

Unit 3

Java as Object Oriented Programming Language-Overview

Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements ,String Handling: String class methods

Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method,

overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length arguments.

Topic	Book To Refer
<p>Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements ,String Handling: String class methods</p> <p>Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method, overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length arguments.</p>	<p>Herbert Schildt, "The Complete Reference Java", 9th Ed, TMH,ISBN: 978-0-07-180856-9.</p> <p>Programming With Java, 3rd Edition, E. Balaguruswamy</p>

Topic	Book To Refer
<p>String Handling: String class methods</p> <p>Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method, overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length arguments.</p>	<p>Herbert Schildt, "The Complete Reference Java", 9th Ed, TMH,ISBN: 978-0-07-180856-9.</p>

Topic	Book To Refer
String Handling: String class methods	Herbert Schildt, "The Complete Reference Java", 9th Ed, TMH,ISBN: 978-0-07-180856-9. Page No- 413-431

Introduction to JAVA Programming



What is Java?

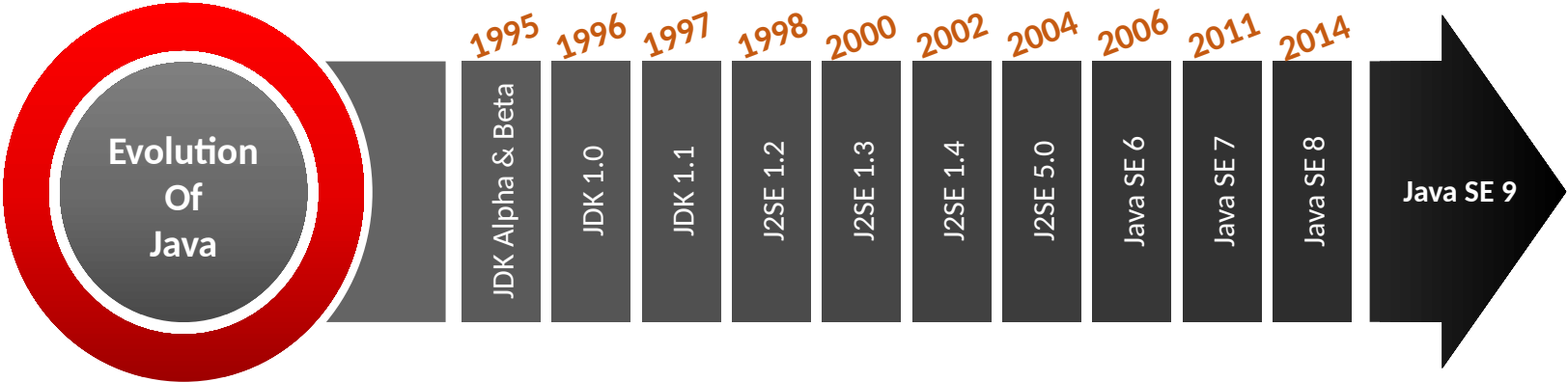
Java is a programming language and a platform. Java is a **high level, robust, object-oriented and secure programming language.**

History of Java

Java was developed by **Sun Microsystems (which is now the subsidiary of Oracle)** in the year 1995. **James Gosling is known as the father of Java.** Before Java, its name was **Oak**. Since Oak was already a registered company, **so James Gosling and his team changed the Oak name to Java.**

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Java Version History



Features of Java

01

Simple

02

Object-Oriented

03

Platform independent

04

Secure language

05

Robust

06

Architecture-neutral

07

Interpreted language

08

Multithreaded language

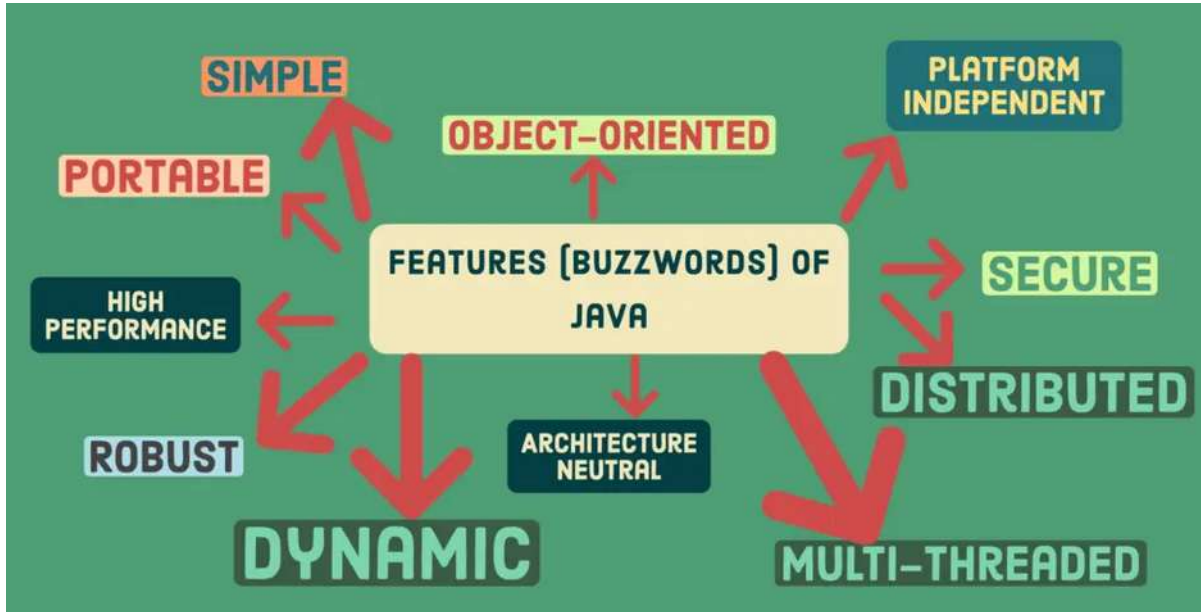
09

Distributed language

10

Dynamic

Features of Java



Java is Simple

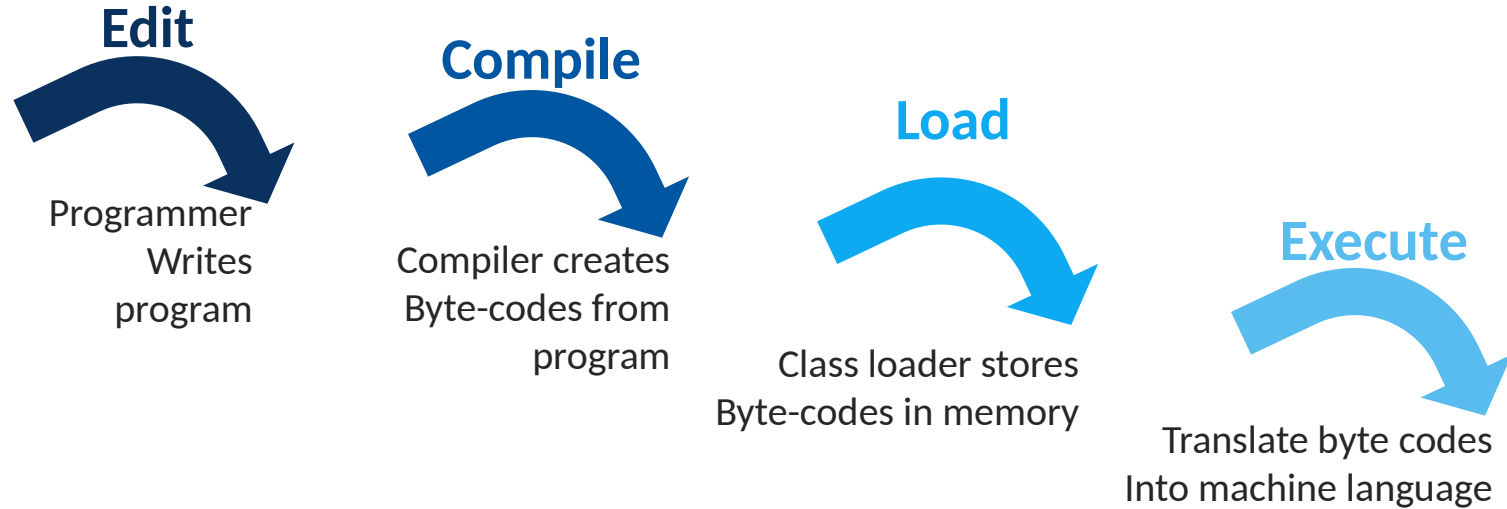
- It is **free from pointer** due to this execution time of application is improved.
[Whenever we write a Java program without pointers then internally it is converted into the equivalent pointer program].
- It has **Rich set of API** (application protocol interface).
- It has **Automatic Garbage Collector** which is always used to collect un-Referenced (unused) Memory location for improving performance of a Java program.
- It contains **user friendly syntax for developing any applications.**

JAVA Compiler and Interpreter



Java Life Cycle

Java Programs Normally Undergo Four Phases



The execution lifecycle of a Java application can be broadly divided into three phases:

1.Compilation: The source code of the application is converted into bytecode using the “javac” compiler.

2.Class Loading: The bytecode is loaded into memory and the necessary class files are prepared for execution.

3. Bytecode Execution: The JVM executes the bytecode and the program runs.

1. Java Bytecode is the intermediate representation of your Java code that is executed by the Java Virtual Machine (JVM).

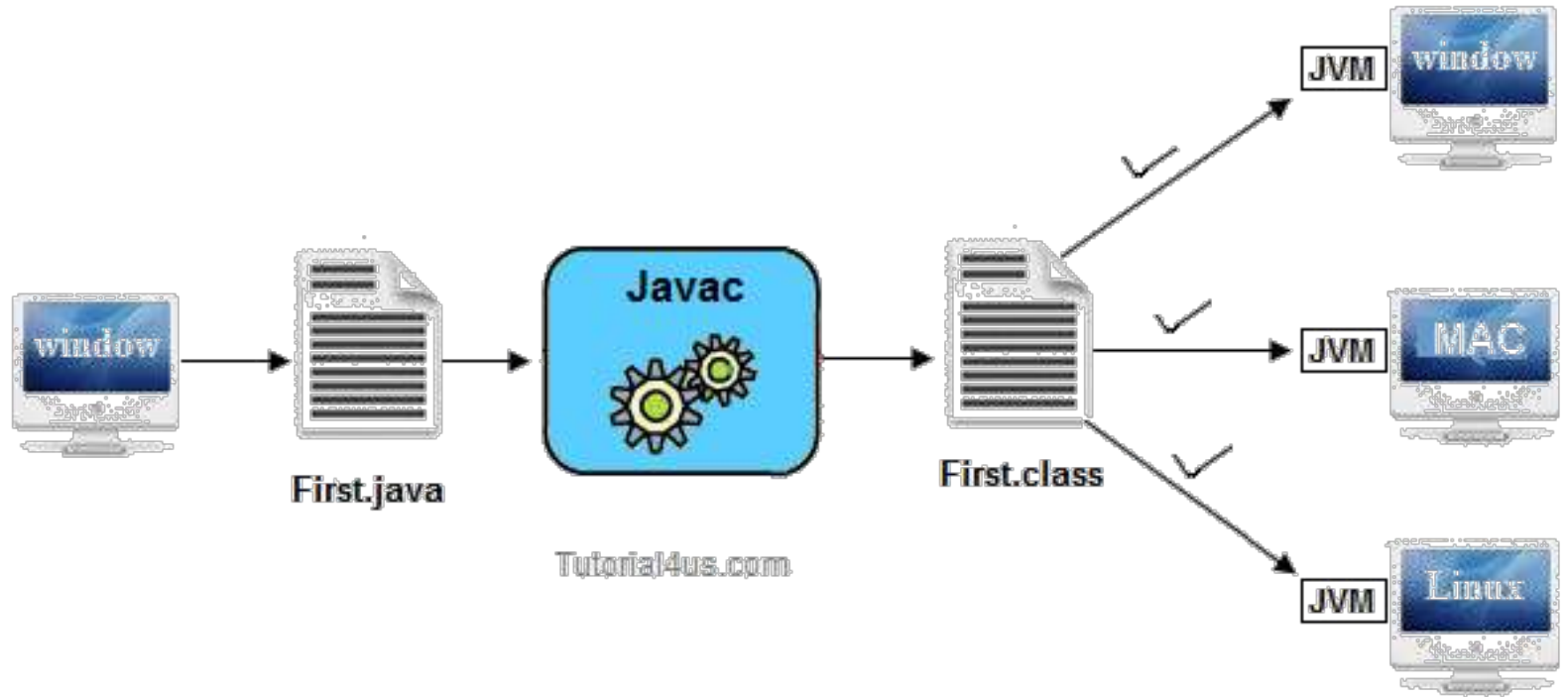
2. When you compile a Java program, the **Java compiler (javac) converts your code into bytecode**, which is a set of instructions that the JVM can understand and execute.

3. This bytecode is platform-independent, meaning the same Java program can run on different devices and operating systems, a principle known as "write once, run anywhere" (WORA).

Java is Object Oriented

- Since it is an object-oriented language, it will support the following features:
 - Class
 - Object
 - Encapsulation
 - Abstraction
 - Inheritance
 - Polymorphism

Java is Platform Independent



1. Java is platform-independent because it uses a virtual machine.
2. The Java programming language and all APIs are compiled into bytecodes.
3. Bytecodes are effectively platform-independent. The virtual machine takes care of the differences between the bytecodes for the different platforms

JVM, JDK ,JRE



Difference between JDK, JRE and JVM

Java Development Kit

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.

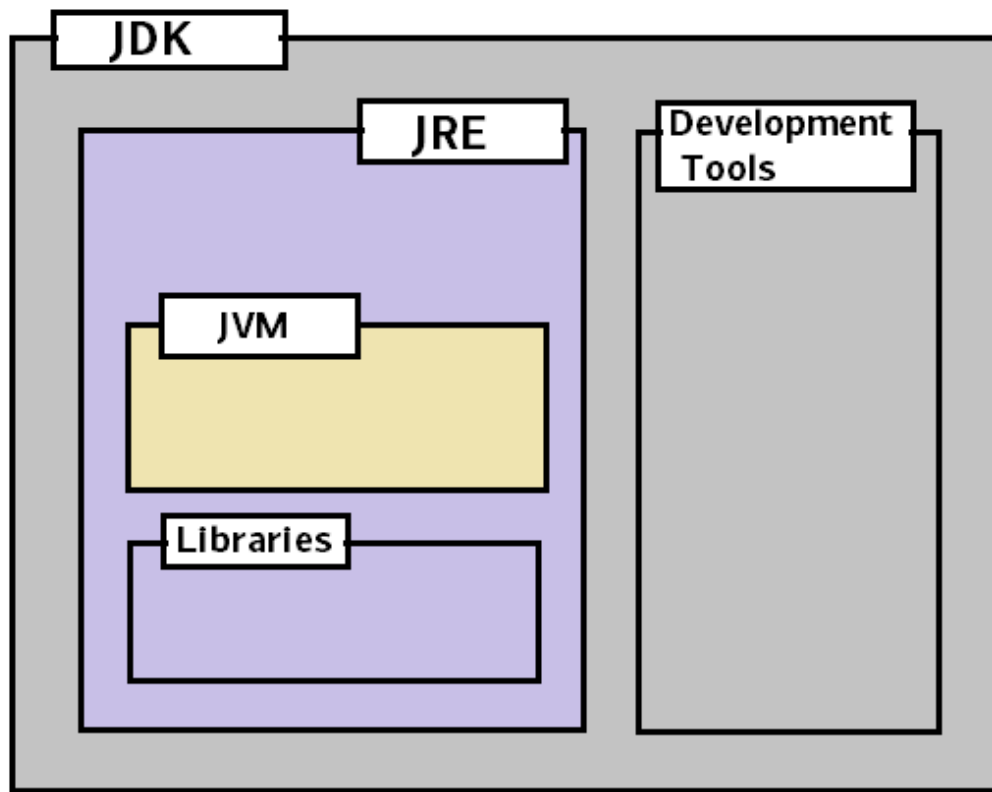
Java Runtime Environment

JRE is used to provide runtime environment. It is the implementation of JVM. It physically exists.

Java Virtual Machine

JVM is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JDK is a software development kit whereas **JRE is a software bundle** that allows Java program to run, whereas **JVM is an environment** for executing bytecode.

