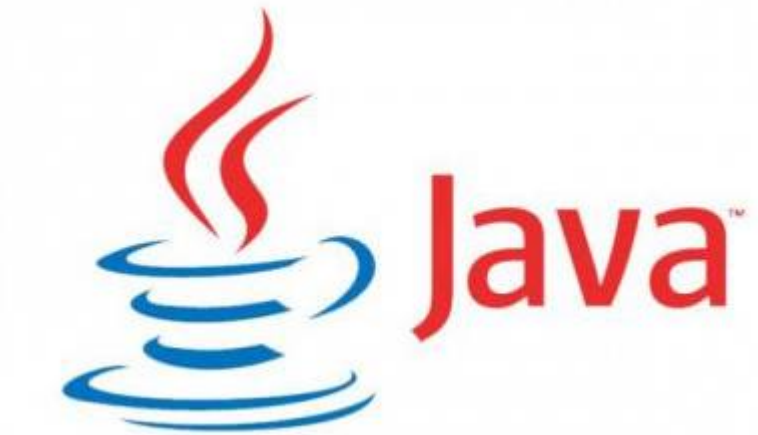
Java Programming

SUBJECT CODE: 18CS52

NUMBER OF CREDITS : 4 = 4:0:0(L-T-P)



Syllabus – Theory

Syllabus – Laboratory

Faculty in-charge

Dr. Smitha Shekar B
Associate Professor
Dept.of CSE, Dr.A.I.T

UNIT-1 CONTENTS

JAVA

OBJECT AND CLASSES

PACKAGE AND INTERFACE

APPLET FUNDAMENTALS

Dr. Smitha Shekar B

UNIT 1- TOPICS THAT WILL BE COVERED

Introduction to Java

History and evolution of Java

An overview of Java

Introduction to classes

Methods
- Overloading methods
- Overriding methods

Package and Interface

Dr. Smitha Shekar B

UNIT 1- TOPICS THAT WILL BE COVERED

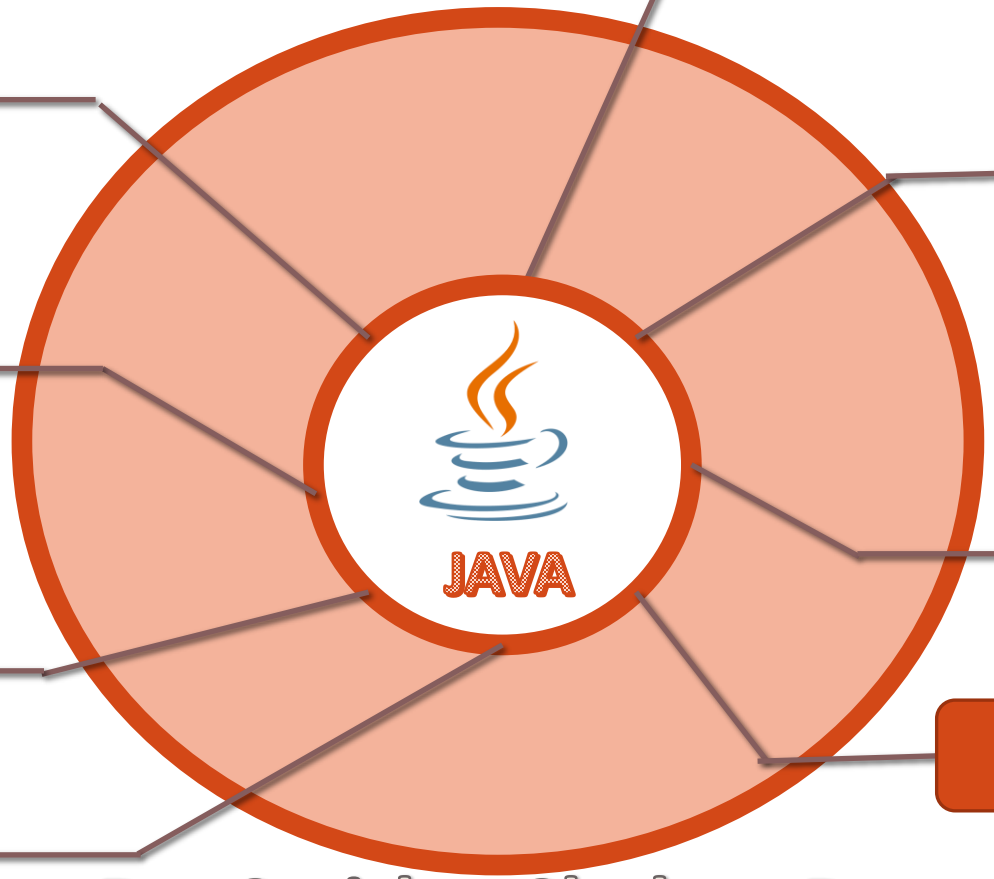
The Applet Class

- Two types of Applets
- Applet basics
- Applet architecture
- An Applet Skeleton
- Simple Applet display methods
- Requesting Repainting
- Using the status window
- The HTML applet tag
- Parsing Parameters to Applets
- getDocumentbase() and getCodebase()
- AppletContext and showApplet()
- The Audio Clip Interface
- The Applet stub interface
- Output to the console

Dr. Smitha Shekar B

Introduction to Java

- High Level Programming Language
- James Gosling is considered father of Java
- Released by SUN microsystems in 1996 for the public
- It was first called as OAK
- Write once run anywhere (WORA)
- Free and Open Software
- In 2010, SUN microsystems was sold off to oracle, along with it Java and other softwares
- In 1995, it was renamed to “Java”



Dr. Smitha Shekar B

History and evolution of Java

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991.

The small team of sun engineers called Green Team.

Initially designed for small, embedded systems in electronic appliances like set-top boxes.

Firstly, it was called "Greentalk" by James Gosling, and the file extension was .gt.

In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

After that, it was called Oak and was developed as a part of the Green project.

Java is an island of Indonesia where the first coffee was produced (called java coffee).

JDK 1.0 released in(January 23, 1996). After the first release of Java, there have been many additional features added to the language.

Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds the new features in Java.

Java Version History

- JDK Alpha and Beta (1995)
- JDK 1.0 (23rd Jan 1996)
- JDK 1.1 (19th Feb 1997)
- J2SE 1.2 (8th Dec 1998)
- J2SE 1.3 (8th May 2000)
- J2SE 1.4 (6th Feb 2002)
- J2SE 5.0 (30th Sep 2004)
- Java SE 6 (11th Dec 2006)
- Java SE 7 (28th July 2011)
- Java SE 8 (18th Mar 2014)
- Java SE 9 (21st Sep 2017)
- Java SE 10 (20th Mar 2018)

Dr. Smitha Shekar B

An overview of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language.

