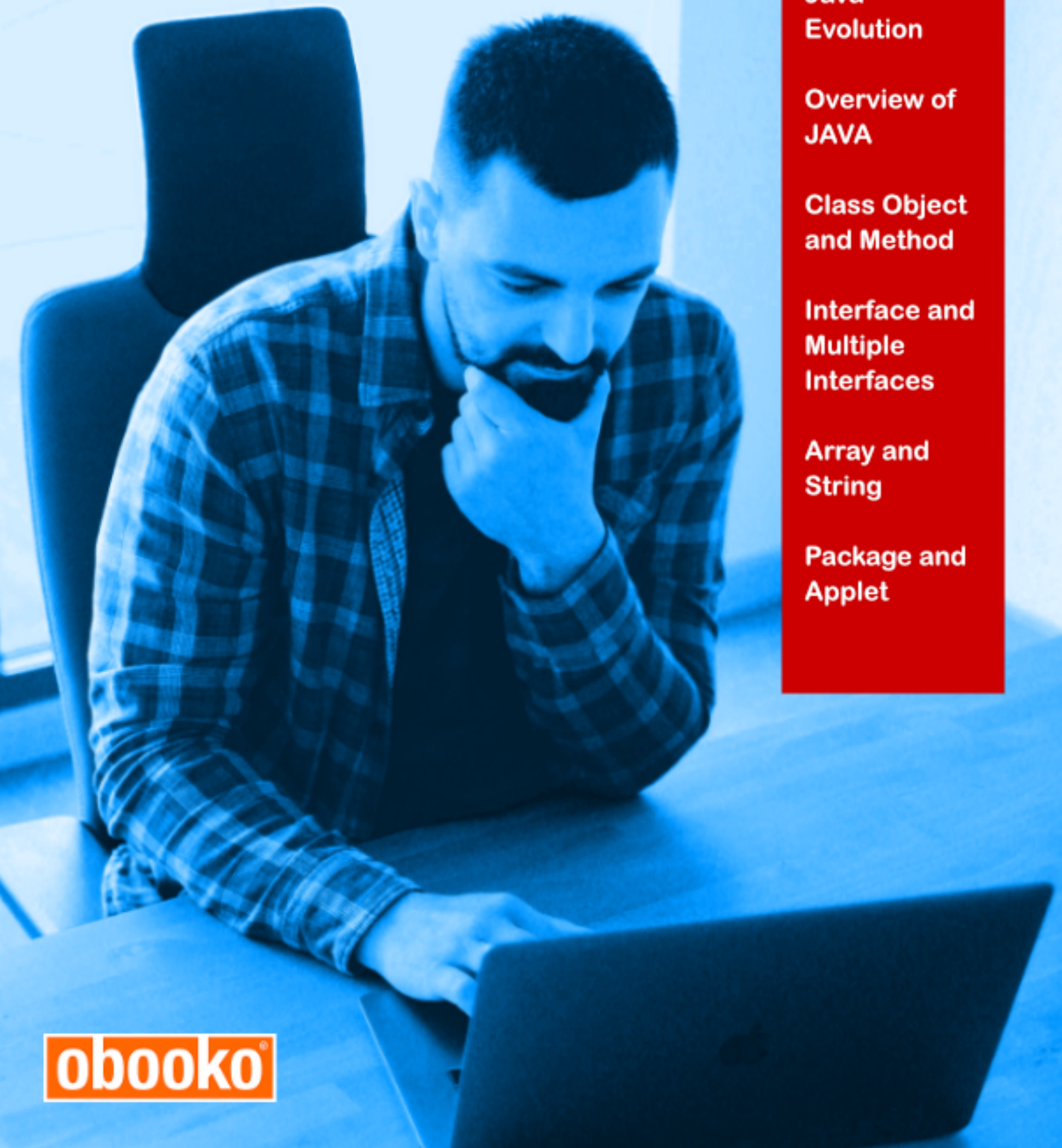
# UNDERSTANDING JAVA

Parmeshwar Korde



- Java Evolution
- Overview of JAVA
- Class Object and Method
- Interface and Multiple Interfaces
- Array and String
- Package and Applet

obooko®

# Understanding Java

Parmeshwar Korde

First Edition

This is an authorized free copy for members of Obooko www.obooko.com



Parmeshwar Korde
Lokmanya Nagar
Near Doordarshan Kendra
Parbhani 431401
Maharashtra India
Korde.parmeshwar@gmail.com

*Cover photo by djordje-petrovic*

## Table of Contents

- JAVA Statements
- JAVA Virtual Machine
- Command Line Arguments
- Constants
- Variables
- Symbolic constants
- Data types
- Type casting
- Standard default values
- Java Statements( Decision making, Branching and looping)

## 3. Class Object and Method

- Introduction
- Class and object
- Constructor
- Method Overloading
- Static members in JAVA
- Nesting Method
- Inheritance
- Method overriding
- Final variables and methods

## 4. Interface and Multiple Interfaces

- Introduction
- Interface
- Extending Interfaces
- Implementing Interfaces
- Multiple Interface

## 5. Array and String

- Introduction
- Array
- Multi-Dimensional Array
- Java String Array
- Java string methods

## 6. Package and Applet

- Introduction
- JAVA API package
- Using system package

- Naming Convention
- Java Package
- Java string methods
- Introduction to applet
- Difference between JAVA APPLET and JAVA Application program
- Preparing to write Applets
- Building Applet Code
- Applet  life cycle

## Chapter 1

# Java Evolution

## 1. Java (programming language)

**Java** is a programming language originally developed by James Gosling at Sun Microsystems (which has since merged into 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. Java applications are typically compiled to byte code (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java is, as of 2012, one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users.

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. Java was originally designed for interactive television, but it was too advanced for the digital cable television industry at the time. The language was initially called Oak after an oak tree that stood outside Gosling's office; it went by the name *Green* later, and was later renamed *Java*, from Java coffee, said to be consumed in large quantities by the language's creators. Gosling aimed to implement a virtual machine and a language that had a familiar C/C++ style of notation. Sun Microsystems released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network- and file-access

restrictions. Major web browsers soon incorporated the ability to run Java applets within web pages, and Java quickly became popular. With the advent of *Java 2* (released initially as J2SE 1.2 in December 1998 – 1999), new versions had multiple configurations built for different types of platforms. For example, *J2EE* targeted enterprise applications and the greatly stripped-down version *J2ME* for mobile applications (Mobile Java). *J2SE* designated the Standard Edition. In 2006, for marketing purposes, Sun renamed new *J2* versions as Java EE, Java ME, and Java SE, respectively.

In 1997, Sun Microsystems approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a de facto standard, controlled through the Java Community Process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) (a subset of the SDK); the primary distinction involves the JRE's lack of the compiler, utility programs, and header files.

On November 13, 2006, Sun released much of Java as free and open source software, (FOSS), under the terms of the GNU General Public License (GPL). On May 8, 2007, Sun finished the process, making all of Java's core code available under free software/open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright.

Sun's vice-president Rich Green said that Sun's ideal role with regards to Java was as an "evangelist." Following Oracle Corporation's acquisition of Sun Microsystems in 2009–2010, Oracle has described itself as the "steward of Java technology with a relentless commitment to fostering a community of participation and transparency". This did not hold Oracle, however, from filing a lawsuit against Google shortly after that for using Java inside the Android SDK. Java software runs on laptops to data centers, game consoles to scientific supercomputers. There are 930 million Java Runtime Environment downloads each year and 3 billion mobile phones run Java. On April 2, 2010, James Gosling resigned from Oracle

## 2. Java Versions

### 2.1 JDK Alpha and Beta (1995)

Alpha and Beta Java public releases had highly unstable APIs and ABIs. The supplied Java web browser was named Web Runner.

### 2.2 JDK 1.0 (January 23, 1996)

Codename Oak. Initial release The first stable version was the JDK 1.0.2. is called Java 1

### 2.3 JDK 1.0 (January 23, 1996)

Codename Oak. Initial release The first stable version was the JDK 1.0.2. is called Java 1

### 2.4 JDK 1.1 (February 19, 1997)

- an extensive retooling of the AWT event model
- inner classes added to the language
- JavaBeans
- JDBC
- RMI
- reflection which supported Introspection only, no modification at runtime was possible.

### 2.5 J2SE 1.2 (December 8, 1998)

Codename Playground. This and subsequent releases through J2SE 5.0 were rebranded retrospectively Java 2 and the version name "J2SE" (Java 2 Platform, Standard Edition) replaced JDK to distinguish the base platform from J2EE (Java 2 Platform, Enterprise Edition) and J2ME

### 2.6 J2SE 1.2 (December 8, 1998)

Codename Playground. This and subsequent releases through J2SE 5.0 were rebranded retrospectively Java 2 and the version name "J2SE" (Java 2 Platform, Standard Edition) replaced JDK to distinguish the base platform from J2EE (Java 2 Platform, Enterprise Edition) and J2ME (Java 2 Platform, Micro Edition). This was a very signaficant release of Java as it tripled the size of the Java platform to 1520 classes in 59 packages. Major additions included:

- strictfp keyword
- the Swing graphical API was integrated into the core classes
- Sun's JVM was equipped with a JIT compiler for the first time
- Java Plug-in
- Java IDL, an IDL implementation for CORBA interoperability
- Collections framework

### 2.7 J2SE 1.3 (May 8, 2000)

- HotSpot JVM included (the HotSpot JVM was first released in April, 1999 for the J2SE 1.2 JVM)
- RMI was modified to support optional compatibility with CORBA
- JavaSound
- Java Naming and Directory Interface (JNDI) included in core libraries (previously available as an extension)
- Java Platform Debugger Architecture (JPDA)
- Synthetic proxy classes

### 2.8 J2SE 1.4 (February 6, 2002)

- regular expressions modeled after Perl regular expressions
- exception chaining allows an exception to encapsulate original lower-level exception
- Internet Protocol version 6 (IPv6) support
- non-blocking IO (named NIO) (New Input/Output) (Specified in JSR 51.)
- logging API (Specified in JSR 47.)
- image I/O API for reading and writing images in formats like JPEG and PNG
- integrated XML parser and XSLT processor (JAXP) (Specified in JSR 5 and JSR 63.)
- integrated security and cryptography extensions (JCE, JSSE, JAAS)
- Java Web Start included (Java Web Start was first released in March, 2001 for J2SE 1.3) (Specified in JSR 56.)
- Preferences API (java.util.prefs)

### 2.9 J2SE 5.0 (September 30, 2004)

- Generics: Provides compile-time (static) type safety for collections and eliminates the need for most typecasts (type conversion). (Specified by JSR 14.)
- Metadata: Also called annotations; allows language constructs such as classes and methods to be tagged with additional data, which can then be processed by metadata-aware utilities. (Specified by JSR 175.)
- Autoboxing/unboxing: Automatic conversions between primitive types (such as int) and primitive wrapper classes (such as Integer). (Specified by JSR 201.)
- Enumerations: The enum keyword creates a typesafe, ordered list of values (such as Day.MONDAY, Day.TUESDAY, etc.). Previously this could only be achieved by non-typesafe constant integers or manually constructed classes (typesafe enum pattern). (Specified by JSR 201.)
- Varargs: The last parameter of a method can now be declared using a type name followed by three dots (e.g. void drawtext(String... lines)). In the calling code any number of parameters of that type can be used and they are then placed in an array to be passed to the method, or alternatively the calling code can pass an array of that type.
- Enhanced for each loop: The for loop syntax is extended with special syntax for iterating over each member of either an array or any Iterable, such as the standard Collection classes, using a construct of the form:

### 2.10 Java SE 6 (December 11, 2006)

- Support for older Win9x versions dropped. Unofficially Java 6 Update 7 is the last release of Java shown to work on these versions of Windows. This is believed to be due to the major changes in Update 10.
- Scripting Language Support (JSR 223): Generic API for tight integration with scripting languages, and built-in Mozilla JavaScript Rhino integration
- Dramatic performance improvements for the core platform, and Swing.
- Improved Web Service support through JAX-WS (JSR 224)
- JDBC 4.0 support (JSR 221).
- Java Compiler API (JSR 199): an API allowing a Java program to select and invoke a Java Compiler programmatically.

- Upgrade of JAXB to version 2.0: Including integration of a StAX parser.
- Support for pluggable annotations (JSR 269)
- Many GUI improvements, such as integration of SwingWorker in the API, table sorting and filtering, and true Swing double-buffering (eliminating the gray-area effect).
- JVM improvements include: synchronization and compiler performance optimizations, new algorithms and upgrades to existing garbage collection algorithms, and application start-up performance.

### 2.11 Java SE 7 (July 28, 2011)

- JVM support for dynamic languages, following the prototyping work currently done on the Multi Language Virtual Machine
- Compressed 64-bit pointers Available in Java 6 with -XX:+UseCompressedOops
- Small language changes (grouped under a project named Coin)
    - Strings in switch
    - Automatic resource management in try-statement
    - Improved type inference for generic instance creation
    - Simplified varargs method declaration
    - Binary integer literals
    - Allowing underscores in numeric literals
    - Catching multiple exception types and rethrowing exceptions with improved type checking
- Concurrency utilities under JSR 166
- New file I/O library to enhance platform independence and add support for metadata and symbolic links. The new packages are java.nio.file and java.nio.file.attribute
- Library-level support for Elliptic curve cryptography algorithms
- An XRender pipeline for Java 2D, which improves handling of features specific to modern GPUs
- New platform APIs for the graphics features originally planned for release in Java version 6
- Enhanced library-level support for new network protocols, including SCTP and Sockets Direct Protocol
- Upstream updates to XML and Unicode

### 2.12 Java SE 8

- Language-level support for lambda expressions (officially, lambda expressions; unofficially, closures) under Project Lambda. There was an ongoing debate in the Java community on whether to add support for lambda expressions. Sun later declared that lambda expressions would be included in Java and asked for community input to refine the feature.
- Parts of project Coin that are not included in Java 7
- JSR 310: Date and Time API
- Tight integration with JavaFX

### 2.13 Java 9

At Java One 2011, Oracle discussed features they hope to have in Java 9, including better support for multi-gigabyte heaps, better native code integration, and a self-tuning JVM.

- Modularization of the JDK under Project Jigsaw

There are plans to add automatic parallelization using OpenCL.

### 2.14 Java 10

There is speculation of removing primitive data types and move towards 64-bit addressing.

## 3. JAVA Features

**Simple and Small**

Java is easily programmable, with a minimum of training, since it is based on C++. Most of the development tools available today are huge in size and complexity, and require powerful development platforms. But Javas's components are comparatively very small in size, enabling construction of compact software that can run 'stand-alone' even on small machines.

**Object Oriented**

An object-oriented design enables the development of small, modular, yet self contained units, each of which could later be used as building block for any specific application. It also facilitates

a clean definition of interfaces and allows 'plug-and-play' features, wherein reusable software components can be distributed easily across diverse hardware platforms. Programmers can even attach their applications and data together, and bundle the combination as a single entity. Java retains many of the object oriented features and the 'look and feel' of C++. Hence, programmers can migrate easily to Java and be productive quickly. To be truly considered as 'object oriented' a programming language should support a minimum of four characteristics.

1) Encapsulation - implements information hiding and modularity (abstraction).
2) Polymorphism - the same message sent to different objects results in behavior that is dependent on the nature of the object receiving the message.
3) Inheritance - you define new classes and behavior based on existing classes to obtain code re-use and code organization.
4) Dynamic binding - objects could come from anywhere possibly across the network. You need to be able to send messages to objects without having to know their specific type at the time you write your code. Dynamic binding provides maximum flexibility while a program is executing.

Now let us see that are objects. They are software programming models. In your everyday life you are surrounded by objects: cars, coffee machines, trees and so on. Software applications contain objects : buttons on user interfaces, spreadsheets, spreadsheet cells and so on. These objects have state and behavior. You can represent all these things with software constructs called objects, which can also be defined by their state and behavior. A car can be modeled by an object. It has state (how fast it is going, in which direction, its fuel consumption and so on) and behavior (starts, stops, turns etc.) Classes are fundamental entities in Java, as they are in any object oriented language. These classes are ways of representing certain sets of data that have something in common and of providing an interface for the use of this data. Encapsulating the data makes it easy to use, safe from outside interference and also make changing the interface and data content easy. It is really simple to reuse a great portion of code already written in Java, without bothering about how it is implemented. Java has a huge repository of reusable code, called the Application Programming Interface (API), which greatly simplifies basic programming.

**Portable and Interpreted**

A software application written in Java, be it a word-processor or spreadsheet, a multimedia tool or virtual reality designer, will run perfectly on any computer, on any platform. A Java application is

a combination of a compiled and interpreted language. The source code written by a programmer is first compiled into Byte Code which is uniform for all machines. This Byte Code is compatible with the Java interpreter which is termed as Java Virtual Machine. The interpreter converts the Byte Code into specific machine code for direct execution, on any machine to which it has been ported. So, you may write your code anywhere and run it anywhere else. This is very useful because most programmers worry whether their programs will run on other systems.

**Architecture-Neutral**

The Java Compiler generates an architecture-neutral object file, 'generic' Byte Code instructions, which have nothing to do with a particular computer architecture. This compiled code is executable on any processor, given the presence of the Java run-time system, which will translate it into native machine code on the fly. The same version of the software runs on all platforms, independent of any CPU or hardware architecture, across networks.

**Dynamic**

By making interconnections between modules at run-time, Java completely avoids dependency on other software components like libraries. Therefore, Java applications are always ready-to-run, without any side-effects, irrespective of changes or revisions that may take place in those external components.

**Distributed**

Java has an extensive library of routines for easy distribution of applications as it abides by industry standard networking protocols like TCP/IP, HTTP, FTP. Java applications can open and access objects across the net via URLs with the same case that programmers are used to when accessing a local file system over a network.

**Automatic Memory Management**

The Java garbage collector keeps tracks of all objects generated, automatically freeing the memory used by objects that have no further use and are not referred to by other existing objects. This contributes enormously to making the code robust.

**Multi-Threaded**

Multithreading is a way of building applications with multiple processes. Support for multithreading enables the efficient execution of programs that potentially have multiple threads of control. Thus, the garbage collector is efficiently run as a background process overcoming the biggest drawback of garbage collector based languages. Multithreading is supported via

inheritable thread class libraries. The result is better interactive response and near real-time features. This enables Java to interface and support many features of modern operating systems and new protocols. Modern network based applications, such a the Hot Java WWW browser, typically need to do several things at the same time. A user working with Hot Java can run several animation concurrently while downloading an image and scrolling the page, Java's multithreading capability provides the means to build applications with many concurrent threads of activity. Multithreading thus results in a high degree of interactivity for the end user.

**Robust**

Java programs are robust because explicit memory manipulations by the programmer are prevented.

**Secured**

Java programs are secured : distributed applications have to exhibit the highest levels of security concerns. A type code verifier in the Java interpreter ensures that the complied code is strictly language compliant, thus trapping all modifications, more so the computer viruses parading as legal code.

## 4. Java and C++

Java is a derivative of C and C++: simple, familiar syntax and with fewer complex features. It does not have many of the poorly understood, confusing and rarely used features of C++. There are no pointers in Java only tightly bounded arrays. This eliminates the possibility of over writing memory and corrupting data unwittingly. Pointers have been the source of raw power in C and C++ programs as well as the primary feature that helped introduce bugs into almost all programs. Java is sophisticated enough to help programmers express complex ideas using arrays and has an efficient automatic garbage collector based memory management scheme. In addition, unlike C or C++, Java does not support structures, enums or functions. Every programmable object is an instance of a 'class' in object oriented Java. This class definition permits both static and dynamic inheritance and hence full reuse of code. Java does not support multiple inheritance whose semantic and usage has been quite controversial in other languages. Following the principles of

structured programs there are no goto statements, no automatic typecasting or operator overloading.

The differences between the C++ and Java programming languages can be traced to their heritage, as they have different design goals.