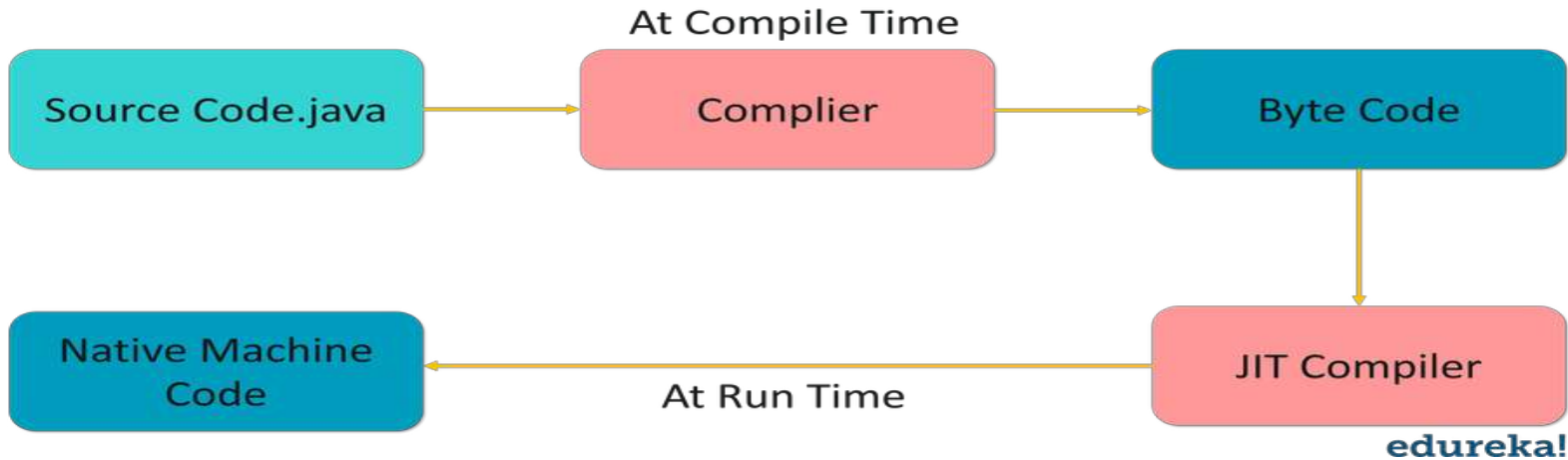


Let's look at some of the important differences between JDK, JRE, and JVM.

1.JDK is for development purpose whereas JRE is for running the java programs.

2.JDK and JRE both contains JVM so that we can run our java program.

3.JVM is the heart of java programming language and provides platform independence.

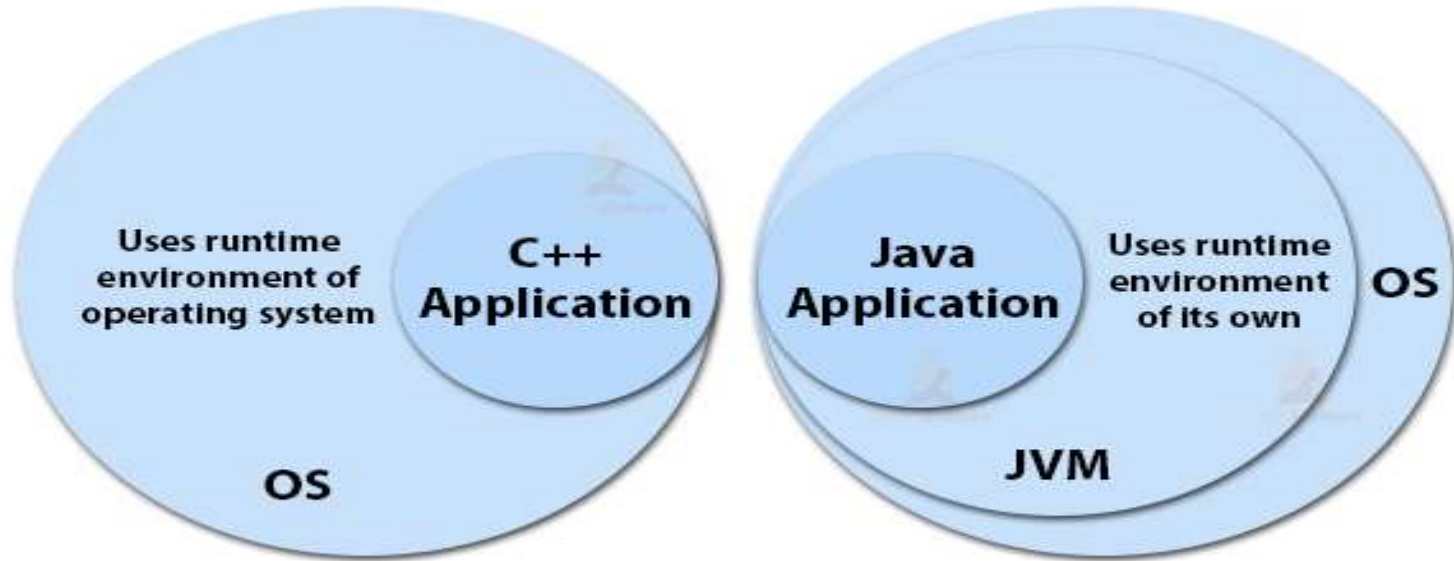


The **JIT compiler** helps improve the performance of Java programs by compiling bytecodes into native machine code at run time. The JIT compiler is enabled by default. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.

Difference between JVM & JIT

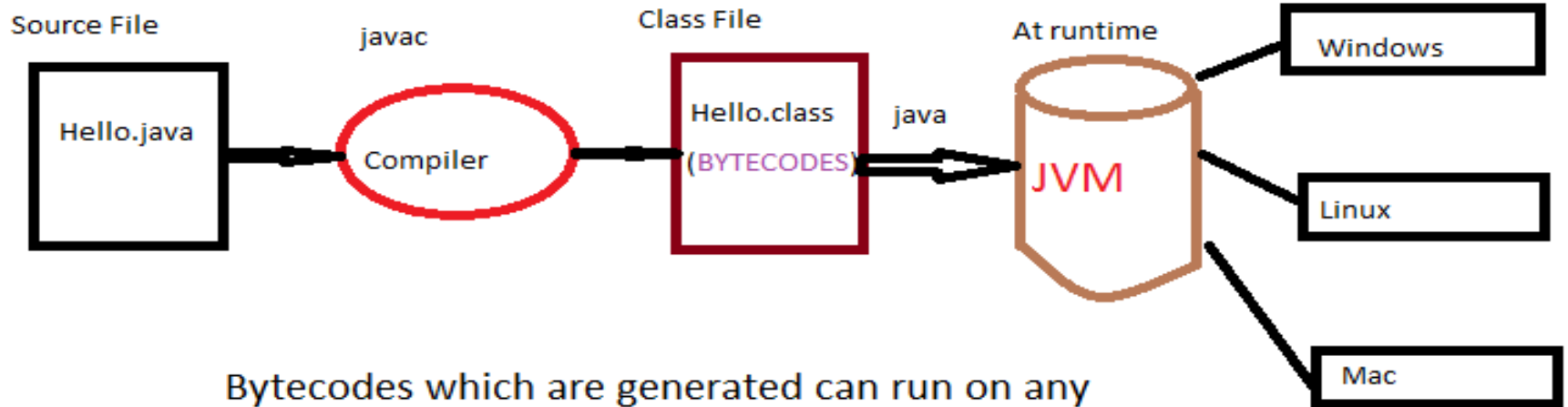
**Java Virtual Machine is an interpreter that converts the bytecode into the machine's native language, whereas JIT (Just In Time) is responsible for improving the environment of Java.

Java is Secure



Security in Applications

Java is Secure

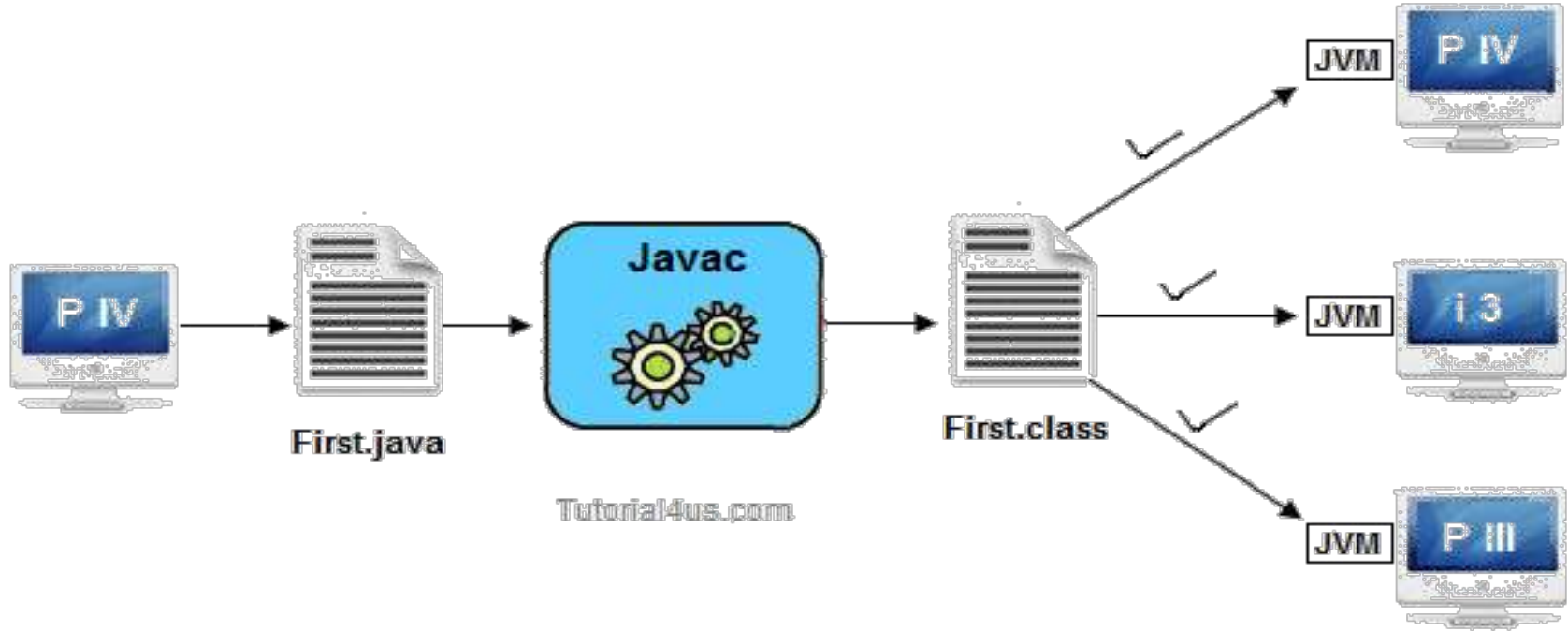


Bytecodes which are generated can run on any machine which has the JVM and are secure because JVM itself checks the code at runtime.

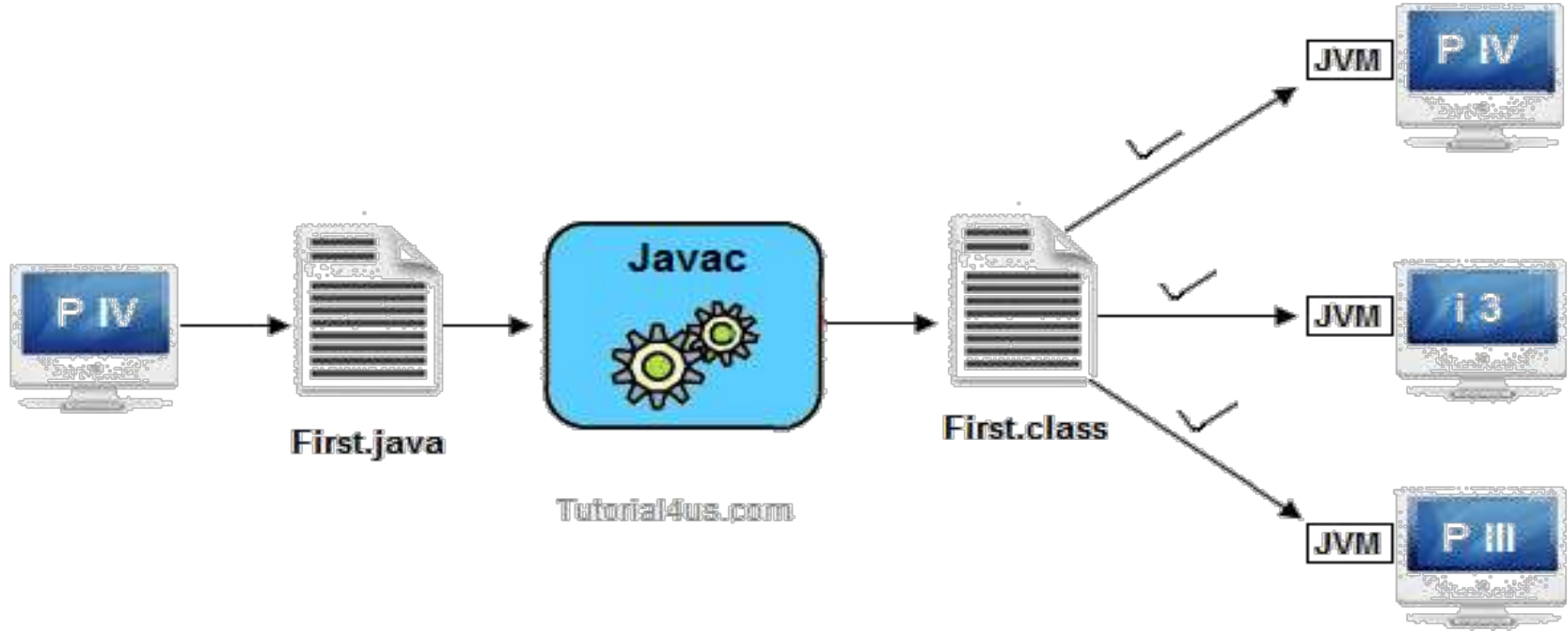
Java is Robust

- Capable of handling run-time errors,
- Supports automatic garbage collection
- Exception handling, and
- Avoids explicit pointer concept.

Java is Architecture Neutral



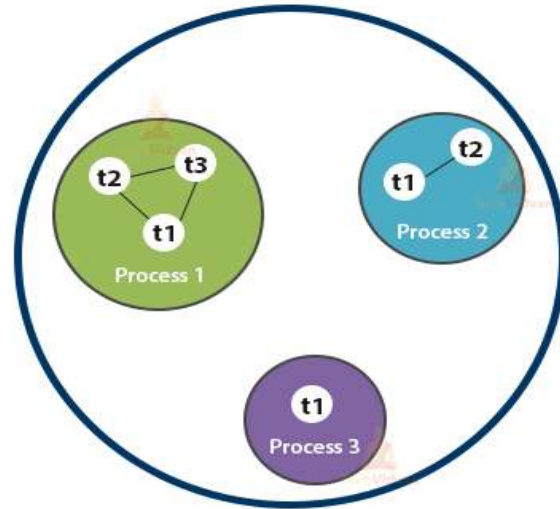
Java is Architecture Interpreted



Java is Multithreading



Multithreading in Java



Operating System

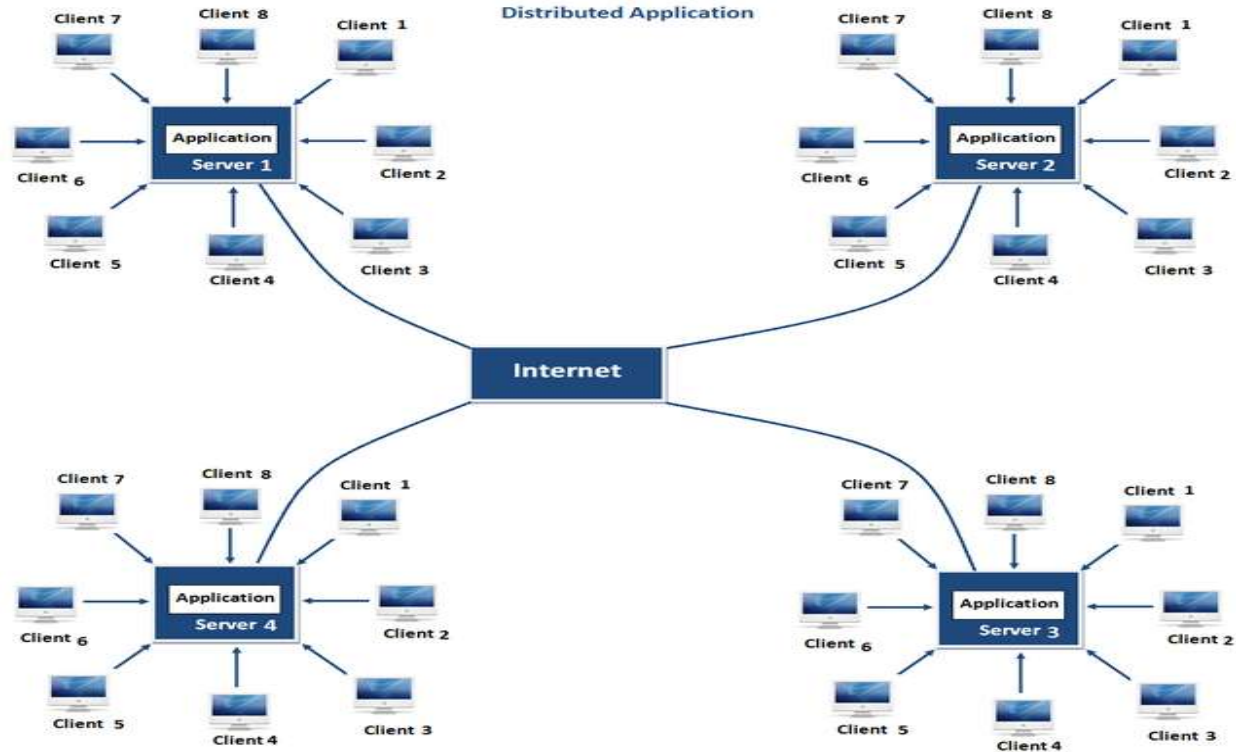
Java is Multithreading

- Multithreaded means **handling multiple tasks simultaneously** or executing multiple portions (functions) of the same program in parallel.
- The code of **java is divided into smaller parts and Java executes them in a sequential and timely manner.**

Java is Distributed

- Multiple programmers **at many locations to work together on a single project.**
- Support **RMI (Remote Method Invocation) and EJB (Enterprise JavaBeans).**
- Extensive library of classes for interacting, **using TCP/IP protocols such as HTTP and FTP**, which makes creating network connections much easier than in C/C++.