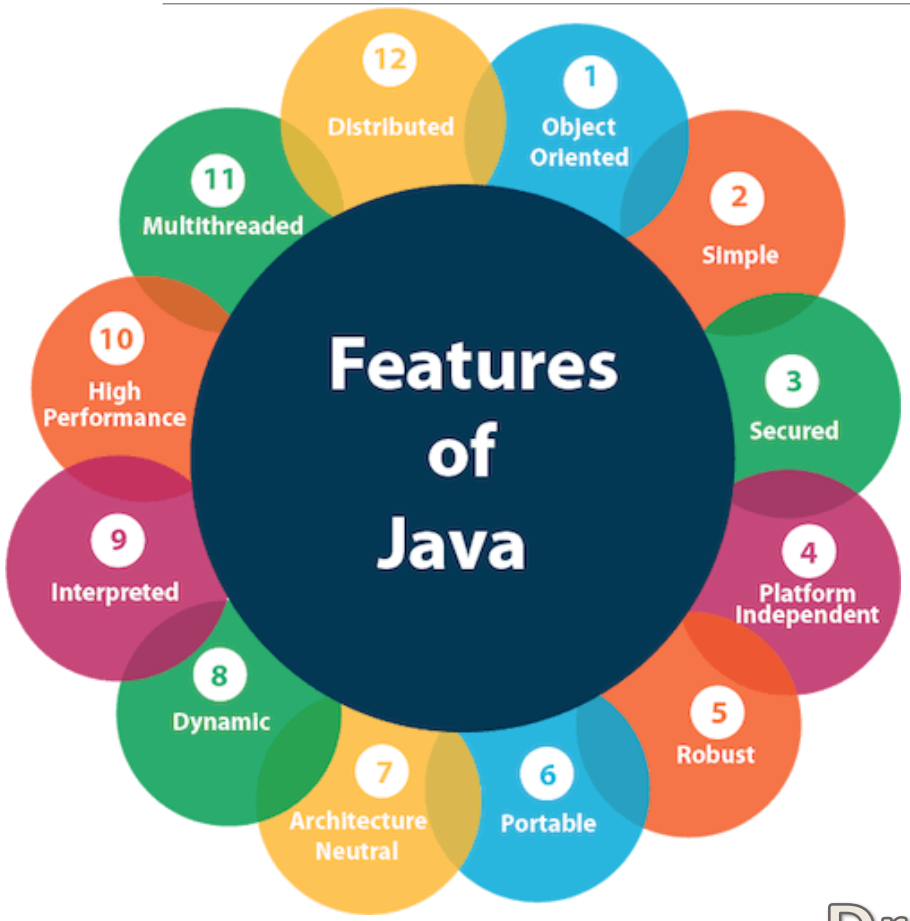
Apart from this, there are also some excellent features which play an important role in the popularity of this language.

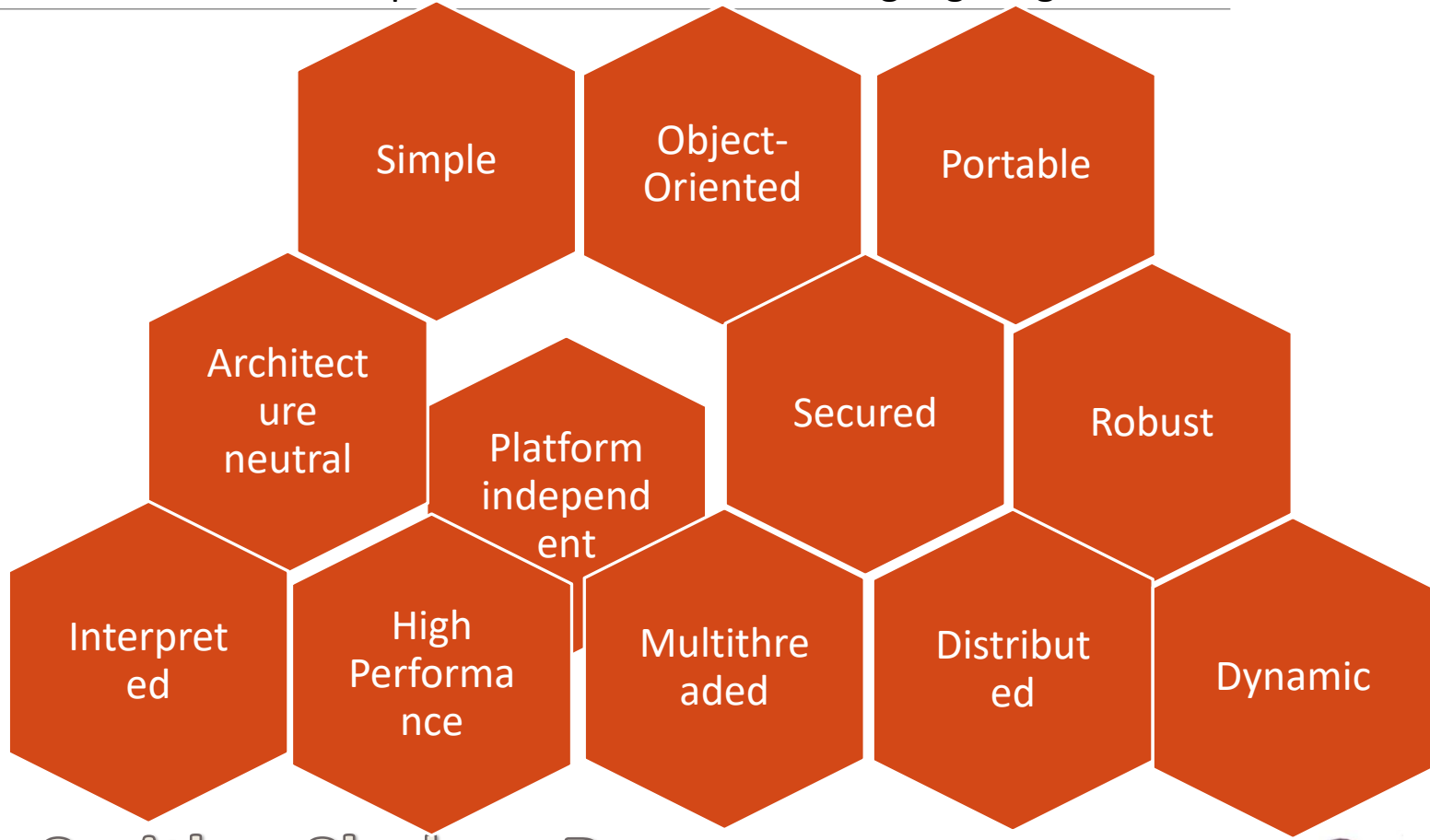
The features of Java are also known as java *buzzwords*.