- C++ was designed for systems and applications programming, extending the C programming language. To this procedural programming language designed for efficient execution, C++ has added support for statically typed object-oriented programming, exception handling, scoped resource management, and generic programming, in particular. It also added a standard library which includes generic containers and algorithms.
- Java was created initially as an interpreter for printing systems but grew to support network computing. It was once used as the base for the "HotJava" thin client system. It relies on a virtual machine to be secure and highly portable. It is bundled with an extensive library designed to provide a complete abstraction of the underlying platform. Java is a statically typed object-oriented language that uses similar (but incompatible) syntax to C++. It was designed from scratch with the goal of being easy to use and accessible to a wider audience. It includes an extensive documentation called Javadoc.

The different goals in the development of C++ and Java resulted in different principles and design trade-offs between the languages. The differences are as follows

| **C++** | **Java** |
|---|---|
| Compatible with C source code, except for a few corner cases. | No backward compatibility with any previous language. The syntax is, however, strongly influenced by C/C++. There are things such as reserved keywords, such as xor and const that don't do anything so that people who write on C++ don't get confused. |
| Write once, compile anywhere (WOCA). | Write once, run anywhere / everywhere (WORA / WORE). |
| Allows procedural programming, functional programming, object-oriented programming, and template meta programming. | Strongly encourages an object-oriented programming paradigm. |

| | |
|---|---|
| Allows direct calls to native system libraries. | Call through the Java Native Interface and recently Java Native Access |
| Exposes low-level system facilities. | Runs in a virtual machine. |
| Only provides object types and type names. | Is reflective, allowing meta programming and dynamic code generation at runtime. |
| Has multiple binary compatibility standards (commonly Microsoft and Itanium/GNU). | Has a binary compatibility standard, allowing runtime check of correctness of libraries. |
| Optional automated bounds checking (e.g., the at() method in vector and string containers). | Normally performs bounds checking. Hot Spot can remove bounds checking. |
| Supports native unsigned arithmetic. | No native support for unsigned arithmetic.(Explain) |
| Standardized minimum limits for all numerical types, but the actual sizes are implementation-defined. Standardized types are available as typedefs (uint8_t, ..., uintptr_t, ...). | Standardized limits and sizes of all primitive types on all platforms. |
| Pointers, references, and pass-by-value are supported. | Primitive and reference data types always passed by value. |
| Explicit memory management. Supports destructors. C++11 replaces the old standard RAII auto_ptr<T> by unique_ptr<T> and adds shared_ptr<T> (smart pointer with reference counter), though third party frameworks exist to provide better garbage collection. | Automatic garbage collection (can be triggered manually). Has a finalize() method that works as a destructor. |
| Supports classes, structs, and unions, and can allocate them on heap or stack. | Only supports classes, and allocates them on the heap. Java SE 6 optimizes with escape analysis to allocate some objects on the stack. |
| Allows explicitly overriding types. | Rigid type safety except for widening conversions. Autoboxing/unboxing added in Java 1.5. |
| The C++ Standard Library has a much more | The standard library has grown with each |

| | |
|---|---|
| limited scope and functionality than the Java standard library but includes language support, diagnostics, general utilities, strings, locales, containers, algorithms, iterators, numerics, input/output, and Standard C Library. The Boost library offers more functionality, including threads and network I/O. Users must choose from a plethora of (mostly mutually incompatible) third-party libraries for GUI and other functionality. | release. By version 1.6, the library included support for locales, logging, containers and iterators, algorithms, GUI programming (but not using the system GUI), graphics, multi-threading, networking, platform security, introspection, dynamic class loading, blocking and non-blocking I/O. It provided interfaces or support classes for XML, XSLT, MIDI, database connectivity, naming services (e.g. LDAP), cryptography, security services (e.g. Kerberos), print services, and web services. SWT offers an abstraction for platform-specific GUIs. |
| Operator overloading for most operators. | The meaning of operators is generally immutable, but the + and += operators have been overloaded for Strings. |
| Full Multiple inheritances, including virtual inheritance. | From classes, only single inheritance is allowed. From Interfaces, Multiple inheritance is allowed. |
| Compile-time templates. | Generics are used to achieve an analogous effect to C++ templates, but they do not translate from source code to byte code due to the use of type erasure by the compiler. |
| Function pointers, function objects, lambdas (in C++11), and interfaces. | No function pointer mechanism. Instead, idioms such as Interface, Adapter, and Listener are extensively used. |
| No standard inline documentation mechanism. Third-party software (e.g. Doxygen) exists. | Javadoc standard documentation. |
| const keyword for defining immutable variables and member functions that do not change the object. | final provides a version of const, equivalent to type* const pointers for objects and plain const for primitive types only. No const member functions, nor any equivalent to const type* |

| | |
|---|---|
| | pointers. |
| Supports the goto statement. It may cause Spaghetti Programming. | Supports labels with loops and statement blocks. |
| Source code can be written to be platform-independent (can be compiled for Windows, BSD, Linux, Mac OS X, Solaris, etc., without modification) and written to take advantage of platform-specific features. Typically compiled into native machine code. | Compiled into byte code for the JVM. Byte code is dependent on the Java platform, but is typically independent of operating system specific features. |

## 5. Differences between C and Java

1. Java is a lot like C but the major difference between Java and C is that Java is an object-oriented language and has mechanism to define classes and objects. In an effort to build a simple and safe; language, the Java team did not include some of the C features in Java.
2. Java does not include the C unique statement keywords **sizeof,** and **typedef.**
3. Java does not contain the data types **struct** and **union.**
4. Java does not define the type modifiers keywords **auto, extern, register, signed,** and **unsigned,**
5. Java does not support an explicit pointer type.
6. Java does not have a preprocessor and therefore we cannot use # **define, # include,** and # **ifdef**statements.
7. Java requires that the functions with no arguments must be declared with empty parenthesis and not with the v**oid** keyword as done in C.
8. Java adds new operators such as **instanceof and >>>.**
9. Java adds labelled **break** and **continue** statements.
10. Java does not have "goto" control structure.
11. Java adds many features required for object-oriented programming.

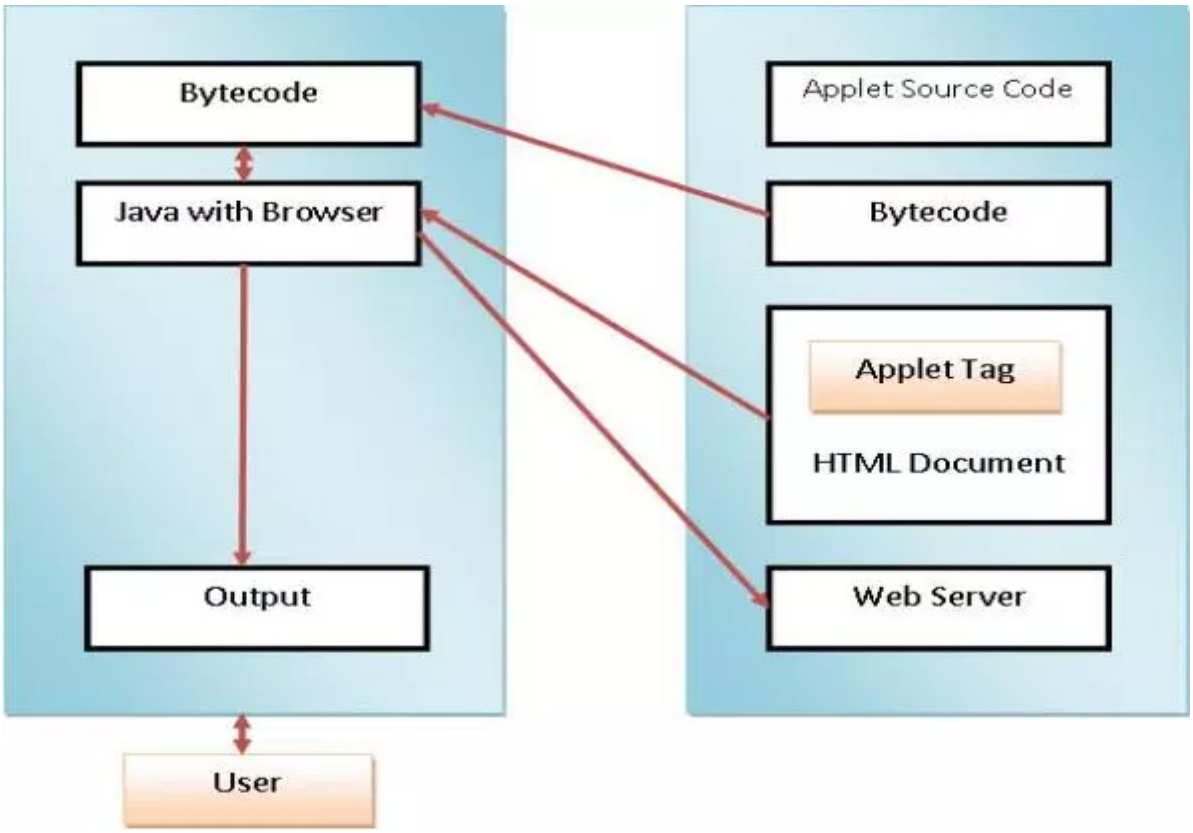
## 6. Java and the Internet

The Java programming language enables the Web documents authors to deliver small application programs to anyone browsing the pages of the html documents. The page becomes alive because it can create game score boards, execute animated cartoons, audio files and video clippings. In addition, it changed the way Internet and WWW worked by allowing architecturally neutral compiled code to be dynamically loaded from anywhere in the network of heterogeneous systems and executed transparently. With Java, data and programs - the twin staples of computing - do not have to be stored on your computer anymore. They can reside anywhere on the Internet, called up by whoever needs them and whenever they need them. The WWW introduced millions of users to the world of the Internet. But for their charm, most Web pages themselves are nearly as static as their book or magazine counterparts. Although the Web is technically equipped to supply video and sound, the user needs to have the right software to make use of them. Java allows you to package software along with data. In other words, when you download a video clip, you also automatically get the software needed to play it. One thing is certain. The Web has become a powerful computer which anybody can use, so costly desktop systems will soon be replaced by a simple computer connected to the Internet and capable of executing any program dynamically loaded from anywhere in the world. Companies will sell computing capabilities over the Internet, just like our electricity boards supply us with power. In the near future, a sizable part of our preferred activity will be over the Internet and our success depends on our ability to adapt to and use the Internet technologies.

## 7. Java and World Wide Web

Java was meant to be used in distributed environments such as Internet. Since, both the Web and Java share the same philosophy, Java could be easily incorporated into the Web system. Before Java, the World Wide Web was limited to the display of still images and texts. However, the incorporation of Java into the Web page has made it capable of supporting animation, graphics, games, and a wide range of special effects. With the support of Web, we can run Java program on someone else's computer access from the Internet.

Java communicates with a Web page through a special tag called <APPLET>.the Figure illustrates the process. It shows the following communication steps:




1. The user sends a request for an HTML document to the remote computer's Web server. The Web server is a program that accepts a request, processes the request, and sends the request document.

2. The HTML document is returned to the user's browser. The document contains the APPLET tag, which identifies the applet.

3. The corresponding applet byte code is transferred to the user's computer. This bytecode had been previously created by the Java compiler using Java source code file for that applet.

4. The Java enabled browser on the user's computer interprets the byte codes and provides output.

5. The user may have further interaction with the applet but with no further downloading from the provider's Web server. This is because the bytecode contains all the information necessary to interpret the applet.

## 8. Web Browsers

A **web browser** (commonly referred to as a **browser**) is a software application for accessing information on the World Wide Web.

A web browser, or simply "browser," is an application used to access and view websites. Common web browsers include Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, and Apple Safari.

The primary function of a web browser is to render HTML, the code used to design or "mark up" WebPages. Each time a browser loads a web page, it processes the HTML, which may include text, links, and references to images and other items, such as cascading style sheets and JavaScript functions. The browser processes these items, then renders them in the browser window. Example of Web Brewers, among of others, include

- Hot JAVA
- Netscape Navigator
- Internet Explorer

## Hot JAVA

**Hotjava** is a web browser from Sun Microsystems that was written in Java. At the time (1996), it was the first web browser that was capable of supporting Java applets. Development of the browser originally began in 1994 under the name "WebRunner". It was criticized for slowness,

which was due to the limitations of the Java Virtual Machine during that time period. Currently, Hotjava is no longer being produced and is no longer supported.

### Netscape Navigator

Netscape Navigator was the first commercially successful Web browser. It was based off the Mosaic browser and was created by a team led by Marc Andreessen, a programmer who co-wrote the code for Mosaic. Netscape Navigator helped influence the development of the Web into a graphical user experience rather than a purely text-based one. In the 1990s, Netscape Navigator was on the leading edge of innovations in Web browsing. Among the many features that became standard after Navigator pioneered them are:

- Displaying a Web page as it loads
- Using JavaScript to create forms and interactive content
- Using cookies to keep session information

### Internet Explorer

Microsoft Internet Explorer (abbreviated IE or MSIE) is a free web browser application produced by Microsoft in 1995. Internet Explorer was designed in response to the first geographical browser, Netscape Navigator. Internet browsing software included on computers with their Windows operating system. This software allows users to view and navigate web pages on the Internet. Internet Explorer is the most widely used browser in the world.

## 9. Hardware and Software Requirements

Java is currently supported on Windows 95, Windows NT, Windows XP, Sun Solaris, Macintosh, and UNIX machines. Though, the programs and examples in this book were tested under Windows 95, the most popular operating system today, they can be implemented on any of the above systems.

The minimum hardware and software requirements for Windows 95 version of Java are as follows:

IBM-compatible 486 system

Minimum of 8 MB memory

Windows 95 software

Windows-compatible sound card, if necessary

A hard drive

CD-ROM drive

Microsoft-compatible mouse

## 10. Java Support Systems

The operations of Java language and Java-enabled browsers on the internet requires a variety of support systems, namely,

- Internet Connection
- Web server
- Web Browser
- HTML- A language for creating hypertext for the web.
- APPLET tag
- Java code
- Bytecode
- Proxy Server-an intermediate server between the requesting client workstation and the original server.
- Mail Server

## 11. Java Environment

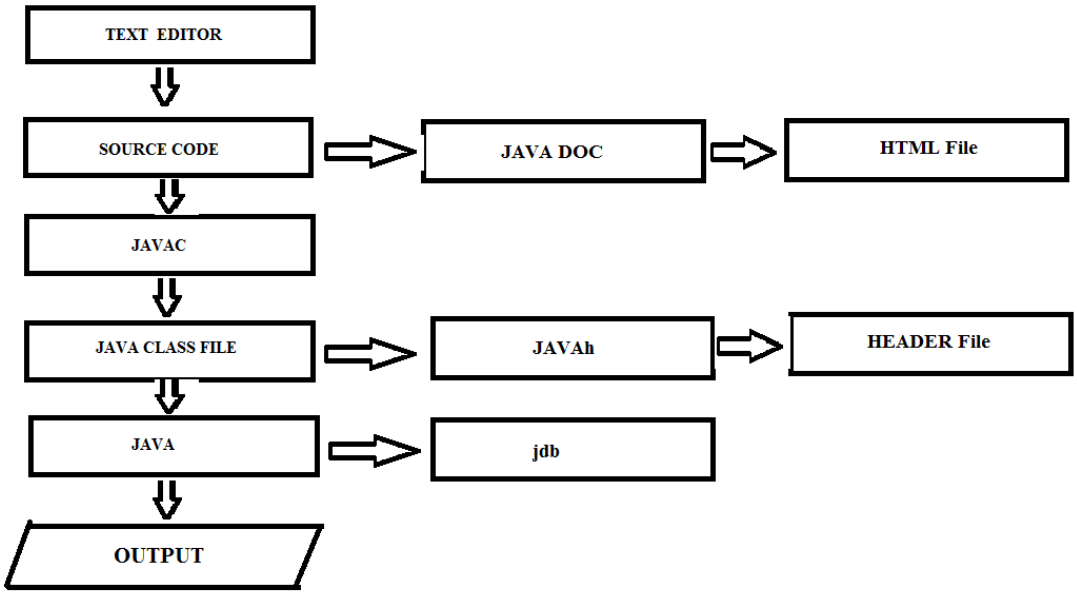
Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as *Java Development Kit* (JDK) and the classes and methods are part of the *Java Standard Library* (JSL), also known as the *Application Programming Interface* (API).

## 11.1 Java Development Kit

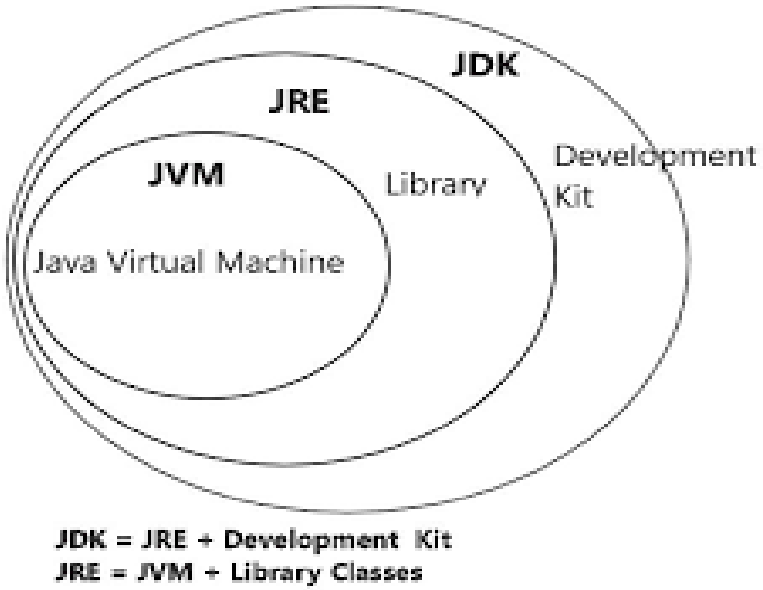
The Java Development Kit comes with a collection of tools that are used for developing and running Java programs. They include:

- applet viewer (for viewing Java applets)
- javac (Java compiler)
- Java (Java interpreter)
- javap (Java disassemble)
- javah (for C header files)
- javadoc (for creating HTML documents)
- jdb (Java debugger)

The following table describes the tools used in java environment




| | |
|---|---|
| Applet viewer | it enables us to run java applets (without actually using a java-compatible browser). |
| Java | java interpreter, which runs applets and applications by reading and interpreting bytecode files. |
| Javac | the java compiler, which translates java source code to bytecode files that the interpreter can understand |
| Javadoc | creates html-format documentation from java source code files, |
| javah | produces header files for use with native methods. |
| javap | java disassembler, which enables us to convert bytecode files into a program description, |
| jdb | java debugger, which helps us to find errors in our programs. |




The way these tools are applied to build and run application programs is illustrated in the figure below. To create a Java program, we need to create a source code file using a text editor. The source code is then compiled using the Java compiler **javac** and executed using the Java interpreter **Java,** The Java debugger **jdb** is used to find errors, if any, in the source code. A compiled Java program can be converted into a source code with the help of Java disassembler javap.

## 11.2 APPLICATION PROGRAMMING INTERFACE (API)

The Java Standard Library (or API) includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

- **Language Support Package:** A collection of classes and methods required for implementing basic features of Java.
- **Utilities Package:** A collection of classes to provide utility functions such as date and time functions.
- **Input / Output Package:** A collection of classes required for input/output manipulation.

- **Networking Package:** A collection of classes for communicating with other computers via Internet.
- **AWT Package:** The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.
- **Applet Package:** This includes a set of classes that allows us to create Java applets. The use of these library classes will become evident when we start developing Java programs.

