## 1. Introduction

### 1.1. Java's Lineage

- Java is related to C++, which is a direct descendent of C. Much of the character of Java is inherited from these two languages.
- From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++.
- The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs.
- Invent of C++: By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called object-oriented programming (OOP).
- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- It took 18 months to develop the first working version.
- This language was initially called "Oak" but was renamed "Java" in 1995.
- Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language.
- The original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target.
- Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU.
- In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.
- However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.
- By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet.
- This realization caused the focus of Java to switch from consumer electronics to Internet programming.

### 1.2. Feature of Java

1.2.1. Simple:
- Java is designed to be easy for the professional programmer to learn and use effectively.
- If you are an experienced C++ programmer, moving to Java will require very little effort.

1.2.2. Object-Oriented:
- Java supports object oriented programming paradigm

1.2.3. Robust:
- Java is robust programming language, because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.
- Better memory management in Java compared to C/C++. In C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. 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.)

1.2.4. Multithreaded:
- Java was designed to meet the real-world requirement of creating interactive, networked programs.
- Java supports **multithreaded programming**, which allows you to write programs that do many **things simultaneously**.

1.2.5. Architecture-Neutral
- Java is designed on the principle of “write once; run anywhere, anytime, forever.”
- It can run even if operating system upgrades, processor upgrades, and changes in core system resources can all combine.

1.2.6. 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.
- 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.

1.2.7. Distributed
- Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
- Java supports distributed computing through Remote Method Invocation (RMI).

1.2.8. Dynamic
- Java enable dynamic link of various code block at runtime.

## 1.3. Java Virtual Machine

- JVM (Java Virtual Machine) Architecture

  JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

  **What is JVM ?**

  It is:

  **1. A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.

  **2. An implementation** Its implementation is known as JRE (Java Runtime Environment).

  **3. Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

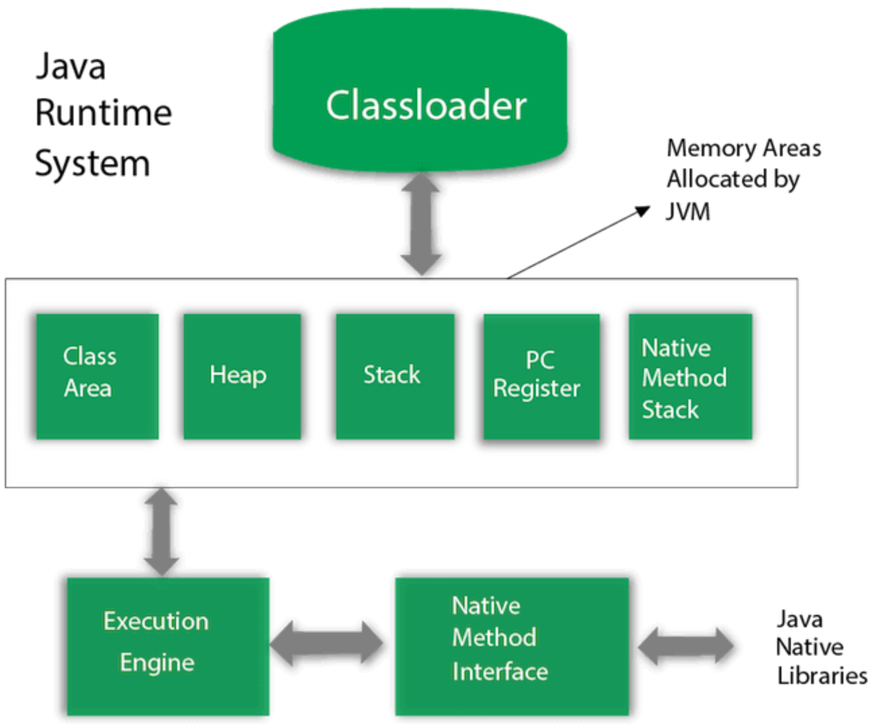
  **What it does ?**

  The JVM performs following operation:

  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment

  JVM provides definitions for the:

  - Memory area
  - Class file format
  - Register set
  - Garbage-collected heap
  - Fatal error reporting etc.