Dr. Smitha Shekar B

Features of Java

A list of most important features of Java language is given below.





Dr. Smitha Shekar B

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Dr. Smitha Shekar B

Object-oriented

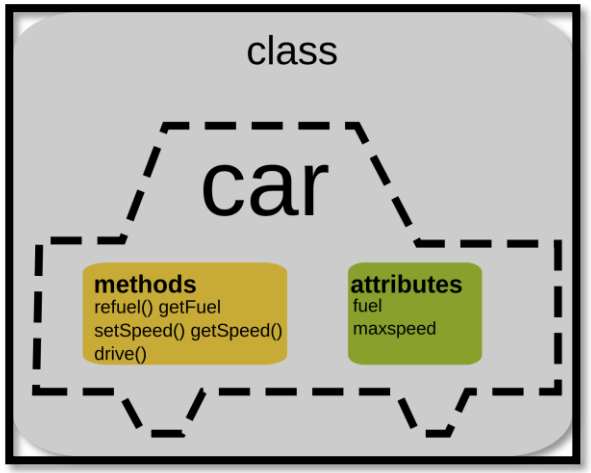
Java is an object-oriented programming language. Everything in Java is an object.

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

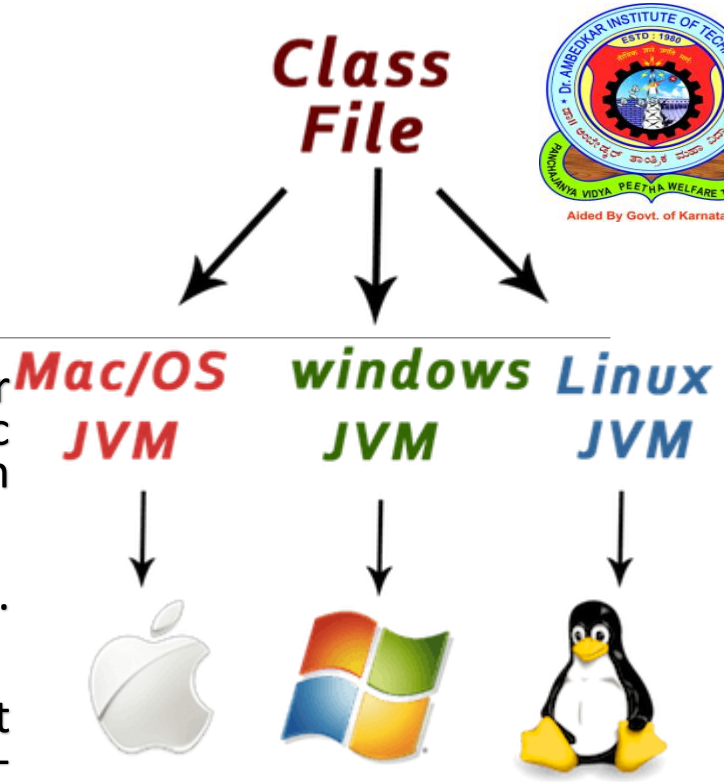
Basic concepts of OOPs are:

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



Dr. Smitha Shekar B

Platform Independent



Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

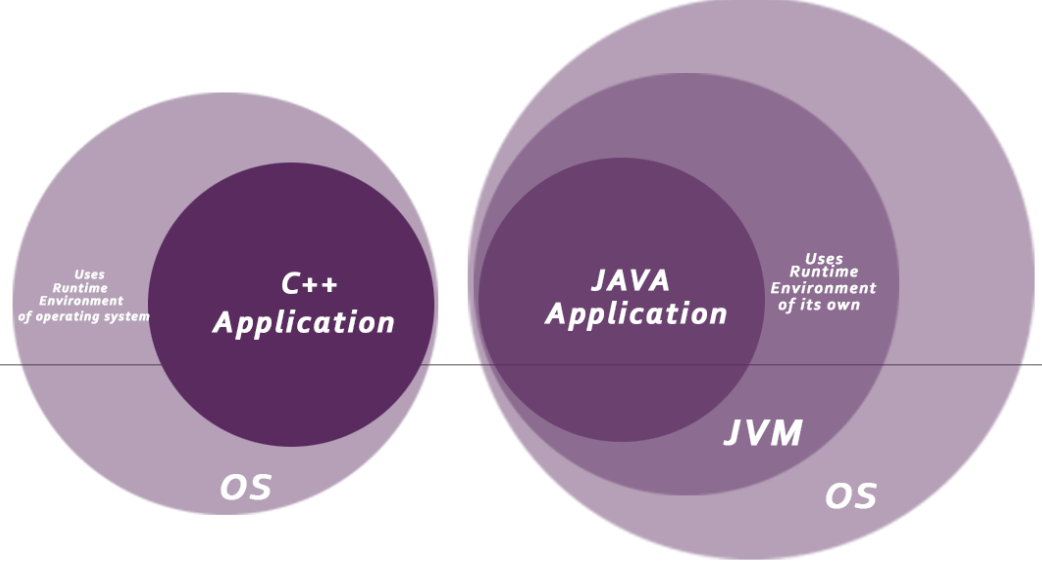
The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

- Runtime Environment
- API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

Dr. Smitha Shekar B

Secured



Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**
- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Dr. Smitha Shekar B

Robust

Robust simply means **strong**. Java is robust because:

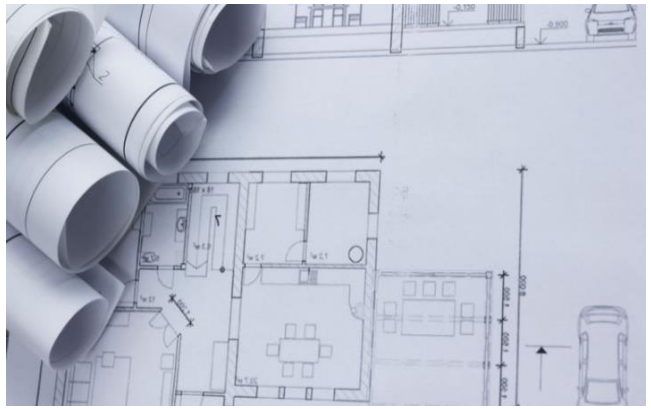
- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Dr. Smitha Shekar B

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, *int* data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.