## Summary

- James Gosling invented Java Programming language at Sun Microsystems and released in 1995
- Java is truly objected oriented programming language.
- Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another.
- In November 13, 2006, Sun released Java as free and open source software
- Different Java versions as JDK Alpha and Beta ,JDK 1.0,JDK 1.1 ,J2SE 1.2 ,J2SE 1.3 ,J2SE 1.4 J2SE 5.0,Java SE 6 ,Java SE 7, Java SE 8,Java 9, Java 10
- Java is Simple and Small, Object Oriented, Portable and Interpreted, Architecture-Neutral , Dynamic, Distributed, Automatic Memory Management, Multi-Threaded, Robust, Secured
- C++ was designed for systems and applications programming, extending the C programming language. To this procedural programming language designed for efficient execution, Java was created initially as an interpreter for printing systems but grew to support network computing. It was once used as the base for the "HotJava" thin client system. It relies
- The Java programming language enables the Web documents authors to deliver small application programs to anyone browsing the pages of the html documents
- Java environment includes a large number of development tools and hundreds of classes and methods
- Java language includes five types of tokens Reserved Keywords, Identifiers, Literals, Operators, Separators
- Java application can accept any number of arguments from the command line

## Review Question

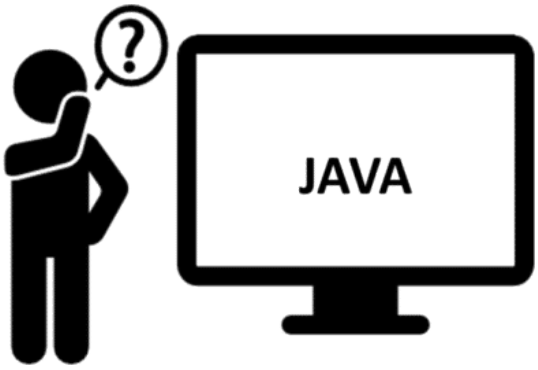
1) Discuss history of Java Programming language in detail
2) What are features of Java Programming language? Discuss in detail.
3) Describe differences between C++ and Java programming language in detail.
4) What are benefits for Internet using Java? Discuss in detail.
5) What is API? Discuss in detail.
6) Describe Java Programming Structure discuss in detail.
7) Discuss different tokens in Java.
8) Describe Java Virtual Machine discuss in detail.

## Chapter 2

# Overview of JAVA

### 1. What is Java?

Java is a programming language which produces software that can be used for gaming, web, business, desktop, database, and other applications. It exhibits most of its syntax from C/C++.




Java is simple, fast, object-oriented, robust, secure, multi-threaded, and platform independent programming language. Java platform consists of several programs, each of which provides a portion of its overall capabilities.

Basically, Java consists of two portions-

- Code
- Platform

Code consists of source code and byte code. Source Code is what a programmer writes in the text editor, and then Java Compiler compiles that and produces byte code. Java platform includes an execution engine, a compiler, and a set of libraries that help to develop programs.

## 2. Installation of Java

*Let's move into the steps to install Java.*

***Step 1: Install JDK:***

- *Visit site @ http://www.oracle.com/technetwork/java/javase/downloads/index.html, choose JDK according to your operating system & download.*

***Step 2: Set PATH to include JDK's directory***

- ***Set PATH permanently in system Environment Variable***

***On Windows:*** *Use the following command:-*

*Right click on My Computer ->Properties ->Advanced System setting ->Environment Variable ->new button in the user variable*

*Provide the required info:*

- *Variable name- PATH*
- *Variable value- C:\jdk1.8.o\bin*

***For Example:***

*If it contains 'C:\Windows\System32' then change your path by 'C:\Windows\System32; C:\Program Files\java\bin'.*

*Click* ***OK*** *button*

***On Linux, UNIX, Solaris***

*Use the following command:*

***export PATH=$PATH:/home/jdk1.6.01/bin/***

*Where JDK is installed in the home directory under Root (home)*

- *Set path temporary in command line*

*Say you have saved a program in D drive, then*

***D:\set PATH=%PATH%; C:\jdk1.8.0\bin;***

## 3. Simple java program

We break the process of programming in Java into three steps

1. *Create* the program by typing it into a text editor and saving it to a file named, say, MyProgram.java.

2. *Compile* it by typing "javac MyProgram.java" in the terminal window.

3. *Run* (or *execute*) it by typing "java MyProgram" in the terminal window.

The first step creates the program; the second translates it into a language more suitable for machine execution (and puts the result in a file named MyProgram.class); the third actually runs the program.

- *Creating a Java program.* A program is nothing more than a sequence of characters, like a sentence, a paragraph, or a poem. To create one, we need only define that sequence characters using a text editor in the same way as we do for e-mail. HelloWorld.java is an example program. Type these character into your text editor and save it into a file named HelloWorld.java.

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

The “Hello World!” program consists of three primary components: the HelloWorld class definition, the main method and source code comments. Following explanation will provide you with a basic understanding of the code:

- **Class definition:** This line uses the keyword **class** to declare that a new class is being defined.

- class HelloWorld

- **HelloWorld** is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace **{** and the closing curly brace **}** .

- **main method:** In Java programming language, every application must contain a main method whose signature is:

public static void main(String[] args)

  - **public**: So that JVM can execute the method from anywhere.
  - **static**: Main method is to be called without object.
  - The modifiers public and static can be written in either order.
  - **void**: The main method doesn't return anything.
  - **main()**: Name configured in the JVM.
  - **String[]**: The main method accepts a single argument:
    - an array of elements of type String.

Like in C/C++, main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

1. The next line of code is shown here. Notice that it occurs inside main( ).

2. System.out.println("Hello, World");

3. This line outputs the string “Hello, World” followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. **System** is a predefined class that provides access to the system, and **out** is the variable of type output stream that is connected to the console.

4. Comments: They can either be multi-line or single line comments.

5. /* This is a simple Java program.

6. Call this file "HelloWorld.java". */

7. This is a multiline comment. This type of comment must begin with /* and end with */. For single line you may directly use // as in C/C++.

## 4. Implementation of JAVA program

- The name of the class defined by the program is HelloWorld, which is same as name of file(HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class and there is at most one public class which contain main() method.

- By convention, the name of the main class(class which contain main method) should match the name of the file that holds the program.

- *Compiling a Java program.* At first, it might seem to you as though the Java programming language is designed to be best understood by the computer. Actually, to the contrary, the language is designed to be best understood by the programmer (that's you). A *compiler* is an application that translates programs from the Java language to a language more suitable for executing on the computer. It takes a text file with the .java extension as input (your program) and produces a file with a .class extension (the computer-language version). To compile HelloWorld.java type the boldfaced text below at the terminal. (We use the % symbol to denote the command prompt, but it may appear different depending on your system.)

  **% javac HelloWorld.java**

If you typed in the program correctly, you should see no error messages. Otherwise, go back and make sure you typed in the program exactly as it appears above.

- *Executing a Java program.* Once you compile your program, you can run it. This is the exciting part, where the computer follows your instructions. To run the HelloWorld program, type the following at the terminal:

  **% java HelloWorld**

  If all goes well, you should see the following response

  Hello, World

- *Understanding a Java program.* The key line with System.out.println() send the text "Hello, World". When we begin to write more complicated programs, we will discuss the meaning of public, class, main, String[], args, System.out, and so on.

- *Creating your own Java program.* For the time being, all of our programs will be just like HelloWorld.java, except with a different sequence of statements in main(). The easiest way to write such a program is to:

  - Copy HelloWorld.java into a new file whose name is the program name followed by .java.
  - Replace HelloWorld with the program name everywhere.
  - Replace the print statement by a sequence of statements.

**Errors.**

Most errors are easily fixed by carefully examining the program as we create it, in just the same way as we fix spelling and grammatical errors when we type an e-mail message.

- *Compile-time errors.* These errors are caught by the system when we compile the program, because they prevent the compiler from doing the translation (so it issues an error message that tries to explain why).

- *Run-time errors.* These errors are caught by the system when we execute the program, because the program tries to peform an invalid operation (e.g., division by zero).

- *Logical errors.* These errors are (hopefully) caught by the programmer when we execute the program and it produces the wrong answer. Bugs are the bane of a programmer's existence. They can be subtle and very hard to find.

## 5. Java program structure

A Java program may contain many classes of which only one class defines a main method. Classes contain data members and methods that operate on the data members of the class. Methods may contain data type declarations and executable statements. To write a Java program, we first define classes and then put them together. A Java program may contain one or more sections as shown below

| | |
|---|---|
| Documentation Section | Suggested |
| Package Statement | Optional |
| Import Statements | Optional |
| Interface Statements | Optional |
| Class Definitions | Optional |
| main Method Class<br>{<br>Main Method Definition<br>} | Essential |

**Documentation Section:** The documentation section comprises a set of comment lines giving the name of the program, the author and other details. Comments explain *why* and *what* of classes and *how* of algorithms. This would greatly help in maintaining the program. Java uses a new style of comment /**....*/ known as *documentation comment.* This form of comment is used for generating documentation automatically.

**Package Statement:** The first statement allowed in a Java file is a *package* statement. This statement declares a *package* name and informs the compiler that the classes defined here belong to this package. Example:

*package student;*

The package statement is optional. That is, our classes do not have to be part of package.

**Import Statements:** The next thing after a package statement (but before any class definitions) may be a number **of** *import* statements. This is similar to the *include* statement in C. Example:

*import student.test;*

This statement instructs the interpreter to load the *test* class contained in the package *student***.** Using import statements, we can have access to classes that are part of other named packages.

**Interface Statements:** An interface is like a class but includes a group of method declarations. This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.

**Class Definitions:** A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program. These classes are used to map the objects of real-world problems. The number of classes used depends on the complexity of the problem.

**Main Method Class:** Since every Java stand-alone program requires a **main** method as its starting point, this class is the essential part of a Java program. A simple Java program may contain only this part. The **main** method creates objects of various classes and establishes communications between them. On reaching the end **of main,** the program gets terminated.

## 6. Java Tokens

Tokens are the ones which are used to prepare a program. The statements of a java program contain expressions, which describe the actions carried out on data. Smallest individual units in a program are known as *tokens.* The compiler recognizes them for building up expressions and statements.

In simple terms, a Java program is a collection of tokens, comments and white spaces. Java language includes five types of tokens. They are:

1)Reserved Keywords

2)Identifiers

3)Literals

4)Operators

5)Separators

**1) Java Character Set:** The smallest units of Java language are the characters used to write Java tokens. These characters are defined by the *Unicode* character set. The Unicode is a 16-bit character coding system and currently supports more than 34,000 defined characters derived from 24 languages.

**2) Keywords:** Keywords are an essential part of a language definition. They implement specific features of the language. Java language has reserved 50 words as keywords. Since keywords have specific meaning in Java, we cannot use them as names for variables, classes, methods and so on. All keywords are to be written in lower-case letters.

**3) Identifiers:** Identifiers are user defined tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules:

They can have alphabets, digits, and the underscore and dollar sign characters.

They must not begin with a digit.

Uppercase and lowercase letters are distinct.

They can be of any length.

**4)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. They are:

Integer literals

Floating point literals

Character literals

String literals

Boolean literals

**5) Operators:** An operator is a symbol that takes one or more arguments and *operates* on them to produce a result. Operators are of many types such as arithmetic, relational and etc.

**6) Separators:** Separators are symbols used to indicate where groups of code are divided and arranged. They basically define the shape and function of our code. They include symbols like braces, comma and etc.

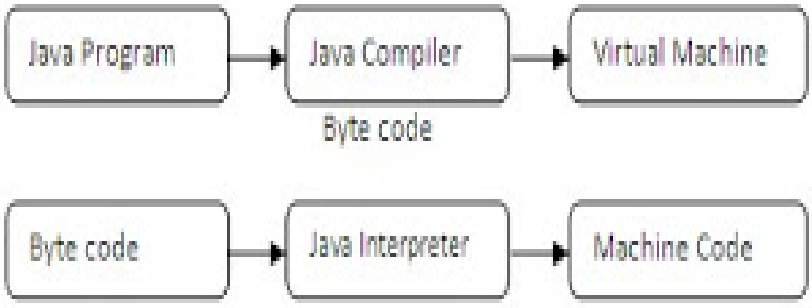
## 7. Java statement

A statement is an executable combination of tokens ending with a semicolon (;) mark. Statements are usually executed in sequential order. However, it is possible to control the flow of execution, if necessary, using special statements. Java implements several types of statements as illustrated in the following Table.

| | |
|---|---|
| **Empty Statement** | These do nothing and are used during program development as a place holder. |
| **Labeled Statement** | Any Statement may begin with a label. Such labels must not be keywords, already declared local variables or previously used labels in this module. Labels in Java are used as the arguments of Jump statements. |
| **Expression Statement** | Most statements are expression statements. Java has seven types of Expression statements Assignment, Pre-Increment, Pre-Decrement, Post-Increment, Post-Decrement, Method Call and Allocation Expression. |
| **Selection Statement** | These select one of several control flows. There are three types of selection statements in Java if, if-else, and switch. |
| **Iteration Statement** | These specify how and when looping will take place. There are three types of iteration statements; while, do and for. |
| **Jump Statement** | Jump Statements pass control to the beginning or end of the current block, or to a labeled statement. Such labels must be in the same block, and continue labels must be on an iteration statement. The four types of Jump statement are *break, continue, return and throw.* |
| **Synchronization Statement** | These are used for handling issues with multithreading. |
| **Guarding Statement** | Guarding statements are used for safe handling of code that may cause exceptions (such as division by zero). These statements use the keywords try, catch, and finally |

## 8. JAVA Virtual Machine

All language compilers translate source code into *machine code* for a specific computer. But, Java compiler produces an intermediate code known as *byte code* for a machine that does not exist. This machine is called the *Java Virtual Machine* and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer




The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine. The interpreter is different for different machines

## 9. Command Line Arguments

Java application can accept any number of arguments from the command line. Command line arguments allow the user to affect the operation of an application. For example, a program might allow the user to specify verbose mode--that is, specify that the application display a lot of trace information--with the command line argument -verbose. When invoking an application, the user

types the command line arguments after the application name. For example, suppose you had a Java application, called Sort, that sorted lines in a file, and that the data you want sorted is in a file named ListOfFriends. If you were using DOS, you would invoke the Sort application on your data file like this:

C:\> **java Sort ListOfFriends**

In the Java language, when you invoke an application, the runtime system passes the command line arguments to the application's main method via an array of Strings. Each String in the array contains one of the command line arguments. In the previous example, the command line arguments passed to the Sort application is an array that contains a single string: "ListOfFriends".

**Echo Command Line Arguments**

This simple application displays each of its command line arguments on a line by itself:

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

**Try this:** Invoke the Echo application with the command line shown in this DOS example:

C:\> **java Echo Drink Hot Java**

```
Drink
Hot
Java
```

You'll notice that the application displays each word--Drink, Hot, and Java--on a line by itself. This is because The Space Character Separates Command Line Arguments.

**Conventions**

There are several conventions that you should observe when accepting and processing command line arguments with a Java application.

**Parsing Command Line Arguments**

Most programs accept several command line arguments that allow the user to affect the execution of the application. For example, the UNIX command that prints the contents of a directory--the ls utility program--accepts arguments that determine which file attributes to print and the order in

which the files are listed. Typically, the user can specify the command line arguments in any order thereby requiring the application to *parse* them.

## 10. Constants

Constants in Java are fixed values those are not changed during the Execution of program java supports several types of Constants those are

**Integer Constants**

Integer Constants refers to a Sequence of digits which Includes only negative or positive Values and many other things those are as follows

- An Integer Constant must have at Least one Digit
- it must not have a Decimal value
- it could be either positive or Negative
- if no sign is Specified then it should be treated as Positive
- No Spaces and Commas are allowed in Name

**Real Constants**

A Real Constant must have at Least one Digit

- it must have a Decimal value
- it could be either positive or Negative
- if no sign is Specified then it should be treated as Positive
- No Spaces and Commas are allowed in Name

- Like 251, 234.890 etc are Real Constants

In The Exponential Form of Representation the Real Constant is Represented in the two Parts The part before appearing e is called mantissa whereas the part following e is called Exponent.

In Real Constant The Mantissa and Exponent Part should be Separated by letter e

The Mantissa Part have may have either positive or Negative Sign

Default Sign is Positive

**Single Character Constants**

A Character is Single Alphabet a single digit or a Single Symbol that is enclosed within Single inverted commas. Like 'S' ,'1' etc are Single Character Constants

**String Constants**

String is a Sequence of Characters Enclosed between double Quotes These Characters may be digits ,Alphabets Like "Hello" , "1234" etc.

**Backslash Character Constants**

Java Also Supports Backslash Constants those are used in output methods For Example \n is used for new line Character These are also Called as escape Sequence or backslash character Constants.

## 11. Variable

A variable is a container that stores a meaningful value that can be used throughout a program. For example, in a program that calculates tax on items you can have a few variables - one variable that stores the regular price of an item and another variable that stores the total price of an item after the tax is calculated on it. Variables store this information in a computer's memory and the value of a variable can change all throughout a program.

**Declaring variables**

One variable in your program can store numeric data while another variable can store text data. Java has special keywords to signify what type of data each variable store. Use these keywords when declaring your variables to set the data type of the variable.

Java data types

| Keyword | Type of data the variable will store | Size in memory |
|---|---|---|
| Boolean | true/false value | 1 bit |
| byte | byte size integer | 8 bits |
| char | a single character | 16 bits |
| double | double precision floating point decimal number | 64 bits |
| float | single precision floating point decimal number | 32 bits |
| int | a whole number | 32 bits |
| long | a whole number (used for long numbers) | 64 bits |
| short | a whole number (used for short numbers) | 16 bits |

Example:

char a Character;

int a Number;

You can assign a value to a variable at the same time that it is declared. This process is known as initialization:

Example:

char aCharacter = 'a'; int aNumber = 10;

Declaring a variable and then giving it a value:

char aCharacter; aCharacter = 'a'; int aNumber; aNumber = 10;

NOTE: A variable must be declared with a data type or an error will be generated! The data type of a variable should be used only once with the variable name - during declaration. After that, you can refer to the variable by its name without the data type.

### Naming variables

Rules that must be followed when naming variables or errors will be generated and your program will not work:

No spaces in variable names

No special symbols in variable names such as !@#%^&*

Variable names can only contain letters, numbers, and the underscore ( _ ) symbol

Variable names cannot start with numbers, only letters or the underscore ( _ ) symbol (but variable names can contain numbers)

Recommended practices (make working with variables easier and help clear up ambiguity in code):

Make sure that the variable name is descriptive of what it stores - For example, if you have a variable which stores a value specifying the amount of chairs in a room, name it "numChairs".

Make sure the variable name is of appropriate length - Should be long enough to be descriptive, but not too long.

Also keep in mind:

Distinguish between uppercase and lowercase - Java is a case sensitive language which means that the variables varOne, VarOne, and VARONE are three separate variables!

When referring to existing variables, be careful about spelling - If you try to reference an existing variable and make a spelling mistake, an error will be generated.

### Printing variables

Variables are printed by including the variable name in a System.out.print() or System.out.println() method. When printing the value of a variable, the variable name

should NOT be included in double quotes. You can also print variables together with regular text. To do this, use the + symbol to join the text and variable values.

```
class PrintText
{
public static void main(String[] args)
{
 //declare some variables
byte aByte = -10;
 int aNumber = 10;
 char aChar = 'b';
 boolean isBoolean = true;
 //print variables alone
 System.out.println(aByte);
 System.out.println(aNumber);
//print variables with text
System.out.println("aChar = " + aChar);
 System.out.println("Is the isBoolean variable a boolean variable? " + isBoolean);
 }
 }
```

Output:

-10 10 aChar = b Is the isBoolean variable a boolean variable? true

## 12. Symbolic Constants in Java

Constants may appear repeatedly in number of places in the program. Constant values are assigned to some names at the beginning of the program, then the subsequent use of these names in the program has the effect of caving their defined values to be automatically substituted in appropriate points. The constant is declared as follows:

**Syntax** : final type symbolicname= value;

Eg final float PI =3.14159;

final int STRENGTH =100;

**Rules:-**

Symbolic names take the some form as variable names. But they one written in capitals to distance from variable names. This is only convention not a rule.