- JVM contains classloader, memory area, execution engine etc.
    - 1) Classloader
        - Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.
            - Bootstrap ClassLoader: This is the first classloader which is the super class of Extension classloader. It loads the rt.jar file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.

            - Extension ClassLoader: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside $JAVA_HOME/jre/lib/ext directory.

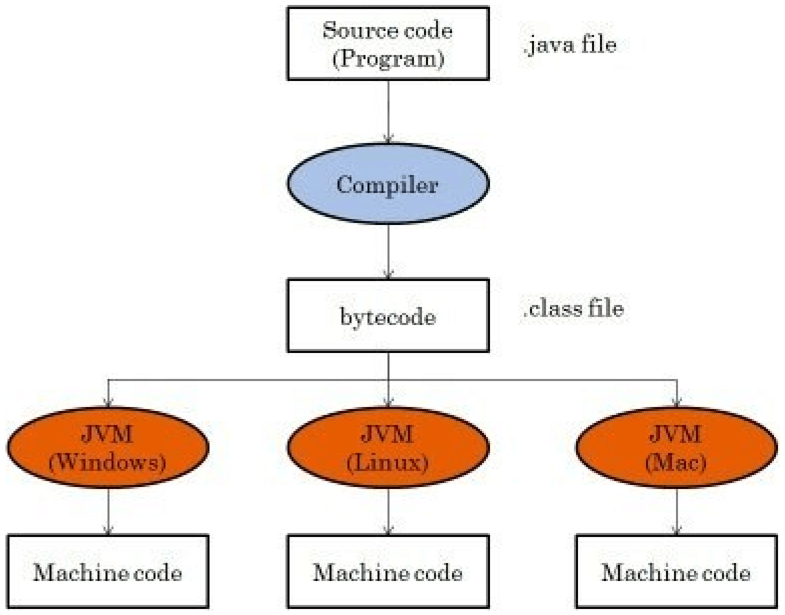
            - System/Application ClassLoader: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

- 2) Class(Method) Area
  - Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
- 3) Heap
  - It is the runtime data area in which objects are allocated.
- 4) Stack
  - Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.
  - Each thread has a private JVM stack, created at the same time as thread.
  - A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
- 5) Program Counter Register
  - PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.
- 6) Native Method Stack
  - It contains all the native methods used in the application.
- 7) Execution Engine
  - A virtual processor
  - Interpreter: Read bytecode stream then execute the instructions.
  - Just-In-Time(JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.
- 8) Java Native Interface
  Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

## 1.4. Byte Code

- What is Java Bytecode?
  - Java bytecode is the instruction set for the Java Virtual Machine.
  - As soon as a java program is compiled, java bytecode is generated. In more apt terms, java bytecode is the machine code in the form of a .class file. With the help of java bytecode we achieve platform independence in java.

- How does it works?



- When we write a program in Java, firstly, the compiler compiles that program and a bytecode is generated for that piece of code.
- After the first compilation, the bytecode generated is now run by the Java Virtual Machine and not the processor in consideration.
- Resources required to run the bytecode are made available by the Java Virtual Machine, which calls the processor to allocate the required resources.

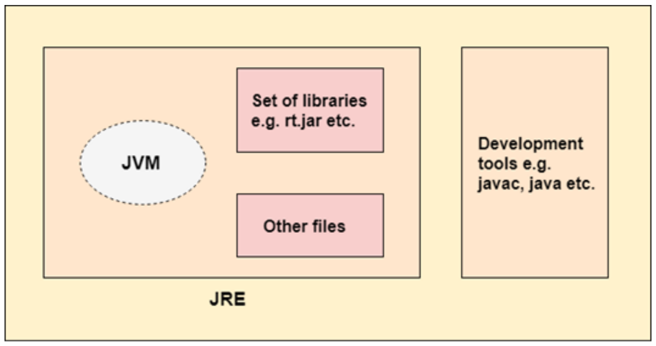
- Advantage of Java Bytecode
  - Platform independence is one of the soul reasons for which James Gosling started the formation of java and it is this implementation of bytecode which helps us to achieve this.
  - Hence bytecode is a very important component of any java program.
  - A point to keep in mind is that bytecodes are non-runnable codes and rely on the availability of an interpreter to execute and thus the JVM comes into play.
  - Bytecode promotes portability which ensures that Java can be implemented on a wide array of platforms like desktops, mobile devices, severs and many more.