Dr. Smitha Shekar B

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

Dr. Smitha Shekar B

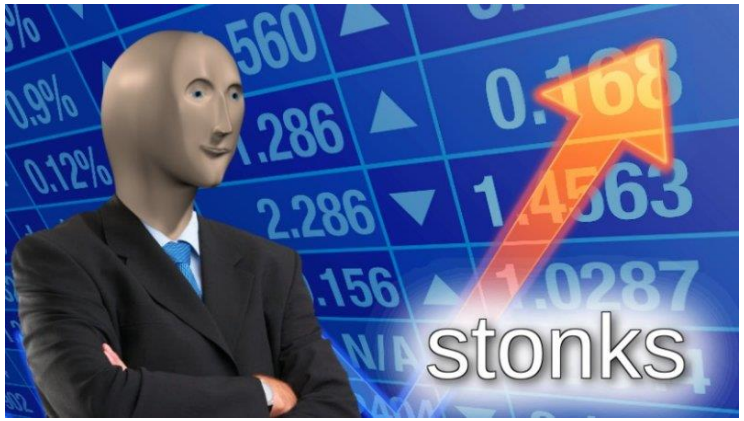
High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.

It is still a little bit slower than a compiled language (e.g., C++).

Java is an interpreted language that is why it is slower than compiled languages,

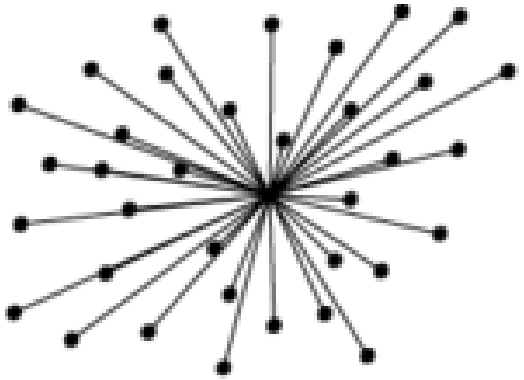
e.g., C, C++, etc.



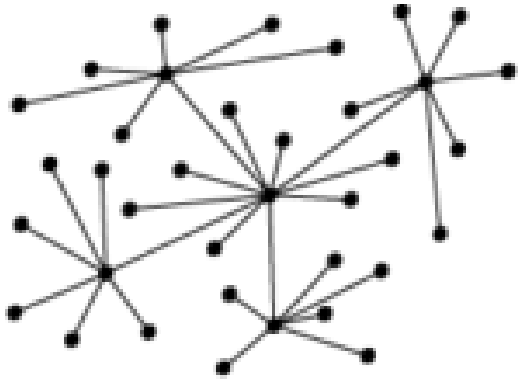
Dr. Smitha Shekar B

Distributed

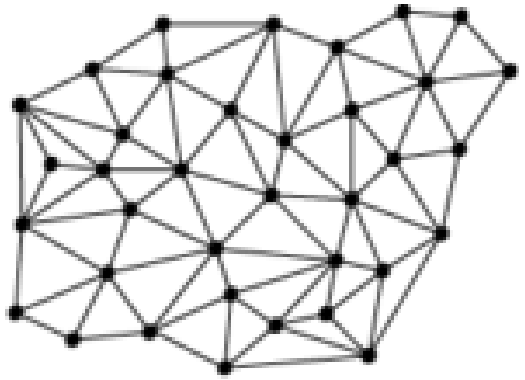
Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.



centralised



decentralised



distributed

Dr. Smitha Shekar B

Multi-threaded

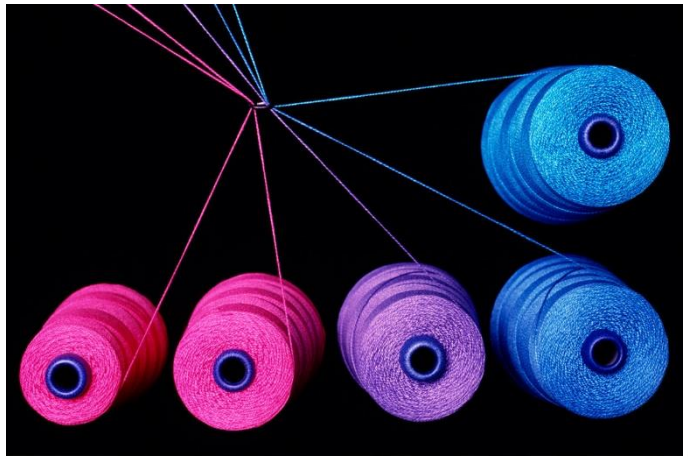
A thread is like a separate program, executing concurrently.

We can write Java programs that deal with many tasks at once by defining multiple threads.

The main advantage of multi-threading is that it doesn't occupy memory for each thread.

It shares a common memory area.

Threads are important for multi-media, Web applications, etc.



Dr. Smitha Shekar B

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

Dr. Smitha Shekar B

C++ vs Java

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent.

Dr. Smitha Shekar B

C++ vs Java

Comparison Index	C++	Java
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comment.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in C language, single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

Dr. Smitha Shekar B

C++ vs Java

C++ Example

File: main.cpp

```
#include <iostream>
using namespace std;
int main() {
   cout << "Hello C++ Programming";
   return 0;
}
```

Java Example

File: Simple.java

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


Java Hello World

The requirement for Java Hello World Example

- For executing any java program, you need to
- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory.
- Create the java program
- Compile and run the java program

Dr. Smitha Shekar B

Java Hello World

Let's create the hello java program:

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

Save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Output: Hello Java

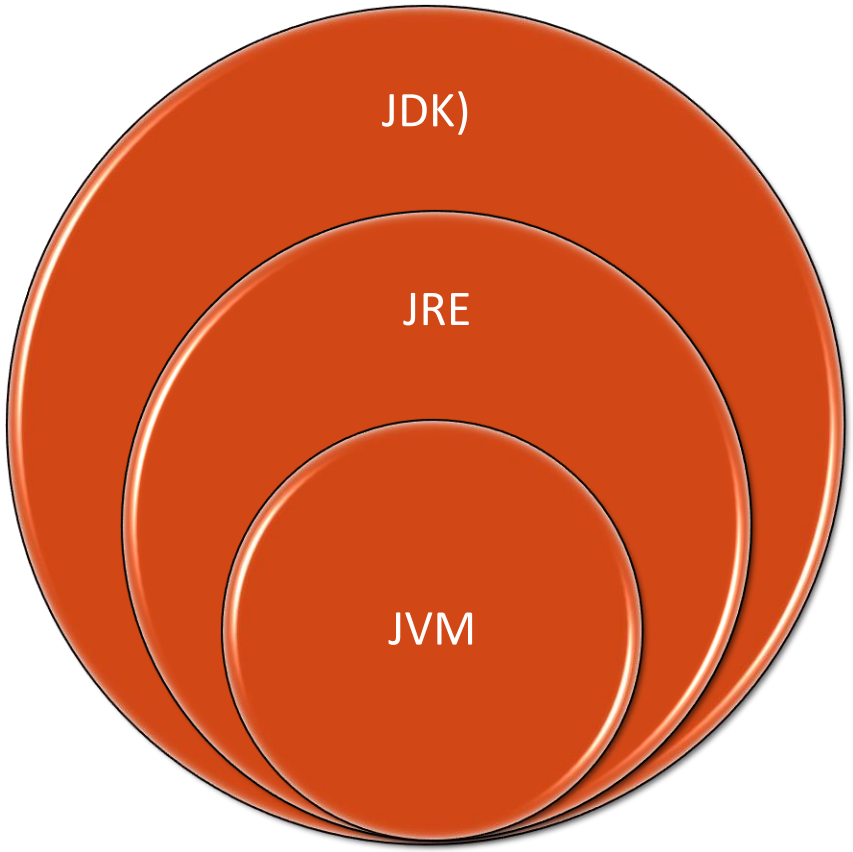
Dr. Smitha Shekar B

Difference between JDK, JRE, and JVM

We must understand the differences between JDK, JRE, and JVM before proceeding further to Java. See the brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page.

Firstly, let's see the differences between the JDK, JRE, and JVM.



Dr. Smitha Shekar B

Difference between JDK, JRE, and JVM

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

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

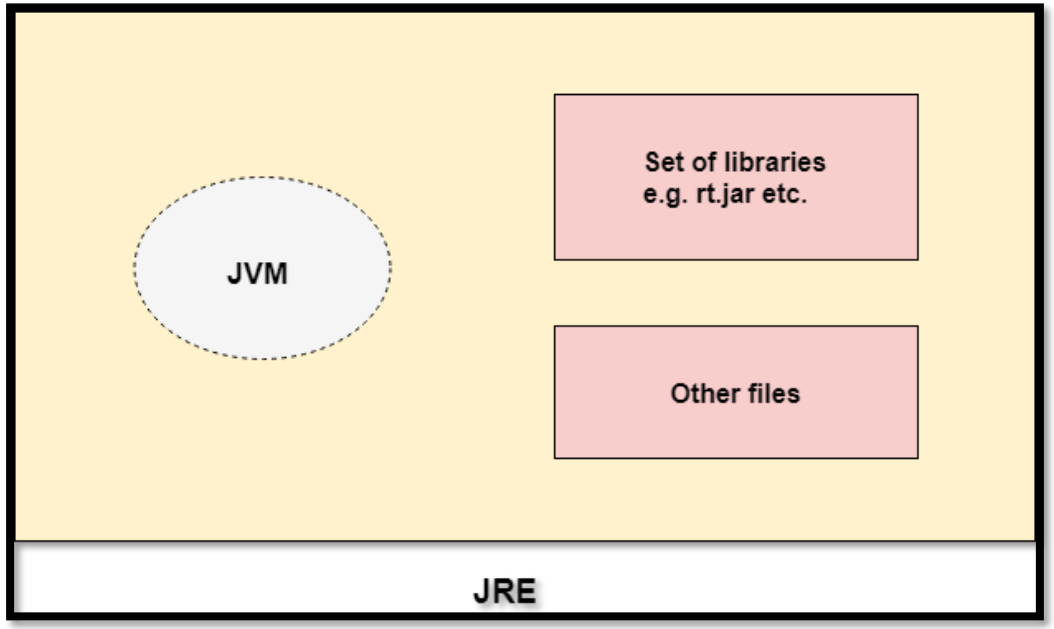
Dr. Smitha Shekar B

Difference between JDK, JRE, and JVM

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



Dr. Smitha Shekar B

Difference between JDK, JRE, and JVM

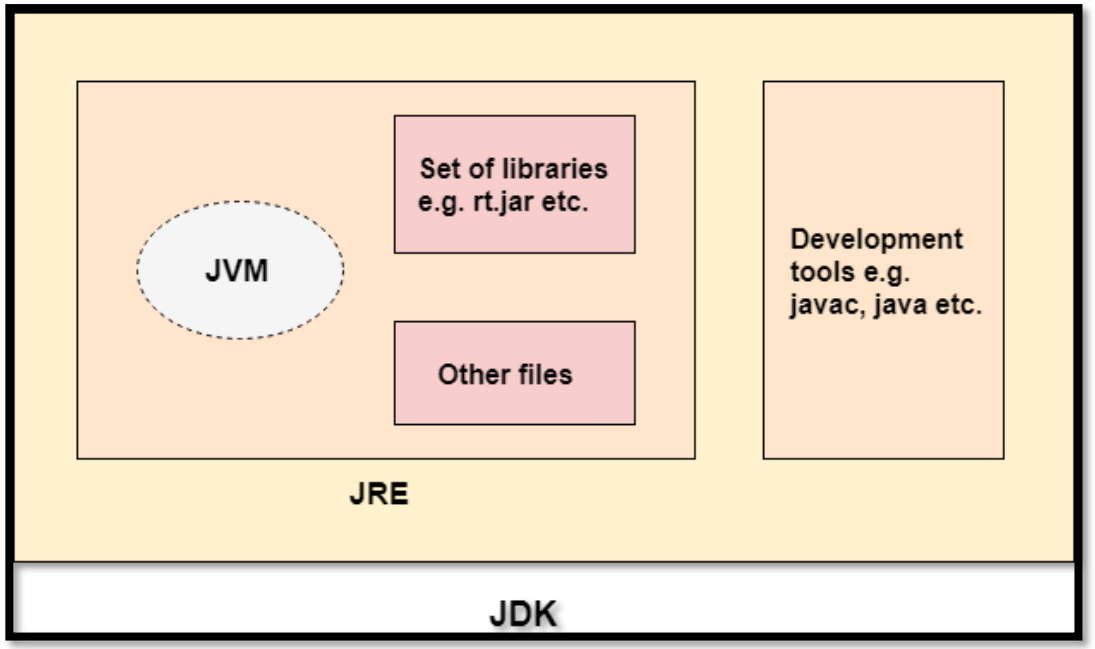
JDK

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

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.