Java is Distributed



Java is Dynamic

- Classes are **not loaded all at once**.
- They jump into action only when **an invoke operation executes or some data about the class is needed in the memory**.
- Java finalizes **invoking instructions during runtime**. Ex- Runtime Polymorphism i.e **function overriding**.

Java Editions

J2SE

**Java 2 Standard
Edition**

Java standard edition is use to develop client-side standalone applications or applets

J2ME

Java 2 Micro Edition

Java micro edition is use to develop applications for mobile devices such as cell phones

J2EE

Java 2 Enterprise Edition

Java enterprise edition is use to develop server-side applications such as Java servlets and Java Server Pages

First “Hello World” program using JAVA

```
 HelloWorld.java
package sct;

public class HelloWorld {
    /**
     * comments
     */
    public static void main(String[] args) {
        // Comment: printing output on console
        System.out.println("Hello World from Java");
    }
}
```



“package sct”

- It is package declaration statement.
- defines a namespace in which classes are stored.
- to organize the classes based on functionality.
- If you omit the package statement, the class names are put into the **default package** **java.lang**, which has no name.

2. “public class HelloWorld”

- This line has various aspects of java programming.
- **public:** This is access modifier keyword **which tells compiler access to class.**
Various values of access modifiers can be **public, protected,private or default (no value).**
- **class:**

3. “Comments”

- **Line comments:** It starts with **two forward slashes (//)** and **continues to the end of the current line.** Line comments do not require an ending symbol.
- **Block comments:** start with a forward slash and **an asterisk (/*)** and **end with an asterisk and a forward slash (* /).**Block comments can also extend

4. “public static void main (String [] args)”:

public: This keyword means that the method is accessible anywhere, including from outside the class it’s declared in.

static: By using ‘static’, we’re saying that the main method can be run without needing an instance of the class.

void: This keyword indicates that the main method doesn’t return any value.

main: ‘main’ is the name of this method. The JVM looks for a method with this name when it starts running a program.

String[] args: This is an array of ‘String’ objects. It’s used to receive any command-line arguments that were passed when the program was started.

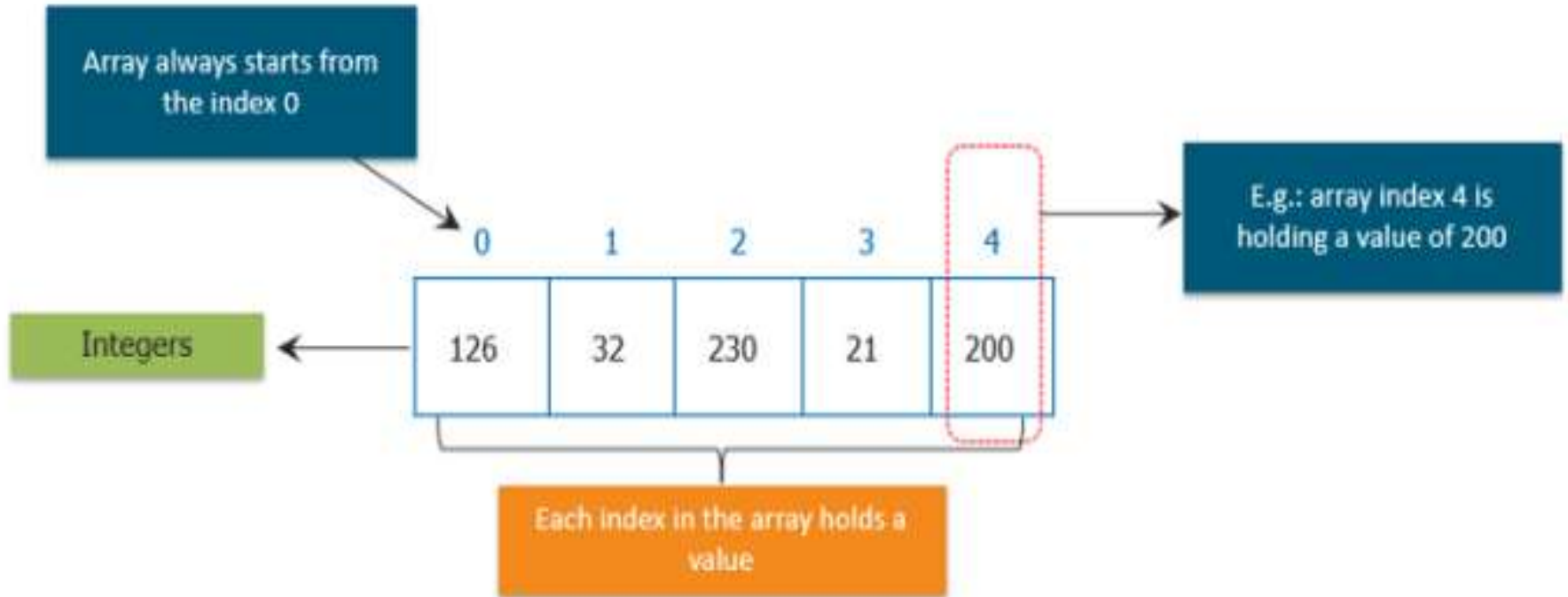
5. `System.out.println("Hello World from Java")` :

- **System:** It is the name of Java utility class.
- **out:** It is an object which belongs to System class.
- **println:** It is utility method name which is used to send any String to the console.
- **“Hello World from Java”:** It is String literal set as argument to println method.

ARRAY in JAVA Programming



JAVA: Introduction to Array Data Type



JAVA: Introduction to Array Data Type

- Arrays in Java are **homogeneous data structures** implemented in Java as objects.
- Arrays **store one or more values** of a specific data type and provide indexed access to store the same.
- A **specific element in an array is accessed by its index.**
- Arrays offer a **convenient means of grouping related information.**

JAVA: Introduction to Array Data Type

- Obtaining an array is a **two-step process**.
 - 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**

JAVA: General Form of Java Array Initialization

The *type* determines what type of data the array will hold

type *var-name*[];



Example:- *int month_days*[];

JAVA: General Form of Java Array Initialization

The **type** determines what type of data the array will hold

new is a special operator that allocates memory.

type array-var = new type[size];

array-var is the array variable that is linked to the array.

size specifies the number of elements in the array

1

data type

size of array

```
int[] a = new int[5];
```

Starts

0	1	2	3	4
0	0	0	0	0

Ends

2

data type

size of array

```
int a[] = new int[5];
```

Index has to be given in
square brackets

Starts

0	1	2	3	4
0	0	0	0	0

Ends

3

data type

size of array

```
int[] a = new int[] {1, 2, 3, 4, 5};
```

Index has to be given in
square brackets

Starts

0	1	2	3	4
1	2	3	4	5

Ends

JAVA: More About Array Initialization...

The **type** determines what type of data the array will hold



```
type var-name[] = {value1, value2, value3, value4,...};
```



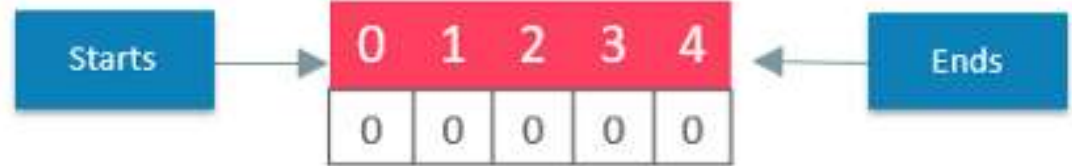
An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements

JAVA: More About Array Initialization...

data type size of array

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

Index has to be given in square brackets



JAVA: Implementing an Array

```
class MyArray{  
  
    public static void main(String args[]){  
  
        int month_days[ ] = {31,28,31,30,31,30,31,30,31,30,31};  
  
        System.out.println("April has " + month_days[3] + days.);  
  
    }  
  
}
```

OUTPUT

April has 30days

JAVA: Accessing a Specific Element in a Java Array



This statement
assigns the value
90 to the second
element of
month_days

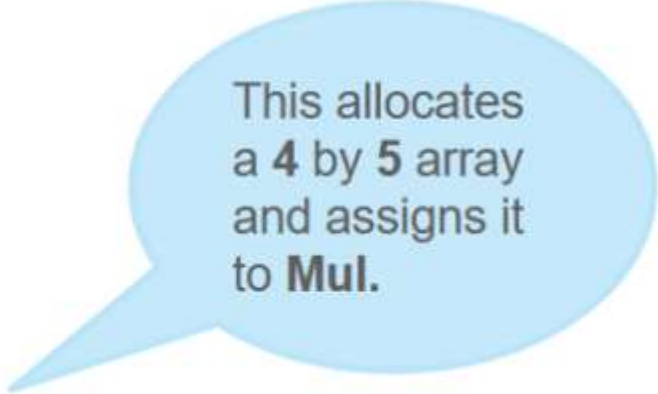
```
month_days[1] = 90;
```

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

OUTPUT

April has 30days

JAVA: Multidimensional Array



This allocates
a **4** by **5** array
and assigns it
to **Mul**.

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

JAVA: Multidimensional Array -Conceptually

edureka!

Left index
determines row.



```
class TwoDArray
{
    //-----
    // Creates a 2D array of integers, fills it with increasing
    // integer values, then prints them out.
    //-----
    public static void main (String[] args)
    {
        int[][] multarry = new int[4][5];
        int i,j,k=0;

        // Load the table with values
        for (i=0; i < 4;i++)
            for (j=0; j < 5; j++)
            {
                multarry[i][j]=k;
                k++;
            }
    }
}
```

```
// Print the table
for (i=0; i < 4;i++)
{
    for (j=0; j < 5; j++)
    {
        System.out.print( multarry[i][j]+" ");
    }

    System.out.println();

}
}
}
```

OUTPUT

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```


JAVA: Multidimensional arrays representation of different data types.

```
int [][]a= new int [2][2];
```

	0	1
0	1	4
1	4	5

2 x 2 dimensional int array

```
char [][]a= new char[3][2];
```

	0	1
0	s	a
1	g	v
2	v	d

JAVA: Multidimensional arrays representation of different data types.

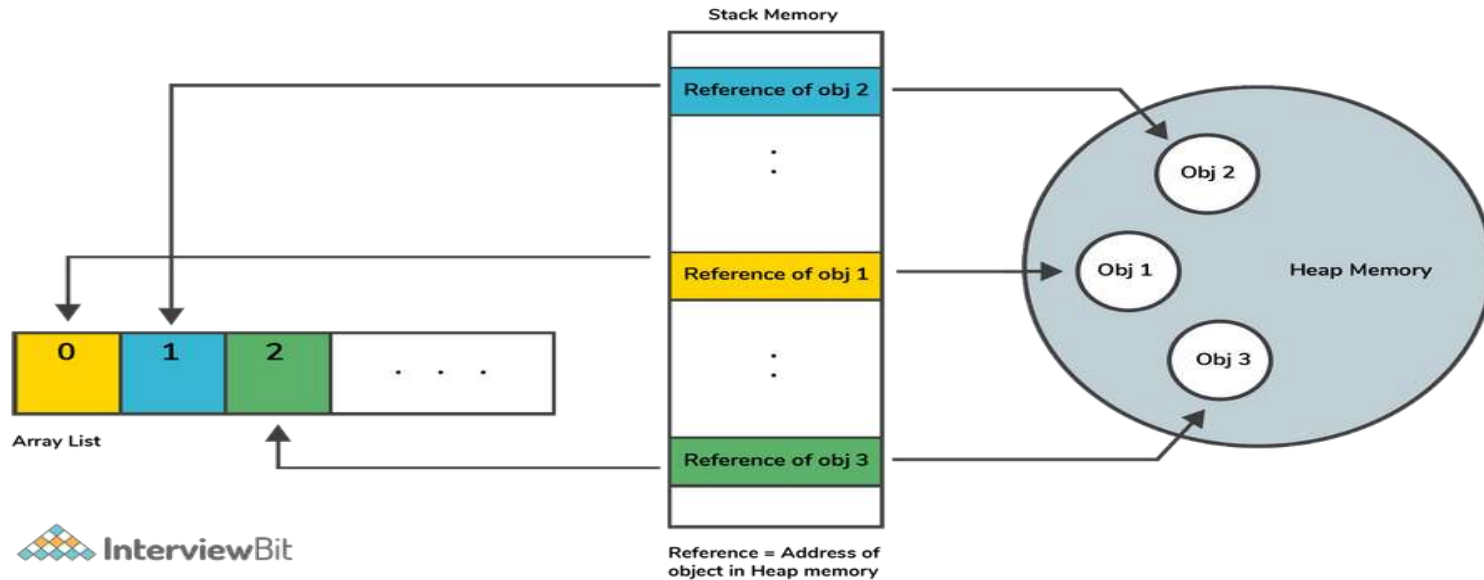
```
float [][]a= new float[5][5];
```

	0	1	2	3	4
0	2.2	3.4	5.0	3.3	1.2
1	7.8	9.0	1.1	2.9	5.5
2	2.0	3.0	7.8	9.8	9.9
3	5.7	6.6	8.8	5.3	2.7
4	1.8	4.4	7.6	1.0	1.1

5 x 5 dimensional float array

Array Vs ArrayList

Sorting of objects in Array List

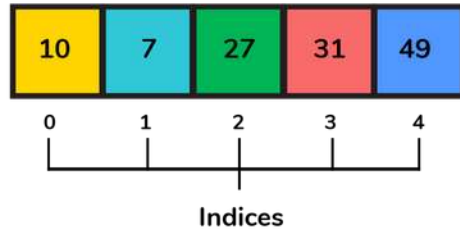


Array Vs ArrayList

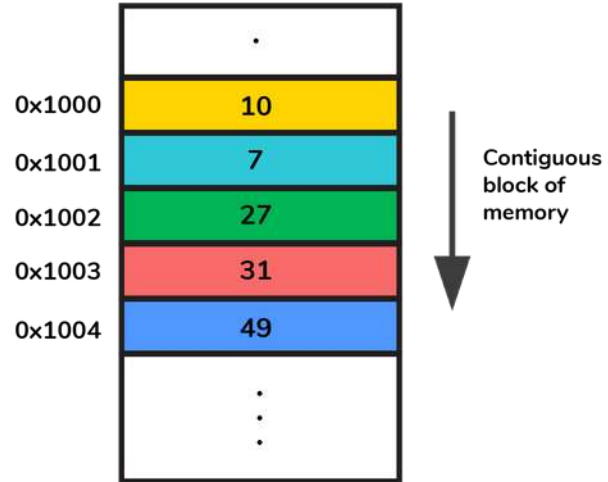
Sorting values in Array

How Array look like?

Array



Array in memory



JAVA: Passing Java Array to a Method

```
class PMethods{
public static void display(int y[])
{
    System.out.println(y[0]);
    System.out.println(y[1]);
    System.out.println(y[2]);

}
public static void main(String args[])
{
    int x[] = { 1, 2, 3 };
    display(x); //Passed array x to method display
}
}
```

OUTPUT

1
2
3

[Click Here](#)

Java Interview Questions on Array

Memory Allocation
and
Java Garbage Collection

Java Heap Space

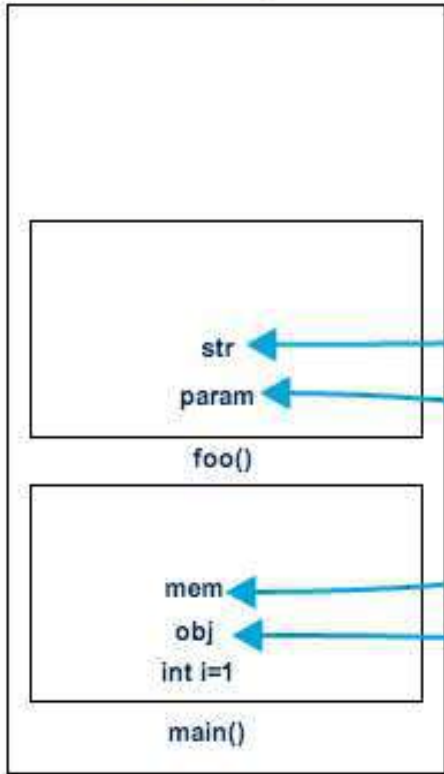
- Used by Java runtime **to allocate memory to Objects and JRE classes.**
- Any **new Object** is always created in **Heap Space.**
- Garbage Collection **runs on the heap memory to free the memory used** by objects that **doesn't have any reference.**
- **All instance and class variables** are also stored in the heap.