## 1.5. JDK

- JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.
- JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

  - Standard Edition Java Platform
  - Enterprise Edition Java Platform

- Micro Edition Java Platform

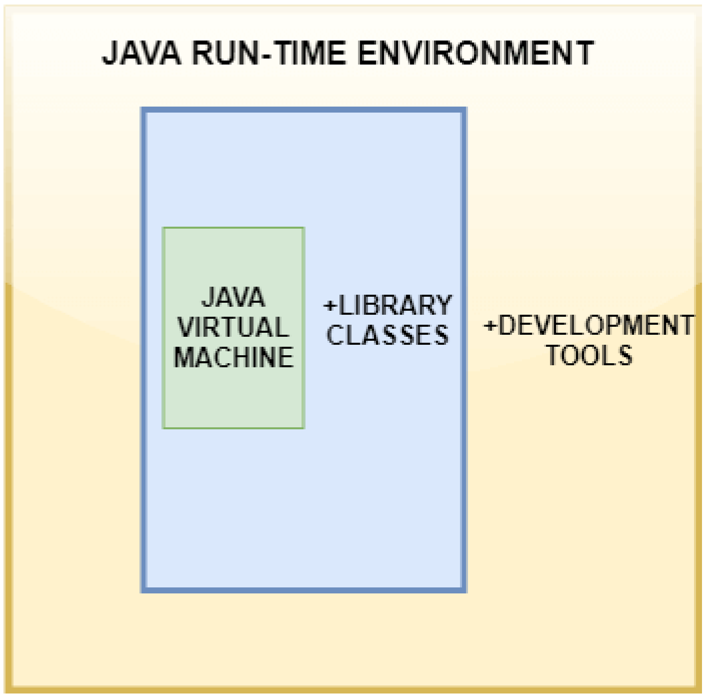

Set of libraries e.g. rt.jar etc.
JVM
Other files
Development tools e.g. javac, java etc.
JRE

JDK

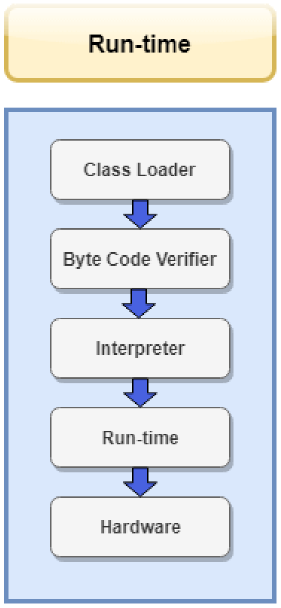
- The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.
- Some main components of JDK
  - appletviewer: This tool is used to run and debug Java applets without a web browser.
  - apt: It is an annotation-processing tool.
  - extcheck: it is a utility that detects JAR file conflicts.
  - idlj: An IDL-to-Java compiler. This utility generates Java bindings from a given Java IDL file.
  - jabswitch: It is a Java Access Bridge. Exposes assistive technologies on Microsoft Windows systems.
  - java: The loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler. Now a single launcher is used for both development and deployment. The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader.
  - javac: It specifies the Java compiler, which converts source code into Java bytecode.
  - javadoc: The documentation generator, which automatically generates documentation from source code comments
  - jar: The specifies the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files.
  - javafxpackager: It is a tool to package and sign JavaFX applications.
  - jarsigner: the jar signing and verification tool.
  - javah: the C header and stub generator, used to write native methods.
  - javap: the class file disassembler.
  - javaws: the Java Web Start launcher for JNLP applications.
  - JConsole: Java Monitoring and Management Console.

1.6. JRE

- What is JRE?
    - Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK).
    - It is a freely available software distribution which has Java Class Library, specific tools, and a stand-alone JVM.
    - The source Java code gets compiled and converted to Java bytecode. If you wish to run this bytecode on any platform, you require JRE.
    - The JRE loads classes, verify access to memory, and retrieves the system resources. JRE acts as a layer on the top of the operating system.
- What does JRE consist of?
    - JRE consists of the following components:
        - Deployment technologies such as deployment, Java plug-in, and Java Web Start.
        - User interface toolkits, including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag, and drop (DnD) and input methods.
        - Integration libraries including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.
        - Other base libraries, including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
        - Lang and util base libraries, including lang and util, zip, Java Archive (JAR), instrument, reflection, Collections, Concurrency Utilities, management, versioning, Logging, Preferences API, Ref Objects and Regular Expressions.
- How does JRE work with JVM?