Dr. Smitha Shekar B

Introduction to Arrays

What is an Array?

Features of Arrays

Advantages

Disadvantages

Types of Arrays

- Single(1-D Array)
- Multi(2-D/3-D Array)

Array Declaration, Creation and Initialisation

Dr. Smitha Shekar B

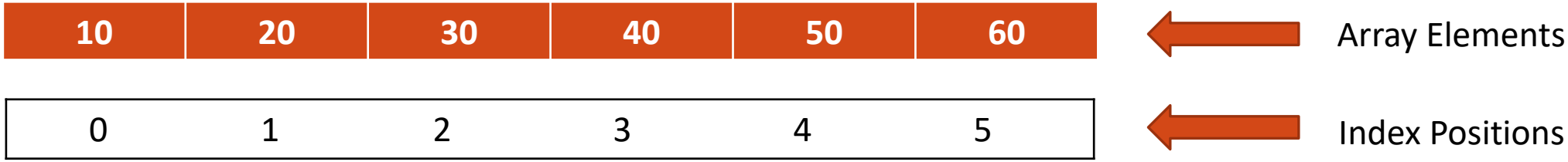
ARRAYS

- An array is a group of contiguous or related data items that share a common name.
- Example: salary[10] , to represent a set of salaries of a group of employees.
- A particular value is indicated by writing a number called index number or subscript in brackets after the array name.
- The complete set of values is referred to as an Array, the individual values are called elements.
- Arrays can be of any variable type.
- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Dr. Smitha Shekar B

What is an Array?

- An Array is an object that holds a fixed number of values of homogeneous or similar data-type.
- Or say, An Array is a Data Structure where we store similar elements.
- The length of an Array is assigned when the array is created and after creation, its length is fixed.
- For example: int[ ] a = new int[6];
- It will create an array of length 6 and index value will start from 0.



Dr. Smitha Shekar B

Features of an Array

- A Java array variable can be declared like any other variables with [ ] after the data-type.
- The variables in the array are ordered and each have an index beginning from 0.
- In Java , Arrays are objects, and thus they occupy memory in ‘Heap Area’.
- The direct super class of an array type is Object.
- They are always created at runtime.
- The elements of an array are stored in consecutive memory locations.

Dr. Smitha Shekar B

Working with Arrays – 1D Array

Declaration:

int a;	String a;
1. **int[ ] a;**	**String[ ] a;**
2. int [ ]a;	String [ ]a;
3. int a[ ];	String a[ ];
4. int[ ]a;	Sting[ ]a;

a is 1D int array

int[] a,b; // a-ID;b-1D

Int []a,b;//a-ID;b-1D

Int a[],b;//a-ID; b-variable

Creation/Initialization

int[] a;

a = new int[3];

int[] a = {10,20,30};

a[0]=10;
a[1]=20;
a[2]=30;

10	20	30	40

int[] a = new int[0]; //

Int[] a = new int[-2];//

Dr. Smitha Shekar B

Advantages of an Array

- Arrays are used to store multiple data items of same type by using only single name.
- We can access any element randomly by using **indexes** provided by arrays.
- Arrays can be used to implement other data structures like Linked list, Stacks,Queues,Trees, Graphs etc.

Dr. Smitha Shekar B

Disadvantages of an Array

- **Fixed Size**: size of the array is mentioned, thus they have fixed size. When array is created , size cannot be changed.
- **Memory Wastage**: Suppose an array of length 100 is created, but only 10 elements are inserted, then 90 blocks are empty and thus memory is wasted.
- **Strongly typed**: Array stores only similar data type, thus strongly typed.
- **Reduce performance**: The elements of array are stored in consecutive memory locations, thus to delete an element in an array we need to traverse through out the array, so this will reduce performance.
- **No methods**: No adding or removal of methods.

Dr. Smitha Shekar B

Examples

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

To create an array of integers, you could write:

int[] myNum = {10, 20, 30, 40};

You **access an array element** by referring to the index number.
This statement accesses the value of the first element in cars.

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
                System.out.println(cars[0]);

// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);


// Now outputs Opel instead of Volvo
```


Array Length

To find out how many elements an array has, use the length property.

Example

```
String[] cars =
{"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);


// Outputs 4
```

```
public class MyClass {

  public static void main(String[] args) {

    String[] cars = {"Volvo", "BMW", "Ford",
"Mazda"};

    System.out.println(cars.length);

  }

}
```

Dr. Smitha Shekar B

PrintArray – 1D

```
class PrintArray
{
 public static void main(String[] args)
{
 int[] a = {10,20,30};
  for(int i=0; i<a.length;i++)
    {
    System.out.println(a[i] + " ");
   }
}
}
```

Output: 10,20,30

Dr. Smitha Shekar B

Loop Through an Array

You can loop through the array elements with the **for** loop, and **use the length** property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array.

Example

```
String[]cars=
{"Volvo", "BMW", "Ford", "Mazda"};
for(int i = 0; i < cars.length; i++)
                    {
                System.out.println(cars[i]);
}
```

```
public class MyClass {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford",
"Mazda"};
    for (int i = 0; i < cars.length; i++) {
      System.out.println(cars[i]);
    }
  }
}
```

