Object Oriented Programming using Java

Jigisha Patel

Introduction to Java

What is Java?

- **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 has a simpler object model and fewer low-level facilities.

- Java applications are typically compiled to bytecode (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.

History

- 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.

- 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.

- 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.

- The Stable release is Java Standard Edition 7 Update 5 (1.7.5) released on June 12, 2012.
- Java 8 released 2014
- Java 9 released 2017
- Java 10 (18.3) released 2018
- Java 11 (18.9) released 2018

Features of Java

- Sun Microsystems officially describe Java with the following features.
  - ✓ Simple
  - ✓ Secure
  - ✓ Portable
  - ✓ Object – Oriented
  - ✓ Robust
  - ✓ Multithreaded
  - ✓ Architecture-neutral
  - ✓ Interpreted
  - ✓ High Performance
  - ✓ Distributed
  - ✓ Dynamic

Simple:

- Java was designed to be easy for the professional programmer to learn and use effectively.
- If you know C++ programming, moving to Java will require very little effort as Java inherits the C/C++ syntax and many object-oriented features of C++.

Object-Oriented:

- Almost everything in Java is Object Oriented.
- All program code and data reside within classes and objects.
- The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects.

➢ Robust :

- The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems.
- Thus, the ability to create robust programs was given a high priority in the design of Java.

- To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development.
- At the same time, Java frees you from having to worry about many of the most common causes of programming errors.

- Because Java is a strictly typed language, it checks your code at compile time.
- However, it also checks your code at run time.

- To better understand how Java is robust, consider two of the main reasons for program failure:
  - ✓ memory management mistakes
  - ✓ mishandled exceptional conditions (run-time errors)

- Memory management can be a difficult, tedious task in traditional programming environments.

- Java virtually eliminates these problems by managing memory allocation and deallocation for you.

- In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.

- Exceptional conditions in traditional environments often arise in situations such as “division by zero” or “file not found,” and they must be managed with clumsy and hard-to-read constructs.

- Java helps in this area by providing object-oriented exception handling.

- In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded:

- Java was designed to meet the real-world requirement of creating interactive, networked programs.

- To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

- Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture - Neutral

- A central issue for the Java designers was that of code longevity and portability.
- The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter the situation of program malfunction due to processor upgrades, Operating System upgrades and changes in core system resources etc.
- Their goal was "write once; run anywhere, any time, forever."
- To a great extent, this goal was accomplished.

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
- This code can be interpreted on any system that provides a Java Virtual Machine (JVM).
- Most previous attempts at crossplatform solutions have done so at the expense of performance.
- Java, however, was designed to perform well on very low-power CPUs.

- As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
- Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed:

- Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.

- The original version of Java (Oak) included features for intraaddress-space messaging.
- This allowed objects on two different computers to execute procedures remotely.

- Java revived these interfaces in a package called Remote Method Invocation (RMI).
- This feature brings an unparalleled level of abstraction to client/server programming.

- **Dynamic:**
  - Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
  - This makes it possible to dynamically link code in a safe and expedient manner.

Java Applications and Java … lets

- Stand-alone Applications
  - Just like any programming language
- Applet
  - Run under a Java-Enabled Browser
- Midlet
  - Run in a Java-Enabled Mobile Phone
- Servlet
  - Run on a Java-Enabled Web Server

Java’s Magic : The Bytecode And JVM

The Bytecode:

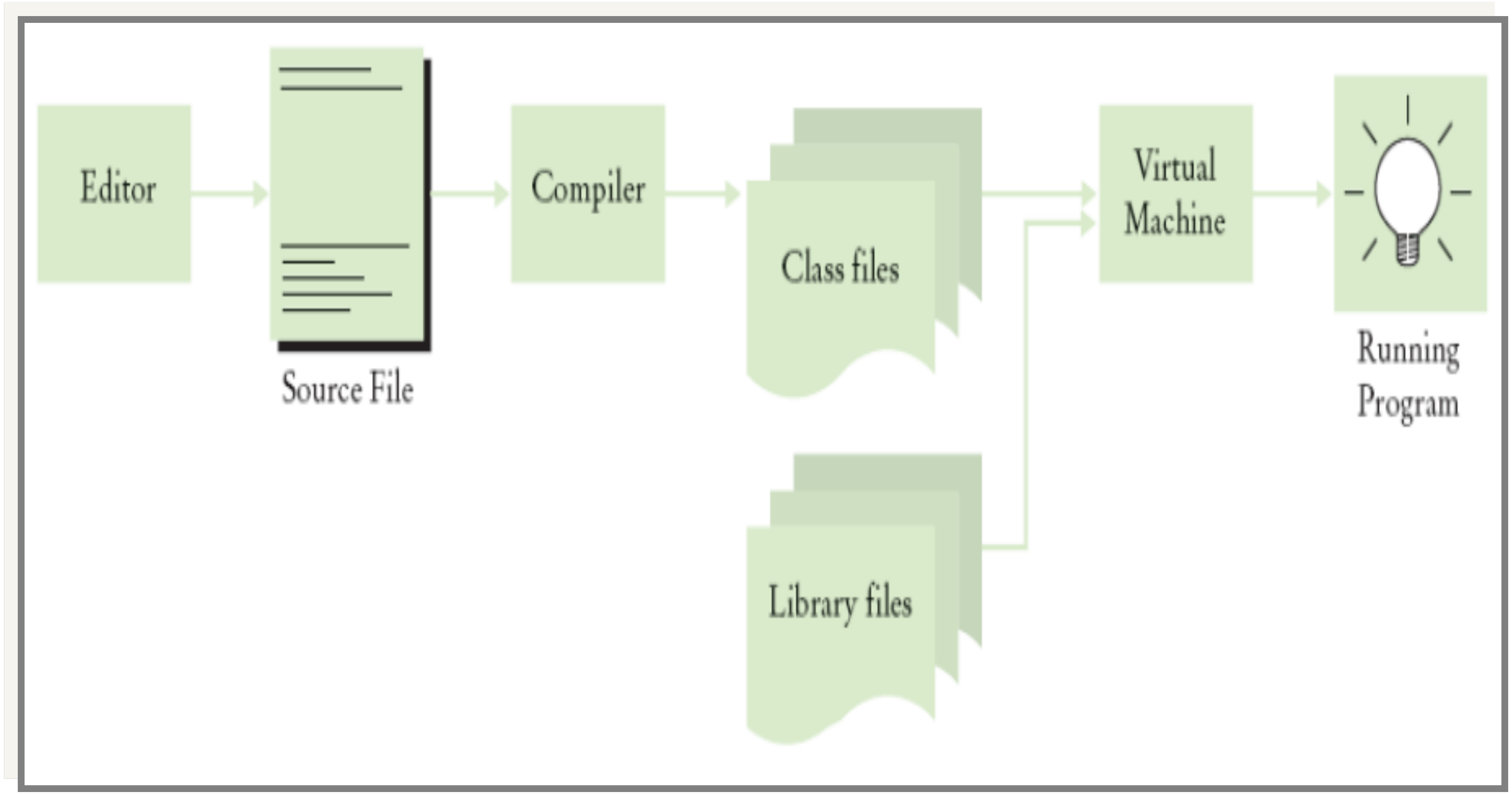
- The Java can support both the security and portability because the **output of java program** is not an executable code but it’s a **Bytecode**.

- **“Bytecode is a highly optimized set of instructions designed to be executed by the Java runtime system, which is called Java Virtual Machine.”**

JVM : Java Virtual Machine

- JVM is interpreter for the Bytecode file.
- All language compiler translate source code into m/c level code for a specific computer.
- The Java compiler produces an intermediate code known as Bytecode for a m/c that is not in readable format.
- JVM is located inside the computer memory and Bytecode is executed by only JVM.

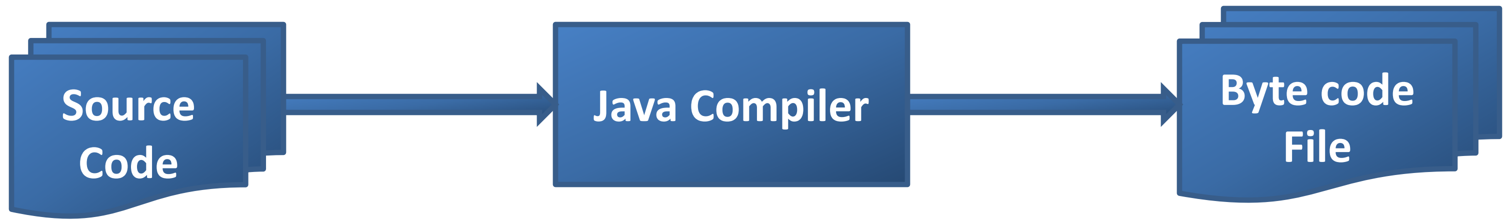
Compilation Process : From source code to running program