- Once you write this program, you have to save it with .java extension. Compile your program. The output of the Java compiler is a byte-code which is platform independent.
- After compiling, the compiler generates a .class file which has the bytecode. The bytecode is platform independent and runs on any device having the JRE.
- From here, the work of JRE begins. To run any Java program, you need JRE. The flow of the bytecode to run is as follows:



- Class Loader

        - At this step, the class loader loads various classes which are essential for running the program. The class loader dynamically loads the classes in the Java Virtual Machine.
        - When the JVM is started, three class loaders are used:
            - Bootstrap class loader
            - Extensions class loader
            - System class loader
    - Byte code verifier
      Byte code verifier can be considered as a gatekeeper. It verifies the bytecode so that the code doesn't make any sort of disturbance for the interpreter. The code is allowed to be interpreted only when it passes the tests of the Bytecode verifier which checks the format and checks for illegal code.
    - Interpreter
      Once the classes get loaded and the code gets verified, then interpreter reads the assembly code line by line and does the following two functions:
        - Execute the Byte Code
        - Make appropriate calls to the underlying hardware

      In this way, the program runs in JRE.

## 1.7. Comments and Java coding convention

### 1.7.1. First Java program

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example
{
       // Your program begins with a call to main().

       public static void main(String args[])
      {
      System.out.println("This is a simple Java program.");
      }
}
```

- The first thing is the name to source file is very important
    - In java, a source file is officially called a **compilation unit**.
    - In Java, all code must reside inside a class.
    - By convention, the name of that class should match the name of the file that holds the program, also match the **case sensitive name.**
- Compiling and run java program
    - C:\>javac Example.java

        - The javac compiler creates a file called Example.class that contains the bytecode version of the program.
    - C:\>java Example
        - To actually run the program, you must use the Java interpreter, called java.
- Comment ( single line and multiline comments)
    - /*
      This is a simple Java program.
      Call this file "Example.java".
      */

- main function
    - public static void main(String args[])
        - All Java applications begin 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.
        - **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.
        - The keyword **static** allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java interpreter before any objects are made.
        - The keyword **void** simply tells the compiler that main( ) does not return a value.

2. Array and String:

   2.1. Single and Multidimensional Array

- In java First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.
- int month_days[];
- Although this declaration establishes the fact that month_days is an array variable, no array actually exists. In fact, the value of month_days is set to null, which represents an array with no value.
- The general form of new as it applies to one-dimensional arrays appears as follows:
- month_days = new int[12];

  or
- int month_days[] = new int[12];

  or
- int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

- multidimensional array
  - int twoD[][] = new int[4][5];

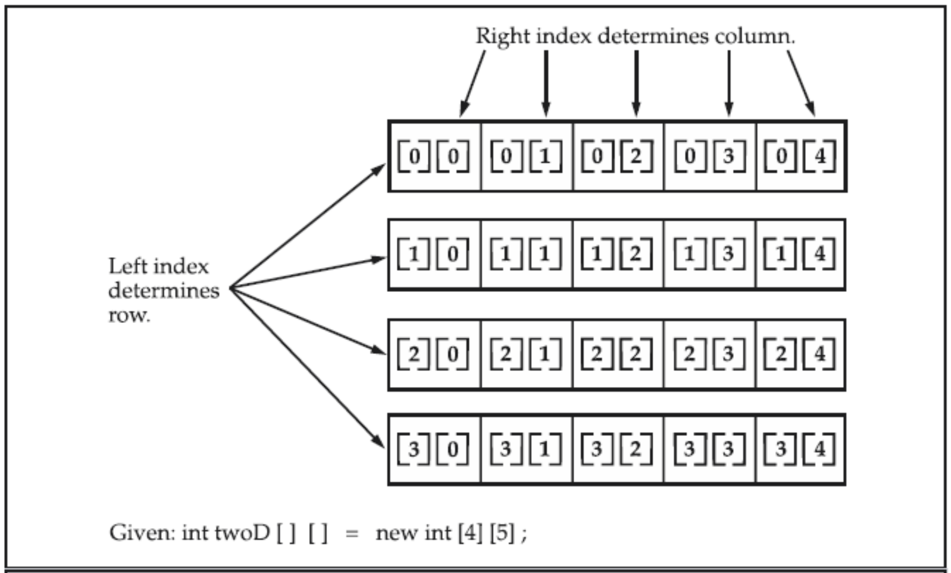
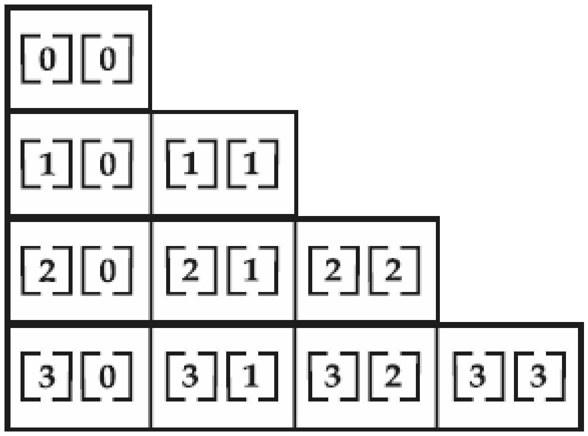


**Figure 3-1.** *A conceptual view of a 4 by 5, two-dimensional array*

  - Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a twodimensional array in which the sizes of the second dimension are unequal.

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }
```



