overloading Methods, Using Object as Parameter, Argument Passing, Returning Objects, Recursion, Introducing Access control, Understanding static, Introducing Nested and Inner classes, Exploring the String class, Using Command line Arguments, Varargs: variable-Length Arguments.

UNIT III:

Inheritance: Basics, Using super, creating a multi level hierarchy, when constructors are executed, method overriding, dynamic method dispatch, using abstract class, using final with inheritance, the object class.

Packages and Interfaces:

Packages, Access protection, Importing Packages, Interfaces, Default Interfaces, Default interface methods, Use static methods in an Interface, Final thoughts on Packages and interface**s.**

Exception Handling:

Exception handling Fundamentals, Exception Types, Uncaught Exceptions, Using try and catch, Multiple catch clauses, Nested try statements, throw, throws, finally, Java Built-in Exceptions, Creating your own exception subclasses, Chained Exceptions, Three Recently added Exceptions features, Using Exceptions.

UNIT IV:

Multithreaded Programming:

The java Thread Model, The main thread , Creating Thread, Creating Multiple Threads, Using isAlive() and join(), Thread Priorities, Synchronization, Interthread Communication, Suspending, resuming and stopping threads, Obtaining a thread state, Using Multithreading.

I/O, Applets, and Other Topics:

I/O basics, Reading Console input, Writing console Output, The PrintWriter class, Reading and writing files, Automatically closing a file, Applet fundamentals, enumerations type wrappers auto boxing annotations, Generics: The general form of a generics class, creating a generic method, generics interfaces.

UNIT V:

Introduction the AWT: Working with windows, graphics and Text:

AWT classes, window fundamentals, working with frame windows, creating a frame window in a an AWT Based applet, creating a window program, displaying information within a window, Graphics, working with color, setting the paint mode, working with fonts, managing text output using font metrics,.

Using AWT controls, Layout Mangers, and Menus:
AWT control fundamentals, Labels, using buttons, applying check boxes, check box group, choice controls, using lists, Managing scroll bars, using a Text field, Using a Text area, understanding layout managers, Menu bars and Menus, dialog boxes, file dialog, Overriding paint().

TEXT BOOKS:
1.”Java The Complete Reference”, Herbert Schildt, MC GRAW HILL Education, 9th Edition,2016.

REFENCE BOOKS:

1. “Programming with Java” T.V.Suresh Kumar, B.Eswara Reddy, P.Raghavan Pearson Edition.
2. “Java Fundamentals - A Comprehensive Introduction”, Herbert Schildt and Dale Skrien, Special Indian Edition, McGrawHill, 2013.
3. “Java – How to Program”, Paul Deitel, Harvey Deitel, PHI.
4. “Core Java”, NageswarRao, Wiley Publishers.
5. “Thinking in Java”, Bruce Eckel, Pearson Education.
6. “A Programmers Guide to Java SCJP”, Third Edition, Mughal, Rasmussen, Pearson.

“Head First Java”, Kathy Sierra, Bert Bates, O’Reilly “SCJP – Sun Certified Programmer for Java Study guide” – Kathy Sierra, Bert Bates, McGrawHill

UNIT-1

1.History of java

Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. For the green team members, it was an

advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

James Gosling

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.
1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.
4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Oak name for java language?
5) **Why Oak?** Oak is a symbol of strength and choosen as a national tree of many countries like 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 name for java language?

7) **Why they choosed java name 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.
8) Java is an island of Indonesia where first coffee was produced (called java coffee).
9) Notice that Java is just a name not an acronym.
10) Originally 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 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)

Java Lineage:

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from--or are responses to--its predecessors.
Java is a high-level, third generation programming language, like C, FORTRAN, Smalltalk, Perl, and many others. You can use Java to write computer applications that play games, store data or do any of the thousands of other things computer software can do. Java is a programming language created by **James Gosling** from Sun Microsystems (Sun) in 1991. The first publicly available version of Java (Java 1.0) was released in 1995. Java is most similar to C. However although Java shares much of C's syntax.The concepts are related to c++.

Birth of C language:
The C Programming Language by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted .The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs.
• Ease-of-use versus power

• Safety versus efficiency
• Rigidity versus extensibility.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++.

. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The Stage Is Set for Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of computer architecture. As of 2016, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers. Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Most special about Java in relation to other programming languages is that it special programs called **applets** that can be downloaded from the Internet and played safely within a web browser. Java language is called as an Fully Object-Oriented Programming language

1.1Following are the differences Between C and C++ :

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used in C++. Polymorphism is the most Important Feature of OOPS.
4. Operator overloading is not possible in C.	4. Operator overloading is one of the greatest Feature of C++.

5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global varioables are not allowed.
8. In C • scanf() Function used for Input. • printf() Function used for output.	8. In C++ • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other Functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	13.In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception Handling is not present.	15. In C++, Exception Handling is done with Try and Catch block.

1.2Difference between c++& java

C++	**Java**
C++ is not a purely object-oriented programming, since it is possible to write C++ programs without using a class or an object.	Java is purely an object-oriented programming language, since it is not possible to write a Java program without using at least one class.
Pointers are available in C++.	We cannot create and use pointers in Java.
Allocating memory and de-allocating memory is the responsibility of the programmer.	Allocation and de-allocation of memory will be taken care of by JVM.
C++ has **goto** statement.	Java does not have **goto** statement.
Automatic casting is available in C++.	In some cases, implicit casting is available.

	But it is advisable that the programmer should use casting wherever possible.
Multiple inheritance feature is available in C++.	No multiple inheritance in Java, but there are means to achieve it.
Operator overloading is available in C++.	It is not available in Java.
#define, typedef and header files are available in C++.	#define, typedef and header are not available in Java, but there are means to achieve them.
There are 3 access specifiers in C++: private, public and protected.	Java supports 4 access specifiers: private, public, protected, and default.
There are constructors and destructors in C++.	Only constructors are there in Java. No destructors are available in this language.

The Creation of Java:

James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan conceived Java at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payane, Frank Yellin, and Tim Lindholm were Key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

The trouble with C and C++(and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time- consuming to create. An easier - and more cost – efficient – solution was needed. In an attempt of find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

- Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (plat form - independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems.

Further, because much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and Unix, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with many types of computers, operating systems, and

CPUs. Even though many types of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritation but low-priority had become a high-profile necessity.

- Java derives much of its character from C and C++. This is by intent. The java designer knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers.

In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ Successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is also a programmer’s language. Second. Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, java is not a language with training wheels. It is a language for professional programmers.

computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted java was the need **for platform-independent programs** destined for distribution on the **Internet**. However, java also embodies changes in the way that people approach the writing of programs. Thus, java is not a language that exists in isolation. Rather, it is the current instance of an ongoing process begun many years ago. This fact alone is enough to ensure java a place in computer language history. Java is to Internet programming what C was to systems programming: a revolutionary force that will change the world.

The c# connection:

Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore .Perhaps the most important example of Java’s influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java.

What's Similar Between C# and Java?

C# and Java are actually quite similar, from an application developer's perspective. The major similarities of these languages are

1.All Objects are References
2. Garbage Collection
3. Both C# and Java are Type-Safe Languages
4.Both C# and Java Are "Pure" Object-Oriented Languages
5.Single Inheritance
6.Built-in Thread and Synchronization Support
7.Formal Exception Handling
8. Built-in Unicode Support

What's Different Between C# and Java?

While C# and Java are similar in many ways, they also have some differences.

1. Formal Exception Handling
2. Java Will Run on "Any" Operating System
3. C# and Java Language Interoperability
4. C# Is a More Complex Language than Java
5. C# and Java Keyword Comparison
6. Operator
7. Delegate
8. Synchronized
9. Threading and synchronization

How java changed the internet

The Internet helped JAVA to the forefront of programming and JAVA in turn had a great impact on the Internet. While simplifying the web programming JAVA innovated a new type of networked program called the applet that changed the way the online world thought about content. JAVA also addressed some of main issues associated with the Internet like portability and security.

1.Java Applets

An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems.

2,Security : As we know when we download a normal program we are taking a risk, because the code that we are downloading might contain a virus of any other harmful code. At the core of the problem is the fact that malicious code can cause its damage as it has gained unauthorized access to system resources. In order for JAVA to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. JAVA achieved this protection by confining an applet to the JAVA execution environment and not allowing its access to other parts of the computer.

3.Portability : It is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. Some means of generating portable executable code was needed. For example :- In case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPU's , operating systems and browsers connected to the internet.

Java's magic: The byte code:

The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a set of instructions designed to be executed by the Java run-time system known as *Java Virtual Machine* (JVM). That is, in its standard form, the JVM is an *interpreter for bytecode.*
Translating a Java program into bytecode helps makes it easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Although the details of JVM will differ from platform to platform, all interpret the same Java bytecode.
The fact that a Java program is interpreted also helps to make it secure. Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system.
Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release. When JIT compiler is part of the JVM, it compiles bytecode into the executable code in real time, on a piece-by-piece, demand basis. It is not possible to compile an entire Java program into executable code all at once, because Java performs various run time checks that can be done only at run time. Instead, the JIT compiles code as it is needed, during execution.