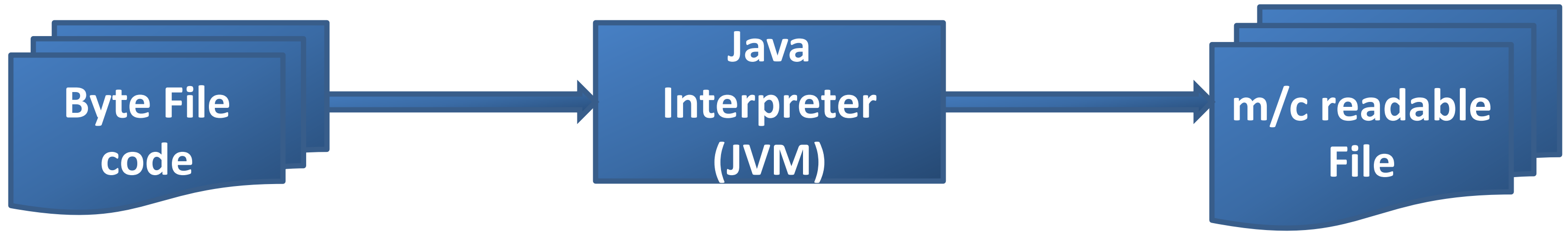
There are two Processes:

- Basically the execution of a Java program can be divided in two processes:

1. **Process of Compilation**



2. **Process of converting Bytecode to m/c code**



Features of the JVM

- The Garbage Collector
  - Java manages memory for you, the developer has no control over the allocation of memory (unlike in C/C++).
  - This is much simpler and more robust (no chance of memory leaks or corruption)
  - Runs in the background and cleans up memory while application is running

- Security
  - Java offers very fine control over what an application is allowed to do
  - E.g. Read/write files, open sockets to remote machines, discover information about the users environment, etc
  - Used in Java Applets to create a “sandbox”. Stops a rogue applet attacking your machine.
  - Makes Java very safe, an important feature in distributed systems

- Class Loading
  - Loading of bytecode into the virtual machine for execution
  - Code can be read from a local disk, over a network, or the Internet
  - Allows downloading of applications and applets

The *Just-In-Time* compiler (JIT)

- Translates bytecode into machine code at runtime
  - 1-time overhead when run initiated
  - Performance increase 10-30 times

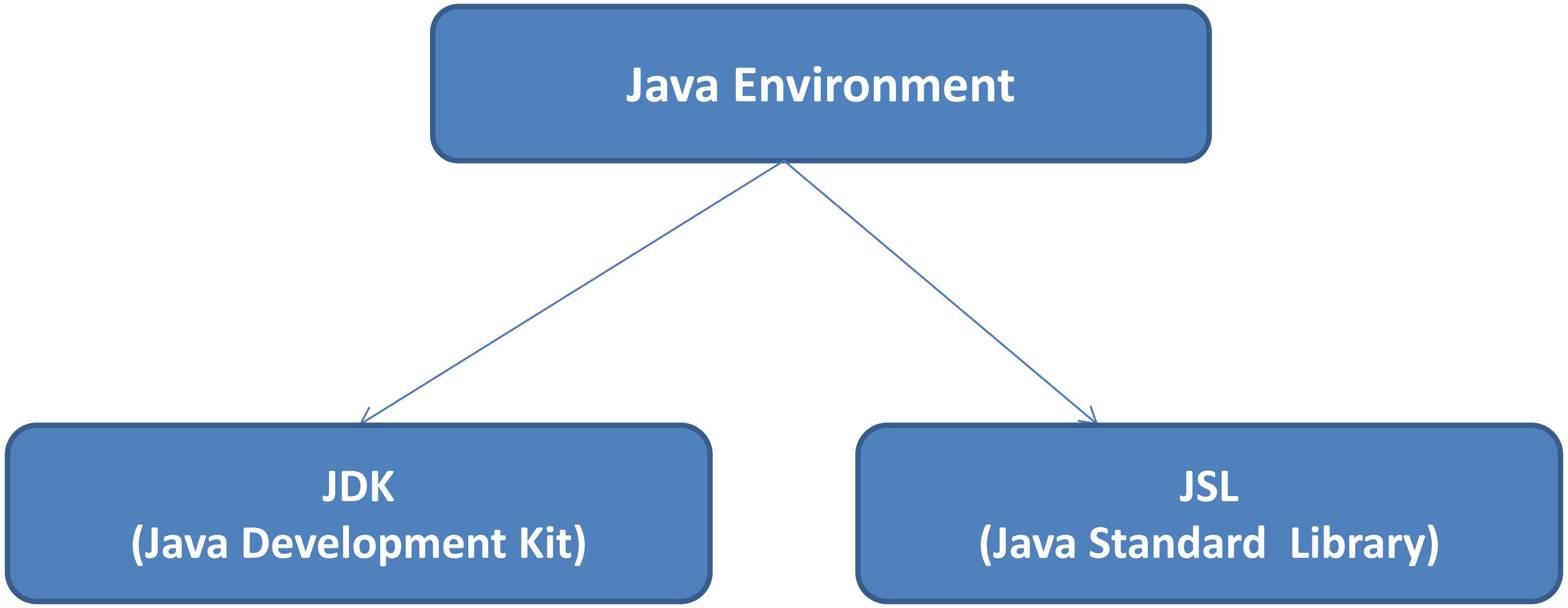
- Also known as "Hot Spot"
- Continually optimises running code to improve performance