Java Stack Memory

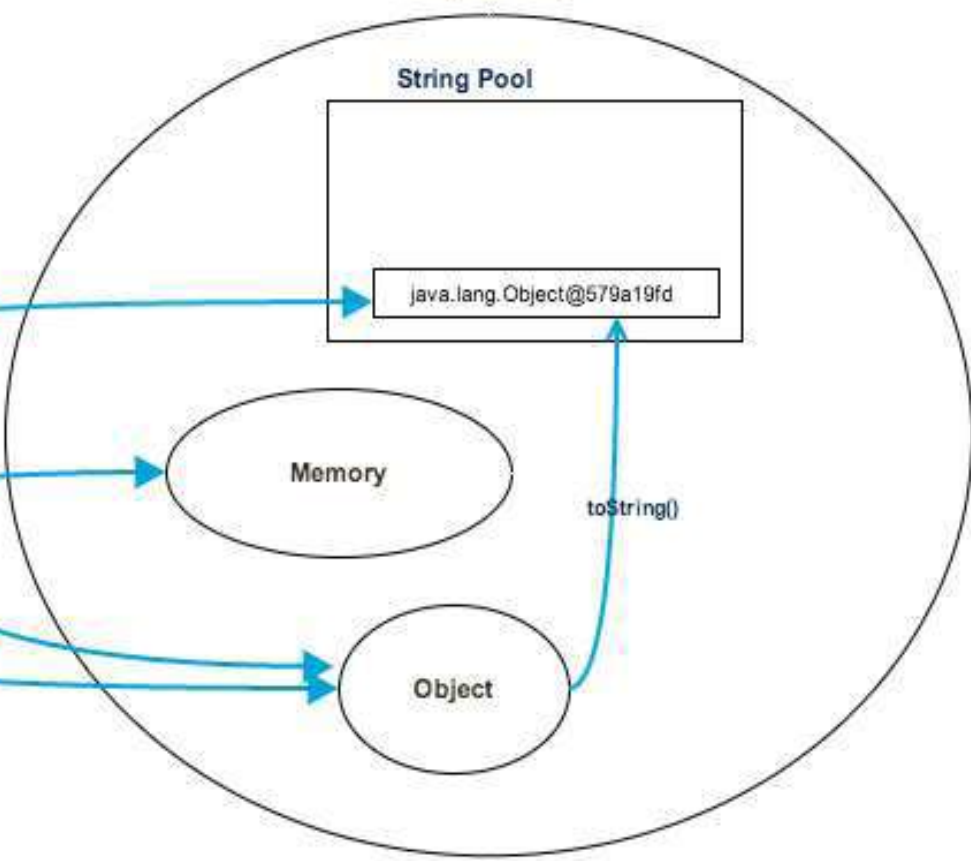
- Used for **execution of a thread**.
- Store method **specific values**, and **“references” to Objects** being used in the method.
- Stack memory is **LIFO (Last-In-First-Out)**
- Whenever a **method is invoked**, a **new block is created** in the stack memory for the method **to hold local primitive values and reference to other objects** in the method.
As soon as **method ends**, the **block becomes unused and become available for next method**.
- Stack memory size is **very less compared to Heap memory**.

```
public class Memory {  
  
    public static void main(String[] args) { // Line 1  
        int i=1; // Line 2  
        Object obj = new Object(); // Line 3  
        Memory mem = new Memory(); // Line 4  
        mem.foo(obj); // Line 5  
    } // Line 9  
  
    private void foo(Object param) { // Line 6  
        String str = param.toString(); //// Line 7  
        System.out.println(str);  
    } // Line 8  
}
```

Stack Memory



Heap Memory



`str`

`param`

`foo()`

`mem`

`obj`

`int i=1`

`main()`

`Memory`

`Object`

`java.lang.Object@579a19fd`

String Pool

`toString()`

String Handling in Java

Creating String in Java

There are two ways to create a String in Java

- String literal

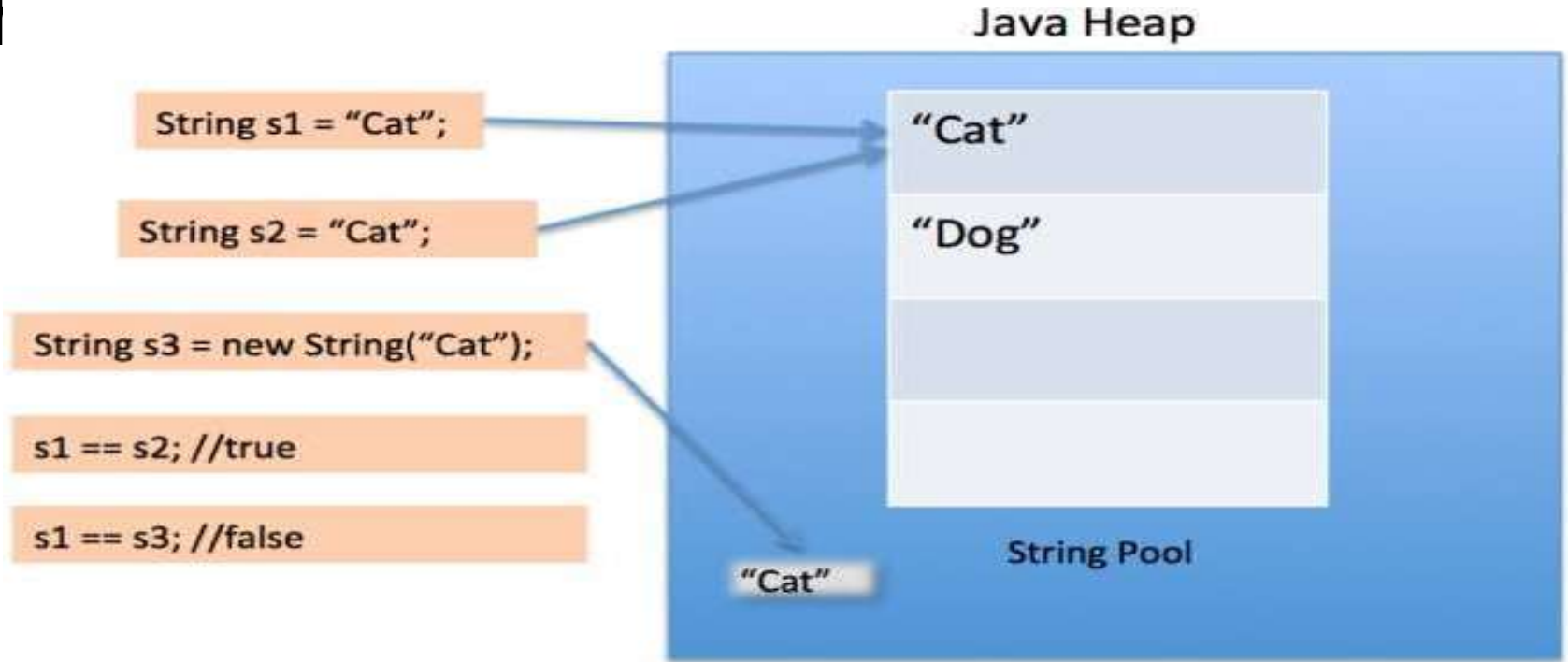
```
String str1 = "Welcome";  
String str2 = "Welcome";
```

- Using “new” keyword

```
String str1 = new String("Welcome");  
String str2 = new String("Welcome");
```

Does it make any difference? Well, yes!

String Pool Concept in Java (String Interning)



String Intern Pool maintained in Java Heap Space

Discussion: How many Strings are getting created here?

```
String str = new String("Cat");
```

String is Immutable in Java

```
String s1 = "Sky";  
String s2 = "Blue"
```

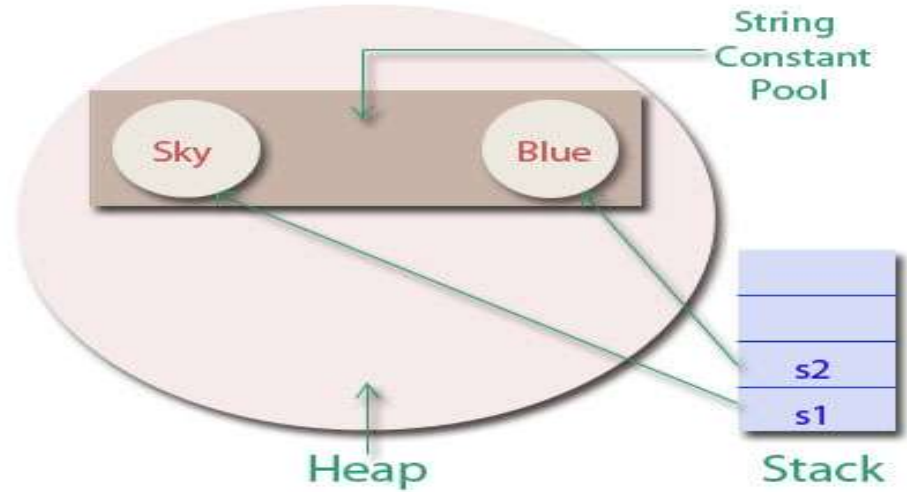


Figure 1

String is Immutable in Java

```
s1 = s1 + s2;
```

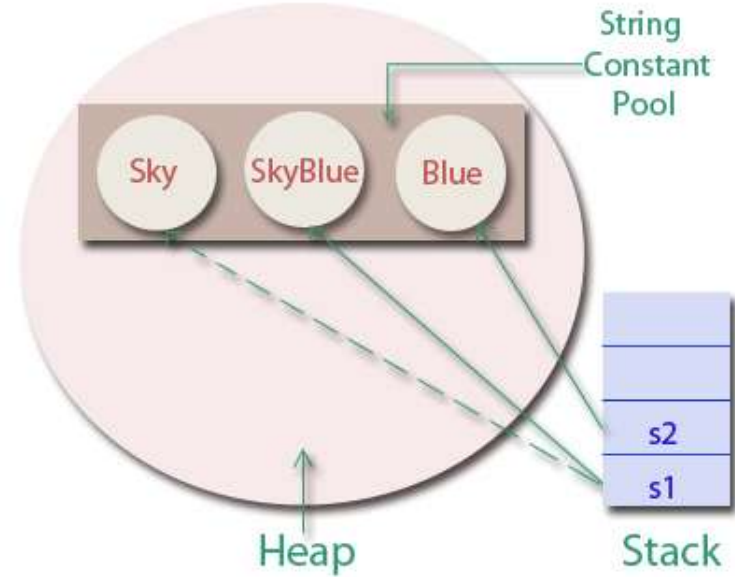


Figure 2

String Pool Concept in Java (String Interning)

- String is **immutable in Java**
- All Strings are stored in **String Pool** (also called String Intern Pool) allocated within **Java Heap Space**
- It is implementation of **String Interning Concept**.
- **String interning** is a method of storing only one copy of each distinct string value, which must be immutable.
- Interning strings makes some string processing tasks **more time- or space-efficient** at the **cost of requiring more time** when the string is created or interned.
- The distinct values are stored in a **string intern pool**.
- Using new operator, we force String class to create a new String object in heap space.

String Pool Concept in Java (String Interning)

```
public class InternExample{  
public static void main(String args[]){  
String s1=new String("hello");  
String s2="hello";  
String s3=s1.intern(); //returns string from pool, now it will be same as s2  
System.out.println(s1==s2);//false because reference variables are pointing to different instance  
System.out.println(s2==s3);//true because reference variables are pointing to same instance  
}}
```

java.lang.String API - Important

```
public class String
```

```
    String(String s)
```

create a string with the same value as s

```
    int length()
```

number of characters

```
    char charAt(int i)
```

the character at index i

```
    String substring(int i, int j)
```

characters at indices i through (j-1)

```
    boolean contains(String substring)
```

does this string contain substring?

```
    boolean startsWith(String pre)
```

does this string start with pre?

```
    boolean endsWith(String post)
```

does this string end with post?

```
    int indexOf(String pattern)
```

index of first occurrence of pattern

```
    int indexOf(String pattern, int i)
```

index of first occurrence of pattern after i

java.lang.String API - Important methods

<code>String concat(String t)</code>	<i>this string with t appended</i>
<code>int compareTo(String t)</code>	<i>string comparison</i>
<code>String toLowerCase()</code>	<i>this string, with lowercase letters</i>
<code>String toUpperCase()</code>	<i>this string, with uppercase letters</i>
<code>String replaceAll(String a, String b)</code>	<i>this string, with as replaced by bs</i>
<code>String[] split(String delimiter)</code>	<i>strings between occurrences of delimiter</i>
<code>boolean equals(Object t)</code>	<i>is this string's value the same as t's?</i>
<code>int hashCode()</code>	<i>an integer hash code</i>

java.lang.String API - Examples

```
String a = new String("now is");  
String b = new String("the time");  
String c = new String(" the");
```

<i>instance method call</i>	<i>return type</i>	<i>return value</i>
a.length()	int	6
a.charAt(4)	char	'i'
a.substring(2, 5)	String	"w i"
b.startsWith("the")	boolean	true
a.indexOf("is")	int	4
a.concat(c)	String	"now is the"
b.replace("t", "T")	String	"The Tim"
a.split(" ")	String[]	{ "now", "is" }
b.equals(c)	boolean	false

java.lang.String API - Examples

```
public class EqualsSample{
    public static void main(String args[]){
        String s1="string";
        String s2="string";
        String s3="swing";
        String s4= "      ABC      ";

        System.out.println(s1.equals(s2)); // true because both
are equal

        System.out.println(s1.equals(s3)); //false because both
are not equal
```

java.lang.String API - Examples

```
System.out.println(s1.length()); // 5 is the length of s1
```

```
System.out.println(s1.compareTo(s2)); //0 as both are equal
```

```
System.out.println(s1.compareTo(s3)); //-3 as 't' in s1 is less than  
'w' in s2
```

```
System.out.println(s4.trim() + ":wordpress.com");  
//ABC.wordpress.com
```

```
System.out.println(s1.concat(s4)); //string      ABC
```

```
System.out.println(s1.toUpperCase()); //STRING
```

```
}
```


java.lang.String API - Examples

```
System.out.println(s1.charAt(4)); // n
```

```
}
```

```
}
```

Converting String to numbers and vice versa

● String to Number

- `int i = Integer.parseInt(str);`
- `Integer i = Integer.valueOf(str);`
- `double d = Double.parseDouble(str);`
- `Double d = Double.valueOf(str);`

Note: Both throw `NumberFormatException` If the String is not valid for conversion

Converting String to numbers and vice versa

- String to Boolean

- `boolean b = Boolean.parseBoolean(str);`

- Any Type to String

- `String s = String.valueOf(value);`

[Click Here](#)

Java Interview Questions on String

Classes and Method

- Core of Java.
- Logical construct upon which the entire Java language is built
- Defines the shape and nature of and object.

Class Fundamentals

- A class is that it defines a **new data type**.
- Once defined, this new type can be used **to create objects of that type**.
- 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
- Two word **object and instance used interchangeably**.

The General Form of a

- **The data** that it contains
and **the code** that
operates on that data

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


The General Form of a Class

- **The data** that it contains and **the code that operates on that data**
- The **data**, or variables, defined within a class **are called instance variables**
- The code is contained within methods
- Collectively, the **methods and variables** defined within a class **are called members of the class.**

The General Form of a Class

- Variables defined **within a class are called instance variables** because each instance of the class (that is, each object of the class) **contains its own copy of these variables.**
- Thus, the data for one object is **separate and unique** from the data for another.

The General Form of a Class

- All methods have the **same general form as main()**.
- However, most methods **will not be specified as static or public**
- the general form of a **class does not specify a main() method**.
- Java classes do not need to have main() method. You only specify one if **that class is the starting point for your program**.
- Further **,some kinds of Java applications, such as applets, don't require a main() method at all**.

A Simple Class

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// This class declares an object of type Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;

        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);
    }
}
```

Declaring Objects

- Obtaining objects of a class is a **two-step process**.
- First, you must **declare a variable of the class type**. This variable does **not define an object**. Instead, it is simply a variable that can refer to an object.
- Second, **you must acquire an actual, physical copy**

Declaring Objects

- The **new operator dynamically allocates** (that is, allocates at **run time**) **memory** for an object **and returns a reference to it.**
- This reference is, more or less, **the address in memory of the object allocated by new**
- **This reference is then stored in the variable.** Thus, in Java, all class objects must be dynamically allocated.