Servlets: java on the server side

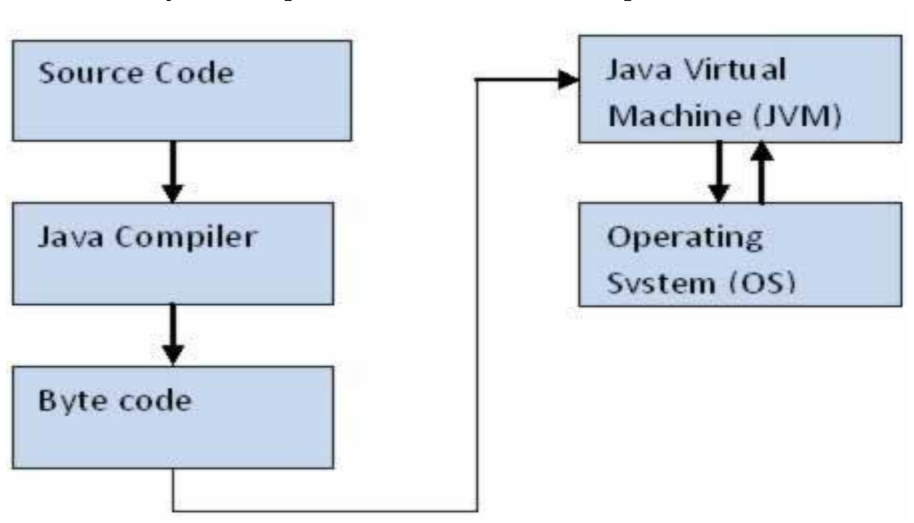
Java would also be useful on the server side. The result was the *servlet*. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet,Java spanned both sides of the client/server connection.
The servlet offers several advantages, including increased performance.Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container.

java Buzzwords:

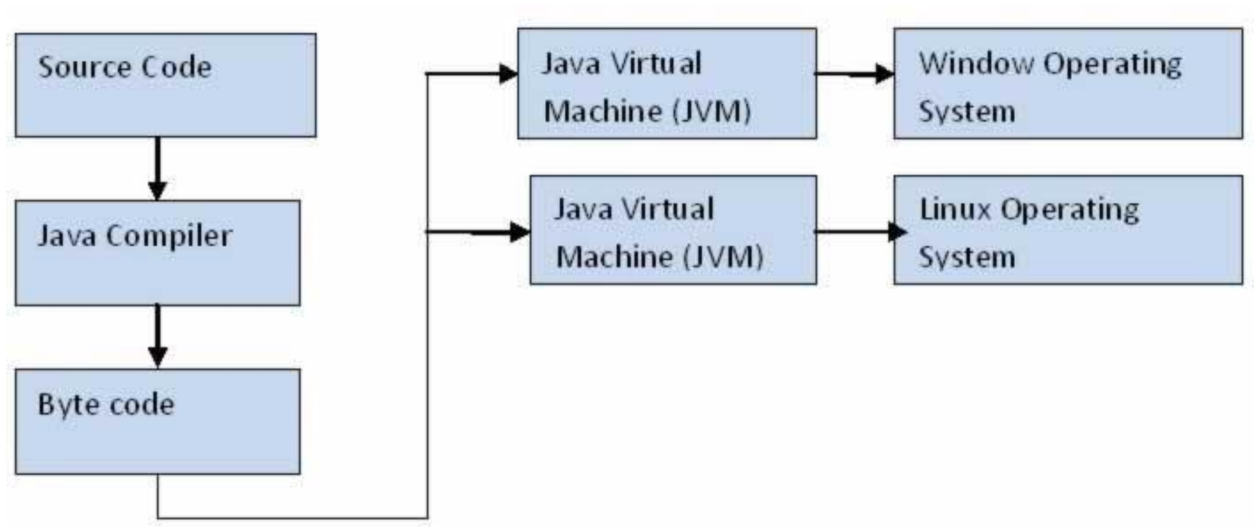
• Simple
• Secure
• Portable

• Object-oriented
• Robust
• Multithreaded
• Architecture-neutral
• Interpreted
• High performance
• Distributed
• Dynamic

1)**Compiled and Interpreter:- Java** has both Compiled and Interpreter Feature Program of java is First Compiled and Then it is must to Interpret it .First of all The Program of java is Compiled then after Compilation it creates Bytes Codes rather than Machine Language.
Then After Bytes Codes are Converted into the Machine Language is Converted into the Machine Language with the help of the Interpreter So For Executing the java Program First of all it is necessary to Compile it then it must be Interpreter .



2**) Platform Independent**:- Java Language is Platform Independent means program of java is Easily transferable because after Compilation of java program bytes code will be created then we have to just transfer the Code of Byte Code to another Computer.
This is not necessary for computers having same Operating System in which the code of the java is Created and Executed After Compilation of the Java Program We easily Convert the Program of the java top the another Computer for Execution.



3) **Object-Oriented**:- We Know that is purely OOP Language that is all the Code of the java Language is Written into the classes and Objects So For This feature java is Most Popular Language because it also Supports Code Reusability, Maintainability etc.

4) **Robust and Secure**:- The Code of java is Robust andMeans ot first checks the reliability of the code before Execution When We trying to Convert the Higher data type into the Lower Then it Checks the Demotion of the Code the It Will Warns a User to Not to do this So it is called as Robust
Secure : When We convert the Code from One Machine to Another the First Check the Code either it is Effected by the Virus or not or it Checks the Safety of the Code if code contains the Virus then it will never Executed that code on to the Machine.

5) **Distributed**:- Java is Distributed Language Means because the program of java is compiled onto one machine can be easily transferred to machine and Executes them on another machine because facility of Bytes Codes So java is Specially designed For Internet Users which uses the Remote Computers For Executing their Programs on local machine after transferring the Programs from Remote Computers or either from the internet.

6) **Simple Small and Familiar**:- is a simple Language Because it contains many features of other Languages like c and C++ and Java Removes Complexity because it doesn't use pointers, Storage Classes and Go to Statements and java Doesn't support Multiple Inheritance

7) **Multithreaded and Interactive:-** Java uses Multithreaded Techniques For Execution Means Like in other in Structure Languages Code is Divided into the Small Parts Like These Code of java is divided into the Smaller parts those are Executed by java in Sequence and Timing Manner this is Called as Multithreaded In this Program of java is divided into the Small parts those are Executed by Compiler of java itself Java is Called as Interactive because Code of java Supports Also CUI and Also GUI Programs

8) **Dynamic and Extensible Code**:- Java has Dynamic and Extensible Code Means With the Help of OOPS java Provides Inheritance and With the Help of Inheritance we Reuse the Code that is Pre-defined and Also uses all the built in Functions of java and Classes

9) **Secure:** Java was designed with security in mind. As Java is intended to be used in networked/distributor environments so it implements several security mechanisms to protect you against malicious code that might try to invade your file system.
For example: The absence of pointers in Java makes it impossible for applications to gain access to memory locations without proper authorization as memory allocation and referencing model is completely opaque to the programmer and controlled entirely by the underlying run-time platform .

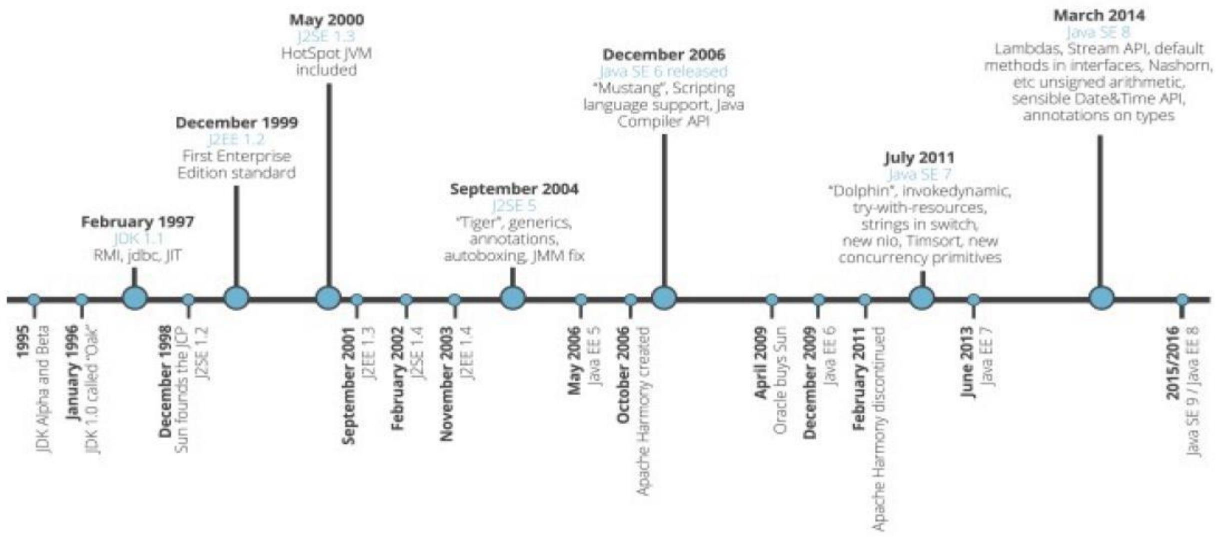
10) **Architectural Neutral**: One of the key feature of Java that makes it different from other programming languages is architectural neutral (or platform independent). This means that the programs written on one platform can run on any other platform without having to rewrite or recompile them. In other words, it follows 'Write-once-run-anywhere' approach.
Java programs are compiled into byte-code format which does not depend on any machine architecture but can be easily translated into a specific machine by a Java Virtual Machine (JVM) for that machine. This is a significant advantage when developing applets or applications that are downloaded from the Internet and are needed to run on different systems.