- Can approach the speed of C++ even though its interpreted

- Now the default for most JVM's
  - Can be turned off if desired
  - JIT can apply statistical optimizations based on runtime usage profile

Java Environment

- Java Environment includes a large number of development tools and hundreds of classes and methods.



JDK (Java Development Kit)

- JDK is a collection of tools that are used for developing and running a Java Program.

1. **javac** - compiler used to compile Java source code.

   Syntax : **javac filename.java (**Source file name, extension is ".java")
2. **Java -** interpreter used to execute Java Bytecodes.

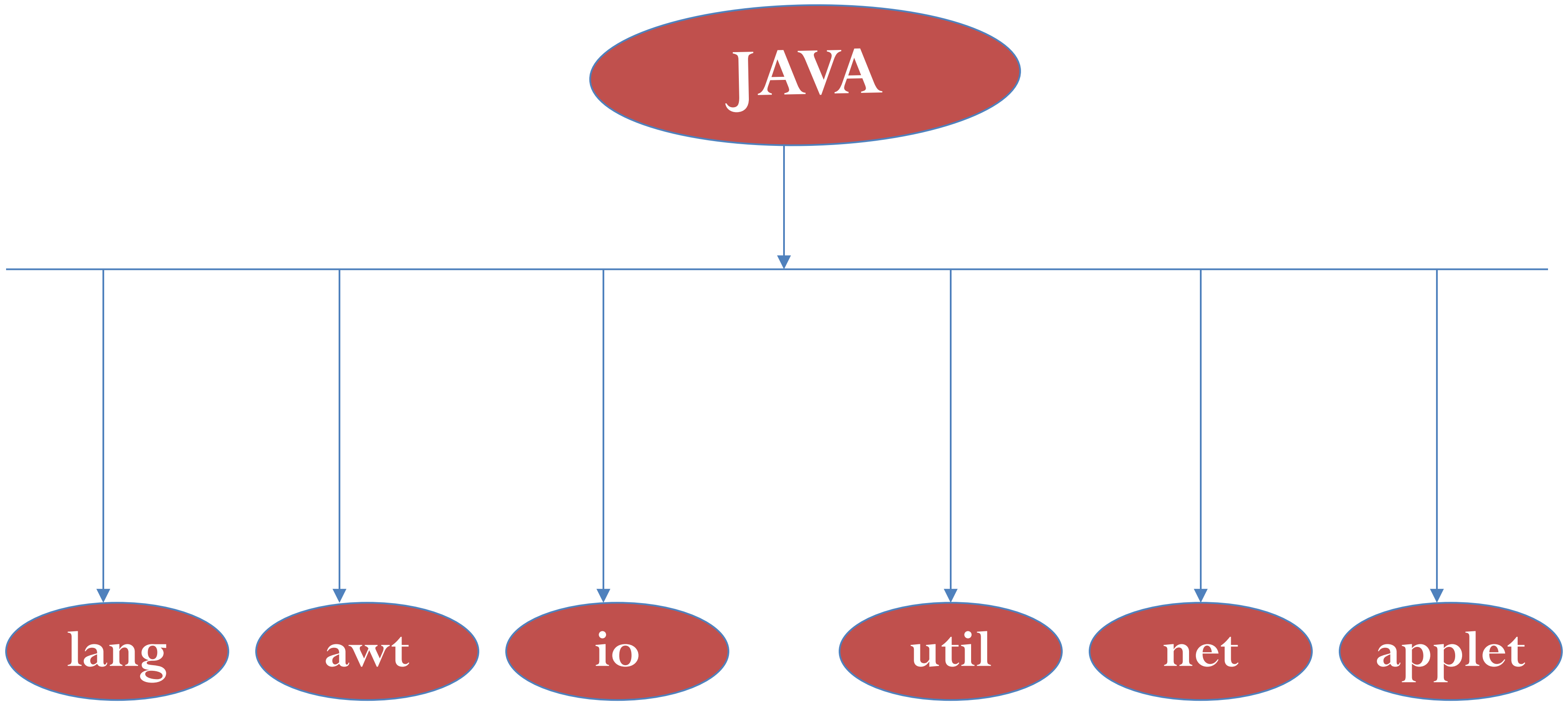
   Syntax: **java filename**
3. **appletviewer** : Used to view and test applets.