- The use of uneven (or, irregular) multidimensional arrays is not recommended for most applications, because it runs contrary to what people expect to find when a multidimensional array is encountered.
- However, it can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

- Alternative Array Declaration Syntax
  - There is a second form that may be used to declare an array:
    type[ ] var-name;
  - For example, the following two declarations are equivalent:
    int al[] = new int[3];
    int[] a2 = new int[3];

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

## 2.2. Command line argument

- Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the String array passed to main( ).

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

Try executing this program, as shown here:

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

When you do, you will see the following output:

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

## 2.3. String

- String is probably the most commonly used class in Java’s class library. The obvious reason for this is that strings are a very important part of programming.
- Every string you create is actually an object of type String.
- Even string constants are actually String objects. For example, in the statement System.out.println("This is a String, too");
- In java, objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:
    - If you need to change a string, you can always create a new one that contains the modifications.
    - Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java.
- Strings can be constructed a variety of ways.
    - The easiest is to use a statement like this:
      String myString = "this is a test";
    - Java defines one operator for String objects: +. It is used to concatenate two strings. For example, this statement,
      String myString = "I" + " like " + "Java.";

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

- The String class contains several methods that you can use.
    - You can test two strings for equality by using equals( ).
    - You can obtain the length of a string by calling the length( ) method.
    - You can obtain the character at a specified index within a string by calling charAt( ).

```
// Demonstrating some String methods.
class StringDemo2 {
        public static void main(String args[]) {
                String strOb1 = "First String";
                String strOb2 = "Second String";
                String strOb3 = strOb1;
                System.out.println("Length of strOb1: " +
                 strOb1.length());
                System.out.println("Char at index 3 in strOb1: " +
                strOb1.charAt(3));

                if(strOb1.equals(strOb2))
                System.out.println("strOb1 == strOb2");
                else
                System.out.println("strOb1 != strOb2");

                if(strOb1.equals(strOb3))
                System.out.println("strOb1 == strOb3");
                else
                System.out.println("strOb1 != strOb3");

                                        }
                        }
```