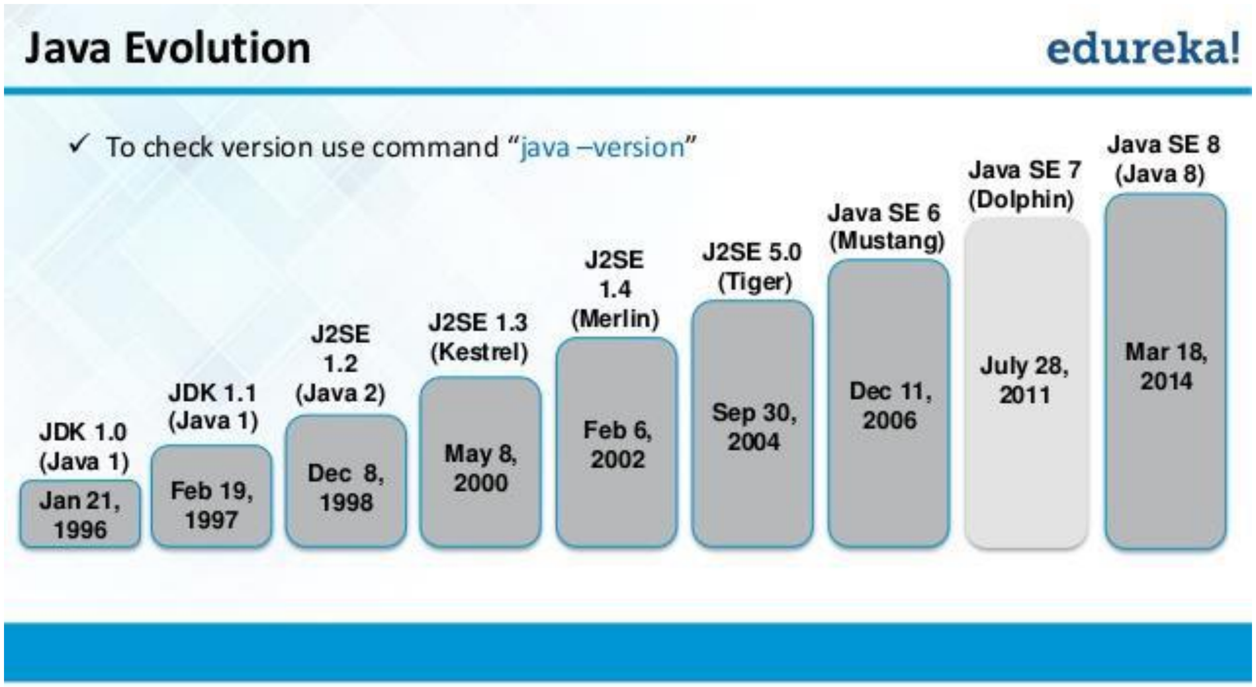
11) **Portable :** The portability actually comes from architecture-neutrality. In C/C++, source code may run slightly differently on different hardware platforms because of how these platforms implement arithmetic operations. In Java, it has been simplified.
Unlike C/C++, in Java the size of the primitive data types are machine independent. For example, an int in Java is always a 32-bit integer, and float is always a 32-bit IEEE 754 floating point number. These consistencies make Java programs portable among different platforms such as Windows, Unix and Mac .

12) **Interpreted** : Unlike most of the programming languages which are either complied or interpreted, Java is both complied and interpreted The Java compiler translates a java source file to bytecodes and the Java interpreter executes the translated byte codes directly on the system that implements the Java Virtual Machine. These two steps of compilation and interpretation allow extensive code checking and improved security .

13) **High performance:** Java programs are complied to portable intermediate form know as bytecodes, rather than to native machine level instructions and JVM executes Java bytecode on. Any machine on which it is installed. This architecture means that Java programs are faster than program or scripts written in purely interpreted languages but slower than C and C++ programsthat compiled to native machine languages.

The Evolution of Java:





<u>An Overview of Java:</u>

<u>Object Oriented Concepts:/The Key Attributes Of Object Oriented Programming</u>

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

6.Encapsulation
7.Dynamic Binding
8.Message passing

1. Class

- A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class.
- A class can contain fields and methods to describe the behavior of an object.
- Methods are nothing but members of a class that provide a service for an object or perform some business logic. Java fields and member functions names are case sensitive. Current states of a class's corresponding object are stored in the object's instance variables. Methods define the operations that can be performed in java programming.

2. Object

- An object is an instance of a class. The relationship is such that many objects can be created using one class. Each object has its own data but its underlying structure (i.e., the type of data it stores, its behaviors) are defined by the class.

Eg:-
Book HitchHiker = new Book("Douglas Adams");
Book ShortHistory = new Book("Bill Bryson", "Black Swan");
Book IceStation = new Book("Alistair MacLean", "Collins");

In the above Example Book is class and here we are creating objects for class by using new operator.

3. Abstraction

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.
- An *abstract class* is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself *must* be declared `abstract`, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
```

```
    abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared `abstract`.

4. Encapsulation

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class. However if we setup public getter and setter methods to update (for e.g. `void setSSN(int ssn)`)and read (for e.g. `int getSSN()`) the private data fields then the outside class can access those private data fields via public methods. This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That’s why encapsulation is known as **data hiding.**

- It improves maintainability and flexibility and re-usability
- The fields can be made read-only or write-only
- User would not be knowing what is going on behind the scene.

5.polymorphism

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

- **If you overload static method in java, it is the example of compile time polymorphism.**

Runtime Polymorphism in Java

- Runtime polymorphism **or** Dynamic Method Dispatch **is a process in which a call to an overridden method is resolved at runtime rather than compile-time.**

Example:

```
class Bike{
  void run(){System.out.println("running");}
}
class Splender extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splender();//upcasting
```

```
     b.run();
  }
}
```

6. Inheritance:

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.In java programming, multiple and hybrid inheritance is supported through **interface** only.

1.Single inheritance is easy to understand. When a class extends another one class only then we call it a single inheritance.

```
class A
{
   public void methodA()
   {
     System.out.println("Base class method");
   }
}

Class B extends A
{
   public void methodB()
   {
     System.out.println("Child class method");
   }
   public static void main(String args[])
   {
     B obj = new B();
     obj.methodA(); //calling super class method
     obj.methodB(); //calling local method
  }
}
```

2.Mutiple Inheritance

Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.

3.Multilevel Inheritance:

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.

```
Class X
{
   public void methodX()
   {
     System.out.println("Class X method");
   }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
   public void methodZ()
   {
     System.out.println("class Z method");
   }
   public static void main(String args[])
   {
     Z obj = new Z();
     obj.methodX(); //calling grand parent class method
     obj.methodY(); //calling parent class method
     obj.methodZ(); //calling local method
  }
}
```

4.Hierarchical Inheritance:

such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.

Exanmple:

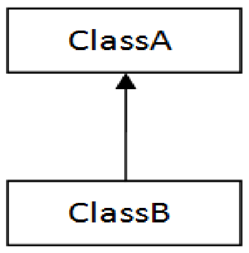
```
Class A
{
  public void methodA()
  {
     System.out.println("method of Class A");
  }
}
Class B extends A
{
  public void methodB()
  {
     System.out.println("method of Class B");
  }
}
Class C extends A
{
 public void methodC()
 {
 System.out.println("method of Class C");
 }
}
Class D extends A
{
  public void methodD()
  {
     System.out.println("method of Class D");
  }
}
Class MyClass
{
  public void methodB()
  {
     System.out.println("method of Class B");
  }
  public static void main(String args[])
  {
     B obj1 = new B();
     C obj2 = new C();
     D obj3 = new D();
     obj1.methodA();
     obj2.methodA();
     obj3.methodA();
  }
```

5.Hybrid Inheritance

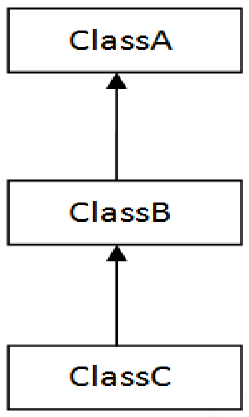
In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.