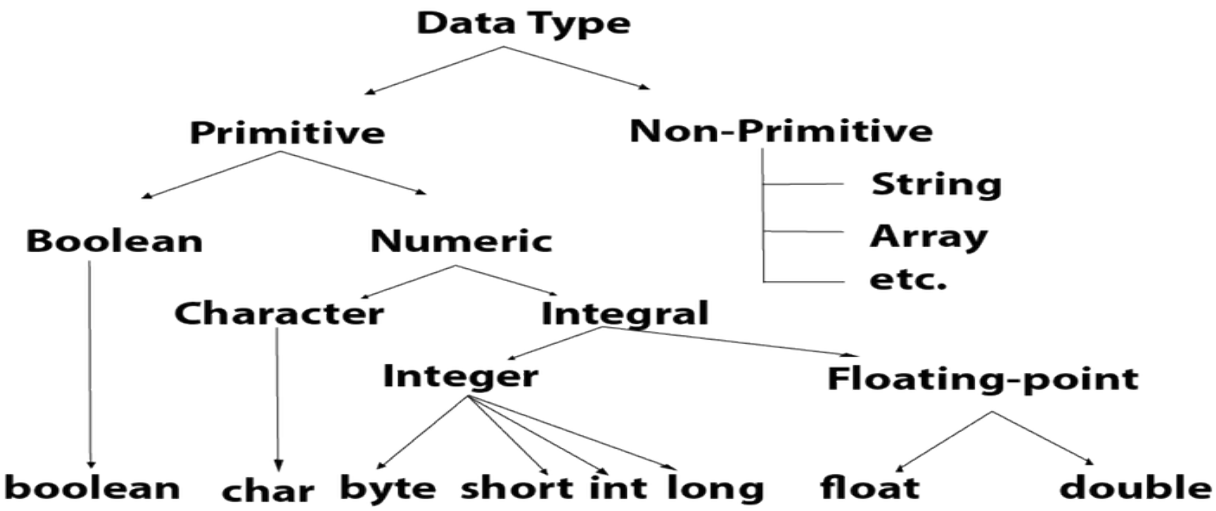
After declaration of symbolic constants they shouldn't be assigned any other value within the program by using an assignment statement.

For eg:- STRENTH = 200 is illegal

Symbolic constants one declared for types there are not done in c & c++ where symbolic constants one defined using the define statement.

They can't be declared inside a method . they should be used only as class data members in the beginning of the class.

## 13. DATA TYPES




Data type defines a set of permitted values on which the legal operations can be performed. In java, all the variables needs to be declared first i.e. before using a particular variable, it must be declared in the program for the memory allocation process. Like

int pedal = 1;

1. This statement exist a field named "pedal" that holds the numerical value as 1. The value assigned to a variable determines its data type, on which the legal operations of java are performed. This behavior specifies that, Java is a strongly-typed programming language.

1) **Primitive Data Types**

**2) Non Primitive Data Types**

**Primitive Data Types**

The primitive data types are predefined data types, which always hold the value of the same data type, and the values of a primitive data type don't share the state with other primitive values. These data types are named by a reserved keyword in Java programming language. There are eight primitive data types supported by Java programming language

***Byte***

The byte data type is an 8-bit signed two's complement integer. It ranges from -128 to127 (inclusive). This type of data type is useful to save memory in large arrays.. We can also use byte instead of int to increase the limit of the code. The syntax of declaring a byte type variable is shown as:

```
byte b = 5;
```

***short***

The short data type is a 16-bit signed two's complement integer. It ranges from -32,768 to 32,767. short is used to save memory in large arrays. The syntax of declaring a short type variable is shown as:

```
short s = 2;
```

***int***

The int data type is used to store the integer values not the fraction values. It is a 32-bit signed two's complement integer data type. It ranges from -2,147,483,648 to 2,147,483,647 that is more enough to store large number in your program. However for wider range of values use long. The syntax of declaring a int type variable is shown as:

```
int num =50 ;
```

***long***

The long data type is a 64-bit signed two's complement integer. It ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Use this data type with larger range of values. The syntax of declaring a long type variable is shown as:

```
long ln =746;
```

***float***

The float data type is a single-precision 32-bit IEEE 754 floating point. It ranges from

1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative). Use a float (instead of double) to save memory in large arrays. We do not use this data type for the exact values such as currency. For that we have to use java.math.BigDecimal class.The syntax of declaring a float type variable is:

```
float f = 105.65;
float f = -5000.12;
```

***double***

This data type is a double-precision 64-bit IEEE 754 floating point. It ranges from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative). This data type is generally the default choice for decimal values. The syntax of declaring a double type variable is shown as:

```
double d = 6677.60;
```

***char***

The char data type is a single 16-bit, unsigned Unicode character. It ranges from 0 to 65,535. They are not integral data type like int, short etc. i.e. the char data type can't hold the numeric values. The syntax of declaring a char type variable is shown as:

```
char caps = 'c';
```

***Boolean***

The boolean data type represents only two values: true and false and occupy is 1-bit in the memory. These values are keywords in Java and represents the two boolean states: on or off, yes or no. We use boolean data type for specifying conditional statements as if, while, do, for. In Java, true and false are not the same as True and False. They are defined constants of the language. The syntax of declaring a boolean type variable is shown as:

```
boolean result = true;
```

The ranges of these data types can be described with default values using the following table:

| Data Type | Default Value (for fields) | Size (in bits) | Minimum Range | Maximum Range |
|---|---|---|---|---|
| Byte | 0 | Occupy 8 bits in memory | -128 | +127 |
| Short | 0 | Occupy 16 bits in memory | -32768 | +32767 |
| int | 0 | Occupy 32 bits in memory | -2147483648 | +2147483647 |
| long | 0L | Occupy 64 bits in memory | -9223372036854775808 | +9223372036854775807 |
| float | 0.0f | Occupy 32-bit IEEE 754 floating point | 1.40129846432481707e-45 | 3.40282346638528860e+38 |
| double | 0.0d | Occupy 64-bit IEEE 754 floating point | 4.94065645841246544e-324d | 1.79769313486231570e+308d |
| char | '\u0000' | Occupy 16-bit, unsigned Unicode character | | 0 to 65,535 |
| boolean | false | Occupy 1- bit in memory | NA | NA |

When we declare a field it is not always essential that we initialize it too. The compiler sets a default value to the fields which are not initialized which might be zero or null. However this is not recommended.

***Integer Data Types***

So far you would have been known about these data types. Now lets take an Integer data type in brief to better understand: As we have told that an integer number can hold a whole number. Java provides four different primitive integer data types that can be defined as byte, short, int, and long that can store both positive and negative values. The ranges of these data types can be described using the following table:

| Data Type | Size (in bits) | Minimum Range | Maximum Range |
|---|---|---|---|
| byte | Occupy 8 bits in memory | -128 | +127 |
| short | Occupy 16 bits in memory | -32768 | +32767 |
| int | Occupy 32 bits in memory | -2147483648 | +2147483647 |
| long | Occupy 64 bits in memory | -9223372036854775808 | +9223372036854775807 |

Examples of floating-point literals are:

0
1
123
-42000

***Floating-point numbers***

A floating-point number represents a real number that may have a fractional values i.e. In the floating type of variable, you can assign the numbers in an in a decimal or scientific notation. Floating-point number have only a limited number of digits, where most values can be represented only approximately. The floating-point types are float and double with

a single-precision 32-bit IEEE 754 floating point and double-precision 64-bit IEEE 754 floating point respectively. Examples of floating-point literals are:

10.0003
48.9
-2000.15
7.04e12

**Non-primitive data types**

The non-primitive data types include Classes, Interfaces, and Arrays.

## 14. TYPE CASTING

It is sometimes necessary to convert a data item of one type to another type. For example when it is necessary to perform some arithmetic using data items of different types (so called mixed mode arithmetic). Under certain circumstances Type conversion can be carried out automatically, in other cases it must be "forced" manually (explicitly).

**Automatic Conversion**

In Java type conversions are performed automatically when the type of the expression on the right hand side of an assignment operation can be safely promoted to the type of the variable on the left hand side of the assignment. Thus we can safely assign: byte -> short -> int -> long -> float -> double.

For example :

//64 bit long integer

long myLongInteger;

//32 bit long integer

int myInteger;

myLongInteger=myInteger;

The extra storage associated with the long integer, in the above example, will simply be padded with extra zeros.

**Explicit Conversion (Casting)**

The above will not work the other way round. For example we cannot automatically convert a long to an int because the first requires more storage than the second and consequently information may be lost. To force such a conversion we must carry out an explicit conversion (assuming of course that the long integer will fit into a standard integer). This is done using a process known as a type cast :

myInteger = (int) myLongInteger

This tells the compiler that the type of myLongInteger must be temporarily changed to a int when the given assignment statement is processed. Thus, the cast only lasts for the duration of the assignment. Java type casts have the following form: (T) N where T is the name of a numeric type and N is a data item of another numeric type. The result is of type T

## 15. Standard default values

In JAVA, every variable has default value, if user does not initialize a variable when it is first created.

Default Value of Data Types in Java:

| Data Type | Default Value (for fields) |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | 'u0000' |
| String (or any object) | null |
| boolean | false |

## 16. JAVA DECISION MAKING, BRANCHING AND LOOPING

We have seen that a Java program is a set of statements, which are normally executed sequentially in the order in which they appear. This happens when no repetitions of certain calculations are necessary. However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions are met.

Java language possesses such decision making capabilities and supports the following statements known as control or decision-making statements.

Decision Making and Branching

if statement

switch statement

Conditional operator statement

**Decision making if statement**

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

if (test expression)

It allows the computer to evaluate the expression first and then , depending on whether the value of the expression (relation or condition) is ' true ' (non-zero) or ' false ' (zero), it transfers the control to a particular statement. This point of program has two paths to follow , one for the true condition and the other for the false condition.

The if statement may be implemented in different forms depending on the complexity of the conditions to be tested.

- Simple if statement
- if...else statement
- Nested if...else statement
- else if ladder.

**Simple if statement**

The general form of a simple if statement is

```
if (test expression)
{
statement-block;
}
statement-x;
```

The statement-block may be a single statement or a group of statements. If the test expression is true, the statement-block will be executed; otherwise the statement-block

will be skipped and the execution will jump to statement-x. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence.

Example:

```
.....
if (x == 1)
{
y = y +10;
}
        System.out.println (y);
.....
```

The program tests the value of x and accordingly calculates y and prints it. If x is 1 then y gets incremented by 1 else it is not incremented if x is not equal to 1. Then the value of y gets printed.

**The if…else statement**

The if....else statement is an extension of the simple if statement. The general form is

```
if (test expression)
{
True-block statement(s)
}
else
{
False-block statement(s)
}
statement-x
```

If the test expression is true , then the true-block statement(s), immediately following the if statement are executed; otherwise the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both.

Example:

```
.....
if (c == 'm')
male = male +1;
else
```

```
fem = fem +1;
.....
```

**Nesting of if…else statements**

When a series of decisions are involved, we may have to use more than one if.....else statement in nested form as follows:

```
if (test condition1)
{
if (test condition 2)
{
statement-1;
}
else
{
statement-2;
}
}
else
{
statement-3;
}
statement-x;
```

If the condition-1 is false statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

**The else if ladder**

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if. It takes the following general form:

```
if (condition 1)
statement 1 ;
else if (condition 2)
statement 2;
```

```
else if (condition 3)
statement 3;
else if (condition n)
statement n;
else
default statement;
statement x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with is executed and the control is transferred to the statement x (skipping the rest of the ladder).

When all the n conditions become false, then the final else containing the default statement will be executed.

Example:
Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Average marks Grade

80-100 Honours
60- 79 First Division
50- 59 Second Division
40- 49 Third Division
0- 39 Fail

This grading can be done using the else if ladder follows:

```
if (marks > 79)
      grade = "Honours";
else
if (marks >59)
      grade = "First Division";
else
if (marks > 49)
      grade = "Second Division";
else
if (marks > 39)
```

```
        grade = "Third division";
else
        grade = "Fail";
```

**The switch Statement**

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

The following rules apply to a switch statement:

The variable used in a switch statement can only be a byte, short, int, or char.

You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.

A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {


  public static void main(String args[]){
    char grade = args[0].charAt(0);


    switch(grade)
    {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
```

```
      case 'D' :
         System.out.println("You passed");
      case 'F' :
         System.out.println("Better try again");
         break;
      default :
         System.out.println("Invalid grade");
    }
    System.out.println("Your grade is " + grade);
  }
}
```

Compile and run above program using various command line arguments. This would produce following result:

```
$ java Test a
Invalid grade
Your grade is a a
$ java Test A
Excellent!
Your grade is a A
$ java Test C
Well done
Your grade is a C
$
```

**The ? : Operator in Java**

The value of a variable often depends on whether a particular boolean expression is or is not true and on nothing else. For instance one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write

```
if (a > b) {
  max = a;
}
else {
  max = b;
}
```

Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it, the conditional operator, ?:. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

(a > b) ? a : b; is an expression which returns one of two values, a or b. The condition, (a > b), is tested. If it is true the first value, a, is returned. If it is false, the second value, b, is returned. Whichever value is returned is dependent on the conditional test, a > b. The condition can be any expression which returns a Boolean value.

**Java Loops - for, while and do...while**

There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

**The while Loop**

A while loop is a control structure that allows you to repeat a task a certain number of times.

- **Syntax:**

The syntax of a while loop is:

```
while(Boolean expression)
{
  //Statements
}
```

When executing, if the boolean_expression result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true. Here key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

- **Example:**

```
public class Test
{
  public static void main(String args[])
{
    int x = 10;
    while( x < 20 )
 {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
```

}

This would produce following result:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**The do...while Loop**

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

**Syntax:**

The syntax of a do...while loop is:

```
do
{
  //Statements
}while(Boolean expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

**Example:**

public class Test

```
{
  public static void main(String args[])
{
    int x = 10;
    do
{
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

**The for Loop:**

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

- **Syntax:**

The syntax of a for loop is:

```
for(initialization; Boolean expression; update)
{
  //Statements
}
```

Here is the flow of control in for loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

**Example:**

```
public class Test {
  public static void main(String args[])
 {
    for(int x = 10; x < 20; x = x+1)
{
       System.out.print("value of x : " + x );
       System.out.print("\n");
    }
```

```
   }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

**Enhanced for loop in Java**

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

**Syntax:**

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
  //Statements
}
```

**Declaration** . The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

**Expression** . This evaluate to the array you need to loop through. The expression can be an array variable or method call that returns an array.

**Example:**

```
public class Test
```

```
{
  public static void main(String args[])
{
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x : numbers )
{
      System.out.print( x );
      System.out.print(",");
    }
    System.out.print("\n");
    String [] names ={"James", "Larry", "Tom", "Lacy"};
    for( String name : names )
{
      System.out.print( name );
      System.out.print(",");
    }
  }
}
```

This would produce following result:

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

**The break statement**

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

**Syntax:**

The syntax of a break is a single statement inside any loop:

```
break;
```

**Example:**

```
public class Test
{
  public static void main(String args[])
{
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x : numbers )
{
      if( x == 30 ) {
          break;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This would produce following result:

```
10
20
```

**The continue statment**

The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.

- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

- **Syntax:**

The syntax of a continue is a single statement inside any loop:

```
continue;
```

- **Example:**

```
public class Test
 {
  public static void main(String args[])
{
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x : numbers )
{
      if( x == 30 )
{
          continue;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

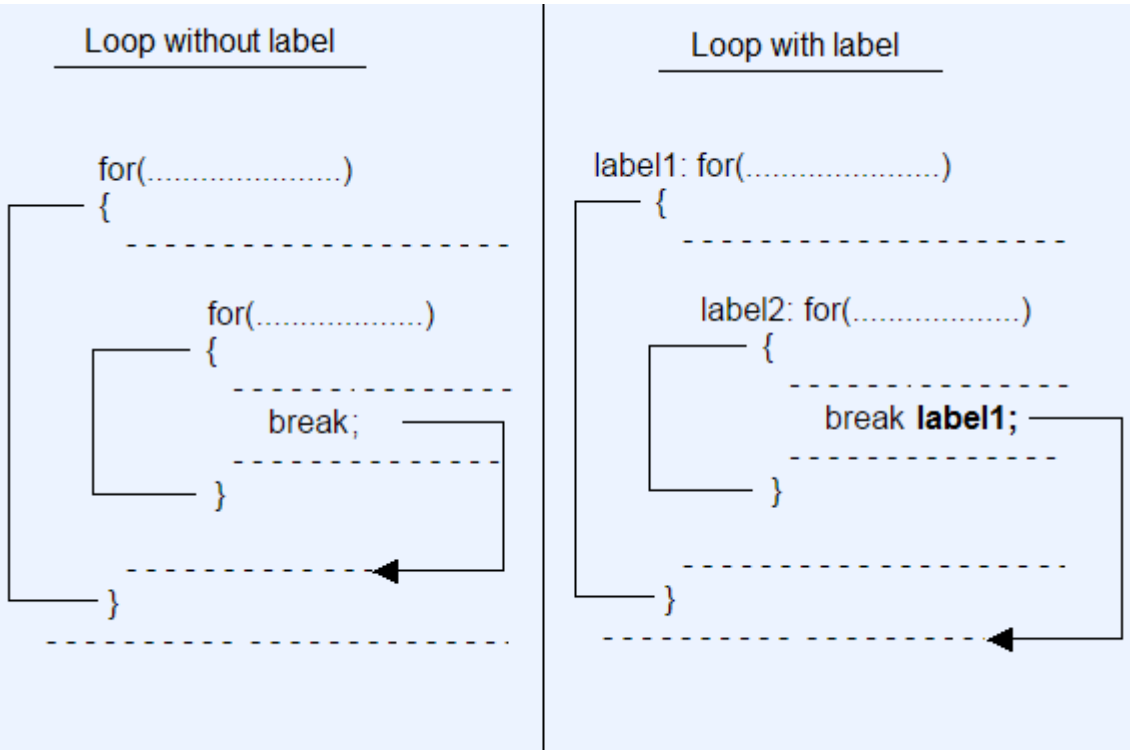
This would produce following result:

```
10
20
40
50
```

## Labelled Loop

According to nested loop, if we put break statement in inner loop, compiler will jump out from inner loop and continue the outer loop again. What if we need to jump out from the outer loop using break statement given inside inner loop? The answer is, we should define **lable** along with colon(:) sign before loop.

**Syntax of Labelled loop**




**Example without labelled loop**

```
        //WithoutLabelledLoop.java
class WithoutLabelledLoop
{
   public static void main(String args[])
   {
      int i,j;

      for(i=1;i<=10;i++)
      {
```

```
            System.out.println();

            for(j=1;j<=10;j++)
            {
               System.out.print(j + " ");

               if(j==5)
                  break;          //Statement 1
            }
         }
      }
   }
```

Output :

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

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

In th above example, statement 1 will break the inner loop and jump outside from inner loop to execute outer loop.

**Example with labelled loop**

```
        //WithLabelledLoop.java
class WithLabelledLoop
{
   public static void main(String args[])
   {
      int i,j;

      loop1:  for(i=1;i<=10;i++)
      {
         System.out.println();

         loop2:  for(j=1;j<=10;j++)
         {
            System.out.print(j + " ");

            if(j==5)
               break loop1;     //Statement 1
         }
      }
   }
}

Output :
        1 2 3 4 5
```

In th above example, statement 1 will break the inner loop and jump outside the outer loop.