```
Volvo
BMW
Ford
Mazda
```

Dr. Smitha Shekar B

Multi-Dimensional Array

An Array having multiple rows and columns is known as Multi-dimensional array.

These are known as Array of arrays, because array is present in another array.

2-Types: 2D, 3D

Dr. Smitha Shekar B

Multi-Dimensional Array – 2D

2D – declaration, creation and initialization

- Matrix Array (Table) Ex. Array[4][3]
- Variable Size Array (Jagged Array)

	Column 1	Column2	Column 3
Row 0	[0][0]	[0][1]	[0][2]
Row 1	[1][0]	[1][1]	[1][2]
Row 2	[2][0]	[2][1]	[2][2]
Row 3	[3][0]	[3][1]	[3][2]

Application : Multiplication Table

Dr. Smitha Shekar B

Intialization Retrive-Printing

a= new int[2][3];

Int[][] a = {{10,20,30},{40,50,60}};

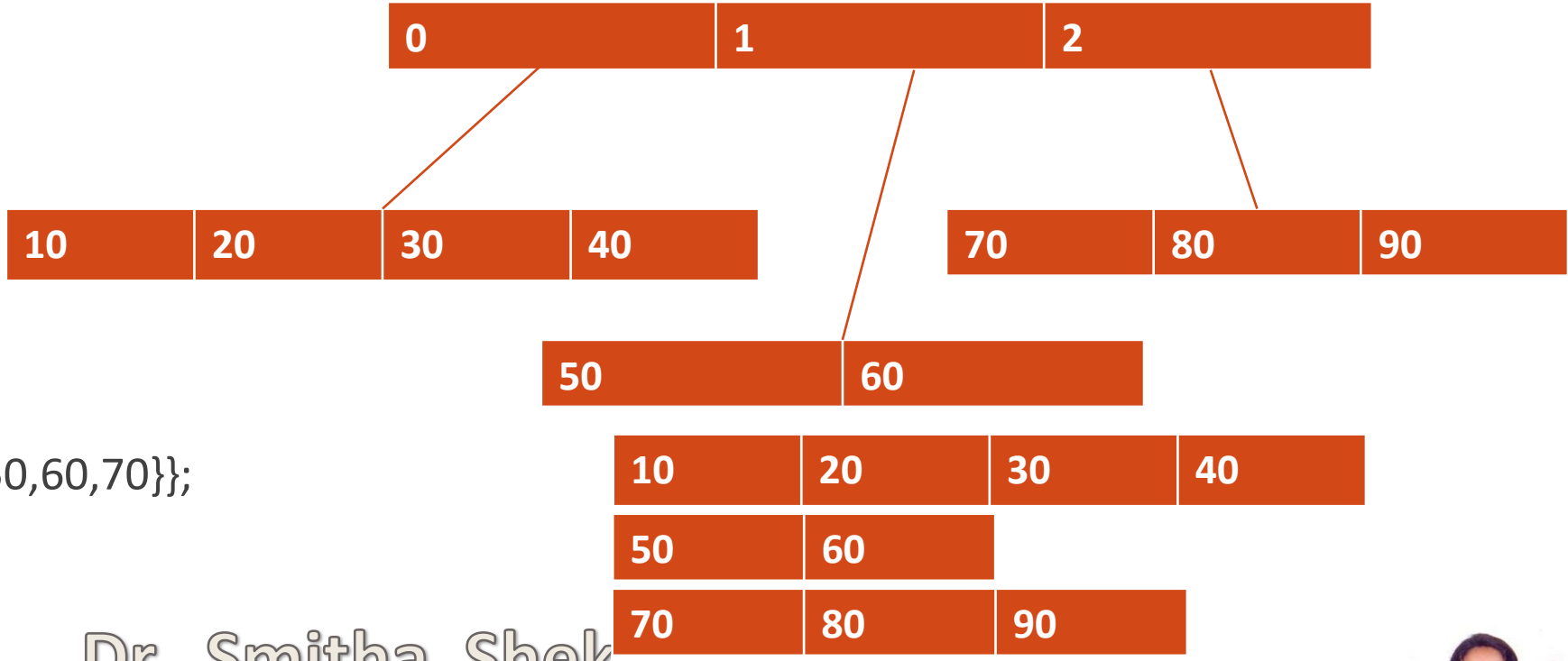
Int[][] a = new int[2][];

a[0]= new int[4];

a[1]= new int[3];

Int[][] a = {{10,20,30,40},{50,60,70}};

Int[][] a={{10,20,30,40},{50,60},{70,80,90};
System.out.println(a),(a[0]),(a[0][0]),(a.length),(a[0].length),(a[0][0].length);



Printing the array elements

```
class ArrayDemo
{
      public static void main(Strings [] args)
         {
            int[][] a = {{10,20,30},{50,60},{70,80,90}};
             System.out.println(a);  // check for (a[0]),(a[0][0]),(a.length), a[0].length), a[0][0].length),


          }

}
```

Dr. Smitha Shekar B

Printing the array elements

```
class ArrayDemo
{
    public static void main(Strings [] args)
      {
        int[][] a = {{10,20,30},{50,60},{70,80,90}};
            for(int i=0;i<a.length,i++)
              {
               for(int j=0;j<a[i].length,j++)
                 {
                    System.out.println(a[i][j]  +  “   “);
                 }
                System.out.println();
              }
        }
}
```