Example:

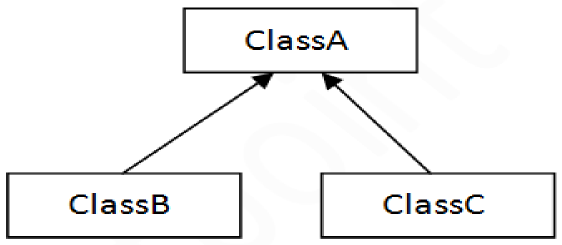
```
interface A
{
     public void methodA();
}
interface B extends A
{
     public void methodB();
}
interface C extends A
{
     public void methodC();
}
class D implements B, C
{
    public void methodA()
    {
         System.out.println("MethodA");
    }
    public void methodB()
    {
         System.out.println("MethodB");
    }
    public void methodC()
    {
         System.out.println("MethodC");
    }
    public static void main(String args[])
    {
         D obj1= new D();
         obj1.methodA();
         obj1.methodB();
         obj1.methodC();
     }
}
```



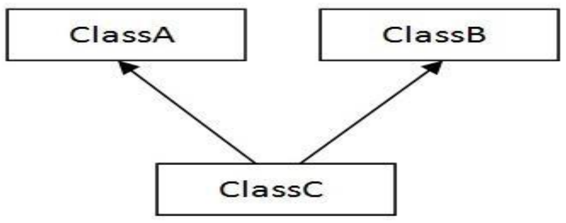
1) Single



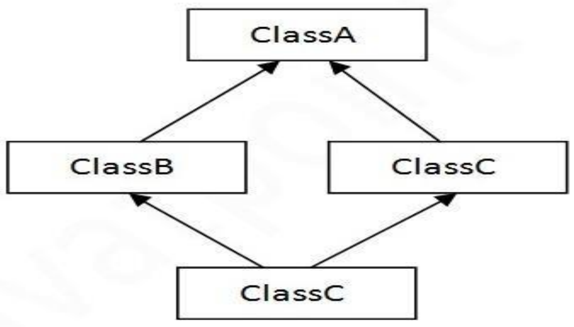
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

7.Dynamic Binding

Association of method definition to the method call is known as binding. There are two types of binding: Static binding and dynamic binding.
Static Binding
The binding which can be resolved at compile time by compiler is known as static or early binding. All the static, private and final methods have always been bonded at compile-time.It resolve the problems related methodoverloading.

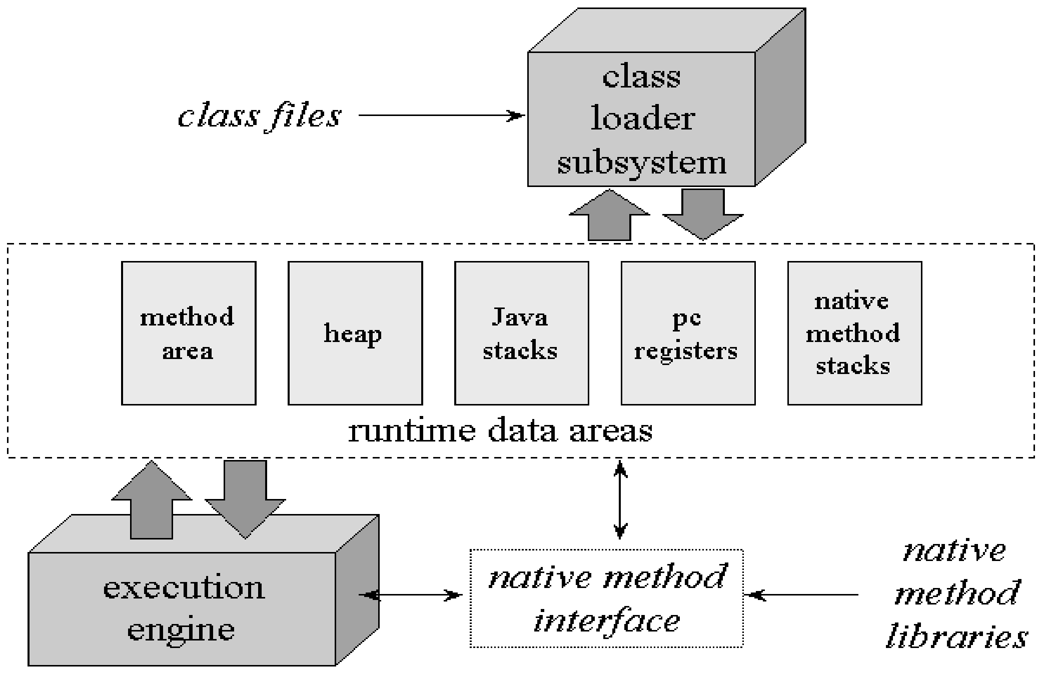
Dynamic Binding
When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method. Thus while calling the overridden method, the compiler gets confused between parent and child class method(since both the methods have same name).
Difference between static and dynamic binding

difference between static and dynamic binding in Java.

1. Static binding happens at compile-time while dynamic binding happens at runtime.
2. Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. Binding of overridden methods happen at runtime.
3. Java uses static binding for overloaded methods and dynamic binding for overridden methods.

8.Message passing
Message passing is a method by which an object sends data to another object or requests other object to invoke method. This is also known as interfacing.It acts like a messenger from one object to other object to convey specific instructions. Message Passing involves specifying the name of objects, the name of the function, and the information to be sent.

<u>JVM ARCHITECTURE</u>



1.Class loader sub system: JVM's class loader sub system performs 3 tasks

a. It loads .class file into memory.

b. It verifies byte code instructions.

c. It allots memory required for the program.

2. Run time data area: This is the memory resource used by JVM and it is divided into 5 parts

a. Method area: Method area stores class code and method code.

b. Heap: Objects are created on heap.

c. Java stacks: Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.

d. Program counter registers: The program counter registers store memory address of the instruction to be executed by the micro processor.

e. Native method stacks: The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

3. Native method interface: Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

4. Native method library: holds the native libraries information.

4. **Execution engine:** Execution engine contains interpreter and JIT compiler, which covert byte code into machine code. JVM uses optimization technique to decide which part to be

interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler.

Java Runtime Environment (JRE)

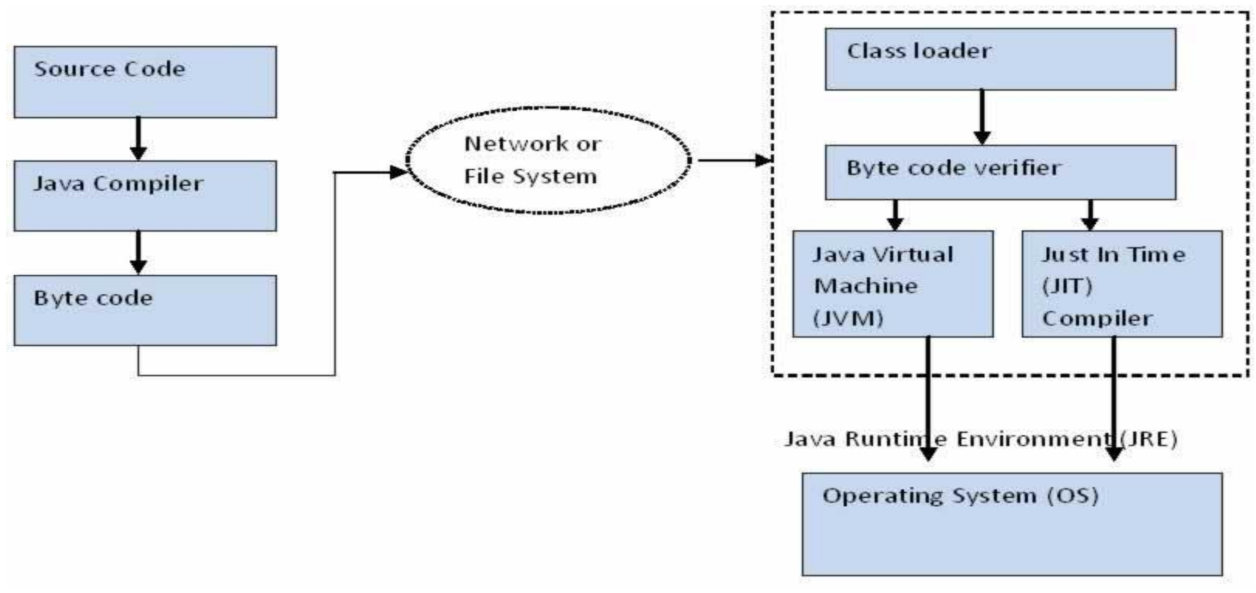
Java Runtime Environment contains JVM, class libraries and other supporting components. As you know the Java source code is compiled into bytecode by Java compiler. This bytecode will be stored in class files. During runtime, this bytecode will be loaded, verified and JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

1. Loads the class ∷This is done by the class loader

2. Verifies the bytecode ∷This is done by bytecode verifier.

3. Interprets the bytecode ∷This is done by the JVM



Class loader

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the bytecode is loaded successfully, then next step is bytecode verification by bytecode verifier.

Byte code verifier

The bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

1. The code follows JVM specifications.
2. There is no unauthorized access to memory.
3. The code does not cause any stack overflows.