   Syntax : **appletviewer [options] url**
4. **javadoc** : This is the Java documentation tool.

   Generates detailed documentation in HTML form for any .java source code or package
5. **jdb**: Java debugger which help to find errors in program.

Application Program Interface

- API or JSL includes hundreds of classes and methods grouped into several functional package.
- Most commonly used packages are :



Packages	Description
java.lang	Language support classes. Imported automatically
java.util	Language utility class such as date.
java.io	Input/output support class.
java.awt	Classes for graphical user interface.
java.net	Classes for networking.
java.applet	Classes for creating and implementing applets.

Several Popular IDEs

- NetBeans
- jEdit
- Eclipse
- JBuilder
- JCreator

- From all of these, we are going to use **NetBeans**.

Anatomy of a Java Program

- Comments
- Package
- Reserved words
- Modifiers
- Statements
- Blocks
- Classes
- Methods
- The main method

General Structure of Java program

Structure of Java Program		
Document Section	-	Suggestion
Package statement	-	Optional
Import Statement	-	Optional
Interface Statement	-	Optional
Class Definition	-	Necessary
Main Method class	-	Necessary
Main method definition	-	Necessary

Document Section

- It is used to specify a comment for understanding the code.

- Three types of comment:
  1. Single Line Comment
     ```
     // insert comments here
     ```
  2. Block Comment
     ```
     /*
      insert comments here
     */
     ```
  3. Documentation Comment
     ```
     /**
      insert documentation
     */
     ```

Package statement

- It is the first statement allowed in a Java file.
- This statement declares a package name and inform the compiler that class defined belongs to this package.

  e.g. package Student;

- The package statement is optional.
- i.e. Classes do not have to be a part of a package.

Import statement

- Next statement after a package statement can be one or more import statements.

- This is similar to #include statement in C.

  e.g. $\text{import } \underset{\text{Package}}{\underset{\downarrow}{\text{student}}}.\underset{\text{Class}}{\underset{\downarrow}{\text{Test}}}$

- It informs the interpreter to load the ‘Test’ class contained in the Student package.

Interface statement

- It is similar to a class but includes a group of methods declaration.

- This is also an optional and use only when we wish to implement multiple inheritance.

Class Definition

- A Java program can contain multiple class definition.

- Classes are primary and essential elements of Java.

Main Method Class

- Every Java stand-alone application requires a ‘main’ method.
- A simple Java program may contain only this part.
- The main method creates objects of various classes and establishes a connection between them.

Sample Program

```
/*
 First Java Program
 */
import java.util.*;

public class JavaMain
{

    public static void main(String[] args)
    {
        // print a message
       System.out.println("Welcome to Java!");
    }
}
```

- **Class:**
  - Every java program includes at least one class definition. The class is the fundamental component of all Java programs.
  - A class definition contains all the variables and methods that make the program work. This is contained in the class body indicated by the opening and closing braces.

- **Braces:**
  - Braces are used for grouping statements or block of codes.
  - The left brace "{" indicates the beginning of a class body, which contains any variables and methods the class needs.
  - The left brace also indicates the beginning of a method body.
  - For every left brace that opens a class or method you need a corresponding right brace "}" to close the class or method.
  - A right brace always closes its nearest left brace.