## 17. Summary

- Java is a simple, object oriented; high performance language Java is truly objected oriented programming language.
- Java constant are Integer Constants, Real Constants ,Single Character Constants, String Constants, Backslash Character Constants
- A variable is a container that stores a meaningful value that can be used throughout a program.
- Data type defines a set of permitted values on which the legal operations can be performed. in java Primitive Data Types are Byte, short int, long, float double, char, Boolean and Integer Data Types, Floating-point numbers
- In the Java programming language, an Decision making, Branching and looping are similar to C and C++ high level languages.
- C language possesses such decision making capabilities and supports the following statements known as control or decision-making statements.
    - if statement
    - switch statement
    - Conditional operator statement

- The if statement may be implemented in different forms depending on the complexity of the conditions to be tested.
    - Simple if statement
    - if...else statement
    - Nested if...else statement
    - else if ladder.
- Java has very flexible three looping mechanisms. You can use one of the following three loops:
    - while Loop
    - do...while Loop
    - for Loop
- The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement
- The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

## Chapter 3

# Classes Object and Method

## 1. Introduction

Java is an object-oriented language. "Object-oriented" is a term that has become so commonly used as to have practically no concrete meaning. This chapter explains just what "object-oriented" means for Java.

If you are a C++ programmer, or have other object-oriented programming experience, many of the concepts in this list should be familiar to you. If you do not have object-oriented experience, don't fear: This chapter assumes no knowledge of object-oriented concepts.

We saw in the last chapter that close analogies can be drawn between Java and C. Unfortunately for C++ programmers, the same is not true for Java and C++. Java uses object-oriented programming concepts that are familiar to C++ programmers, and it even borrows from C++ syntax in a number of places, but the analogies between Java and C++ are not nearly as strong as those between Java and C. C++ programmers may have an easier time with this chapter than C programmers will, but they should still read it carefully and try not to form preconceptions about Java based on their knowledge of C++.

As we'll see, Java supports garbage collection and dynamic method lookup. This actually makes it a closer relative, beneath its layer of C-like syntax, to languages like Smalltalk than to C++.

### 1.1 Classes and Objects

A class is a collection of data and methods that operate on that data. The data and methods, taken together, usually serve to define the contents and capabilities of some kind of object. A method is the object-oriented term for a procedure or a function.

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type it can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class.

A class is declared using **class** keyword. Class contains both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**. Thus, the instance variables and methods are known as class members. **class** is also known as a user defined datatype.

**A class and an object can be related as follows:** Consider an ice tray(like of cube shape) as a class. Then ice cubes can be considered as the objects which is a blueprint of its class i.e of ice tray.

C++ programmers should note that methods go inside the class definition in Java, not outside with the :: operator as they usually do in C++.

***Defining Class***

Java is a Purely Object Oriented Programming Language. Classes are the fundamental building blocks of a Java program. A class can contain any of the following variable types Local variables, Instance variables, Class variables

A class can have any number of methods to access the value of various kinds of methods.

```
class ClassName
{
  // variables
  // methods
}
```

You can define an Employee class as follows:

```
class Employee
{
 int age;
double salary;
}
```

### 1.2 What Is an Object?

An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. Object determines the behavior of the class. When you send a message to an object, you are asking the object to invoke or execute one of its methods.

From a programming point of view, an object can be a data structure, a variable or a function. It has a memory location allocated. The object is designed as class hierarchies.

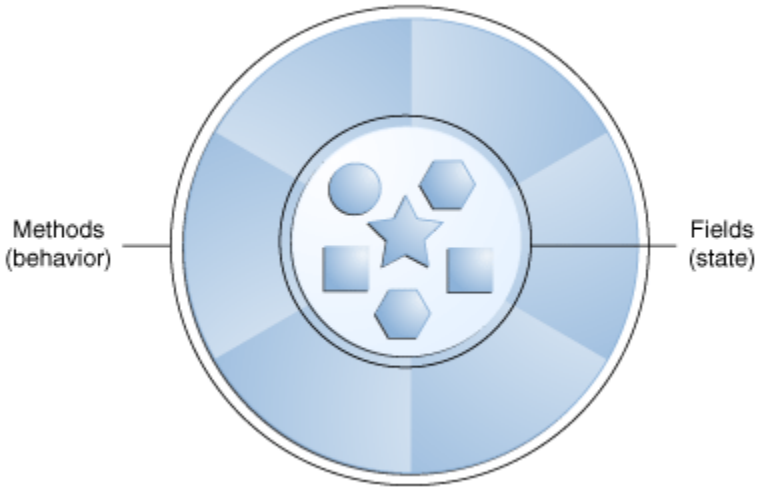
## Syntax

```
ClassName ReferenceVariable = new ClassName();
```

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

**Object Definitions:**

- An object is *a real-world entity*.
- An object is *a basic runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.



A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

```
        //Java Program to demonstrate having the main method in
        //another class
        //Creating Student class.
class Student
{
 int id;
 String name;
}
        //Creating another class TestStudent1 which contains the main method
        class TestStudent1
        {
         public static void main(String args[])
        {
          Student s1=new Student();
          System.out.println(s1.id);
          System.out.println(s1.name);
         }
        }
```

Output:

```
0
null
```

### Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

```
class Employee{
   int id;
   String name;
   float salary;
   void insert(int i, String n, float s) {
       id=i;
       name=n;
```

```
    salary=s;
   }
   void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
   Employee e1=new Employee();
   Employee e2=new Employee();
   Employee e3=new Employee();
   e1.insert(101,"ajeet",45000);
   e2.insert(102,"irfan",25000);
   e3.insert(103,"nakul",55000);
   e1.display();
   e2.display();
   e3.display();
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

## Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
```

```
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

```
55
45
```

**Object and Class Example: Initialization through method**

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insert Recordmethod(). Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n)
{
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
```

```
}
```

Output:

```
111 Karan
222 Aryan
```

## 2. Constructor

A java constructor has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type, since constructors never return a value. Constructors may include parameters of various types. It is called when an instance of the object is created, and memory is allocated for the object. It is a special type of method which is used to initialize the object.

**When is a constructor called?**

Every time an object is created using new () keyword, at least one constructor is called. It calls a default constructor.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

*Syntax*

```
class className {
 same  className(inputParameters)
{
   //set fields and run methods needed on instantiation here
   }
}
```

When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition. Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.

The only action taken by the implicit default constructor is to call the superclass constructor using the super() call. Constructor arguments provide you with a way to provide parameters for the initialization of an object.

Below is an example of a cube class

```
public class Cube1
{
    int length;
    int breadth;
    int height;
    public int getVolume() {
        return (length * breadth * height);
    }
    Cube1(int l, int b, int h)
 {
        length = l;
        breadth = b;
        height = h;
    }
    public static void main(String[] args) {
        Cube1 cubeObj1, cubeObj2;
```

```
        cubeObj1 = new Cube1();
        cubeObj2 = new Cube1(10, 20, 30);
        System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());
    }
}
```

## 3. Method Overloading

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

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

let's get back to the point, when user say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

**Three ways to overload a method**

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.

```
add(int, float)
add(float, int)
```

## Method Overloading: changing no. of arguments

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

*Output:*

```
22
33
```

## Method Overloading: changing data type of arguments

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

*Output:*

```
22
24.9
```

## 4. Static Members in java

The class level members which have static keyword in their definition are called static members.

Types of Static Members:

Java supports four types of static members

1. Static Variables
2. Static Blocks
3. Static Methods
4. Main Method (static method)

- All static members are identified and get memory location at the time of class loading by default by JVM in Method area.
- Only static variables get memory location, methods will not have separate memory location like variables.
- Static Methods are just identified and can be accessed directly without object creation.

### Java – static variable with example

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory. Like variables we can have static block, static method and static class.

**Static variable Syntax**

static keyword followed by data type, followed by variable name.

```
static data_type variable_name;
```

As we mentioned above that the static variables are shared among all the instances of the class, they are useful when we need to do memory management. In some cases we want to have a common value for all the instances like global variable then it is much better to declare them static as this can save memory (because only single copy is created for static variables).

lets understand this with an example:

```
class VariableDemo {

  static int count=0;
  public void increment()
  {
     count++;
  }
  public static void main(String args[])   {
     VariableDemo obj1=new VariableDemo();
     VariableDemo obj2=new VariableDemo();
     obj1.increment();
     obj2.increment();
     System.out.println("Obj1: count is="+obj1.count);
     System.out.println("Obj2: count is="+obj2.count);
  }
}
```

***Output:***

```
Obj1: count is=2
Obj2: count is=2
```

As you can see in the above example that both the objects are sharing a same copy of static variable that's why they displayed the same value of count.

**Static Variable can be accessed directly in a static method**

```
class JavaExample{
  static int age;
  static String name;
  //This is a Static Method
  static void disp(){
     System.out.println("Age is: "+age);
     System.out.println("Name is: "+name);
  }
  // This is also a static method
  public static void main(String args[])
  {
              age = 30;
              name = "Steve";
     disp();
  }
}
```

Output:

```
Age is: 30
Name is: Steve
```

## 5. Nesting of method

The method of class can be called by an object of that class using dot operator. However there is an exception to this. A method can be called by using its name by another method of the same class. This is known as nesting of method

This is a Java Program to Show the Nesting of Methods. When a method in java calls another method in the same class, it is called Nesting of methods.

Enter length, breadth and height as input. After that we first call the volume method. From volume method we call area method and from area method we call perimeter method. Hence we get perimeter, area and volume of cuboid as output.

Here is the source code of the Java Program to Show the Nesting of Methods. The Java program is successfully compiled and run on a Windows system. The program output is also shown below.

```
import java.util.Scanner;
public class Nesting_Methods
{
int perimeter(int l, int b)
        {
            int pr = 12 * (l + b);
             return pr;
        }
        int area(int l, int b)
        {
            int pr = perimeter(l, b);
            System.out.println("Perimeter:"+pr);
            int ar = 6 * l * b;
            return ar;
        }
        int volume(int l, int b, int h)
    {
          int ar = area(l, b);
          System.out.println("Area:"+ar);
          int vol ;
          vol = l * b * h;
          return vol;
        }
        public static void main(String[] args)
        {
          Scanner s = new Scanner(System.in);
          System.out.print("Enter length of cuboid:");
          int l = s.nextInt();
          System.out.print("Enter breadth of cuboid:");
          int b = s.nextInt();
          System.out.print("Enter height of cuboid:");
          int h = s.nextInt();
          Nesting_Methods obj = new Nesting_Methods();
          int vol = obj.volume(l, b, h);
          System.out.println("Volume:"+vol);
```

```
    }
}
```

**Output:**

```
Enter length of cuboid:5
Enter breadth of cuboid:6
Enter height of cuboid:7
Perimeter:132
Area:180
Volume:210
```

## 6. Inheritance in Java

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

- **Super Class/Parent Class:** Super class is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

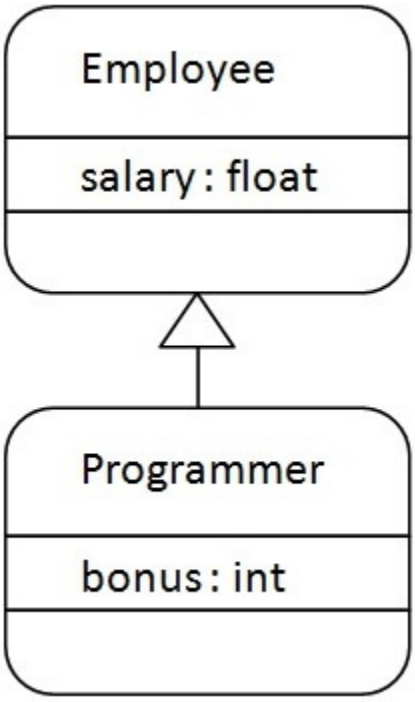
**The syntax of Java Inheritance**

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

**Inheritance Example**




As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
```

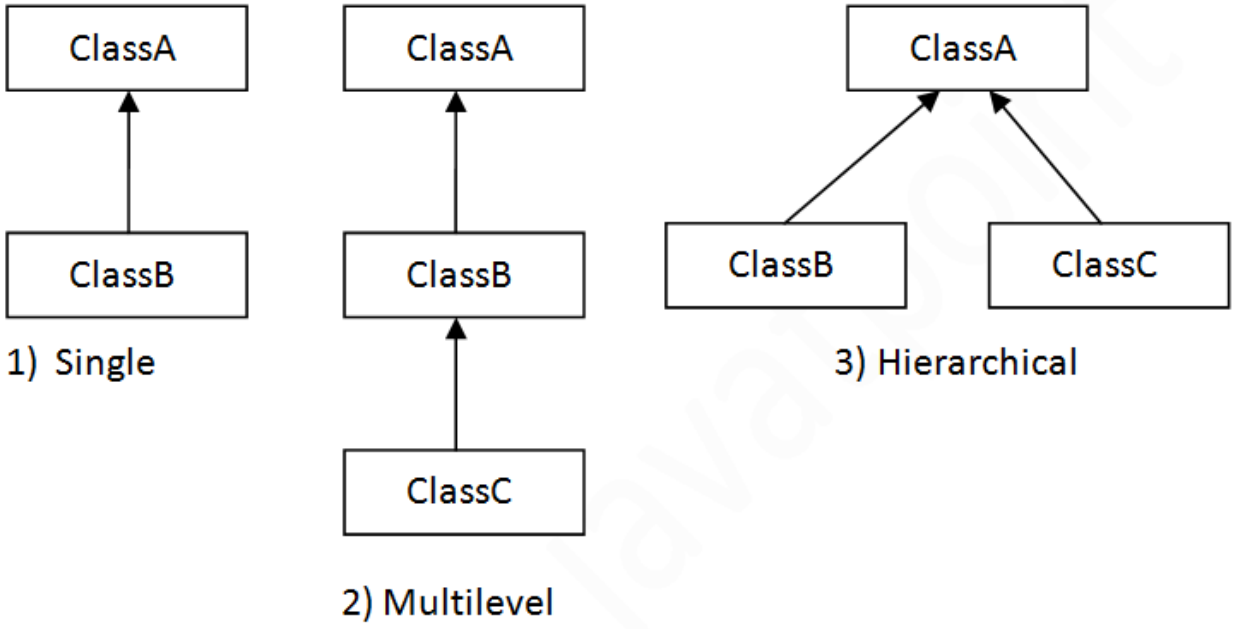
```
    int bonus=10000;  }
    public static void main(String args[]){
      Programmer p=new Programmer();
      System.out.println("Programmer salary is:"+p.salary);
      System.out.println("Bonus of Programmer is:"+p.bonus);
    }
    }
```

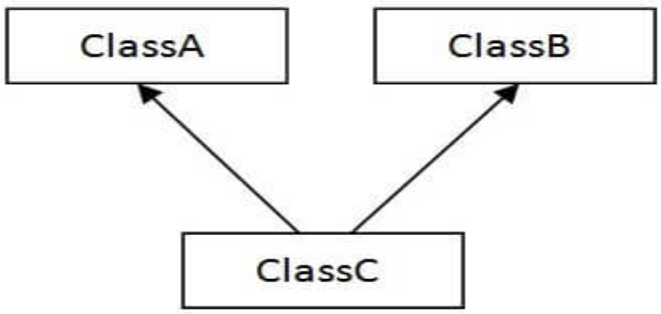
Output

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

## Types of inheritance in java

On the basis of class, there can be four types of inheritance in java: single, multilevel and hierarchical, multiple inheritances.

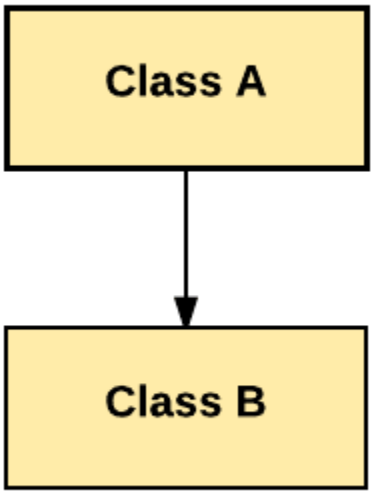





4) Multiple

1. **Single Inheritance:**

   In Single Inheritance one class extends another class (one class only).



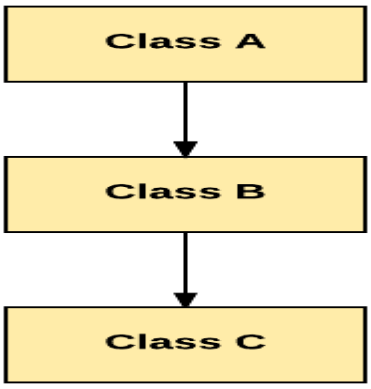

In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

*class Doctor {*
*void Doctor_Details() {*

```
  System.out.println("Doctor Details...");
 }
}

class Surgeon extends Doctor {
 void Surgeon_Details() {
  System.out.println("Surgen Detail...");
 }
}

public class Hospital {
 public static void main(String args[]) {
  Surgeon s = new Surgeon();
  s.Doctor_Details();
  s.Surgeon_Details();
 }
}
```

## 2. Multilevel Inheritance:

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.




As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

Multilevel Inheritance example program in Java

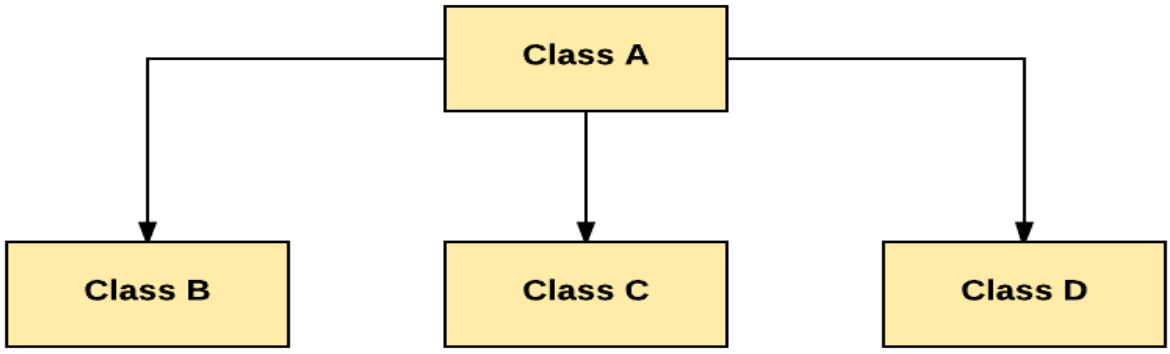
```
class Car
{
  public Car()
  {
        System.out.println("Class Car");
  }
  public void vehicleType()
  {
        System.out.println("Vehicle Type: Car");
  }
}
class Maruti extends Car
{
  public Maruti()
  {
        System.out.println("Class Maruti");
  }
  public void brand()
  {
        System.out.println("Brand: Maruti");
  }
  public void speed()
  {
        System.out.println("Max: 90Kmph");
  }
}
 class Maruti800 extends Maruti
{

  public Maruti800()
  {
        System.out.println("Maruti Model: 800");
  }
  public void speed()
  {
        System.out.println("Max: 80Kmph");
  }
  public static void main(String args[])
  {
         Maruti800 obj=new Maruti800();
         obj.vehicleType();
         obj.brand();
         obj.speed();
  }
}
```

*Output:*

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max: 80Kmph
```

3. **Hierarchical Inheritance:**

   In Hierarchical Inheritance, one class is inherited by many sub classes.




   As per above example, Class B, C, and D inherit the same class A.