4. There are no illegal data conversions in the code such as float to object references.
Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs.

Just in Time Compiler

You might have noticed the component "Just in Time" (JIT) compiler in Figure 3. This is a component which helps the program execution to happen faster.

As we discussed earlier when the Java program is executed, the byte code is interpreted by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster.

If the JIT Compiler library exists, when a particular bytecode is executed first time, JIT complier compiles it into native machine code which can be directly executed by the machine in which the Java program runs. Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser. This compilation happens when the byte code is about to be executed and hence the name "Just in Time".

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

Simple program

To create a simple java program, you need to create a class that contains main method.

Requirement for Java Program

- install the JDK if you don't have installed it,
- set path of the jdk/bin directory.

  Eg: computer->properties->advanced->environmentvariables->new ->

  variable name:path

  variablevalue: C:\Program Files\Java\jdk1.6.0_26\bin

  below->new->

  variable name:classpath

  variablevalue: C:\Program Files\Java\jdk1.6.0_26\bin;

  and then press "ok"

- create the java program in **notepad** and save **as “classname.java”**

  eg:-Hai.java

- compile and run the java program

  To compile java program

  C:\>javac Hai.java

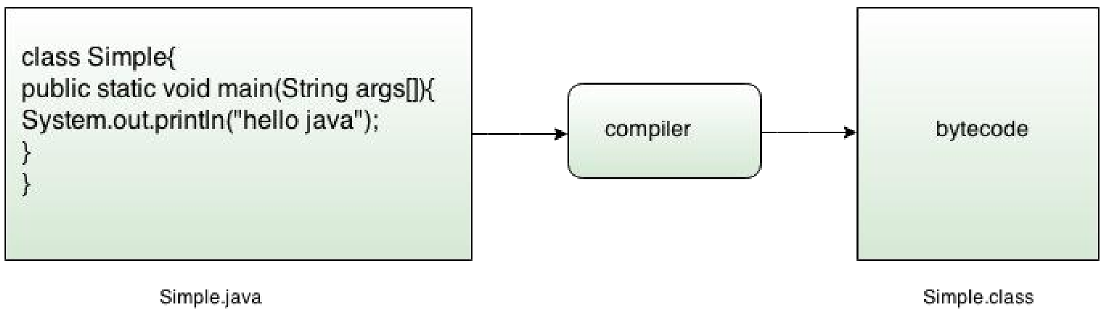
  To run program

  C:\>java Hai

Program:

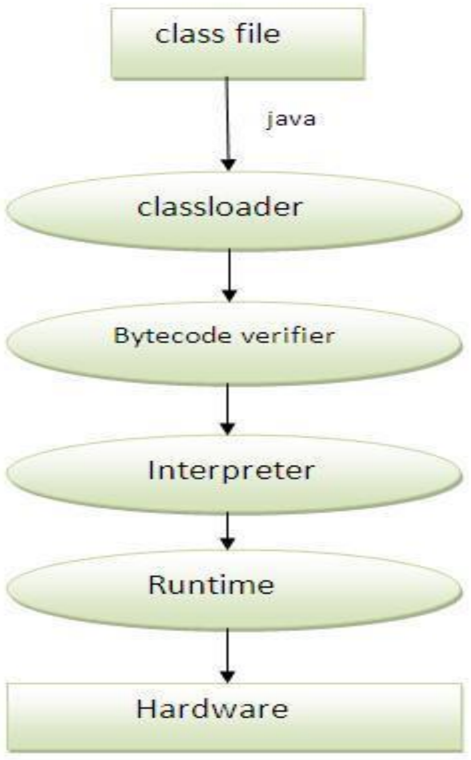
```
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

<u>Compile Time:</u>

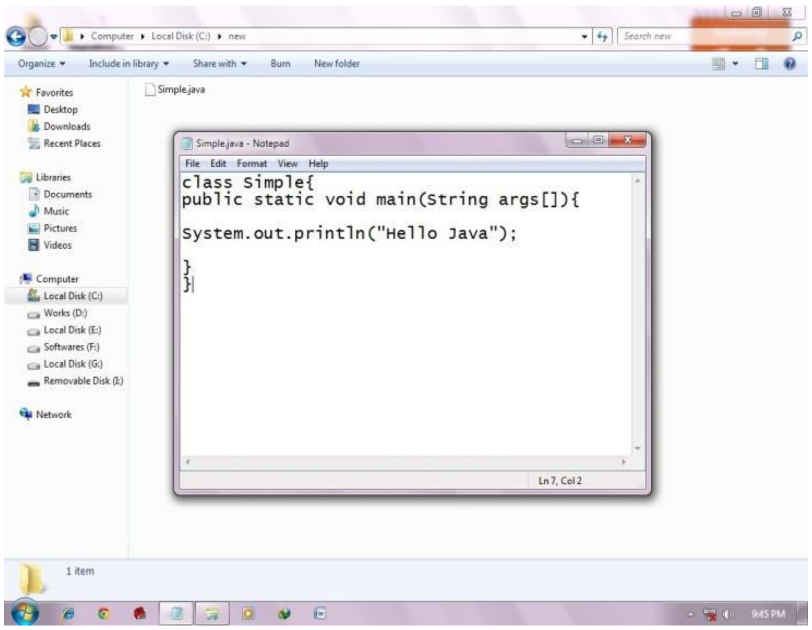


```
class Simple{
public static void main(String args[]){
System.out.println("hello java");
}
}
```

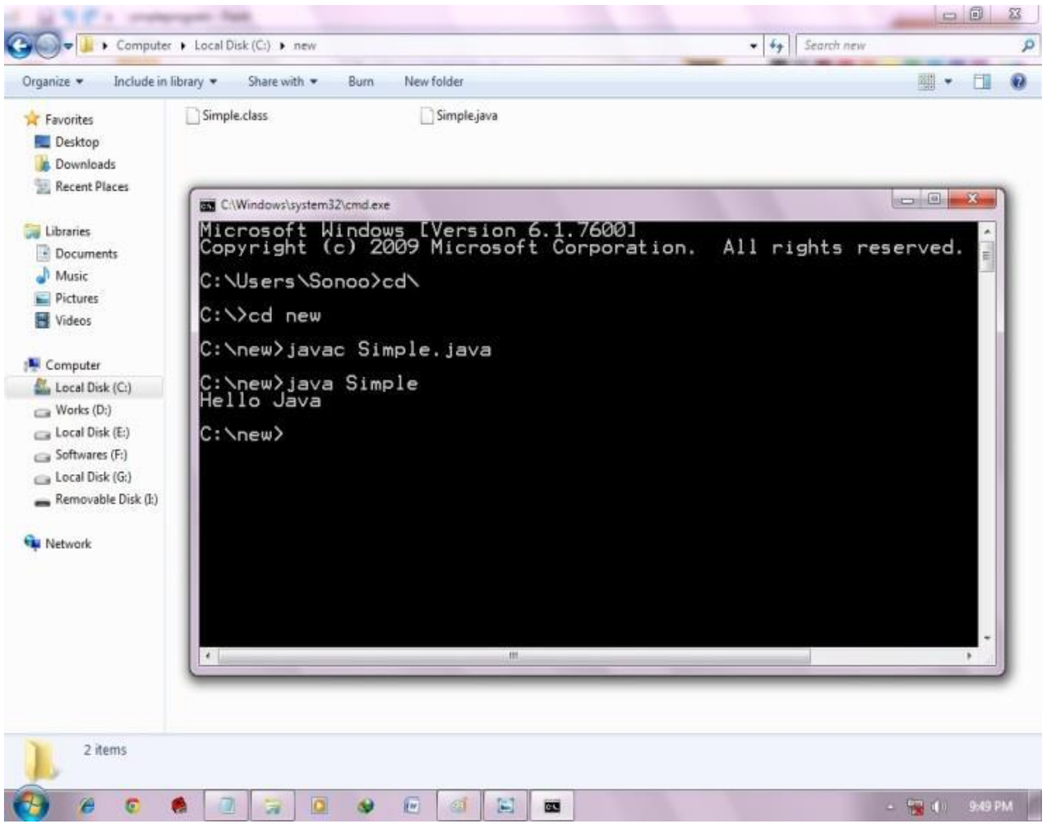
<u>Run Time:</u>



- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create 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 startup of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.



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

System.out.println("Hello Java");

}
}
```



```
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>
```

A Second Short Program

```
/*
Here is another short example.
Call this file "Example2.java".
*/
class Example2 {
public static void main(String args []) {
int num; // this declares a variable called num
num = 100; // this assigns num the value 100
System.out.println("This is num: " + num);
num = num * 2;
System.out.print("The value of num * 2 is ");
System.out.println(num);
}
}
```

When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

Two Control Statements

The if Statement

The Java **if** statement works much like the IF statement in any other language. Further, it is syntactically identical to the **if** statements in C, C++, and C#. Its simplest form is shown here:

if*(condition) statement;*

Here, *condition* is a Boolean expression. If *condition* is true, then the statement is executed.If *condition* is false.