```
Box mybox = new Box ();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object  
mybox = new Box(); // allocate a Box object
```

Statement

Box mybox;

image

Effect



mybox

mybox = new Box();



mybox



Box object

Statement

Box mybox;

image

Effect



mybox

mybox = new Box();



mybox



Box object

A Closer Look at new

- The new operator **dynamically allocates (that is, allocates at run time) memory for an object.**
- It has this general form:

```
class-var = new classname ( );
```

A Closer Look at new

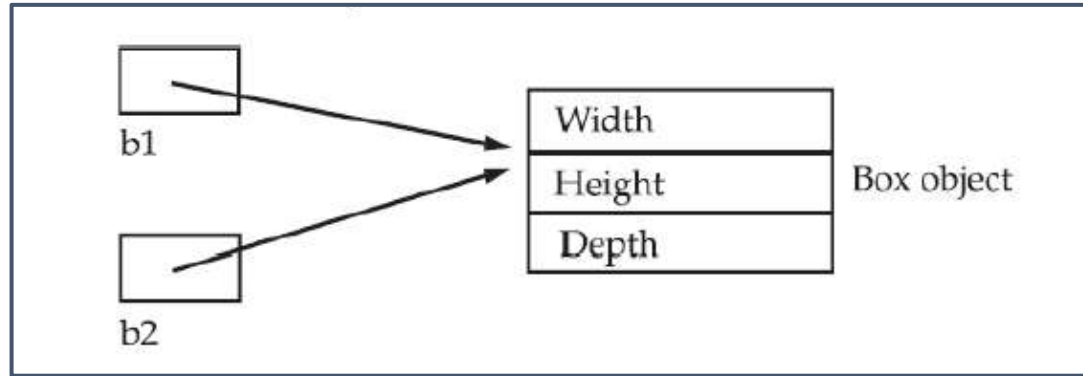
- The class name **followed by parentheses specifies the constructor for the class.**
- A constructor defines **what occurs when an object of a class is created.**
- Constructors are an **important part** of all classes and **have many significant attributes.**
- Most real-world classes **explicitly define their own constructors within their class definition.**

A Closer Look at new

- if no explicit constructor is specified, then Java **will automatically supply a default constructor**
- Java's **primitive types** are not implemented as objects. Rather, they are **implemented as “normal” variables**.
- **Advantage of new** : program can **create as many or as few objects** as it needs during the execution of your program.
- **memory is finite**, it is possible that new will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, **a run-time exception will occur**.

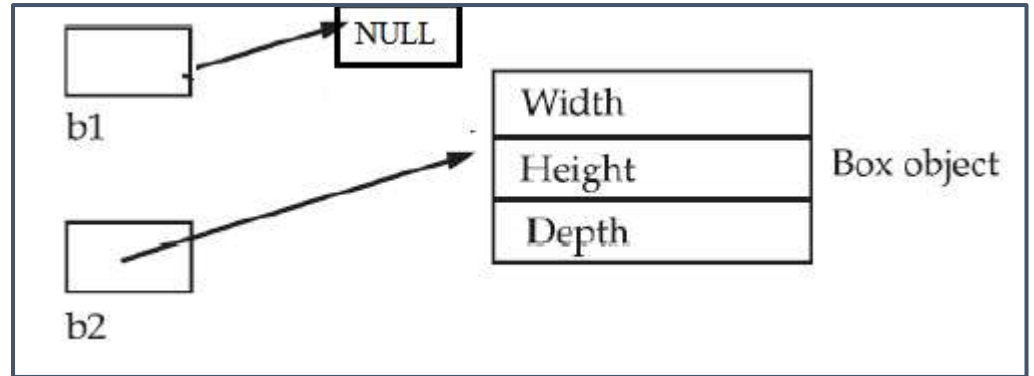
Assigning Object Reference Variables

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



Assigning Object Reference Variables

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



Assigning Object Reference Variables

REMEMBER

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

general form of a method:

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



Adding a Method to the Class

- In fact, methods **define the interface to most classes**. This allows the class **implementor to hide the specific layout of internal data structures** behind cleaner method abstractions.
- In addition **to defining methods that provide access to data**, you can also define **methods that are used internally by the class itself**.

```
// This program includes a method inside the box class.
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // display volume of a box  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```

```
class BoxDemo3 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
  
        // 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;  
  
        // display volume of first box  
        mybox1.volume();  
  
        // display volume of second box  
        mybox2.volume();  
    }  
}
```

image

Adding a Method to the Class

- **The instance variables** width, height, and depth **are referred to directly,**
- **without** preceding them with an **object name or the dot operator.**
- When an instance variable is accessed by **code that is not part of the class** in which that instance variable is defined, it **must be done through an object, by use of the dot operator.**

Returning a Value

- The type of **data returned by a method must be compatible with the return type** specified by the method.
- For example, if the return type of some method is boolean , you could not return an integer.
- The variable **receiving the value returned by a method** (such as vol in this case) must also be **compatible with the return type specified for the method**

Returning a 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);
    }
}
```

Adding a Method That Takes Parameters

```
int square(int i)
{
    return i * i;
}
```

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

Returning a Value

- A **parameter** is a variable defined by a method **that receives a value when the method is called.**
- For example, in `square()`, **`i` is a parameter.**
- An **argument** is a value that is passed to a method when it is invoked.
For example, `square(100)` passes **100** as an argument.
- Inside `square()`, the parameter **`i` receives that value.**

Modified program using parameterized methods

```
// This program uses a parameterized method:
```

```
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);  
    }  
}
```


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** when the object is created, **before the new operator completes.**

Constructor

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

```
class BoxDemo6 {
    public static void main(String[] args) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // 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);
    }
}
```

Constructor

- The **default constructor automatically initializes all instance variables** to their **default values**,
 - which are zero for numeric types,
 - null for reference types ,
 - and false for boolean
- The default constructor **is often sufficient for simple classes**, but it usually won't do for more sophisticated ones.
- Once you define **your own constructor**, the **default constructor is no longer used**.

Parameterized Constructor

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

```
class BoxDemo7 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

        // 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);
    }
}
```

The this Keyword



The this Keyword

```
// A redundant use of this.  
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

The this Keyword and Instance

```
// Use this to resolve name-space collisions.  
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

The 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**.
- That is, this is always **a reference to the object on which the method was invoked**.
- You can use **this any where a reference to an object of the current class' type is permitted**.

Instance Variable Hiding

- Java to declare **two local variables with the same name inside the same** or enclosing scopes.
- Interestingly, **you can have local variables, including formal parameters** to methods, which overlap with the names of the class' instance variables.
- Because this lets you refer directly to the object, **you can use it to resolve any namespace collisions** that might occur between instance variables and local variables.

The this Keyword and Instance Variable Hiding

- The use of this in such a **context can sometimes be confusing**, and some programmers are **careful not to use local variables and formal parameter names that hide instance variables**.
- Of course, other programmers believe the contrary—that it is a **good convention to use the same names for clarity, and use this to overcome the instance variable hiding**.
- **It is a matter of taste which approach you adopt.**

Garbage Collection

- It is **automatic deallocation**.
- 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**.
- There is **no explicit need to destroy objects as in C++**.
- only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.
- **different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.**

The finalize() Method

- Sometimes an object will **need to perform some action when it is destroyed.**
- For example, **if an object is holding some non-Java resource** such as a file handle or character font, then you might want to **make sure these resources are freed before an object is destroyed.**
- To handle such situations, Java provides a mechanism called finalization
- you can define **specific actions that will occur when an object is just about to be reclaimed by the garbage collector.**

The finalize() Method

- To add a finalizer to a class, you simply define **the finalize() Method**.
- The Java **run time calls that method whenever it is about to recycle an object of that class**.
- Inside the finalize() method, you will specify those actions that must be performed before an object is destroyed.

```
protected void finalize( )  
{  
    // finalization code here  
}
```

The finalize() Method

- It is important to **understand that finalize() is only called just prior to garbage collection.**
- It is not called when **an object goes out-of-scope**
- For example. This means that you cannot know when—or even if—finalize() will be executed. Therefore, your program should provide **other means of releasing system resources, etc., used by the object.**
- **It must not rely on finalize() for normal program operation**

Polymorphism in Java

- The process of representing **one form in multiple forms** is known as **Polymorphism**.
- Polymorphism is derived from 2 greek words: **poly** and **morphs**.
- The word "**poly**" means **many** and "**morphs**" means **forms**. So polymorphism means many forms.

Unit 3

Java as Object Oriented Programming Language-Overview

Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements ,String Handling: String class methods

Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method,

overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length

arguments.

Polymorphism in Java



In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

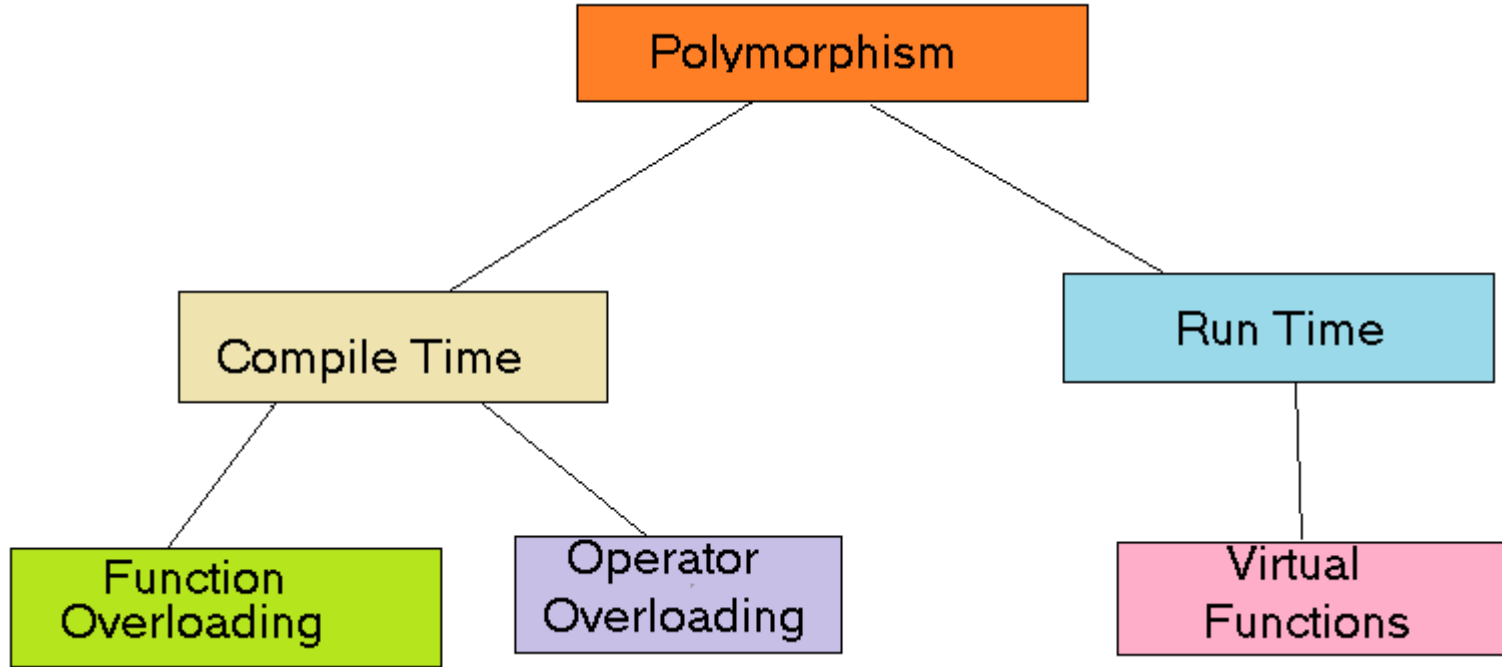
At Home behave like Son Sitesbay.com



Polymorphism



Polymorphism in Java



Types of Polymorphism in Java