   Hierarchical Inheritance example program in Java

```
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");
  }
}
```

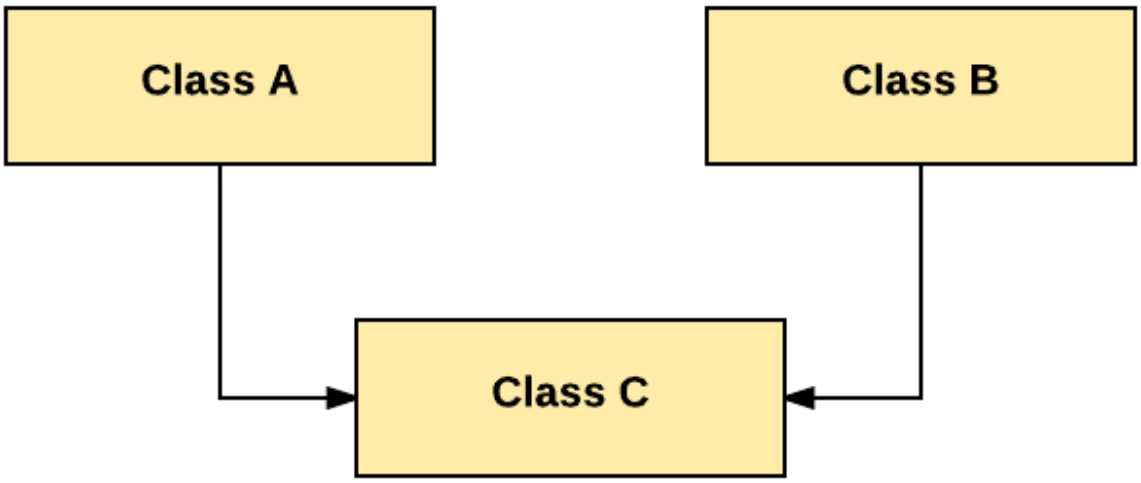
```
class C extends A
{
  public void methodC()
  {
     System.out.println("method of Class C");
  }
}
class D extends A
{
  public void methodD()
  {
     System.out.println("method of Class D");
  }
}
class JavaExample
{
  public static void main(String args[])
  {
     B obj1 = new B();
     C obj2 = new C();
     D obj3 = new D();
     //All classes can access the method of class A
     obj1.methodA();
     obj2.methodA();
     obj3.methodA();
  }
}
```

*Output:*

```
method of Class A
method of Class A
method of Class A
```

4. **Multiple Inheritance:**

   In Multiple Inheritance, one class extending more than one class. Java does not support multiple inheritance.




As per above diagram, Class C extends Class A and Class B both.

## 7. Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

In another form declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
```

```
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

*Output:*

```
Bike is running safely
```

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its super class.
- Method overriding is used for runtime polymorphism

## 8. Final (Keyword) in Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

### 1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

**Example of final variable**

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class

Output: Compile Time Error
```

## 2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
Output:Compile Time Error
```

**3) Java final class**

If you make any class as final, you cannot extend it.

*Example of final class*

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

*Output: Compile Time Error*

## 9. Summary

- Java is true objected oriented programming language and therefore the underlying structure of all Java programs is class.
- A class is a collection of data and methods that operate on that data
- Objects are key to understanding object-oriented technology.
- A class can contain three variable types. Local variables, Instance variables, Class variables
- In Java there are four types of Access Modifiers private protected, default, public
- In Java user have facility to declare a class is the subclass of another class within The Class Declaration
- A java constructor has the same name as the name of the class to which it belongs.
- Like methods, constructors can also be overloaded. Since the constructors in a class all have the same name as the class, />their signatures are differentiated by their parameter lists
- A subclass inherits all of the member variables within its superclass that are accessible to that subclass.
- A java constructor has the same name as the name of the class to which it belongs
- Inheritance is a compile-time mechanism in Java that allows you to extend a class
- There are four types of inheritance namely Single Inheritance, Multi Level Inheritance, Multiple Inheritance, Hierarchical Inheritance
- Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface
- 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 sub-classed
- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.
- Final in java is very important keyword and can be applied to class, method, and variables in Java.

## Chapter 4

# Interfaces- Multiple Interfaces

### 1. Introduction

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

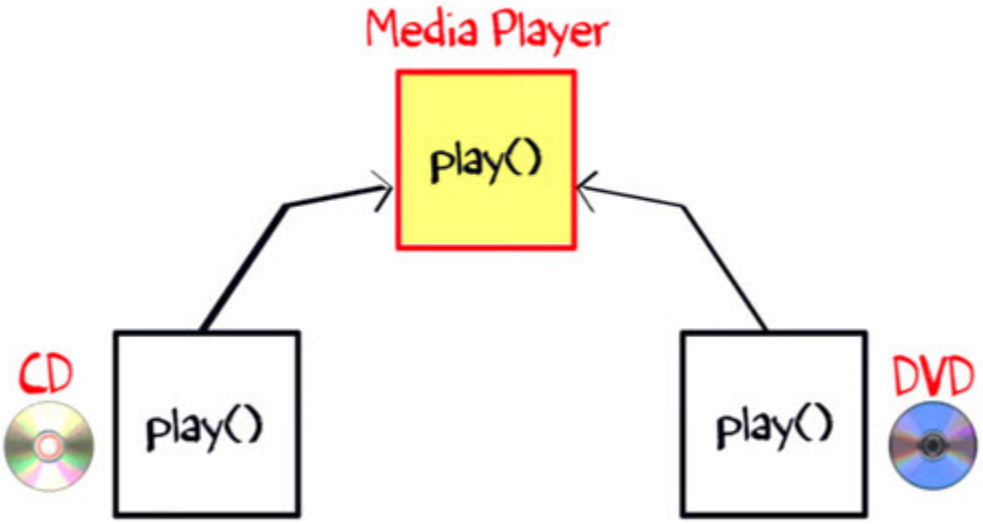
In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

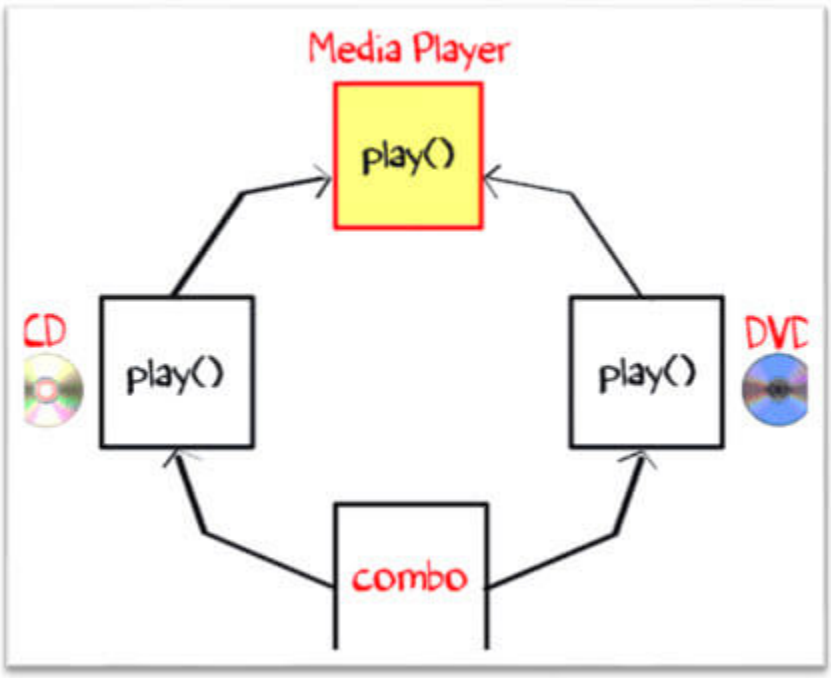
It cannot be instantiated just like the abstract class.

**Why is an Interface required?**

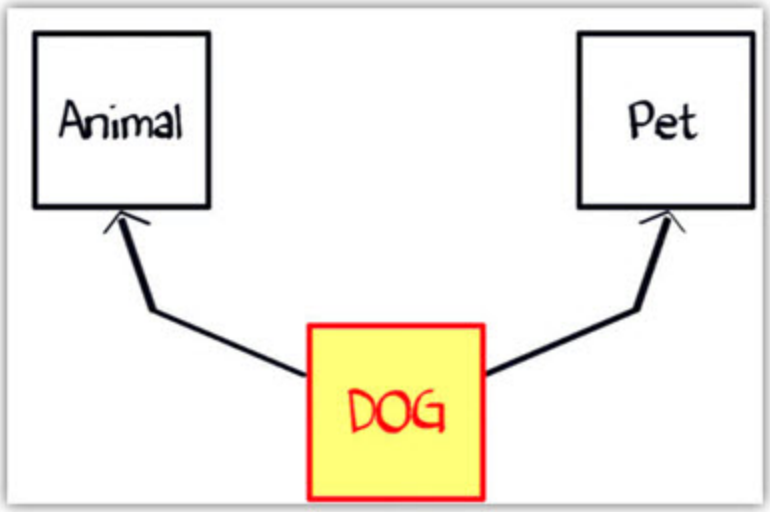
To understand the concept of Java Interface better, let see an example. The class "Media Player" has two subclasses: CD player and DVD player. Each having its unique implementation method to play music.




Another class "Combo drive" is inheriting both CD and DVD (see image below). Which play method should it inherit? This may cause serious design issues. And hence, Java does not allow multiple inheritance.



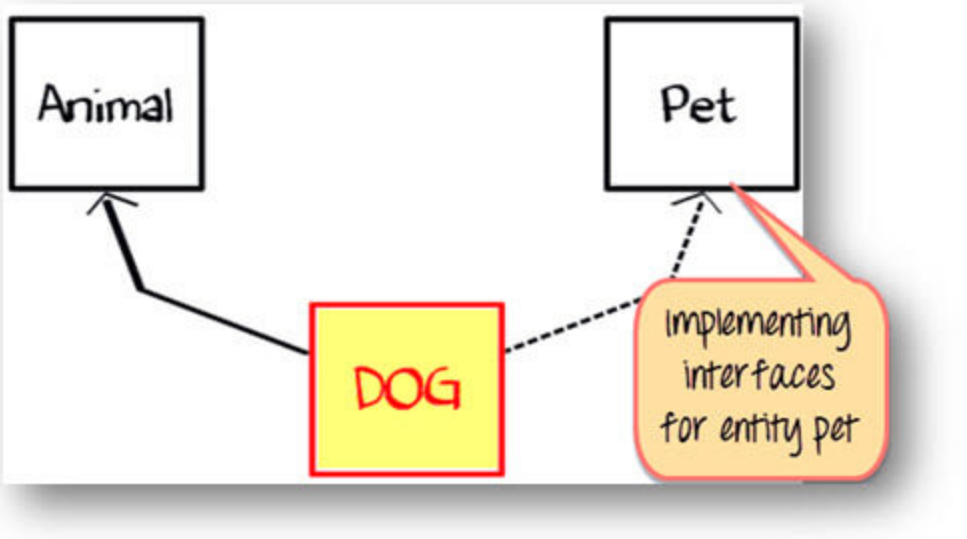

Suppose you have a requirement where class "dog" inheriting class "animal" and "Pet" (see image below). But you cannot extend two classes in Java. So what would you do? The solution is Interface.




The rulebook for interface says,

- An interface is 100% abstract class and has only abstract methods.
- Class can implement any number of interfaces.

Class Dog can extend to class "Animal" and implement interface as "Pet".




**How to declare an interface?**

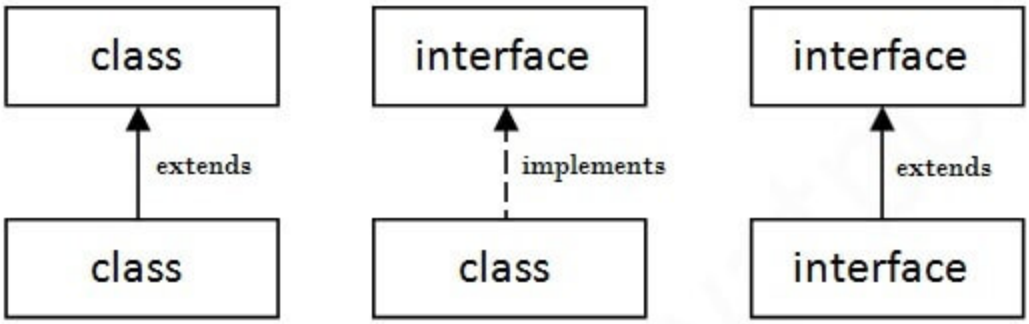
An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

**Syntax:**

```
interface <interface_name>
{

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.




## Example of an Interface in Java

```
interface MyInterface
{
```

```
  /* compiler will treat them as:
   * public abstract void method1();
   * public abstract void method2();
   */
  public void method1();
  public void method2();
}
class Demo implements MyInterface
{
  /* This class must have to implement both the abstract methods
   * else you will get compilation error
   */
  public void method1()
  {
        System.out.println("implementation of method1");
  }
  public void method2()
  {
        System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
        MyInterface obj = new Demo();
        obj.method1();
  }
}
```

Output:

*implementation of method1*

**Points to remember**

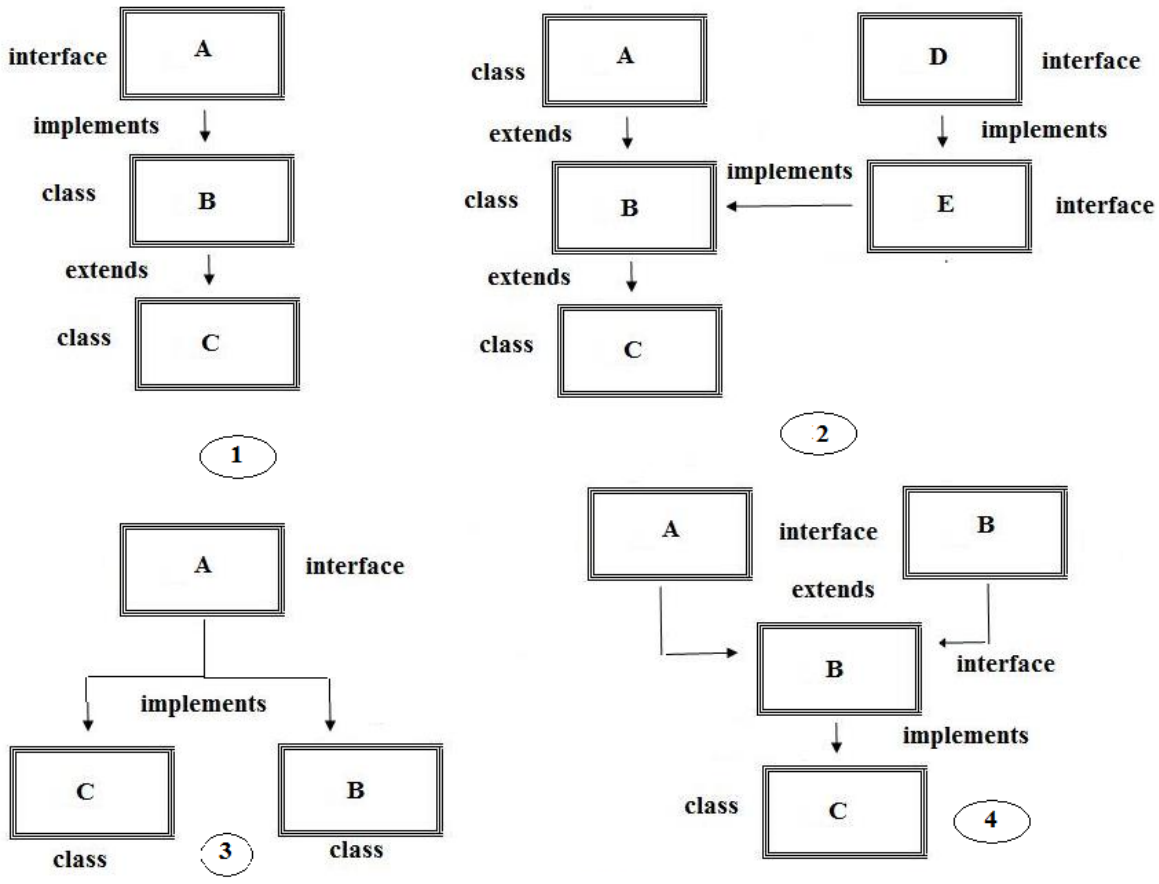
- The class which implements the interface needs to provide functionality for the methods declared in the interface
- All methods in an interface are implicitly public and abstract
- An interface cannot be instantiated
- An interface reference can point to objects of its implementing classes
- An interface can extend from one or many interfaces. A class can extend only one class but implement any number of interfaces

## 2. Implementing Interfaces

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
interface Pet{
  public void test();
}
class Dog implements Pet{
   public void test(){
     System.out.println("Interface Method Implemented");
  }
   public static void main(String args[]){
     Pet p = new Dog();
     p.test();
  }
}
```

## Various forms of interface implementation



**Difference between Class and Interface**

| Class | Interface |
|---|---|
| In class, you can instantiate variable and create an object. | In an interface, you can't instantiate variable and create an object. |
| Class can contain concrete(with implementation) methods | The interface cannot contain concrete(with implementation) methods |
| The access specifiers used with classes are private, protected and public. | In Interface only one specifier is used- Public. |

## 3. Multiple inheritances in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.


**Multiple Inheritance in Java**

```
interface Printable
{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable
```

```
{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
Output:Hello
      Welcome
```

## 4. Summary

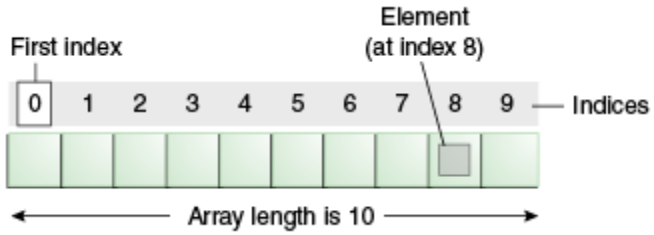
- In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, and nested types
- The Java programming language does not permit multiple inheritance, but interfaces provide an alternative.
- Once a class implements an interface you can use an instance of that class as an instance of that interface
- It is possible for an interface to inherit from another interface, just like classes can inherit from other classes
- Interfaces are a way to achieve polymorphism in Java

## Chapter 5

# Array and String

## 1. Introduction

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. This section discusses arrays in greater detail.




*An array of ten elements*

Each item in an array is called an element, and each element is accessed by its numerical index. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

### 1.1 Declaring a Variable to Refer to an Array

First let us get in to **declaration of array which holds primitive types**. The declaration of array states the type of the element that the array holds followed by the identifier and square braces which indicates the identifier is array type.

Declaring an array which holds elements of integer type.

```
int aiMyArray[];
```

The above statement creates an array reference on the stack.

**Different way of declaring an array –**

Both the below statements are valid and same!!

```
int []aiMyArray;
int aiMyArray[];
```

Similarly, you can declare arrays of other types:

byte[] anArrayOfBytes;

short[] anArrayOfShorts;

long[] anArrayOfLongs;

float[] anArrayOfFloats;

double[] anArrayOfDoubles;

boolean[] anArrayOfBooleans;

char[] anArrayOfChars;

String[] anArrayOfStrings;

You can also place the square brackets after the array's name:

// this form is discouraged

float anArrayOfFloats[];

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

The following program, ArrayDemo, creates an array of integers, puts some values in it, and prints each value to standard output.

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

```
{
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // etc.
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: "
                        + anArray[0]);
        System.out.println("Element at index 1: "
                        + anArray[1]);
        System.out.println("Element at index 2: "
                        + anArray[2]);
        System.out.println("Element at index 3: "
                        + anArray[3]);
        System.out.println("Element at index 4: "
                        + anArray[4]);
        System.out.println("Element at index 5: "
                        + anArray[5]);
        System.out.println("Element at index 6: "
                        + anArray[6]);
        System.out.println("Element at index 7: "
                        + anArray[7]);
        System.out.println("Element at index 8: "
                        + anArray[8]);
        System.out.println("Element at index 9: "
                        + anArray[9]);
    }
}
```

*The output from this program is:*

```
Element at index 0: 100
Element at index 1: 200
```

*Element at index 2: 300*
*Element at index 3: 400*
*Element at index 4: 500*
*Element at index 5: 600*
*Element at index 6: 700*
*Element at index 7: 800*
*Element at index 8: 900*
*Element at index 9: 1000*

In a real-world programming situation, you'd probably use one of the supported looping constructs to iterate through each element of the array, rather than write each line individually as shown above. However, this example clearly illustrates the array syntax. You'll learn about the various looping constructs (for, while, and do-while) in the Control Flow section.