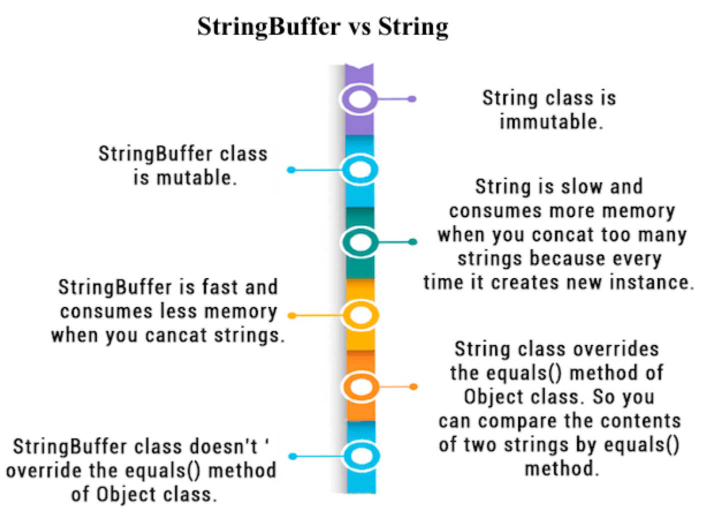
- you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
      class StringDemo3 {
                     public static void main(String args[])
                     {
                     String str[] = { "one", "two", "three" };
                     for(int i=0; i<str.length; i++)
                     System.out.println("str[" + i + "]: " + str[i]);
                     }
                        }
```

```
class HelloWorld {

    public static void main(String[] args)

        {

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

        int i;

        Scanner sc = new Scanner(System.in);

        int length = 5 ;

        String arr[] = new String[length];

        for( i=0; i<length; i++)

        {

         System.out.print("Enter string : ");

         arr[i] = sc.nextLine();

        }

    // to read the already populated string array1

       for(i=0; i<length; i++)

         {

         System.out.println(arr[i]);

         }

       }

}
```

## 2.4. String Buffer

- Difference between String and String Buffer Class


StringBuffer vs String
String class is immutable.
StringBuffer class is mutable.
String is slow and consumes more memory when you concat too many strings because every time it creates new instance.
StringBuffer is fast and consumes less memory when you cancat strings.
String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.
StringBuffer class doesn't ' override the equals() method of Object class.

```
public class ConcatTest{
    public static String concatWithString()    {
        String t = "ADIT";
        for (int i=0; i<10000; i++){
            t = t + "Computer";
        }
        return t;
    }
    public static String concatWithStringBuffer(){
        StringBuffer sb = new StringBuffer("ADIT");
        for (int i=0; i<10000; i++){
            sb.append("Computer");
        }
        return sb.toString();
    }
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        concatWithString();
        System.out.println("Time taken by Concating with String:
"+(System.currentTimeMillis()-startTime)+"ms");
        startTime = System.currentTimeMillis();
        concatWithStringBuffer();
        System.out.println("Time taken by Concating with  StringBuffer:
"+(System.currentTimeMillis()-startTime)+"ms");
    }
}
```

## 2.5. for-each loop

- The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java.
- It is mainly used to traverse the array or collection elements.
- It is known as the for-each loop because it traverses each element one by one.
- The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order.
- Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.
- Syntax:
```
for(data_type variable : array | collection){
//body of for-each loop
}
```
- Example:
```
class ForEachExample1{
  public static void main(String args[]){
   //declaring an array
   int arr[]={12,13,14,44};
   //traversing the array with for-each loop
   for(int i:arr){
     System.out.println(i);
   }
 }
}
```

## 2.6. Wrapper Class

- The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.
  - Autoboxing: The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);  }}
```

- Unboxing: The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.
  Java program to convert object into primitives