```
graph TD; A[Types of Polymorphism in Java] --> B[Static Polymorphism/Compile-time Polymorphism/Early Binding]; A --> C[Dynamic Polymorphism/Runtime Polymorphism/Late Binding]; B --- D[Examples of Static Polymorphism]; D --- E[Method overloading]; D --- F[Constructor overloading]; D --- G[Method hiding]; C --- H[Example of Dynamic Polymorphism]; H --- I[Method overriding];
```

Static Polymorphism/Compile-time Polymorphism/Early Binding

Examples of Static Polymorphism

- ➔ Method overloading
- ➔ Constructor overloading
- ➔ Method hiding

Dynamic Polymorphism/Runtime Polymorphism/Late Binding

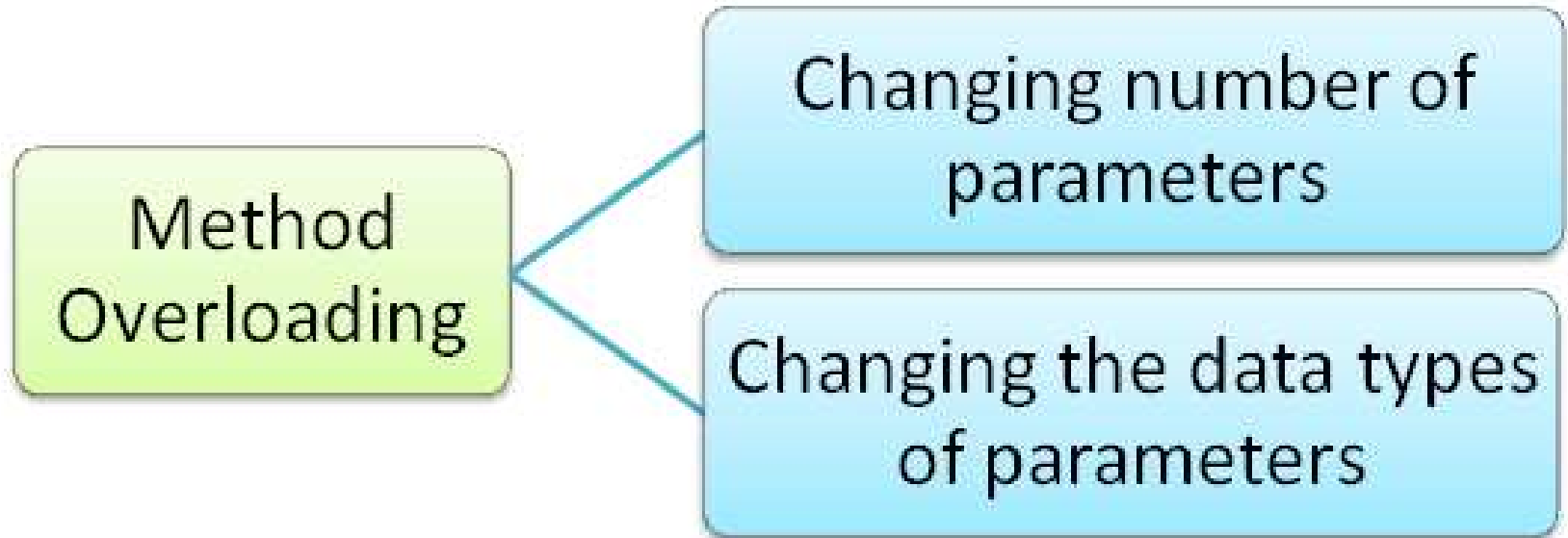
Example of Dynamic Polymorphism

- ➔ Method overriding

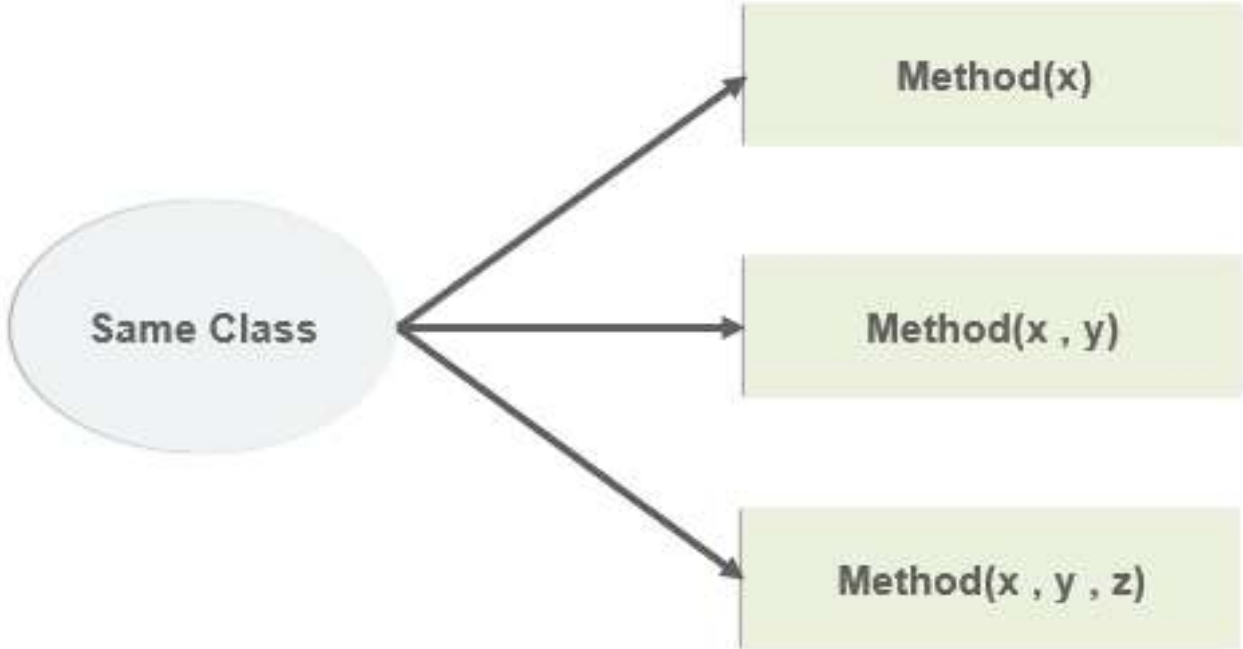
Method Overloading

- In Java it is possible to **define two or more methods within the same class that share the same name**, as long as their parameter declarations are different
- 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**
- The **return type alone is insufficient to distinguish two versions of a method.**

Method Overloading



Method Overloading



Method Overloading

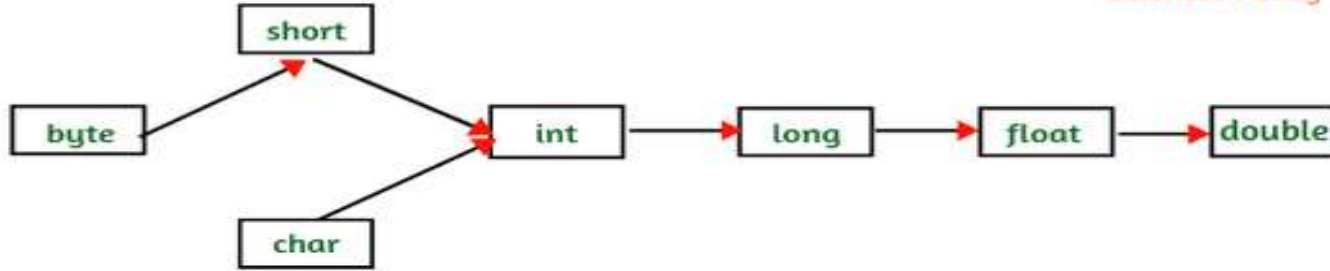


Method Overloading and automatic Type Promotion

- In some cases Java's **automatic type conversions can play a role in overload resolution**
- Java will employ its automatic type conversions **only if no exact match is found**

Method Overloading and automatic Type

Sciencetech Easy



Small size consumes less memory.

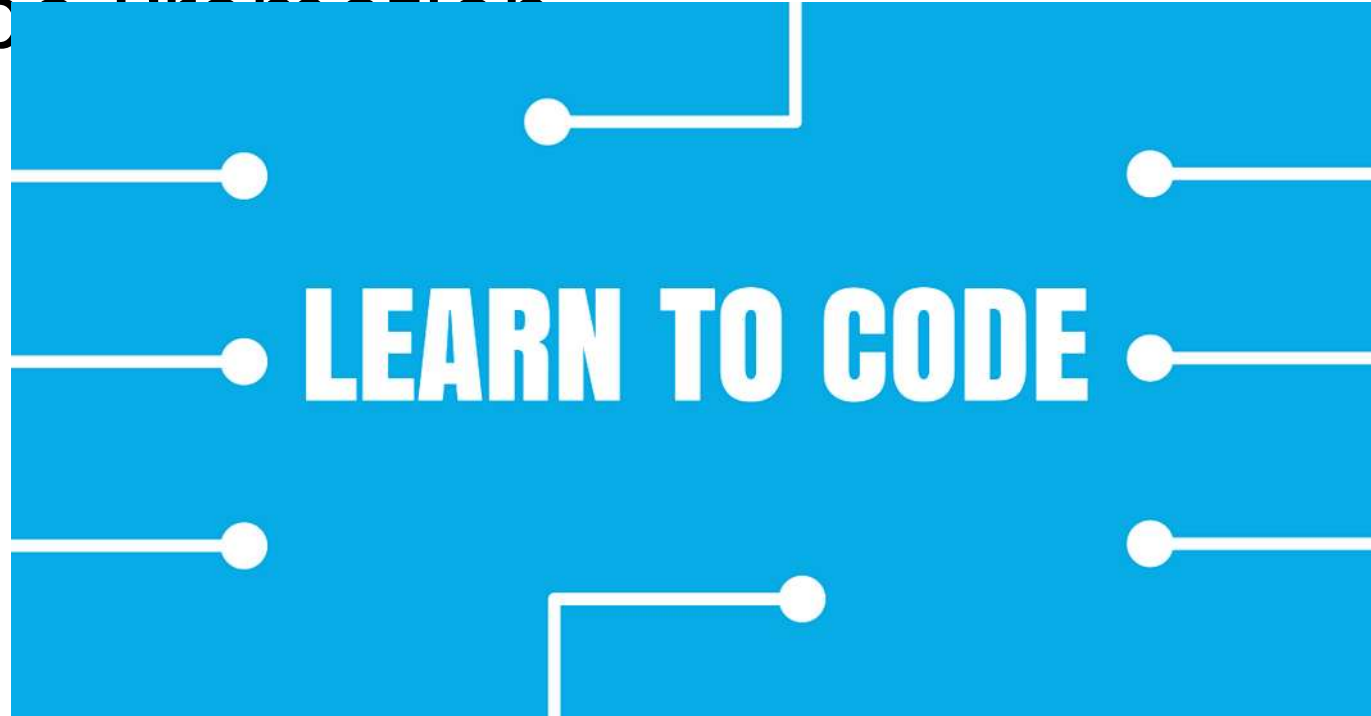
Small size primitive data type can be promoted to large size primitive data type.

Large size consumes more memory.

Large size primitive data type cannot be promoted to small size primitive data type.

Fig: All possible automatic type promotions in method overloading.

Method Overloading and automatic Type Promotion



Constructor Overloading

Beginnersbook.com

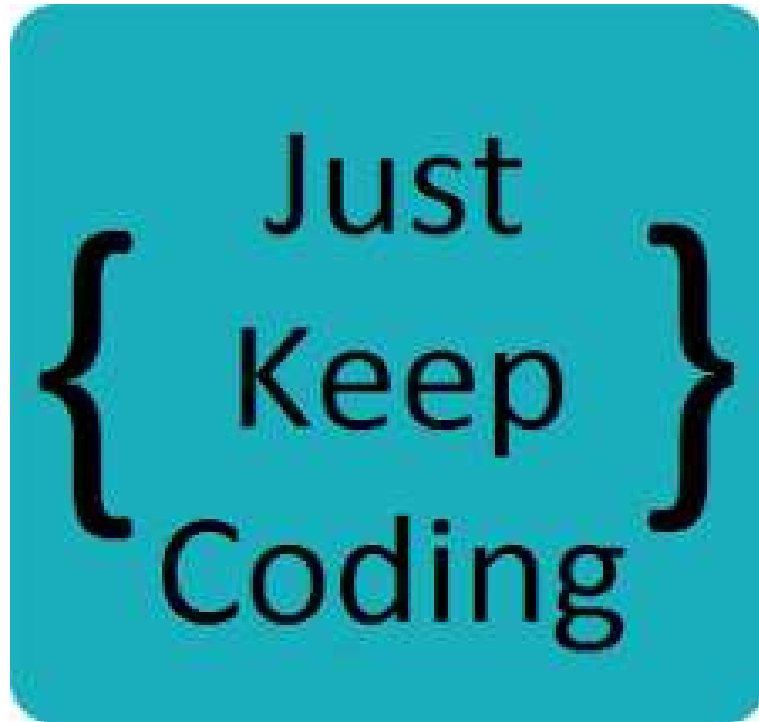
```
public class MyClass{  
    ....  
    MyClass() { ←  
        this("BeginnersBook.com");  
    }  
    MyClass(String s) { ←  
        this(s, 6);  
    }  
    MyClass(String s, int age) { ←  
        this.name =s;  
        this.age = age;  
    }  
    public static void main(String args[]) {  
        MyClass obj = new MyClass();  
        ....  
    }  
}
```

The diagram illustrates constructor overloading. It shows three constructors for the `MyClass` class: a no-argument constructor, a constructor with a `String` parameter, and a constructor with a `String` and an `int` parameter. In the `main` method, the line `MyClass obj = new MyClass();` is annotated with a long arrow pointing to the no-argument constructor. Additionally, there are two shorter arrows: one from `this("BeginnersBook.com");` in the no-argument constructor to the `MyClass(String s)` constructor, and another from `this(s, 6);` in the `MyClass(String s)` constructor to the `MyClass(String s, int age)` constructor.

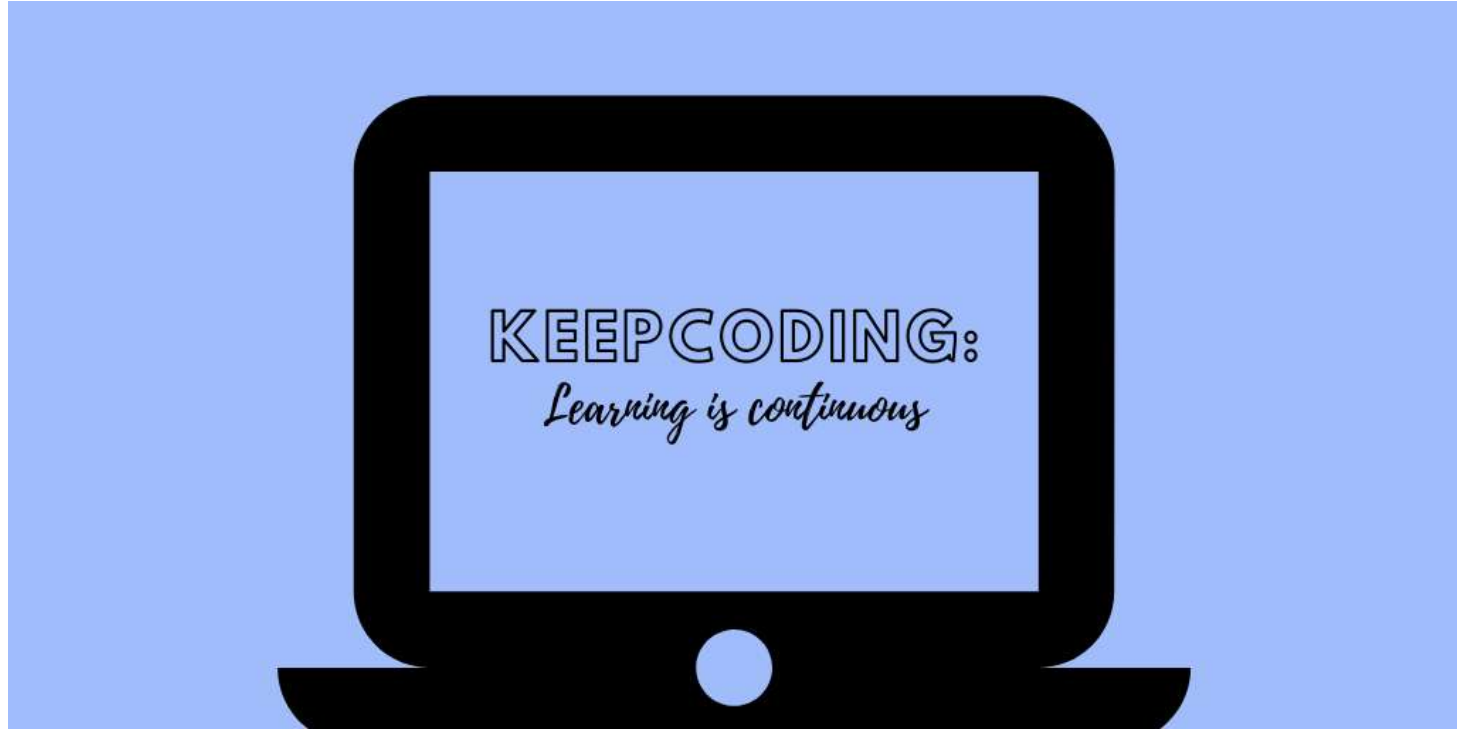
Constructor Overloading

- Java Constructor overloading is a technique in which a class can have **any number of constructors that differ in parameter list.**
- The compiler differentiates these constructors by taking into account **the number of parameters in the list and their type.**

Constructor Overloading



Objects as Parameters



Copy constructor

Copy Constructor in Java