- **public static void main (String args[])** method:
  - All Java applications begins execution by calling main().
  - The **public** keyword is an **access specifier**, which allows the programmer to control the visibility of class members.
  - When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.

- In the case of main(), it must be declared as public, since it must be called by code outside of its class when program is started.

- The keyword **static** is the storage class which allows main() to be called without having to instantiate a particular instance of the class.
- **void** is the return type of the main method.

- **String args[]**: Declares a parameter named args, which is an array of String. It represents command-line arguments.

• A statement is always terminated with a **semicolon** and may span multiple lines in your source code.

- **System.out.println()**
  - This line outputs the string “Welcome to Java!” followed by a new line on the screen.
  - **System** : it is a predefine class that provide access to the system
  - **out** : out is an output stream that is connected to the console.
  - **println()**: it is a method used to display data to the screen.

Importance of main() method

- **main()** is simply a starting place for your program.
- A complex program will have dozens of classes, only one of which will need to have a main() method to get things started.
- Java is a **case-sensitive** language. So Main() is different from main().
- The Java compiler will compile classes that do not have a main() method but Java has no way to run these classes.
- So you must have to write at least one main() method in your program.

Naming Convention

- **Class:**
  - Class names should be nouns, in mixed case with the first letter of **each internal word capitalized**.
  - Try to keep your class names simple and descriptive.
  - Use whole words-avoid acronyms and abbreviations.
  - e.g. class LibrarySystem, class StudentInformation

- **Method:**
  - Methods should be verbs, in mixed case with the **first letter lowercase**, with the first letter of **each internal word capitalized**.
  - e.g. display(), displayData()

- **Variable:**
  - Variable names should be short yet meaningful.
  - The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use.

  - Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.
  - e.g. int i, String name

- **Interface:**
  - Interface names should be capitalized like class names.
  - e.g. interface Result

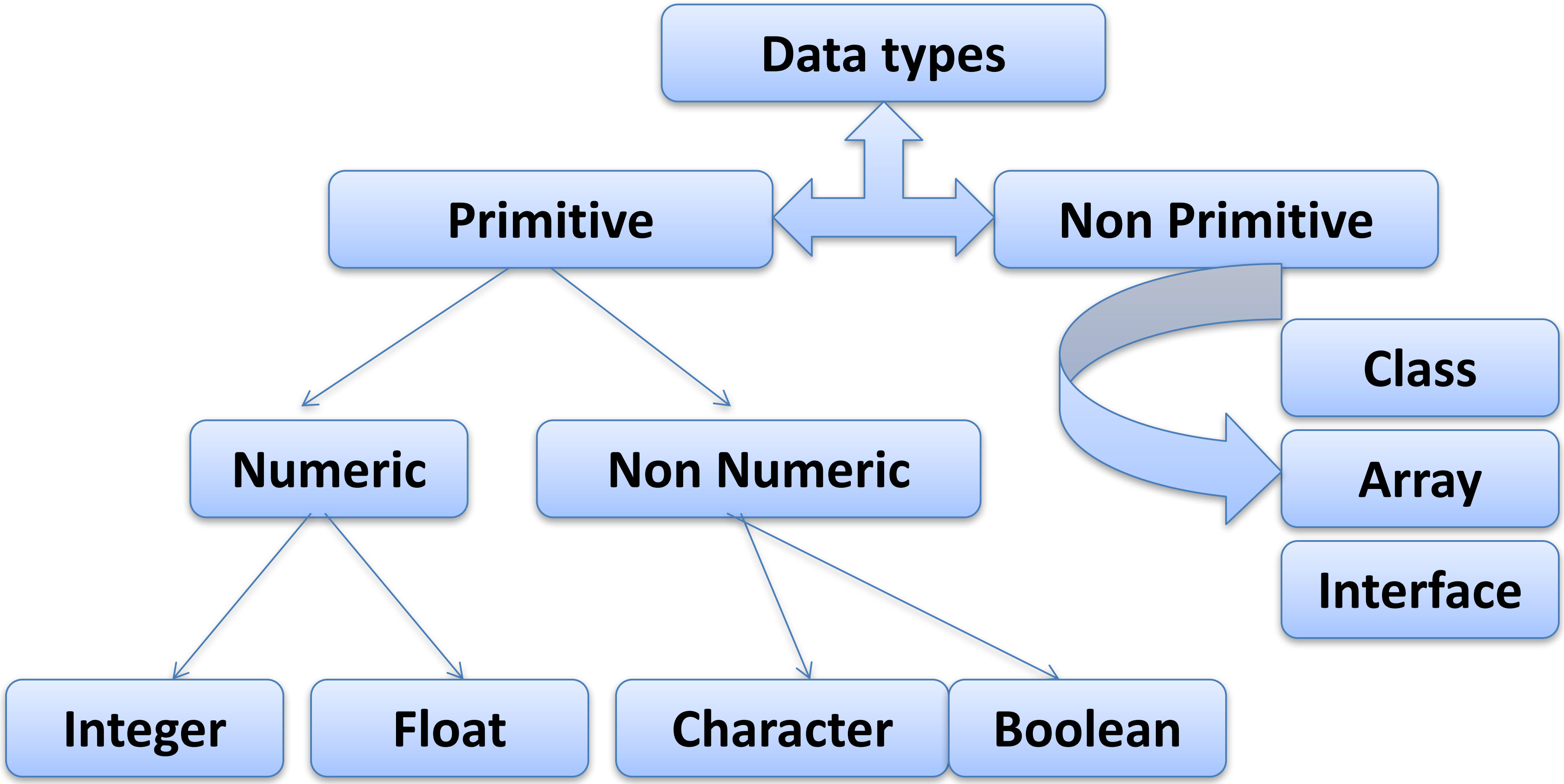
- **Constant:**
  - The names of variables declared class constants and of ANSI constants should be **all uppercase** with words **separated by underscores** ("_").
  - e.g. static final int MIN_WIDTH = 4;
    static final int MAX_WIDTH = 999;