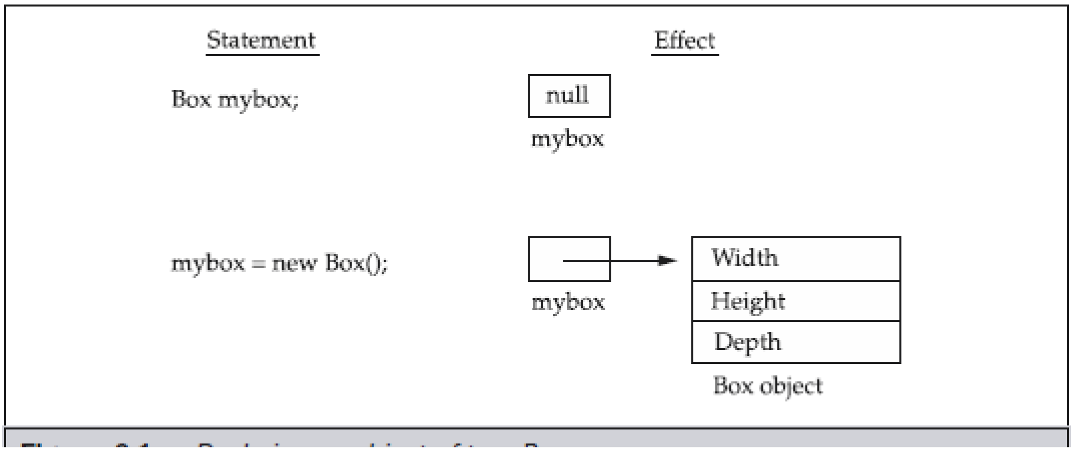
```
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

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

3. Class Object and Method:
   - A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)
   - A class is a template for an object, and an object is an instance of a class.
   - Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.
   - The data, or variables, defined within a class are called instance variables.
   - The methods and variables defined within a class are called members of the class.
   - A Simple Class:

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

- As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.
- To actually create a Box object, you will use a statement like the following:
  Box mybox = new Box(); // create a Box object called mybox

- After this statement executes, mybox will be an instance of Box. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.



- If no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with Box.
- It is important to understand that new allocates memory for an object during run
- time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.

Example:

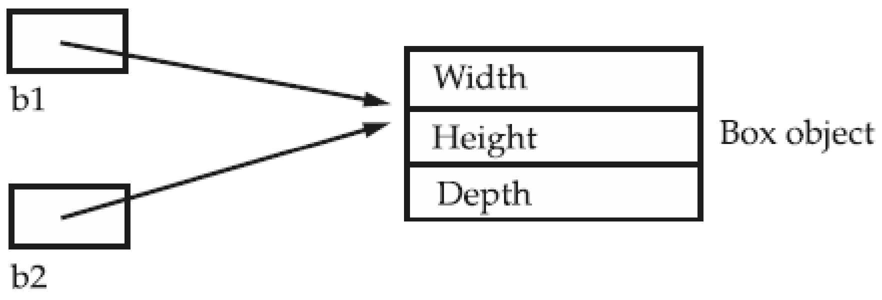
```
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();

double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
```

```
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
        }
}
```

- Assigning Object Reference Variables:

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

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



  ```
  Box b1 = new Box();
  Box b2 = b1;
  // ...
  b1 = null;
  ```

  - Here, b1 has been set to null, but b2 still points to the original object.

- Adding a Method:
  - Method that returns value:

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

```
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

- Method that accept value:

```
class Box {

double width;

double height;

double depth;

// compute and return volume

double volume() {

return width * height * depth;

}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

- Constructors
    - Automatic initialization is performed through the use of a constructor.
    - A constructor initializes an object immediately upon creation.
    - It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
    - Constructors look a little strange because they have no return type, not even void.

```
Box( )
{
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
```

  - Parameterized constructor

```
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
```

- this keyword:
  - Sometimes a method will need to refer to the object that invoked it.
  - To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the current object.

```
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

- Overloading Methods:
  - It is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
  - When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
  - Thus, overloaded methods must differ in the type and/or number of their parameters.
  - While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
  - Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm.
  -

```
void test() {
System.out.println("No parameters");
}
```

```
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

- Overloading Constructors

```
// constructor used when all dimensions specified
Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box()
{
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
```

```
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
```

- Using Objects as Parameters:

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}


// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
```

- A Closer Look at Argument Passing:
    - There are two ways that a computer language can pass an argument to a subroutine.
    - The first way is call-by-value and second way is call-by-reference.
    - **In Java uses both approaches, depending upon what is passed.**
        - In call-by-value, method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

```
// Simple types are passed by value.
class Test {
void meth(int i, int j)
  {
i *= 2;
j /= 2;
  }
```

```
    }

class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

- In call-by-reference, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.
- When you pass an object to a method, the situation changes dramatically, because objects are passed by reference.
- When you pass object to a method, the parameter that receives it will refer to the same object as that referred to by the argument.