## 1.2 Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

```
// create an array of integers

anArray = new int[10];
```

If this statement were missing, the compiler would print an error like the following, and compilation would fail:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element

anArray[1] = 200; // initialize second element

anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
```

System.out.println("Element 2 at index 1: " + anArray[1]);

System.out.println("Element 3 at index 2: " + anArray[2]);

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between { and }.

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<3;i++)//length is the property of array
System.out.println(a[i]);
}}
```

*Output:*

```
33
3
4
5
```

Java program to sort array elements using bubble sort. Bubble sort algorithm is known as the simplest sorting algorithm.

In bubble sort algorithm, array is traversed from first element to last element. Here, current element is compared with the next element. If current element is greater than the next element, it is swapped.

```
class BubbleSortExample
{
  public static void main(String[] args)
{
            int arr[] ={3,60,35,2,45,320,5};

            System.out.println("Array Before Bubble Sort");
            for(int i=0; i < arr.length; i++){
                  System.out.print(arr[i] + " ");
            }
int temp = 0;
      for(int i=0; i <8; i++){
            for(int j=1; j <7); j++){
                  if(arr[j-1] > arr[j]){
                        //swap elements
                        temp = arr[j-1];
                        arr[j-1] = arr[j];
                        arr[j] = temp;
                  }

            }
      }

            System.out.println("Array After Bubble Sort");
            for(int i=0; i < arr.length; i++){
                  System.out.print(arr[i] + " ");
            }

      }
}
```

***Output:***

*Array Before Bubble Sort*
*3 60 35 2 45 320 5*
*Array After Bubble Sort*
*2 3 5 35 45 60 320*

## 2. Multidimensional Array in Java

A multidimensional array is an array containing one or more arrays. In such case, data is stored in row and column based index (also known as matrix form).

**Syntax to Declare Multidimensional Array in Java**

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

**Example to instantiate Multidimensional Array in Java**

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

**Example to initialize Multidimensional Array in Java**

arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

*Output:*

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

## Addition of 2 Matrices in Java

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

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
```

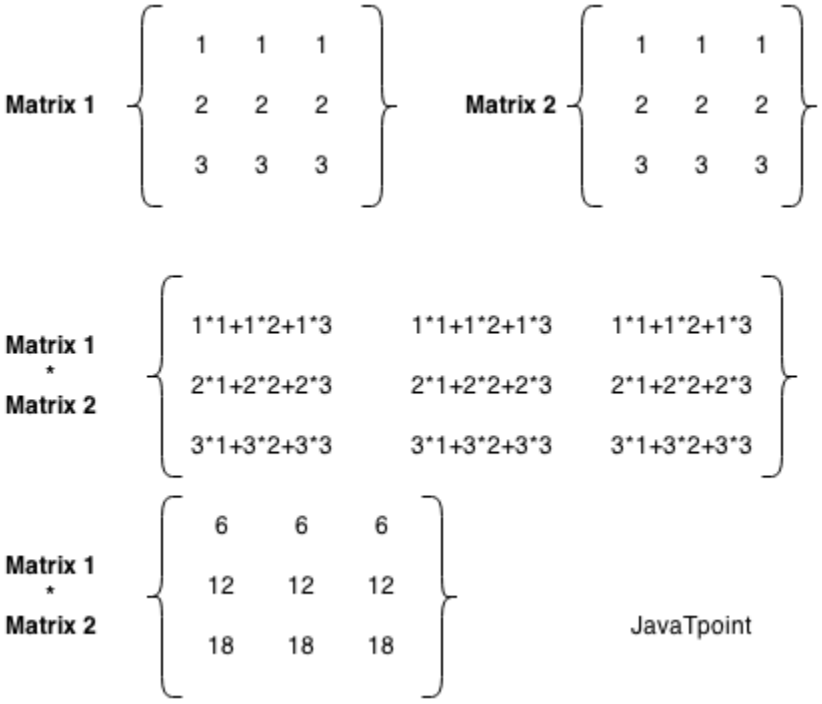
```
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```

***Output:***

```
2 6 8
6 8 10
```

## Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.




Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
```

```
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3];  //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}}
```

***Output:***

```
6 6 6
12 12 12
18 18 18
```

## 3. Java String Array

Java String array is used to hold fixed number of Strings. String array is very common in simple java programs. Java String array is basically an array of objects.

There are two ways to declare string array – declaration without size and declare with size.

### Java String Array Declaration

String[] strArray; //declare without size

String[] strArray1 = new String[3]; //declare with size

**Java String Array Initialization**

String[] strArray1 = new String[] {"A","B","C"};
String[] strArray2 = {"A","B","C"};

//initialization after declaration
String[] strArray3 = new String[3];
strArray3[0] = "A";
strArray3[1] = "B";
strArray3[2] = "C";

```
class JavaStringArrayExample {

  public static void main(String args[]) {
  // declare a string array with no initial size
  // String[] schoolbag;

  // declare string array and initialize with values in one step
  String[] schoolbag = { "Books", "Pens", "Pencils", "Notebooks" };

  // print the third element of the string array
  System.out.println("The third element is: " + schoolbag2[2]);
  }
 }
```

***Output***

*The third element is: Pencils*

***The second way uses the enhanced*** `for loop` ***example, which was introduced in Java*** .

```
class JavaStringArrayExample
{

 public static void main(String args[]) {

  String[] schoolbag = { "Books", "Pens", "Pencils", "Notebooks" };

  // iterate all the elements of the array
  int size = schoolbag.length;
  System.out.println("The size of array is: " + size);

  for (int i = 0; i < size; i++) {
   System.out.println("Index[" + i + "] = " + schoolbag[i]);
  }
 }
}
```

***Output:***

```
The size of array is: 4

Index[0] = Books

Index[1] = Pens

Index[2] = Pencils
```

## 4. Java String – String Functions

- **Java String length ()**: The Java String length () method tells the length of the string. It returns count of total number of characters present in the String. For example:

```
public class Example{
public static void main(String args[]){
String s1="hello";
String s2="whatsup";
System.out.println("string length is: "+s1.length());
System.out.println("string length is: "+s2.length());
}}
```

Here, String length() function will return the length 5 for s1 and 7 for s2 respectively.

- **Java String compareTo()**: The Java String compareTo() method compares the given string with current string. It is a method of *'Comparable'* interface which is implemented by String class. Don't worry, we will be learning about String interfaces later. It either returns positive number, negative number or 0. For example:

```
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="hemlo";
String s4="flag";
System.out.println(s1.compareTo(s2)); // 0 because both are equal
System.out.println(s1.compareTo(s3)); //-1 because "l" is only one time lower than "m"
System.out.println(s1.compareTo(s4)); // 2 because "h" is 2 times greater than "f"
}}
```

This program shows the comparison between the various string. It is noticed that

if s1 > s2, it returns a positive number

if s1 < s2, it returns a negative number

if s1 == s2, it returns 0

- **Java String concat() :** The Java String concat() method combines a specific string at the end of another string and ultimately returns a combined string. It is like appending another string. For example:

```
public class ConcatExample{
```

```
public static void main(String args[]){
String s1="hello";
s1=s1.concat("how are you");
System.out.println(s1);
}}
```

The above code returns “hellohow are you”.

- **Java String IsEmpty()** : This method checks whether the String contains anything or not. If the java String is Empty, it returns true else false. For example:

```
public class IsEmptyExample{
public static void main(String args[]){
String s1="";
String s2="hello";
System.out.println(s1.isEmpty());      // true
System.out.println(s2.isEmpty());      // false
}}
```

- **Java String Trim()** : The java string trim() method removes the leading and trailing spaces. It checks the unicode value of space character (‘\u0020’) before and after the string. If it exists, then removes the spaces and return the omitted string. For example:

```
public class StringTrimExample{
public static void main(String args[]){
String s1="  hello   ";
System.out.println(s1+"how are you");       // without trim()
System.out.println(s1.trim()+"how are you"); // with trim()
}}
```

In the above code, the first print statement will print “hello how are you” while the second statement will print “hellohow are you” using the trim() function.

- **Java String toLowerCase()** : The java string toLowerCase() method converts all the characters of the String to lower case. For example:

```
public class StringLowerExample{
```

```
public static void main(String args[]){
String s1="HELLO HOW Are You?";
String s1lower=s1.toLowerCase();
System.out.println(s1lower);}
}
```

The above code will return "hello how are you".

- **Java String toUpper()** : The Java String toUpperCase() method converts all the characters of the String to upper case. For example:

```
public class StringUpperExample{
public static void main(String args[]){
String s1="hello how are you";
String s1upper=s1.toUpperCase();
System.out.println(s1upper);
}}
```

The above code will return "HELLO HOW ARE YOU".

- **Java String ValueOf()**: This method converts different types of values into string.Using this method, you can convert int to string, long to string, Boolean to string, character to string, float to string, double to string, object to string and char array to string. The signature or syntax of string valueOf() method is given below:

public static String valueOf(boolean b)

public static String valueOf(char c)

public static String valueOf(char[] c)

public static String valueOf(int i)

public static String valueOf(long l)

public static String valueOf(float f)

public static String valueOf(double d)

public static String valueOf(Object o)

Let's understand this with a programmatic example:

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

```
int value=20;
String s1=String.valueOf(value);
System.out.println(s1+17);       //concatenating string with 10
}}
```

In the above code, it concatenates the Java String and gives the output – 2017.

- **Java String replace()**: The Java String replace() method returns a string, replacing all the old characters or CharSequence to new characters. There are 2 ways to replace methods in a Java String.

  ```
  public class ReplaceExample1{
  public static void main(String args[]){
  String s1="hello how are you";
  String replaceString=s1.replace('h','t');
  System.out.println(replaceString); }}
  ```

  In the above code, it will replace all the occurrences of 'h' to 't'. Output to the above code will be "tello tow are you". Let's see the another type of using replace method in java string:

- **Java String replace(CharSequence target, CharSequence replacement) method :**

  ```
  public class ReplaceExample2{
  public static void main(String args[]){
  String s1="Hey, welcome to Edureka";
  String replaceString=s1.replace("Edureka","Brainforce");
  System.out.println(replaceString);
  }}
  ```

  In the above code, it will replace all occurrences of "Edureka" to "Brainforce". Therefore, the output would be " Hey, welcome to Brainforce".

- **Java String contains()** :The java string contains() method searches the sequence of characters in the string. If the sequences of characters are found, then it returns true otherwise returns false. For example:

  ```
  class ContainsExample{
  public static void main(String args[]){
  String name=" hello how are you doing?";
  ```

```
System.out.println(name.contains("how are you"));  // returns true
System.out.println(name.contains("hello"));        // returns true
System.out.println(name.contains("fine"));         // returns false
}}
```

In the above code, the first two statements will return true as it matches the String whereas the second print statement will return false because the characters are not present in the string.

- **Java String equals()** : The Java String equals() method compares the two given strings on the basis of content of the string i.e Java String representation. If all the characters are matched, it returns true else it will return false. For example:

```
public class EqualsExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="hi";
System.out.println(s1.equalsIgnoreCase(s2));   // returns true
System.out.println(s1.equalsIgnoreCase(s3));   // returns false
}
}
```

- **Java String equalsIgnoreCase():** This method compares two string on the basis of content but it does not check the case like equals() method. In this method, if the characters match, it returns true else false. For example:

```
public class EqualsIgnoreCaseExample{
public static void main(String args[]){
String s1="hello";
String s2="HELLO";
String s3="hi";
System.out.println(s1.equalsIgnoreCase(s2));   // returns true
System.out.println(s1.equalsIgnoreCase(s3));   // returns false
}}
```

In the above code, the first statement will return true because the content is same irrespective of the case. Then, in the second print statement will return false as the content doesn't match in the respective strings.

- **Java String toCharArray():** This method converts the string into a character array i.e first it will calculate the length of the given Java String including spaces and then create an array of char type with the same content. For example:

```
StringToCharArrayExample{
public static void main(String args[]){
String s1="Welcome to Edureka";
char[] ch=s1.toCharArray();
for(int i=0;i<ch.length;i++){
System.out.print(ch[i]);
}}}
```

The above code will return "Welcome to Edureka".

- **Java StringGetBytes()** : The Java string getBytes() method returns the sequence of bytes or you can say the byte array of the string. For example:

```
public class StringGetBytesExample {
public static void main(String args[]){
String s1="ABC";
byte[] b=s1.getBytes();
for(int i=0;i<b.length;i++){
System.out.println(b[i]);
}
}}
```

In the above code, it will return the value 65,66,67.

- **Java String IsEmpty()** : This method checks whether the String is empty or not. If the length of the String is 0, it returns true else false. For example:

```
public class IsEmptyExample{
public static void main(String args[]) {
String s1="";
String s2="hello";
System.out.prinltn(s1.isEmpty());     // returns true
System.out.prinltn(s2.isEmpty());     // returns false
```

```
}}
```

In the above code, the first print statement will return true as it does not contain anything while the second print statement will return false.

- **Java String endsWith()** : The Java String endsWith() method checks if this string ends with the given suffix. If it returns with the given suffix, it will return true else returns false. For example:

```
public class EndsWithExample{
public static void main(String args[]) {
String s1="hello how are you";
System.out.println(s1.endsWith("u"));       // returns true
System.out.println(s1.endsWith("you"));     // returns true
System.out.println(s1.endsWith("how"));     // returns false
}}
```

## 5. String Methods

Here is the list of methods supported by String class −

| Sr.No. | Method & Description |
|---|---|
| 1 | **char charAt(int index)**<br>Returns the character at the specified index. |
| 2 | **int compareTo(Object o)**<br>Compares this String to another Object. |
| 3 | **int compareTo(String anotherString)**<br>Compares two strings lexicographically. |
| 4 | **int compareToIgnoreCase(String str)** |

| | |
|---|---|
| | Compares two strings lexicographically, ignoring case differences. |
| 5 | **String concat(String str)**<br>Concatenates the specified string to the end of this string. |
| 6 | **boolean contentEquals(StringBuffer sb)**<br>Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer. |
| 7 | **static String copyValueOf(char[] data)**<br>Returns a String that represents the character sequence in the array specified. |
| 8 | **static String copyValueOf(char[] data, int offset, int count)**<br>Returns a String that represents the character sequence in the array specified. |
| 9 | **boolean endsWith(String suffix)**<br>Tests if this string ends with the specified suffix. |
| 10 | **boolean equals(Object anObject)**<br>Compares this string to the specified object. |
| 11 | **boolean equalsIgnoreCase(String anotherString)**<br>Compares this String to another String, ignoring case considerations. |
| 12 | **byte getBytes()**<br>Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array. |
| 13 | **byte[] getBytes(String charsetName)**<br>Encodes this String into a sequence of bytes using the named charset, storing the |

| | |
|---|---|
| | result into a new byte array. |
| 14 | **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**<br><br>Copies characters from this string into the destination character array. |
| 15 | **int hashCode()**<br><br>Returns a hash code for this string. |
| 16 | **int indexOf(int ch)**<br><br>Returns the index within this string of the first occurrence of the specified character. |
| 17 | **int indexOf(int ch, int fromIndex)**<br><br>Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. |
| 18 | **int indexOf(String str)**<br><br>Returns the index within this string of the first occurrence of the specified substring. |
| 19 | **int indexOf(String str, int fromIndex)**<br><br>Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. |
| 20 | **String intern()**<br><br>Returns a canonical representation for the string object. |
| 21 | **int lastIndexOf(int ch)**<br><br>Returns the index within this string of the last occurrence of the specified character. |

| | |
|---|---|
| 22 | **int lastIndexOf(int ch, int fromIndex)**<br><br>Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. |
| 23 | **int lastIndexOf(String str)**<br><br>Returns the index within this string of the rightmost occurrence of the specified substring. |
| 24 | **int lastIndexOf(String str, int fromIndex)**<br><br>Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. |
| 25 | **int length()**<br><br>Returns the length of this string. |
| 26 | **boolean matches(String regex)**<br><br>Tells whether or not this string matches the given regular expression. |
| 27 | **boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)**<br><br>Tests if two string regions are equal. |
| 28 | **boolean regionMatches(int toffset, String other, int ooffset, int len)**<br><br>Tests if two string regions are equal. |
| 29 | **String replace(char oldChar, char newChar)**<br><br>Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. |
| 30 | **String replaceAll(String regex, String replacement**<br><br>Replaces each substring of this string that matches the given regular expression |

| | |
|---|---|
| | with the given replacement. |
| 31 | **String replaceFirst(String regex, String replacement)**<br>Replaces the first substring of this string that matches the given regular expression with the given replacement. |
| 32 | **String[] split(String regex)**<br>Splits this string around matches of the given regular expression. |
| 33 | **String[] split(String regex, int limit)**<br>Splits this string around matches of the given regular expression. |
| 34 | **boolean startsWith(String prefix)**<br>Tests if this string starts with the specified prefix. |
| 35 | **boolean startsWith(String prefix, int toffset)**<br>Tests if this string starts with the specified prefix beginning a specified index. |
| 36 | **CharSequence subSequence(int beginIndex, int endIndex)**<br>Returns a new character sequence that is a subsequence of this sequence. |
| 37 | **String substring(int beginIndex)**<br>Returns a new string that is a substring of this string. |
| 38 | **String substring(int beginIndex, int endIndex)**<br>Returns a new string that is a substring of this string. |
| 39 | **char[] toCharArray()**<br>Converts this string to a new character array. |

| | |
|---|---|
| 40 | **String toLowerCase()**<br><br>Converts all of the characters in this String to lower case using the rules of the default locale. |
| 41 | **String toLowerCase(Locale locale)**<br><br>Converts all of the characters in this String to lower case using the rules of the given Locale. |
| 42 | **String toString()**<br><br>This object (which is already a string!) is itself returned. |
| 43 | **String toUpperCase()**<br><br>Converts all of the characters in this String to upper case using the rules of the default locale. |
| 44 | **String toUpperCase(Locale locale)**<br><br>Converts all of the characters in this String to upper case using the rules of the given Locale. |
| 45 | **String trim()**<br><br>Returns a copy of the string, with leading and trailing whitespace omitted. |
| 46 | **static String valueOf(primitive data type x)**<br><br>Returns the string representation of the passed data type argument. |

## Summary

- An array is a container object that holds a fixed number of values of a single type
- An array is a sequence of variables of the same data type (homogenous).
- The data type can be any of Java's primitive types or a class (user-defined as well).
- Each variable in the array is an element.
- Java, as with most languages, supports multi-dimensional arrays - 1-dimensional, 2-dimensional, 3-dimensions-------n-dimensions.
- Strings are used for storing text.
- A String contains a collection of characters surrounded by double quotes.
- **String is a Final class**; i.e once created the value cannot be altered. Thus String objects are called immutable.
- The Java Virtual Machine(JVM) creates a memory location especially for Strings called **String Constant Pool**. That's why String can be initialized without 'new' keyword.
- String class falls under **java.lang.String hierarchy**. But there is no need to import this class. Java platform provides them automatically.
- String **reference can be overridden but that does not delete the content**;

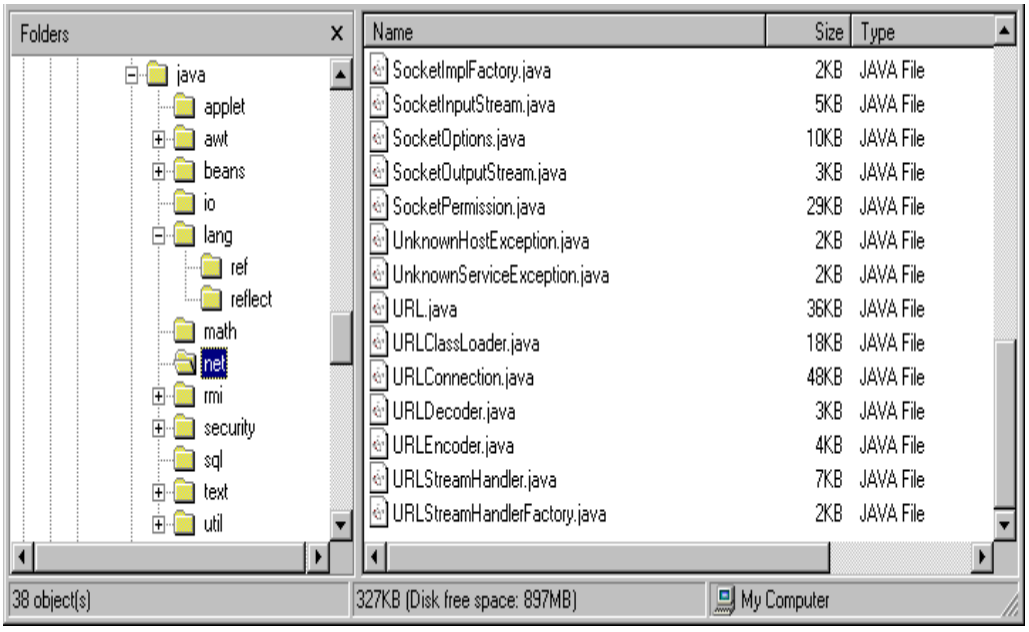
# Chapter 6

# Package and Applets

## 1. Introduction

Many times when we get a chance to work on a small project, one thing we intend to do is to put all java files into one single directory. It is quick, easy and harmless. However if our small project gets bigger, and the number of files is increasing, putting all these files into the same directory would be a nightmare for us. In java we can avoid this sort of problem by using Packages.

Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to. An obvious example of packaging is the JDK package from SUN (java.xxx.yyy) as shown below:




*Basic structure of JDK package*

Basically, files in one directory (or package) would have different functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network. In GUI applications,

it's quite common for us to see a directory with a name "ui" (user interface), meaning that this directory keeps files related to the presentation part of the application. On the other hand, we would see a directory called "engine", which stores all files related to the core functionality of the application instead.

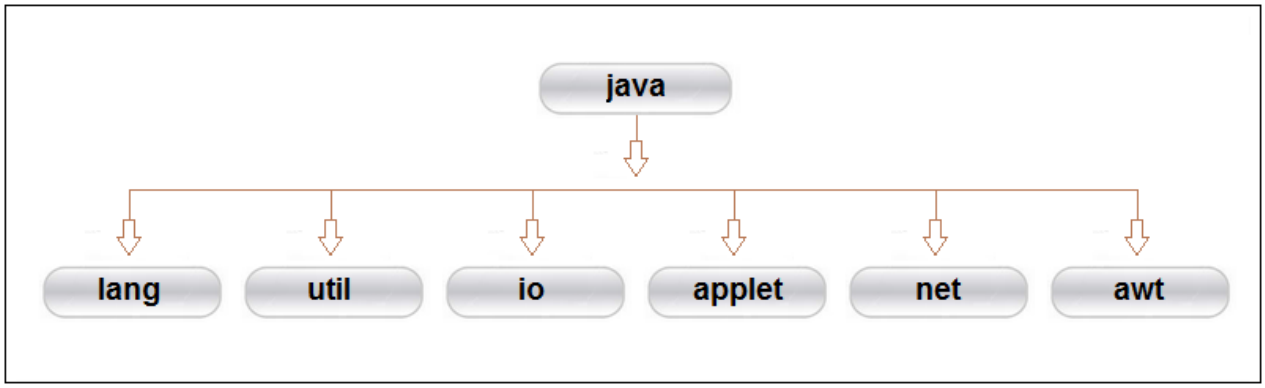
A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

## 2. JAVA API Packages

The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment.

Java **API (Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API. Following figure shows the system packages that are frequently used in the programs.




The Java API, included with the JDK, describes the function of each of its components. In Java programming, many of these components are pre-created and commonly used. Thus, the programmer is able to apply prewritten code via the Java API. After referring to the

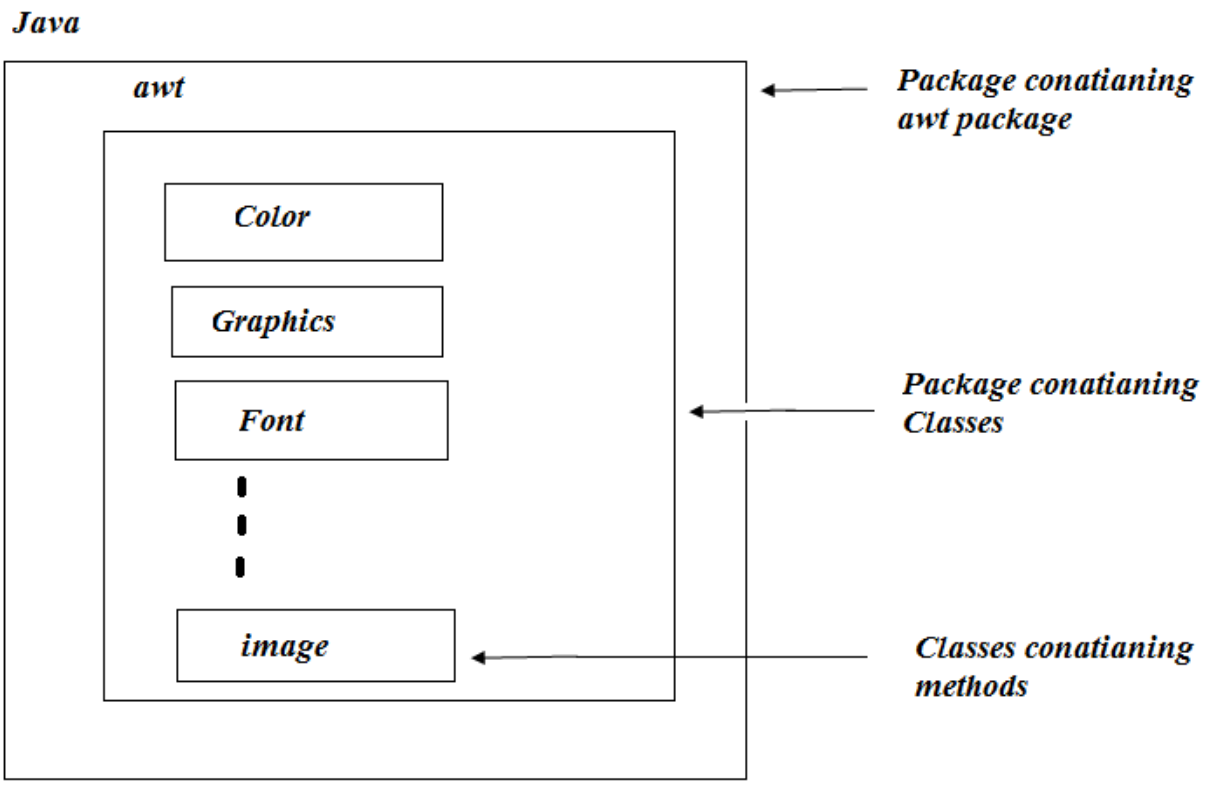
available API classes and packages, the programmer easily invokes the necessary code classes and packages for implementation.

## 3. Java System Packages and Their Classes

| | |
|---|---|
| **java.lang** | Language support classes. They include classes for primitive types, string, math functions, thread and exceptions. |
| **java.util** | Language utility classes such as vectors, hash tables, random numbers, data, etc. |
| **java.io** | Input/output support classes. They provide facilities for the input and output of data. |
| **java.applet** | Classes for creating and implementing applets. |
| **java.net** | Classes for networking. They include classes for communicating with local computers as well as with internet servers. |
| **java.awt** | (Abstract Window Tookit)<br>Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on. |

## 4. Using System Packages

The packages are organized in a hierarchical structure as shown in fig. This shows that the package name **java** contains the package **awt (Abstract Window Toolkit)** consist of various classes for graphical user interface.




*Hierarchical representation of Java.awt package*

**There are two ways of accessing the classes stored in packages.**

The first approach is to use class of the class that we want to use.

The syntax is

***import packagename.classname;***

example

***import java.awt.color;***

import class color and therefore the class name can now be directly used in the program.

The Second approach is to access all class contained in specified packages.

The syntax is

***import packagename.*;***

example

***import java.awt.*;***

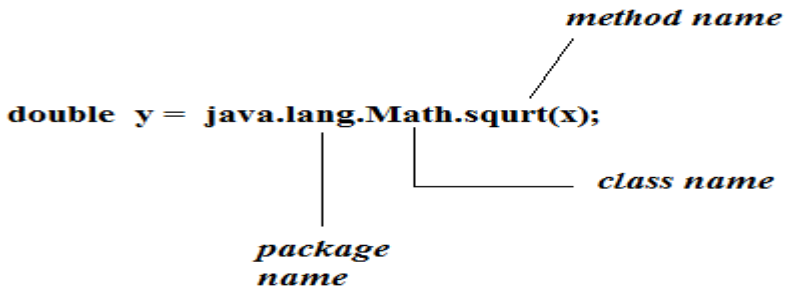
will bring all classes of **java.awt** package.

## 5. Naming Conventions

Package can be named using the Java standard name rules.

1) Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
2) A Java package name consists of a set of name components separated by periods (.).
    - Each name component must be a valid Java identifier.
    - A component name must not start with an upper-case letter.
3) All class name by convention begin with upper case letters

For example



In this example class name Math invoke the squrt ().

Every package name must be unique to make the best use of package. Duplicate names will cause the runtime errors. Java also suggest the use of domain name as prefix to the preferred package names

**pbn.scp.mypackage**

In this example pbn denotes city name and scp denote collage name, we can create a hierarchy of package within package by separating levels of dots.

## 6. JAVA PACAGE

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Packages is a way of grouping a variety of classes or interfaces collectively. The grouping is usually done according to functionality. The Java packages act as containers for Java classes.

Package in java can be categorized in two form, built-in package and user-defined package.

### How to create a package

Suppose we have a file called HelloWorld.java, and we want to put this file in a package world. First thing we have to do is to specify the keyword package with the name of the package we want to use (world in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

*// only comment can be here*

*package world;*

*public class HelloWorld {*

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

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierarchy of the class. In our case, we have the world package, which requires only one directory. So, we create a directory world and put our HelloWorld.java into it.

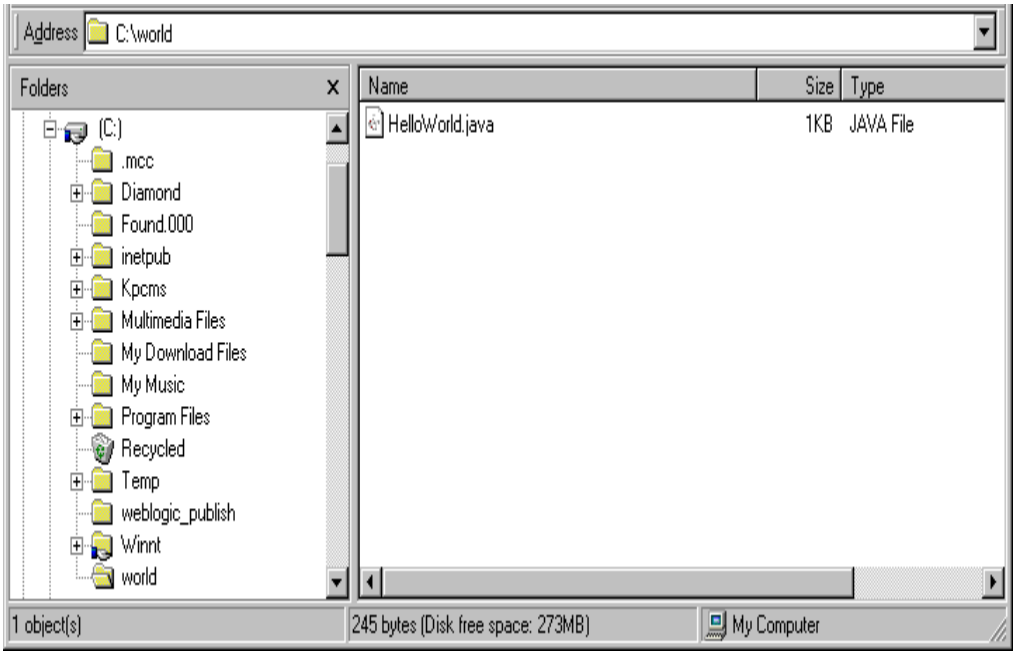



*Figure: HelloWorld in world package (C:\world\HelloWorld.java)*

That's it!!! Right now we have HelloWorld class inside world package. Next, we have to introduce the world package into our CLASSPATH.

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;

3. fully qualified name.

**Using packagename.***

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

***Example of package that import the packagename.****

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

**Using packagename.classname**

If you import package.classname then only declared class of this package will be accessible.

***Example of package by import package.classname***

```
//save by A.java
```

```
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

*Output:Hello*

**Using fully qualified name**

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

***Example of package by import fully qualified name***

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

*Output:Hello*

## 7. Java Applet

A Java applet is a small Java program that can be transferred via the Internet and run by a Java-compatible Web browser. The main difference between Java-based applications and applets is that applets are typically executed in an AppletViewer or Java-compatible Web browser. All applets import the java.awt package.

Applets are used to make the web site more dynamic and entertaining.

- All applets are sub-classes (either directly or indirectly) of *java.applet.Applet* class.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at main() method.
- Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

### Difference between JAVA APPLET and JAVA Application program

- **Definition of Application and Applet** – Applets are feature rich application programs that are specifically designed to be executed within an HTML web document to execute small tasks or just part of it. Java applications, on the other hand, are stand-alone programs that are designed to run on a stand-alone machine without having to use a browser.
- **Execution of Application and Applet**– Applications require main method() to execute the code from the command line, whereas an applet does not require main method() for execution. An applet requires an HTML file before its execution. The browser, in fact, requires a Java plugin to run an applet.
- **Compilation of Application and Applet**–Application programs are compiled using the "javac" command and further executed using the java command. Applet programs, on the other hand, are also compiled using the "javac" command but are executed either by using the "appletviewer" command or using the web browser.

- **Security Access of Application and Applet** – Java application programs can access all the resources of the system including data and information on that system, whereas applets cannot access or modify any resources on the system except only the browser specific services.
- **Restrictions of Application and Applet** – Unlike applications, applet programs cannot be run independently, thus require highest level of security. However, they do not require any specific deployment procedure during execution. Java applications, on the other hand, run independently and do not require any security as they are trusted.

| **Application program** | **Applet** |
|---|---|
| Applications are stand-alone programs that can be run independently without having to use a web browser. | Applets are small Java programs that are designed to be included in a HTML web document. They require a Java-enabled browser for execution. |
| Java applications have full access to local file system and network. | Applets have no disk and network access. |
| It requires a main method() for its execution. | It does not require a main method() for its execution. |
| Applications can run programs from the local system. | Applets cannot run programs from the local machine. |
| An application program is used to perform some task directly for the user. | An applet program is used to perform small tasks or part of it. |
| It can access all kinds of resources available on the system. | It can only access the browser specific services. |

## Preparing to write Applets

We have created a simple Java program with single main ( ) method having different variables, created object, set of methods. Here we will creating applet therefore we will need to know

- When to use applets
- How an applets works
- What sort of feature an applets has and
- Where to start when we first create our own applets.

***Why we need applet?***

- When we need something dynamic to be included in the display of a webpage.
- When we require some "flash" outputs. For example applets that produce sounds, animated or special effects on webpage.
- When we want to create a program and make it available on internet.

***Do you know about applet?***

When we design applet application, before we must make sure that Java is installed with Java applet viewer or Java enabled browser properly or not. Following steps and testing executed for designing applets.

- Building an applet code (.java file)
- Creating an executable applet (.class file)
- Designing a web page using HTML tag
- Preparing <APPLET> tag
- Incorporating <APPLET> tag into the webpage
- Creating HTML file
- Testing the applet code

## Building Applet Code

Every applet import Java.awt package that contain Graphics class. All output operation of an applet is performed using the methods defined in the Graphics class. The general format of applet is

```
import java.awt.*;
import java. Applet.*;
--------------------
--------------------
Public class appletclassname extends Applet
{
 ----------------------------
-----------------------------// Applet operation code
-----------------------------
}
---------------------------
---------------------------...
}
```

The appletclassname is the main class for the applet. When applet is loaded , java creates an instance of this class, and then a series of Applet class method are called on that instance to execute the code.

**A "Hello, World" Applet**

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;


public class HelloWorldApplet extends Applet {
  public void paint (Graphics g) {
    g.drawString ("Hello World", 25, 50);
  }
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet

- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

**The Applet Class**

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

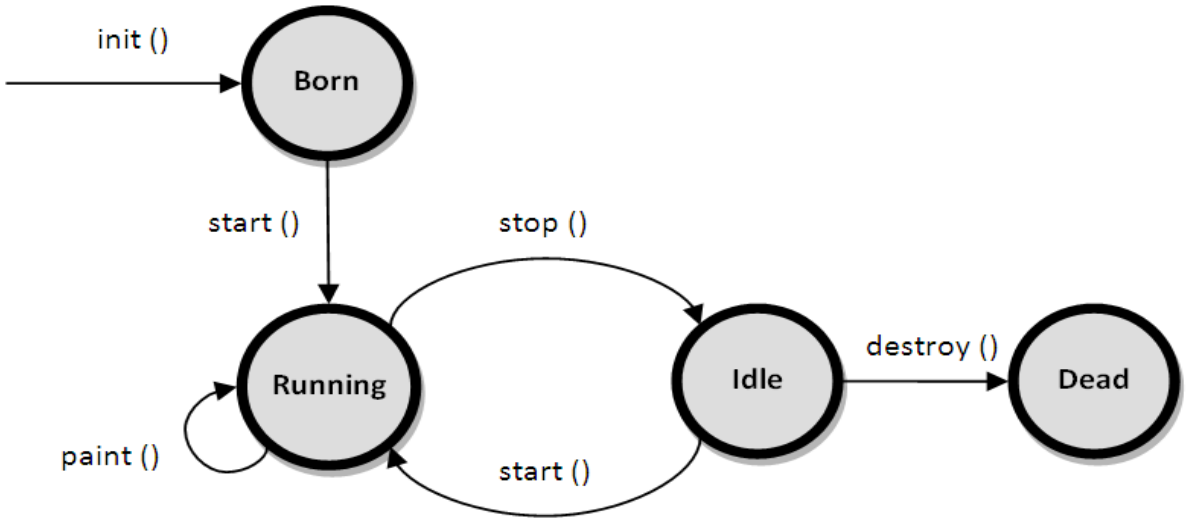
## 8. Applet life cycle

When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )




***An applet state transition diagram***

Let's look more closely at these methods.

- **init( ) :**

  The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.

- **start( ):**

  The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )** is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

- **paint( ):**

The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.

**paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.

The **paint( )** method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

- **stop( ):**

  The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When**stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

- **destroy( ):**

  The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

## 9. Summary

- **Java packages** are a mechanism to divide your **Java** code into coherent groups of classes, called **packages.**
- Packages are divided into two categories Built-in Packages (packages from the Java API) and User-defined Packages (create your own packages)
- The Java API is a library of prewritten classes that are free to use.
- User defined **java package** is a group of similar types of classes, interfaces and sub-packages.

- **Applet** is a **Java** program that can be run into a web page.
- Applets are feature rich application programs that are specifically designed to be executed within an HTML web document to execute small tasks or just part of it.
- JAVA stand-alone programs that are designed to run on a stand-alone machine without having to use a browser.
- When an applet begins, methods are called, in this sequence init( ), start( ), paint ( ),When an applet is terminated, the following sequence of method call stop( ), destroy( )