Operator	**Meaning**
<	Less than
>	Greater than
==	Equal to

Here is a program that illustrates the **if** statement:

```
/*
Demonstrate the if.
Call this file "IfSample.java".
*/
class IfSample {
public static void main(String args[]) {
int x, y;
x = 10;
y = 20;
if(x < y) System.out.println("x is less than y");
x = x * 2;
if(x == y) System.out.println("x now equal to y");
x = x * 2;
if(x > y) System.out.println("x now greater than y");
// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}
```
The output generated by this program is shown here:
```
x is less than y
x now equal to y
x now greater than y
```
The for Loop:

for*(initialization; condition; iteration) statement;*

In its most common form, the *initialization* portion of the loop sets a loop control variable to an initial value. The *condition* is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the **for** loop continues to iterate. If it is false, the loop terminates. The *iteration* expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the **for** loop:
```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
class ForTest {
public static void main(String args[]) {
int x;
for(x = 0; x<10; x = x+1)
System.out.println("This is x: " + x);
}
}
```
This program generates the following output:
```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```
Using blocks of codes

Java allows two or more statements to be grouped into *blocks of code*, also called *code blocks*.This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement.

Consider this **if** statement
```
if(x < y) { // begin a block
x = y;
y = 0;
} // end of block
```
The following program uses a block of code as the target of a **for** loop.

```
/*
Demonstrate a block of code.
Call this file "BlockTest.java"
*/
class BlockTest {
public static void main(String args[]) {
int x, y;
y = 20;
// the target of this loop is a block
for(x = 0; x<10; x++) {
System.out.println("This is x: " + x);
System.out.println("This is y: " + y);
y = y - 2;
}
}
}
```

The output generated by this program is shown here:

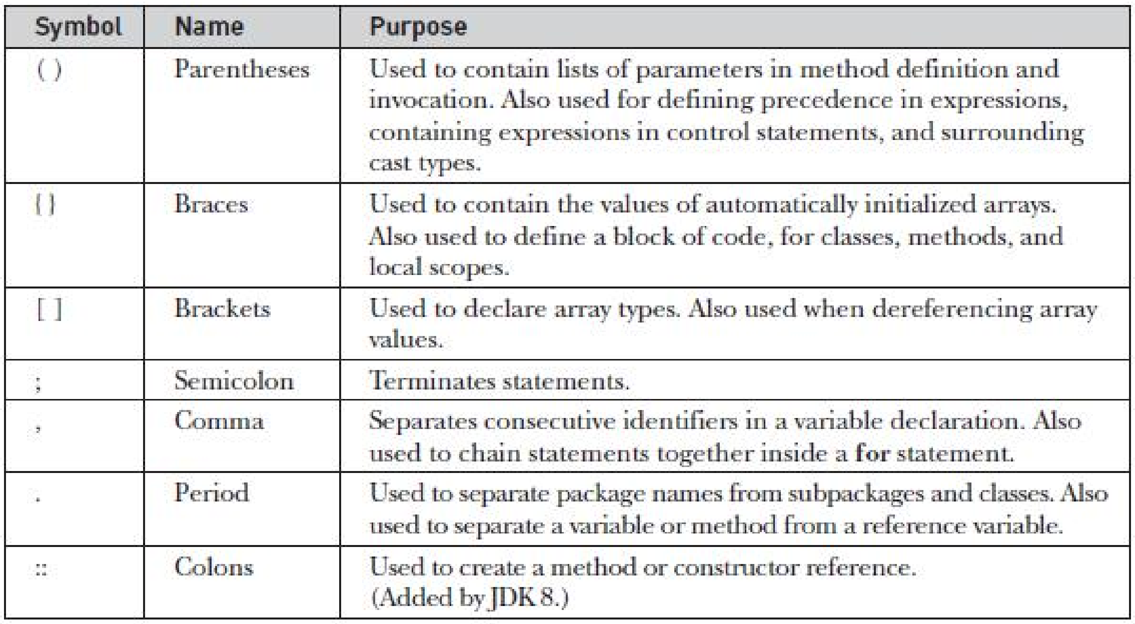
```
This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

Lexical Issues: Java programs are a collection of whitespace, identifiers, literals,comments, operators, separators, and keywords.

Whitespace

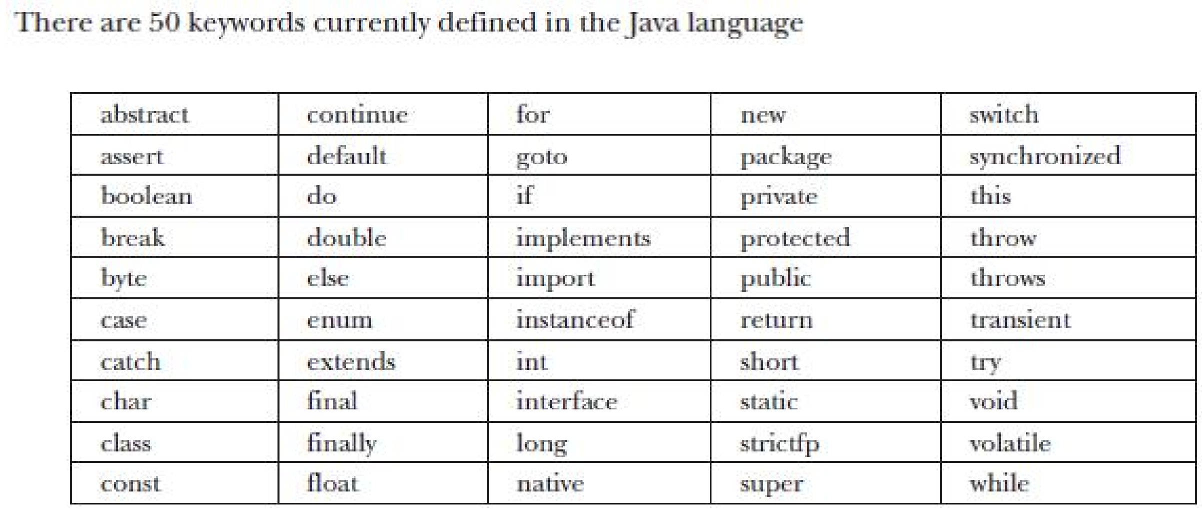
Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers
Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) 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
1.AvgTemp 2. count 3.a4 4. $test 5.this_is_ok
Invalid identifier names include these:
1.2count 2. high-temp 3. Not/ok
Literals
A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:
1.100 2.98.6 3.'X' 4."This is a test"
Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.
Comments
As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */.
Separators
In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table.



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

Java Keywords



There are 50 keywords currently defined in the Java language

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

Access modifiers:	private, protected, public
Class, method , variable modifiers:	abstract, class, extends, final, implements, interface, native, new, static, strictfp, synchronized, transient, volatile
Flow control:	break, case, continue, default, do, else, for, if, instance of, return, switch, while
Package control:	import, package
Primitive types:	boolean, byte, char, double, float, int, long, short
Error handling:	assert, catch, finally, throw, throws, try
Enumeration:	enum
Others:	super, this, void
Unused:	const, goto

Java class libraries:

Two of Java's built-in methods:
println( ) and print( ). As mentioned, these methods are available through System.out. System is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI).

List of Library Classes in Java:

Library classes	Purpose of the class
Java.io	Use for input and output functions.
Java.lang	Use for character and string operation.
Java.awt	Use for windows interface.
Java.util	Use for develop utility programming.
Java.applet	Use for applet.
Java.net	Used for network communication.
Java.math	Used for various mathematical calculations like power, square root etc.

We will come across different library classes, which basically deal with input/output operation.

Data Types, Arrays and Variables:

Primitive types: boolean, byte, char, double, float, int, long, short

• **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.

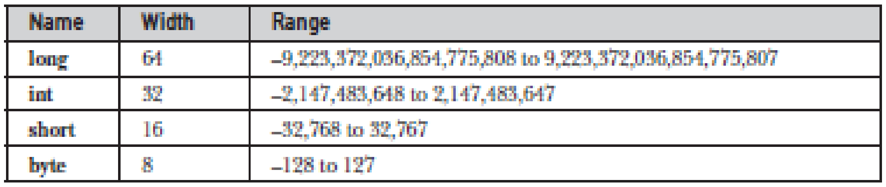
• **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.

• **Characters** This group includes **char**, which represents symbols in a character set,like letters and numbers.

• **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

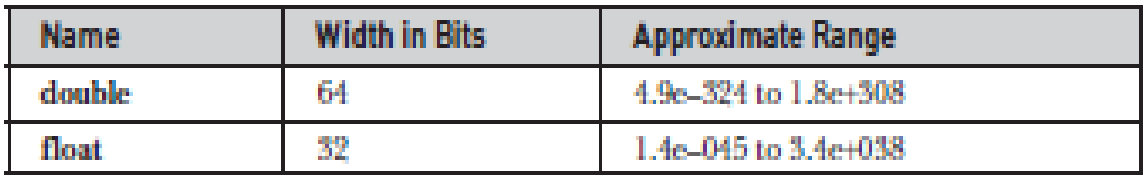


Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you're working with a stream of data from a network or file.