```
// Objects are passed by reference.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}

// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

- Returning Objects:
  - A method can return any type of data, including class types that you create.

```
class Test
{
            int a;
            Test(int i)
            {
            a = i;
             }
            Test incrByTen()
            {
            Test temp = new Test(a+10);
            return temp;
            }
}

class RetOb
{
    public static void main(String args[])
    {
            Test ob1 = new Test(2);
            Test ob2;
            ob2 = ob1.incrByTen();
            System.out.println("ob1.a: " + ob1.a);
            System.out.println("ob2.a: " + ob2.a);
            ob2 = ob2.incrByTen();
            System.out.println("ob2.a after second increase: "+ ob2.a);
    }
}
```

- Passing and returning array of object in function:

```
import java.util.Scanner;
public class array {
  public int max(int [] array) {
    int max = 0;
    for(int i=0; i<array.length; i++ ) {
      if(array[i]>max) {
        max = array[i];
      }
    }
    return max;
  }

  public int min(int [] array) {
    int min = array[0];

    for(int i = 0; i<array.length; i++ ) {
      if(array[i]<min) {
        min = array[i];
      }
    }
    return min;
  }
}
class Example
{
public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the array range");
    int size = sc.nextInt();
    int[] arr = new int[size];
    System.out.println("Enter the elements of the array ::");

    for(int i=0; i<size; i++) {
      arr[i] = sc.nextInt();
    }
    array m = new array();
    System.out.println("Maximum value in the array is::"+m.max(arr));
    System.out.println("Minimum value in the array is::"+m.min(arr));
  }
}
```

- Returning array:
  - Remember:
    - A method can return a reference to an array.
    - The return type of a method must be declared as an array of the correct data type.

```
import java.util.Arrays;
```

```
class ReturnArrayExample1
{
public static void main(String args[])
{
int[] a=numbers();          //obtain the array
for (int i = 0; i < a.length; i++) //for loop to print the array
System.out.print( a[i]+ " ");
}
public static int[] numbers()
{
int[] arr={5,6,7,8,9};  //initializing array
return arr;
}
}
```

- Static
  - There will be times when you will want to define a class member that will be used independently of any object of that class. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
  - Methods declared as static have several restrictions:
    - They can only call other static methods.
    - They must only access static data.
    - They cannot refer to this or super in any way.
  - static variable
  - static method

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
        static int a = 3;
        static int b;
        static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        }
        static {
        System.out.println("Static block initialized.");
        b = a * 4;
        }
        public static void main(String args[]) {
```

```
        meth(42);
        }
}
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

```
class StaticDemo {
            static int a = 42;
            static int b = 99;
            static void callme() {
            System.out.println("a = " + a);
                              }
            }
class StaticByName {
                  public static void main(String args[]) {
                  StaticDemo.callme();
                  System.out.println("b = " + StaticDemo.b);
                                                  }
                  }
```

- Introducing final
  - A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared.
    - Example of final variable
      - 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
```

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

```
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();
  }
}
```

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

```
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();
  }
}
```

Q) Is final method inherited?
Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Q) What is blank or uninitialized final variable?
Yes, but only in constructor.

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }  }
```

Q) Can we declare a constructor final?
No, because constructor is never inherited.

- garbage collection:
  - In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
  - Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection.
  - It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
  - Advantage of Garbage Collection:
    - It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
    - It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.
  - How can an object be unreferenced?
    - By nulling the reference

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

    - By assigning a reference to another

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

    - By anonymous object

      ```
      new Employee();
      ```

  - finalize() method
    The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

    ```
    protected void finalize(){}
    ```

  - gc() method
    The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

    ```
    public static void gc(){}
    public class TestGarbage1{
     public void finalize(){System.out.println("object is garbage collected");}
     public static void main(String args[]){
    ```

```
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

- access modifier (default, public, private, protected):
    - The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
    - The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
    - There are four types of Java access modifiers:
        - Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
        - Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
        - Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
        - Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

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

- 1) Private: The private access modifier is accessible only within the class.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

- 2) Default: If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

```
package pack;
class A{
  void msg(){System.out.println("Hello");}
}

package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

- 3) Protected: The protected access modifier is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class. It provides more accessibility than the default modifier.

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

//save by B.java
```

```
package mypack;
import pack.*;

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

- 4) Public: The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

```
//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();
  }
}
```