- **Packages:**
  - The prefix of a unique package name is always written in **all-lowercase .**
  - e.g. package mypackage

Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- Identifiers **must not begin with a number**.
- Java Identifiers are **case-sensitive**.
- Some **valid** identifiers are PID, name, Address1
- Some **invalid** identifiers are 1Address, n-l, Student/ID

Data types

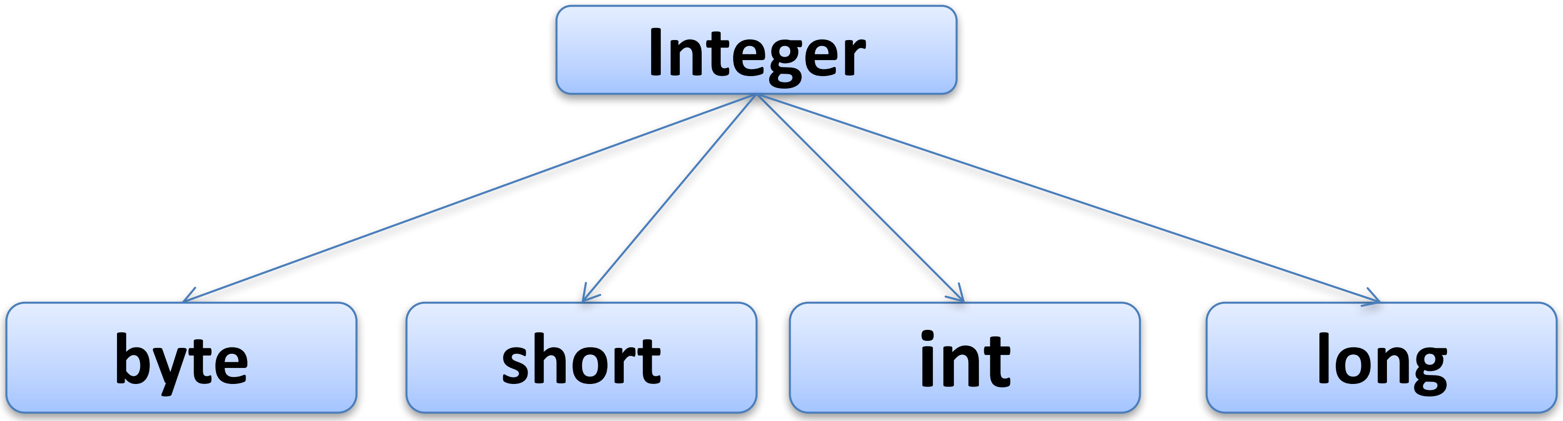

Data types
Primitive
Non Primitive
Numeric
Non Numeric
Integer
Float
Character
Boolean
Class
Array
Interface