Ex:
declares two **byte** variables called **b** and **c**:
byte b, c;
<u>short</u>
short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least used Java type. Here are some examples of **short** variable declarations:
short s;
short t;
<u>int</u>
The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
Ex:-int lightspeed;
<u>long</u>
long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value.
Ex:- long distance;
<u>Floating Point:</u>
There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively.



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

<u>float</u>
The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision.
Ex:-float hightemp, lowtemp;
<u>double</u>
Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision
ex:-double pi, r, a;
<u>Characters</u>
In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses *Unicode* to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.
Unicode required 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always.
<u>Ex:-</u> char ch1, ch2;
Boolean:
<u>Booleans</u>
Java has a primitive type, called **boolean**, for logical values. It can have only one of two

possible values, **true** or **false**.
Ex-boolean b;
Example:

```
public class PrimitiveDemo {
public static void main(String[] args) {
byte b =100;
short s =123;
int v = 123543;
int calc = -9876345;
long amountVal = 1234567891;
float intrestRate = 12.25f;
double sineVal = 12345.234d;
boolean flag = true;
boolean val = false;
char ch1 = 88; // code for X
char ch2 = 'Y';
System.out.println("byte Value = "+ b);
System.out.println("short Value = "+ s);
System.out.println("int Value = "+ v);
System.out.println("int second Value = "+ calc);
System.out.println("long Value = "+ amountVal);
System.out.println("float Value = "+ intrestRate);
System.out.println("double Value = "+ sineVal);
System.out.println("boolean Value = "+ flag);
System.out.println("boolean Value = "+ val);
System.out.println("char Value = "+ ch1);
System.out.println("char Value = "+ ch2);
}
  }
```

Summary of Data Types

Primitive Type	**Size**	**Minimum Value**	**Maximum Value**	**Wrapper Type**
char	16-bit	Unicode 0	Unicode 216-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-215 (-32,768)	+215-1 (32,767)	Short
int	32-bit	-231 (-2,147,483,648)	+231-1 (2,147,483,647)	Integer

long	64-bit	-263 (-9,223,372,036,854,775,808)	+263-1 (9,223,372,036,854,775,807)	Long
float	32-bit	Approx range 1.4e-045 to 3.4e+038		Float
double	64-bit	Approx range 4.9e-324 to 1.8e+308		Double
boolean	1-bit	true or false		Boolean

Literals:

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Java language specifies five major types of literals. Literals can be any number, text, or other information that represents a value.

- **Integer literals**
- **Floating literals**
- **Character literals**
- **String literals**
- **Boolean literals**

Each of them has a type associated with it. The type describes how the values behave and how they are stored.

Integer literals:

Integer data types consist of the following primitive data types: int,long, byte, and short. 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.
Examples:

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

Floating-point literals:

Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix. However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float. We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as:

0.0314 *10^2 (i.e 3.14).
6.5E+32 (or 6.5E32) Double-precision floating-point literal
7D Double-precision floating-point literal
.01f Floating-point literal

Character literals:
char data type is a single 16-bit Unicode character. We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'. You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc. Below table shows a set of these special characters.

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\d	Octal
\xd	Hexadecimal
\ud	Unicode character

If we want to specify a single quote, a backslash, or a non-printable character as a character literal use an escape sequence. An escape sequence uses a special syntax to represents a character. The syntax begins with a single backslash character. You can see the below table to view the character literals use Unicode escape sequence to represent printable and non-printable characters.

'u0041'	Capital letter A
'\u0030'	Digit 0
'\u0022'	Double quote "
'\u003b'	Punctuation ;
'\u0020'	Space
'\u0009'	Horizontal Tab

String Literals:
The set of characters in represented as String literals in Java. Always use "double quotes" for String literals. There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same values.

""	The empty string
"\""	A string containing

"This is a string"	A string containing 16 characters
"This is a " + "two-line string"	actually a string-valued constant expression, formed from two string literals

Null Literals
The final literal that we can use in Java programming is a Null literal. We specify the Null literal in the source code as 'null'. To reduce the number of references to an object, use null literal. The type of the null literal is always null. We typically assign null literals to object reference variables. For instance
s = null;

Boolean Literals:
The values true and false are treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value.

Example
boolean chosen = true;

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 optional initialize.

Declaring a Variable

type identifier [= *value*][, *identifier* [= *value*] …];
type is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value.
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'.
The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization
Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:
// Demonstrate dynamic initialization.
class DynInit {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized

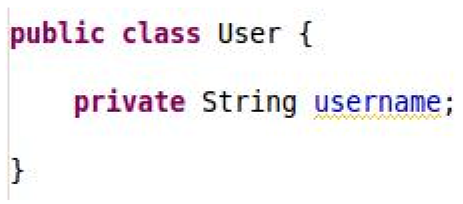
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}

Scope of the Variable:

Scope and Lifetime of **Variables**. The **scope** of a **variable** defines the section of the code in which the **variable** is visible. As a general rule, **variables** that are defined within a block are not accessible outside that block. The lifetime of a **variable** refers to how long the **variable** exists before it is destroyed.

Class Level Scope

In Java, there are some variables that you want to be able to access from anywhere within a Java class. The scope of these variables will need to be at the class level, and there is only one way to create variables at that level – just inside the class but outside of any methods. Let's take a look at an example:

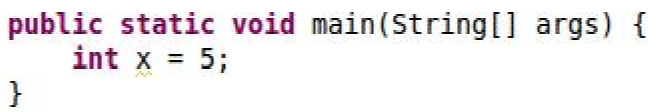


```java
public class User {

    private String username;

}
```

You will notice that the variables have been defined at the top of the class, before any methods. This is simply a convention; you can define your class-level variables anywhere in the class (so long as it's outside of any methods in the class). However, it is good practice to put these variables at the top so it's easier to see where they all are. Also note that the access identifier does not have anything to do with the variable's scope within the class! If you're not sure what access identifiers are, I strongly recommend checking out our tutorial about them by clicking here.

Method Scope

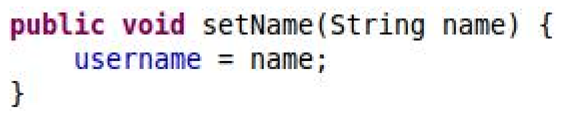
Some variables you might want to make temporary and preferably they are used for only one method. This would be an example of method scope. Here's a pretty typical example of a variable in method scope using an example of Java's main method:



```java
public static void main(String[] args) {
    int x = 5;
}
```

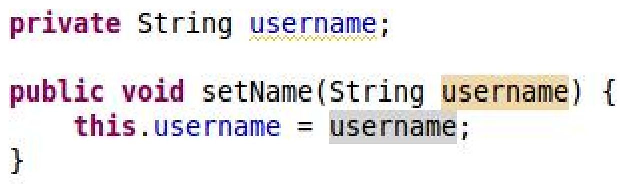
The variable x in the example is created inside the method. When the method ends, the variable reference goes away and there is no way to access that variable any longer. You cannot use that same variable x in another method because it only exists in the main method's scope and that is it.

Here's another example of method scope, except this time the variable got passed in as a parameter to the method:



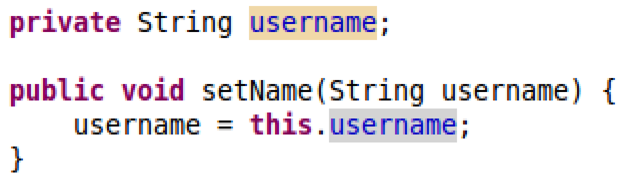
```
public void setName(String name) {
    username = name;
}
```

The above example is the typical example of a setter method. The purpose of a setter method as you might recall is to set a class variable to a particular value from somewhere outside of the class. The above example is a pretty clean example of this. Now let's look at the same example, except now we'll use a conflicting variable name.



```
private String username;

public void setName(String username) {
    this.username = username;
}
```

So, what's going on with the variable scope here? And what is "this"? The this keyword helps Java to differentiate between the local scope variable (the method scope variable) and the class variable. This.username tells Java that you are referencing the class variable. So, the setter above sets the method-scope variable called name to the class variable called name. The variable scope here is not in conflict because Java knows which variable to access because of the "this" keyword. You could also do this:



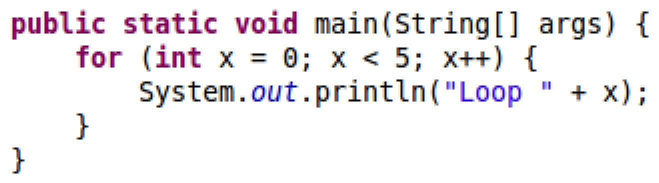
```
private String username;

public void setName(String username) {
    username = this.username;
}
```

However the above is something you probably won't be doing often (if at all!) because you're setting the local, temporary variable to the value of the more permanent one.