```
constructor 1{}
```

```
Constructor2(constructor 1)
```

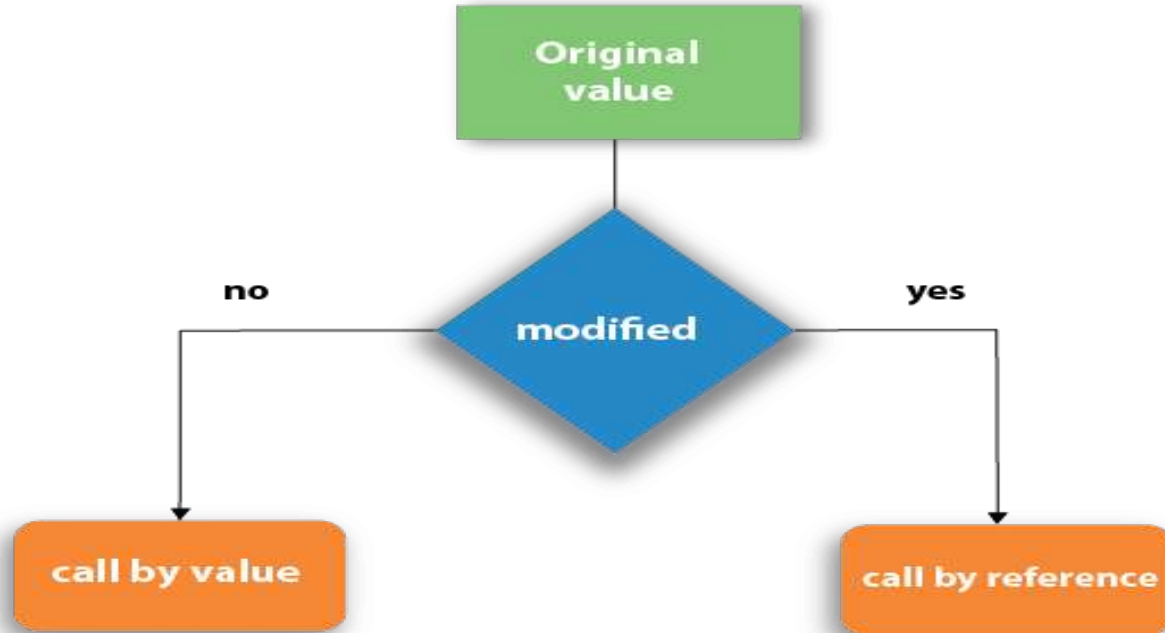
```
{}
```

A closer look at argument Passing

There are two ways that a computer language can pass an argument to a subroutine

- Call by value
- Call by reference

A closer look at argument Passing

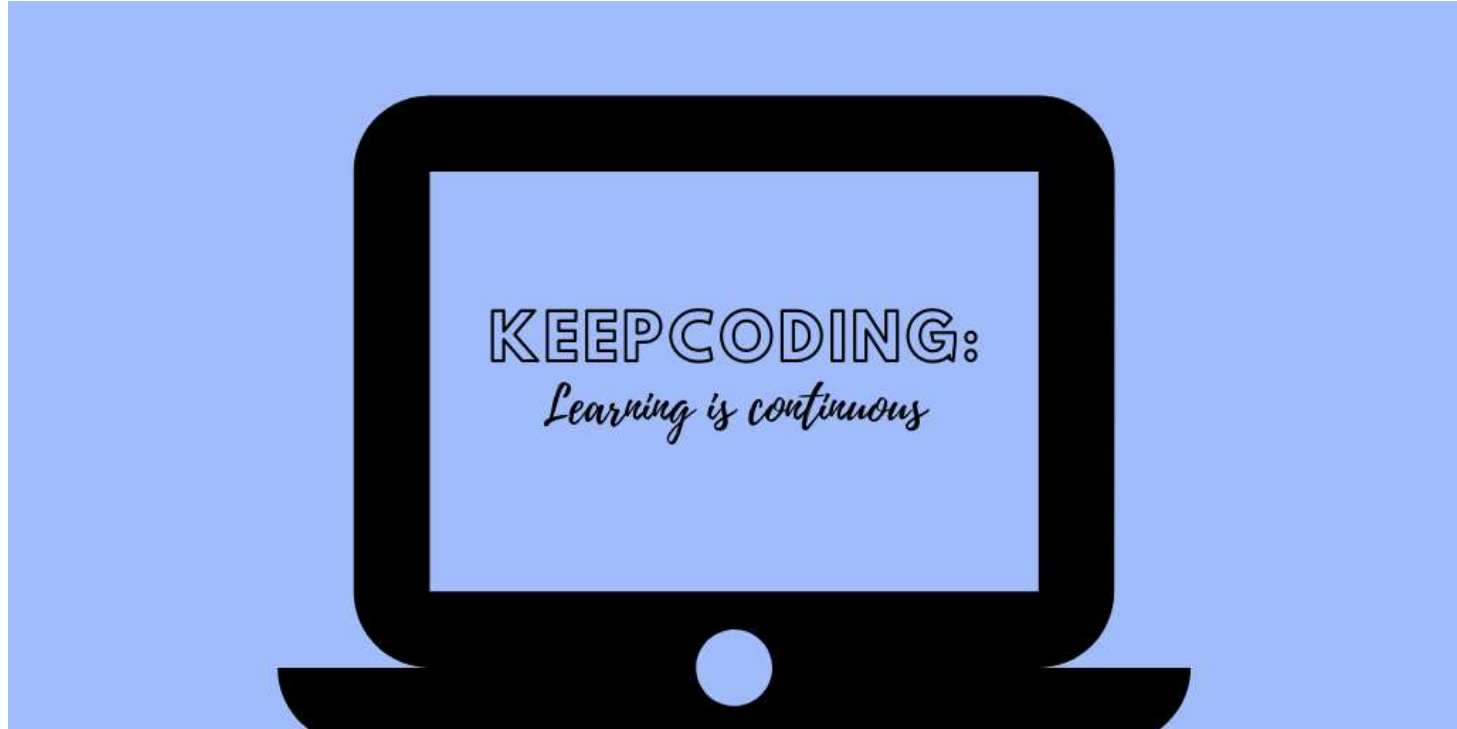


A closer look at argument Passing

Call by Value : When a primitive type is passed to a method

Call by Reference : objects are implicitly passed to a method

Returning Objects



Unit 3

Java as Object Oriented Programming Language-Overview

Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements ,String Handling: String class methods

Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method,

overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable -length

arguments.

Recursion

- Java supports **recursion**
- Recursion is the process **of defining something in terms of itself**
- As it relates to Java programming, **recursion is the attribute that allows a method to call itself**
- A method that calls itself is **said to be recursive**

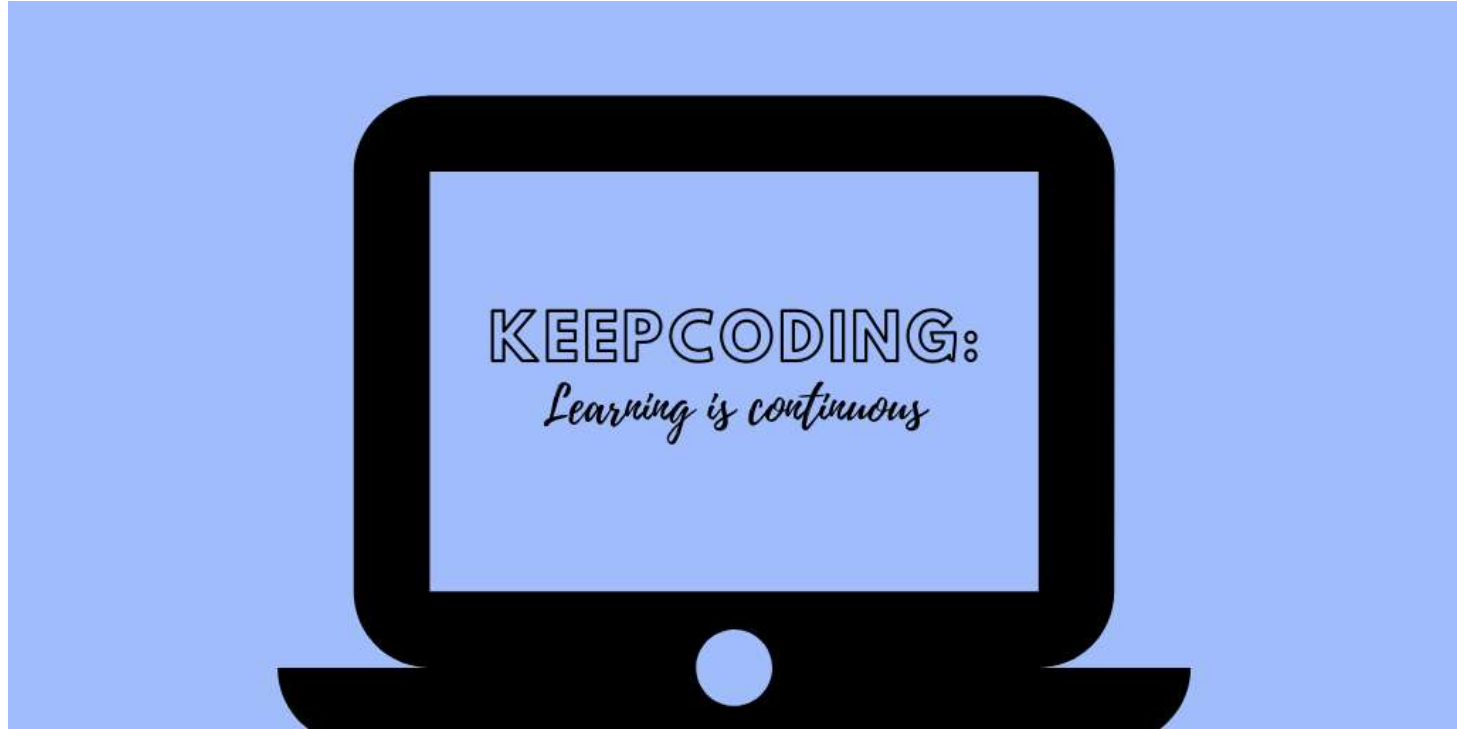
Recursion

- Recursive versions of many **routines may** execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls
- Because storage for parameters and local variables is on the stack and each **new call creates a new copy of these variables, it is possible that the stack could be exhausted**
- If this occurs, **the Java run-time system will cause an exception**

Recursion

- The main advantage to recursive methods is that they can be used to create **clearer and simpler versions of several algorithms than can their iterative relatives**

Recursion



Introducing Access Control

- Java's access specifiers are public, private, and protected
- **protected applies only when inheritance** is involved
- When a member of a **class is modified by the public specifier, then that member can be accessed by any other code**
- When a member of a **class is specified as private, then that member can only be accessed by other members of its class**

Introducing Access Control

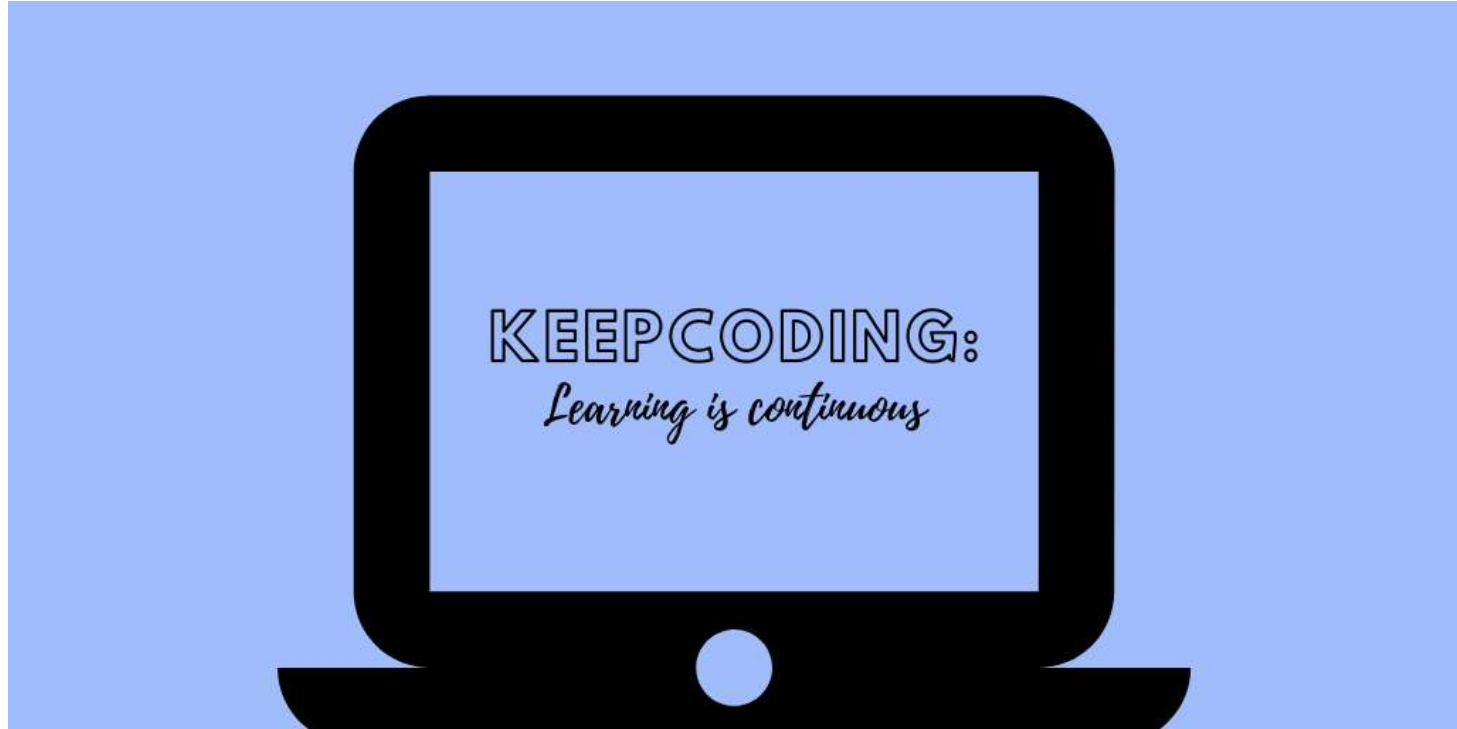
Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

Introducing Access Control

- Java's access specifiers are public, private, and protected
- **protected applies only when inheritance** is involved
- When a member of a **class is modified by the public specifier, then that member can be accessed by any other code**
- When a member of a **class is specified as private, then that member can only be accessed by other members of its class**

Introducing Access Control



Understanding 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
- The most common example of a **static member is main()**
- **main() is declared as static because it must be called before any objects exist**
- Instance variables declared as static are, **essentially, global variables**

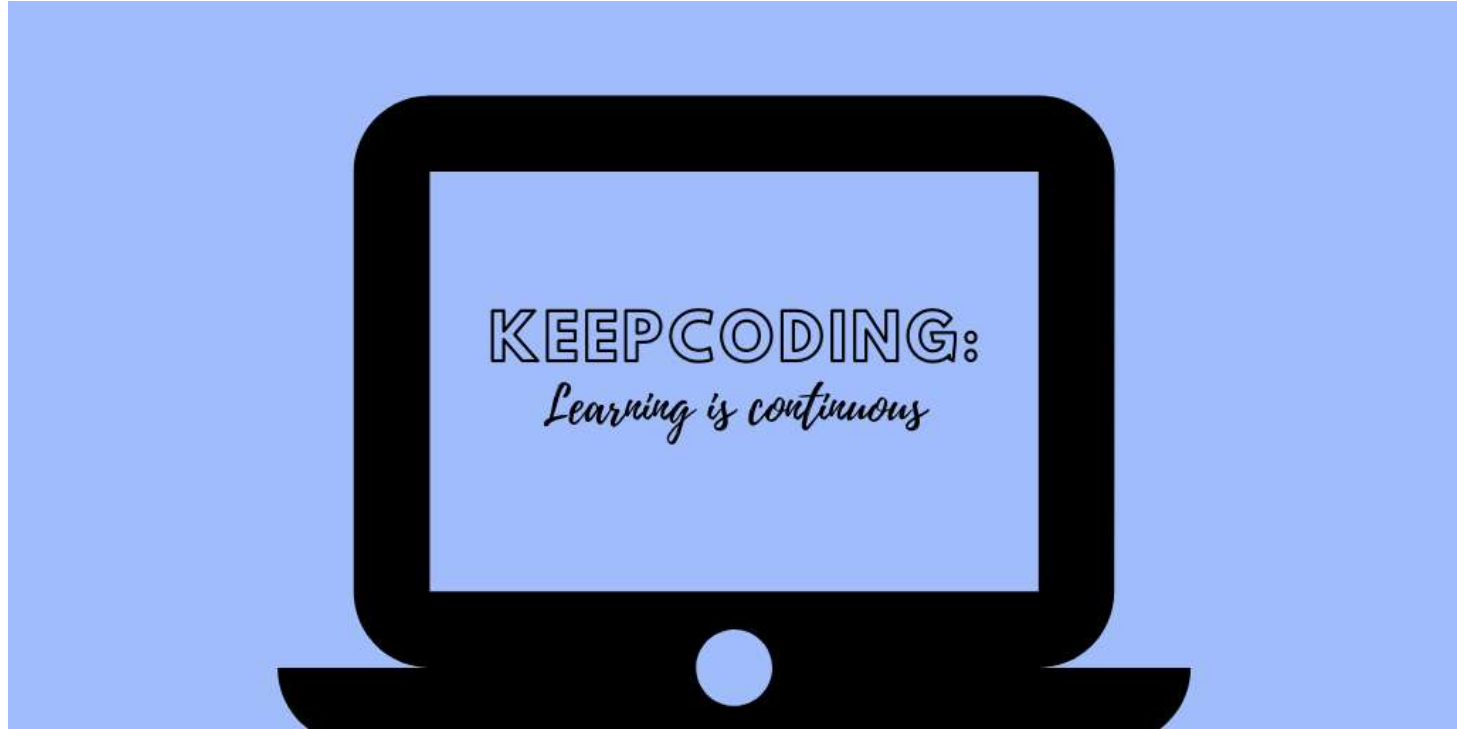
Methods declared as static have several restrictions:

- They can only **call other static methods**
- They must **only access static data**
- They cannot **refer to this or super in any way**

Methods declared as static have several restrictions:

- They can only **call other static methods**
- They must **only access static data**
- They cannot **refer to this or super in any way**
- **We can declare a static block which gets executed exactly once, when the class is first loaded**

Understanding static



Introducing final

- A variable can be declared as final
- Doing so prevents its **contents from being modified**
- We **must initialize a final variable when it is declared**
- `final int FILE_NEW = 1;`
- `final int FILE_OPEN = 2;`

Introducing final

- Variables declared **as final do not occupy memory on a per-instance basis**
- The keyword **final** can also **be applied to methods, but its meaning is substantially different** than when it is applied to variables

Introducing Nested and Inner Classes

- It is possible to define **a class within another class**
- The **scope of a nested class is bounded by the scope of its enclosing class**
- If **class B is defined within class A, then B is known to A, but not outside of A**
- A nested class **has access to the members, including private members, of the class in which it is nested**

Introducing Nested and Inner Classes

- However, **the enclosing class does not have access to the members of the nested class**
- There are two types of **nested classes: static and non-static**
- A **static nested class** is one which has the **static modifier applied**
- **static innerclass must access its enclosing class by creating an object.**

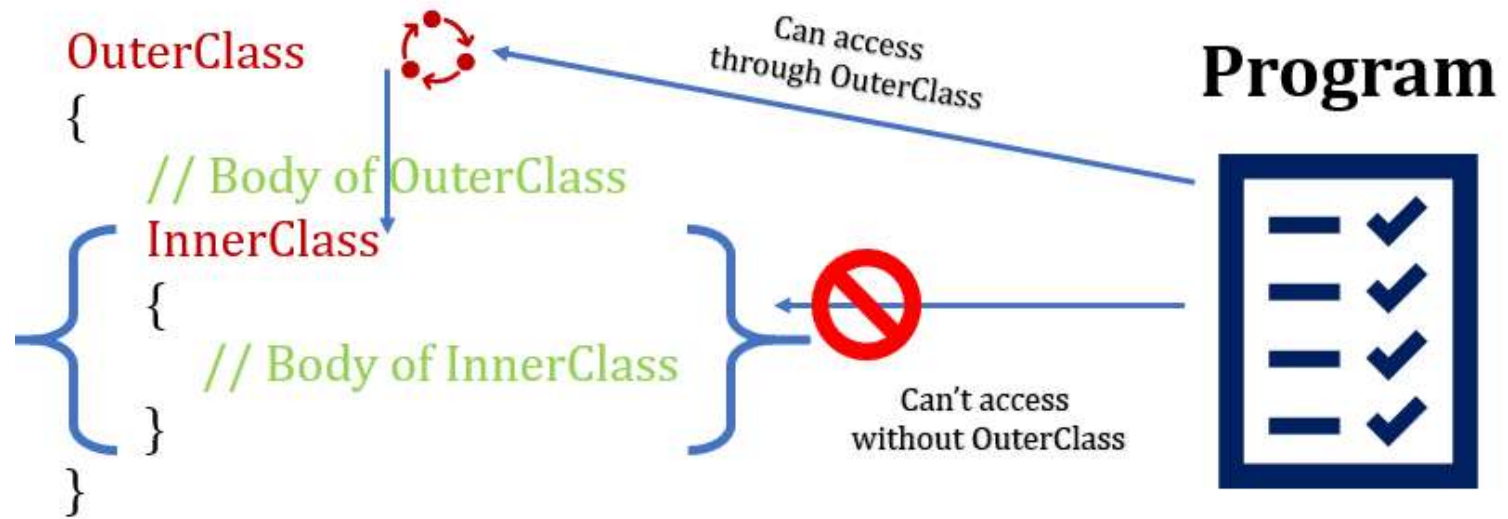
Introducing Nested and Inner Classes

- The most important type of **nested class is the inner class**
- An inner class is a **non-static nested class**
- It has access to **all of the variables and methods of its outer class**

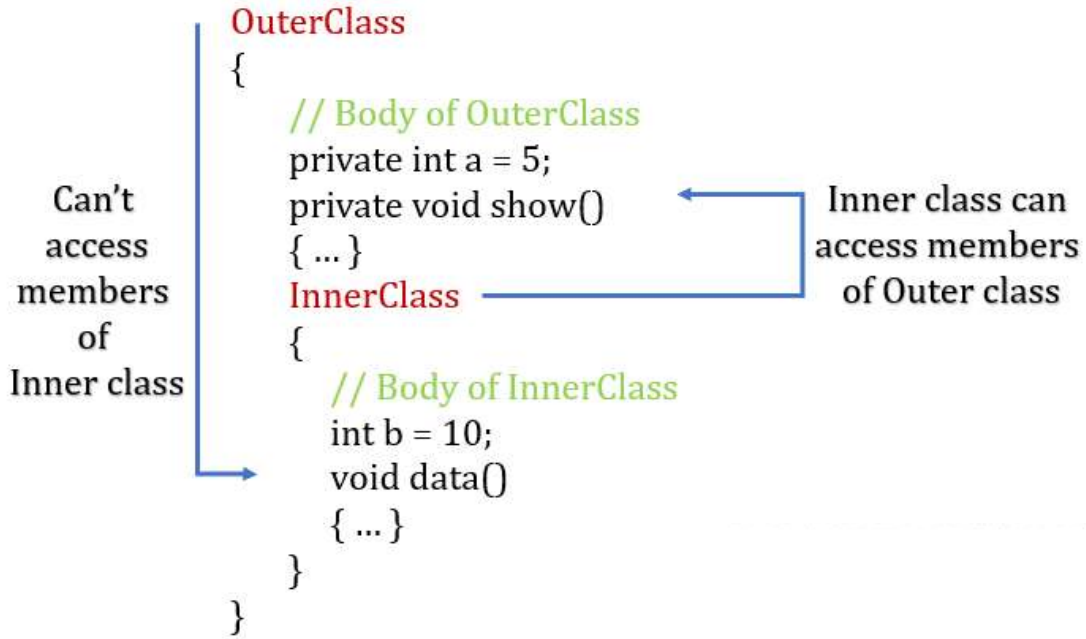
Introducing Nested and Inner Classes

- It is important to **realize that class Inner is known only within the scope of class Outer**
- The Java compiler **generates an error message if any code outside of class Outer attempts to instantiate class Inner**

Introducing Nested and Inner Classes



Introducing Nested and Inner Classes



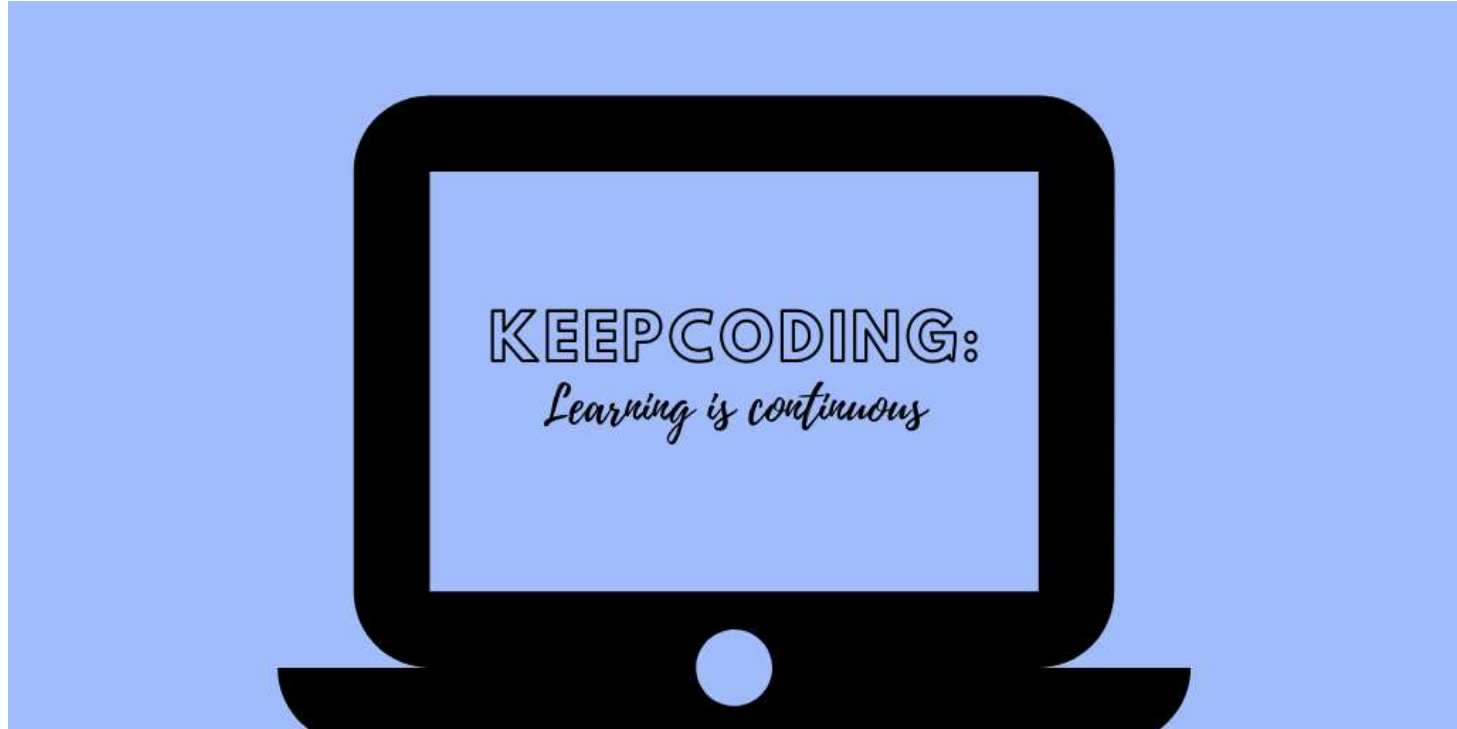
Introducing Nested and Inner Classes

- It is important to **realize that class Inner is known only within the scope of class Outer**
- The Java compiler **generates an error message if any code outside of class Outer attempts to instantiate class Inner**
- While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet

Introducing Nested and Inner Classes

- It is important to **realize that class Inner is known only within the scope of class Outer**
- The Java compiler **generates an error message if any code outside of class Outer attempts to instantiate class Inner**
- While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet

Introducing Nested and Inner Classes



Example of Nested Class