Primitive Data Types in Java

Type	Kind	Memory	Range
byte	integer	1 byte	-128 to 127
short	integer	2 bytes	-32768 to 32767
int	integer	4 bytes	-2147483648 to 2147483647
long	integer	8 bytes	-9223372036854775808 to -9223372036854775807
float	floating point	4 bytes	±3.40282347 x 10^{38} to ±3.40282347 x 10^{-45}
double	floating point	8 bytes	±1.76769313486231570 x 10^{308} to ±4.94065645841246544 x 10^{-324}
char	single character	2 bytes	all Unicode characters
boolean	true or false	1 bit	

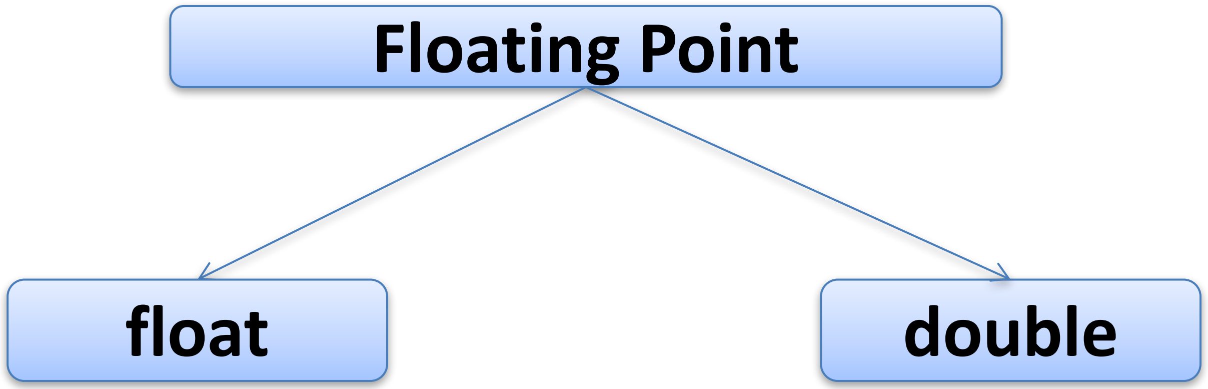
Integer Types :

- Integer types can hold whole numbers such as 123, -98, and 5634.
- The size of the value that can be stored depends on the integer data type we choose.
- Java supports four types of Integer as shown in the figure.
- Java does not support the concept of unsigned types and therefore all Java values are signed meaning they can be positive or negative.



Floating Point Types :

- Integer types can hold only whole numbers and therefore we use another type known as floating point type to hold numbers containing fractional parts such as 25.36 and -12.23 (known as floating point constants).
- There are two kinds of floating point as shown in the below figure.



- The **float** type values are single-precision numbers while the **double** types represents double precision numbers.
- To declare floating number

  1.23f or 1.23F
- Double-precision types are used when we need greater precision in storage of floating point numbers.

Character Type :

- In order to store character constant in memory, Java provides a character data type called **char**.
- Java uses **Unicode** to represent characters.
- Unicode defines a fully international character set that can represent all of the characters found in all human language.
- To declare a character variable,

  char c;

Boolean Type :

- It is used for logical values.
- It can have only one of two possible values true or false.

e.g.
```
boolean b = false;
boolean a = true;
```

Declaring a variable

- A variable can be used to store value of any data type..
- Declaration does three things :
    1. It tells the compiler what the variable name is.
    2. It specifies what type of data the variable will hold.
    3. The place of declaration in program decides the scope of the variable.
- A variable must be declare before it is used in the program.
- The basic form of a variable declaration is :

type identifier [= value][, identifier [= value] ...] ;

- E.g.: **int count;**
- Java allows variables to be initialized dynamically.
- E.g.: **double** c = 2 * 2;

Scope & Life of variable

- Java allows variables to be declared within any block.
- A block is begin with an opening curly bracket and ended by a closing curly bracket.

- A block defines **scope**.
- Thus, each time you start a new block, you are creating a new scope.

- The scope define by a method begins with its opening curly brace.
- If that method includes parameters, they are also include within that method's scope.
- As a general rule, **variable declared inside a scope are not accessible to the outside code**.

- Scopes can be nested. The outer scope encloses the inner scope.
- Variables declared in the outer scope are visible to the inner scope.
- Variables declared in the inner scope are not visible to the outside scope.