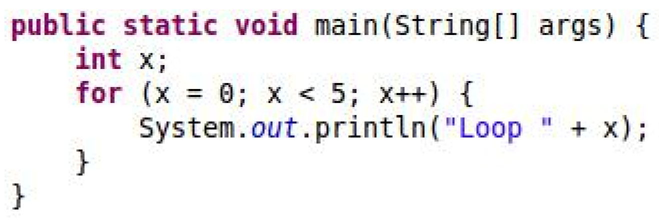
Loop Scope

Have you ever had trouble accessing a variable when dealing with for loops and while loops? Variable scope plays a big role in your difficulty, so let's figure out what's going on there.

Any variables created inside of a loop are **LOCAL TO THE LOOP**. This means that once you exit the loop, the variable can no longer be accessed! This includes any variables created in the loop signature. Take a look at the following two for loop examples:



```
public static void main(String[] args) {
    for (int x = 0; x < 5; x++) {
        System.out.println("Loop " + x);
    }
}
```



```
public static void main(String[] args) {
    int x;
    for (x = 0; x < 5; x++) {
        System.out.println("Loop " + x);
    }
}
```

In the first example, x can ONLY be used inside of the for loop. In the second example, you are free to use x inside of the loop as well as outside of the loop because it was declared outside of the loop (it has been declared at method scope).

Type Conversion and Casting

Assigning a value of one type to a variable of another type is known as **Type Casting**.

Java supports two types of castings – **primitive data type casting** and **reference type casting**. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting. Now let us go for data type casting.

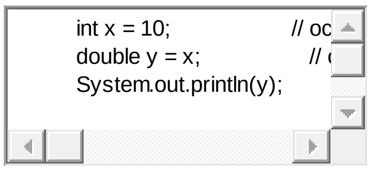
Java data type casting comes with 3 flavors.

1. **Implicit casting**
2. **Explicit casting**
3. **Boolean casting.**

1. automatic type conversion.: **Implicit casting (widening conversion)**

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as **automatic type conversion**.

Examples:



```
int x = 10;                // oc
double y = x;              // (
System.out.println(y);
```

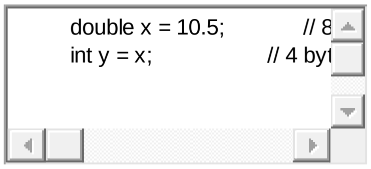
1 int x = 10; // occupies 4 bytes

```
double y = x;                 // occupies 8 bytes

System.out.println(y);        // prints 10.0
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

2. Casting Incompatible Types: Explicit casting (narrowing conversion)

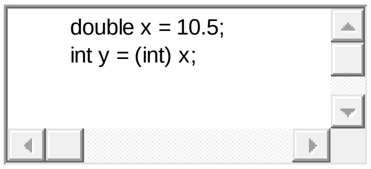
A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires **explicit casting**; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.



```
double x = 10.5;          // 8
int y = x;                // 4 byt
```

```
double x = 10.5;            // 8 bytes

int y = x;                  // 4 bytes ;  raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.



```
double x = 10.5;
int y = (int) x;
```

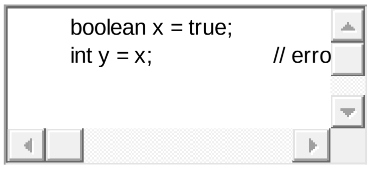
```
double x = 10.5;

int y = (int) x;
```

The double **x** is explicitly converted to int **y**. The thumb rule is, on both sides, the same data type should exist.

3. Boolean casting

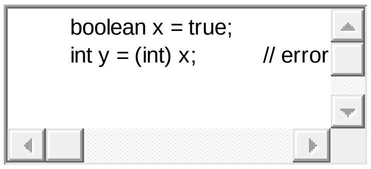
A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is **incompatible** for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.



```
boolean x = true;
int y = x;                // erro
```

```
boolean x = true;
int y = x;                    // error
```



```
boolean x = true;
int y = (int) x;          // error
```

```
boolean x = true;
int y = (int) x;          // error
```

byte –> **short** –> **int** –> **long** –> **float** –> **double**

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, **byte** can be assigned to **short** implicitly but **short** to **byte** requires explicit casting.

Automatic Type Promotion in Expressions
In addition to assignments, there is another place where certain type conversions may occur: in expressions. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For
example, examine the following expression:
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a*b** is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.
As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte**

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

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

which yields the correct value of 100.

The Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows: First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands are **double**, the result is **double**.

```
class Promote {

public static void main(String args[]) {

byte b = 42;

char c = 'a';

short s = 1024;

int i = 50000;

float f = 5.67f;

double d = .1234;

double result = (f * b) + (i / c) - (d * s);

System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));

System.out.println("result = " + result);

}

}
```

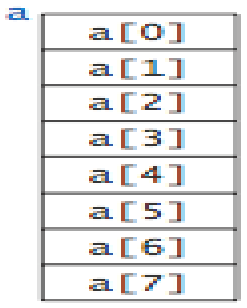
Let’s look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, **f * b, b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i/c, c** is promoted to **int**, and the result is of type **int**. Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double.**

Arrays

An array stores a sequence of values that are all of the same type. We want not just to store values but also to be able to quickly access each individual value. The method that we use to refer to individual values in an array is to number and then *index* them—if we have N values, we think of them as being numbered from 0 to N-1.



An array

Each item in an array is called an *element*, and each element is accessed by its numerical *index*.

- **Array** is a data structure in java that can hold one or more values in a single variable.
- Array in java is a collection of similar type of values.
- Java has two types of arrays – single dimensional and multidimensional arrays.
- Array index starts at 0.

How to declare an array

ArrayDataType[] ArrayName;

OR

ArrayDataType ArrayName[];

Where ArrayDataType defines the data type of array element like int, double etc.

In java, initialize an array can be done by using **new** keyword as well:

int arrayName = new int[10];

int[]ArrList = {1, 2, 3, 4,5};

The above array is five elements length.

Eg:-

```
public class array_ex {
    public static void main(String []args) {
        int arrex[] = {10,20,30}; //Declaring and initializing an array of three elements
       for (int i=0;i&lt;arrex.length;i++){
             System.out.println(arrex[i]);
          }
    }
}
```

(or)

```
public class array_ex {
    public static void main(String []args) {
        int arrex[] = new int[3]; //declaring array of three items
        arrex[0] =10;
        arrex[1] =20;
        arrex[2] =30;
        //Printing array
        for (int i=0;i&lt;arrex.length;i++){
            System.out.println(arrex[i]);
          }
    }
}
```

String array in Java

```
public class array_ex {
    public static void main(String []args) {
        String strState[] = new String[3]; //declaring array of three items
        strState[0] ="New York";
        strState[1] ="Texas";
        strState[2] ="Florida";
        //Printing array
        for (int i=0;i&lt;strState.length;i++){
            System.out.println(strState[i]);
          }
    }
}
```

Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Definition of One Dimensional Array

- One dimensional array is a list of variables of same type that are accessed by a common name. An individual variable in the array is called an array element. Arrays forms a way to handle groups of related data.

Eg:-

int account_numbers[] = new int[10];

eg:-

```
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
```

Multidimensional Arrays

The array we used in the last example was a *one dimensional* array. Arrays can have more than one dimension, these arrays-of-arrays are called *multidimensional arrays*. They are very similar to standard arrays with the exception that they have multiple sets of square brackets after the array identifier. A two dimensional array can be though of as a grid of rows and columns.

```
int twoD[][] = new int[4][5];
```

```
eg:-
```

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

Declaration of allocating memory to multi dimensional array
int iarr[][] = new int[2][3];

// **Initializing elements**
iarr[0][0] = 1;
iarr[0][1] = 2;
iarr[0][2] = 3;
iarr[1][0] = 4;
iarr[1][1] = 5;
iarr[1][2] = 6;

//**Display array elements**
System.out.println(iarr[0][0]);
System.out.println(iarr[0][1]);
System.out.println(iarr[0][2]);
System.out.println(iarr[1][0]);
System.out.println(iarr[1][1]);
System.out.println(iarr[1][2]);

// **Or Use for loop to display elements**
for (int i = 0; i < iarr.length; i = i + 1)

```
{
    for(int j=0; j < iarr[i].length; j = j + 1)
   {
        System.out.print(iarr[i][j]);
        System.out.print(" ");
    }
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] *var-name*;

Here, the square brackets follow the type specifier, and not the name of the array variable.
For example, the following two declarations are equivalent:

int al[] = new int[3];
int[] a2 = new int[3];

The following declarations are also equivalent:

char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

int[] nums, nums2, nums3; // create three arrays

creates three array variables of type **int**. It is the same as writing

int nums[], nums2[], nums3[]; // create three arrays.

String:

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are object. The String class implements immutable character strings, which are read-only once the string object has been created and initialized. All string literals in Java programs, are implemented as instances of String class.

The easiest way to create a Java String object is using a string literal:

1. String str1 = "I can't be changed once created!";

Creating Strings

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

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

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

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

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

Pointers:

Java cannot allow pointers, because doing so would allow Java programs to breach the firewall between the Java execution environment and the host computer. (Remember, a pointer can be given any address in memory—even addresses that might be outside the Java run-time system.) Since C/C++ make extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one.