```
class TestMemberOuter1{  
  
    private int data=30;  
  
    class Inner{  
  
        void msg(){System.out.println("data is  
"+data);} /)msg() complete  
  
    } // Inner class Complete
```

```
public static void main(String args[]){  
  
    TestMemberOuter1 obj=new TestMemberOuter1();  
  
    TestMemberOuter1.Inner in=obj.new Inner();  
  
    in.msg();  
  
    }  
  
}
```

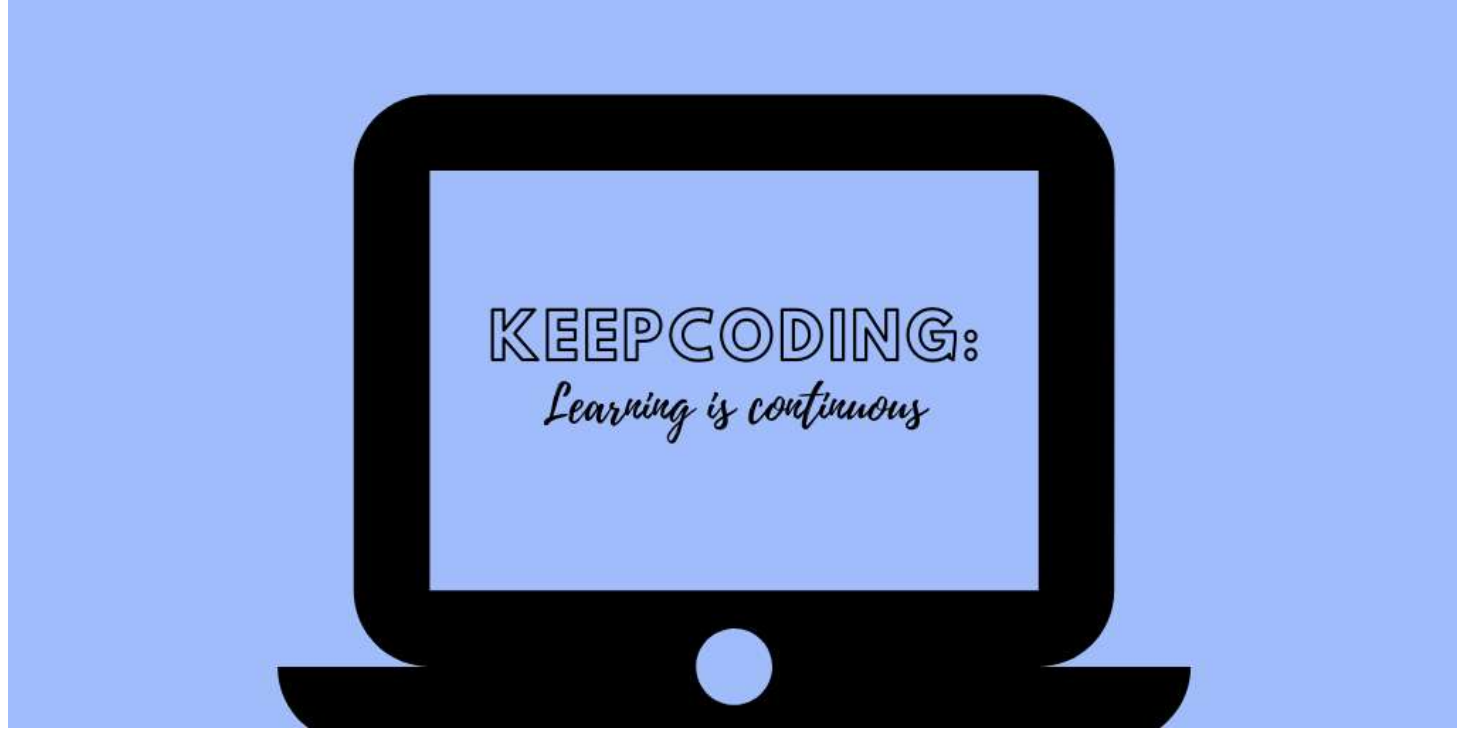
Command Line Argument

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

Command Line Argument

- When command-line arguments are supplied to JVM, JVM wraps these and supplies them to `args[]`. It can be confirmed that they are actually wrapped up in an `args` array by checking the length of `args` using `args.length`.
- Internally, JVM wraps up these command-line arguments into the `args[]` array that we pass into the `main()` function. We can check these arguments using `args.length` method. JVM stores the first

Using Command-Line Arguments



Program for Command Line argument

Steps to Run COmmand Line Argument

Save the program as Hello.java

Open the command prompt window and compile the program- `javac Hello.java`

After a successful compilation of the program, run the following command by writing the arguments- `java Hello`

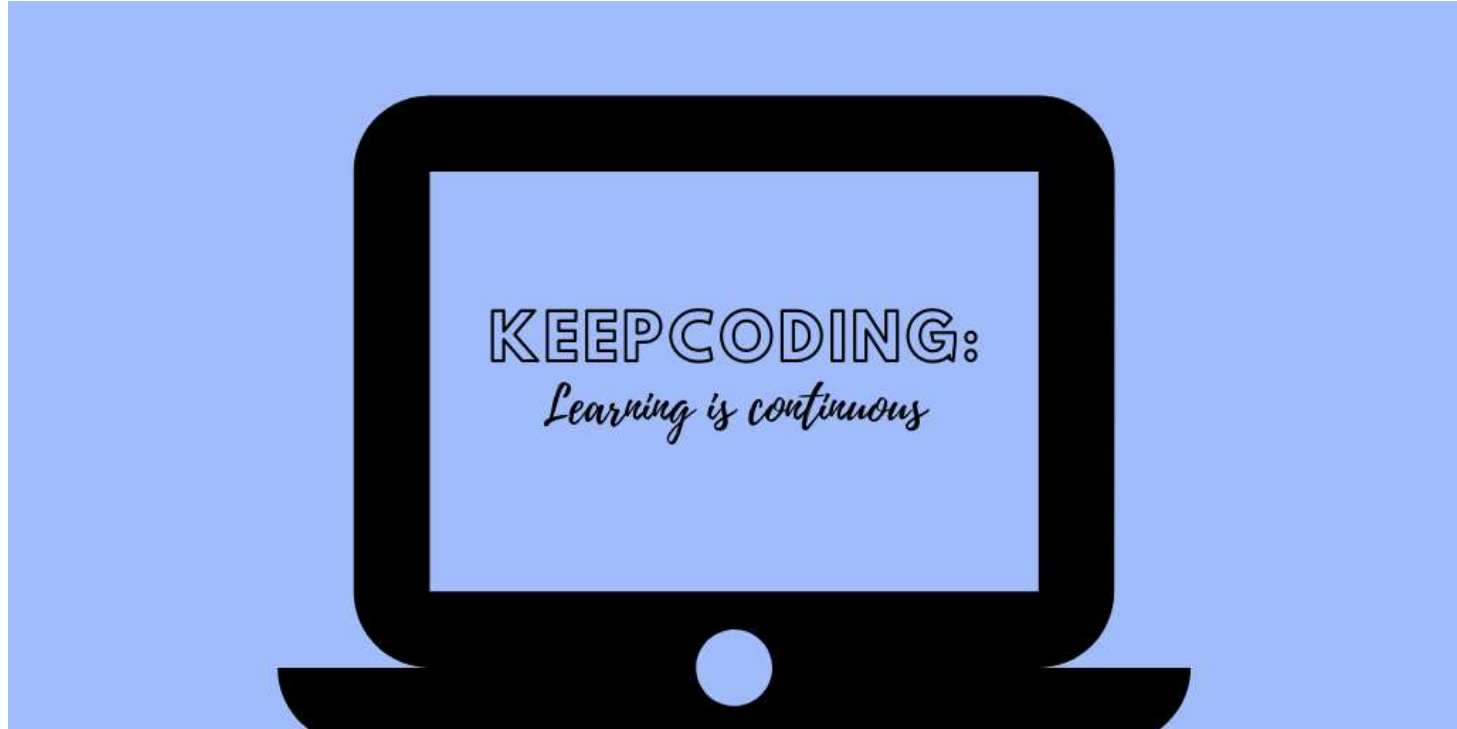
For example – `java Hello Geeks at GeeksforGeeks`

Press Enter and you will get the desired output.

Command Line Argument

```
gfg@gfg-Lenovo-G50-80:~$ javac a.java
gfg@gfg-Lenovo-G50-80:~$ java Hello
No command line arguments found.
gfg@gfg-Lenovo-G50-80:~$ java Hello Geeks at GeeksforGeeks
The command line arguments are:
Geeks
at
GeeksforGeeks
gfg@gfg-Lenovo-G50-80:~$
```

Var args-Variable length Arguments



Variable Arguments

A method with variable length arguments(Varargs) in Java can have zero or multiple arguments.

Variable length arguments are most useful when the number of arguments to be passed to the method is not known beforehand.

They also reduce the code as overloaded methods are not required.

Variable Arguments

```
public class Demo {  
    public static void Varargs(String... str) {  
        System.out.println("\nNumber of arguments are: " +  
str.length);  
        System.out.println("The argument values are: ");  
        for (String s : str)  
            System.out.println(s);  
    }  
}
```

Variable Arguments

```
public static void main(String  
args[]) {  
    Varargs("Apple", "Mango",  
"Pear");  
    Varargs();  
    Varargs("Magic");  
}  
}
```

Output

Number of arguments are: 3

The argument values are:

Apple

Mango

Pear

Number of arguments are: 0

The argument values are:

Number of arguments are: 1

The argument values are:

Magic