```
10
20
30
50
60
70
80
90
```

Dr. Smitha Shekar B

Multi-Dimensional Array – 3D

3D – declaration, creation and initialization

- **Ex. Array[2][3][2]**
- **Declaration**
- **Creation**
- **Initialization**

Declaration

int[][][] a;

int[] [][]a;
int[] []a[];
int [][] a[];
int []a[][];

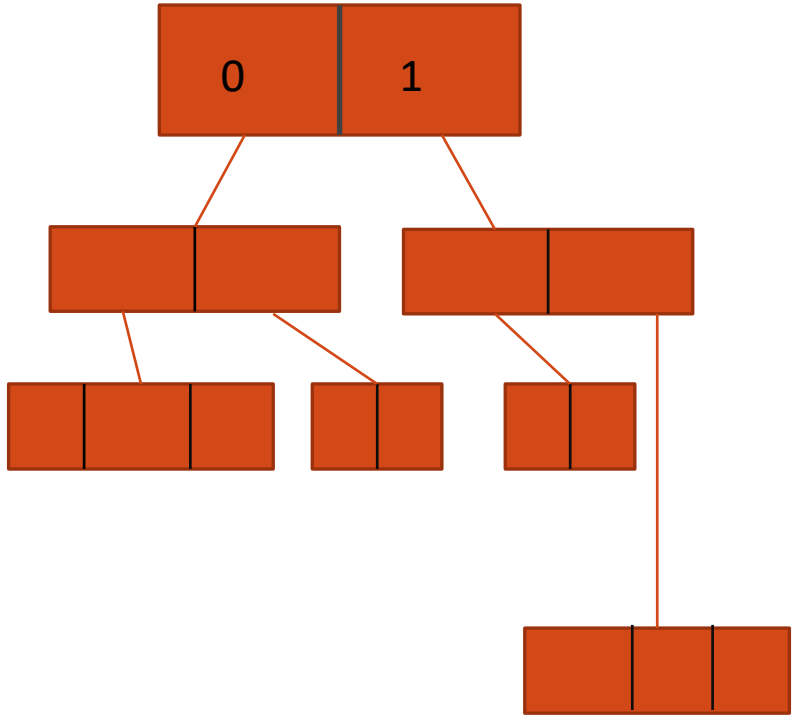
Creation

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

a[0] = new int[2][];
a[0][0]=new int[3];
a[0][1]=new int[2];

a[1]=new int[2][];
a[1][0]=new int[2];
a[1][1]=new int[3];

Index: a[0][0][0]=10;
a[0][0][1]=20;
a[1][0][0]=30; **check** for a[0][0];//CTE



Dr. Smitha Shekar B

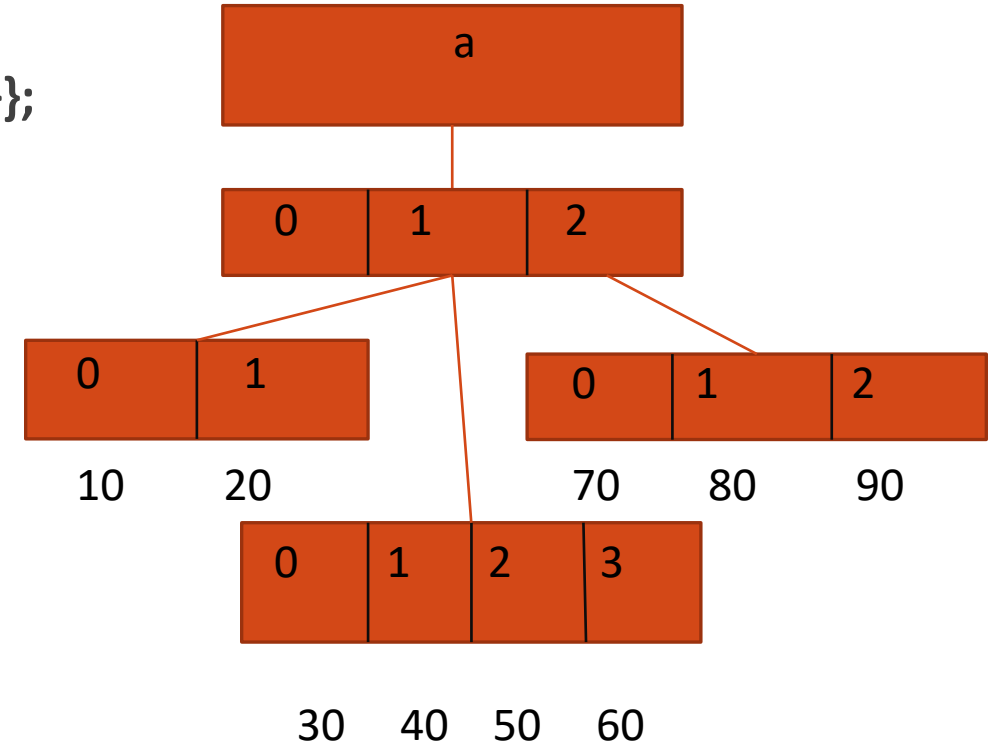
Multi-Dimensional Array – 3D

3D – declaration, creation and initialization

Single line: int[][][] a = {{{10,20},{30,40,50,60},{70,80,90}}};

Check for:

S.o.pln(a);	**S.o.pln(a.length);**
S.o.pln(a[0]);	**S.o.pln(a[0].length);**
S.o.pln(a[0][0]);	**S.o.pln(a[0][0].length);**
S.o.pln(a[0][0][0]);	**S.o.pln(a[0][0][0].length);**



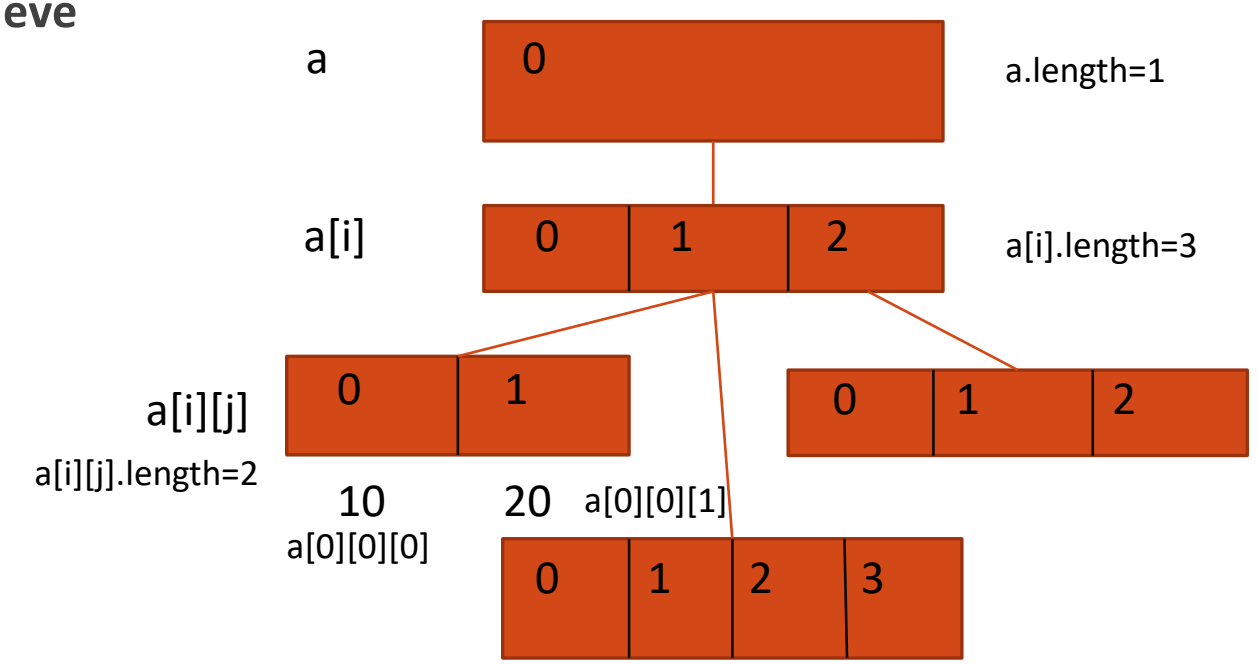
Dr. Smitha Shekar B

Multi-Dimensional Array – 3D

3D – declaration, creation, initialization and retrieve

int[][][] a = {{{10,20},{30,40,50,60},{70,80,90}}};

```
for(int i=0;i<a.length;i++)
{
  for(int j=0;j<a[i].length;j++)
   {
     for(int k=0;k<a[i][j].length;k++)
      {
       System.out.println(a[i][j][k] + " ");
       }
     System.out.println();
   }
}
```



Dr. Smitha Shekar B

Java String

- In Java , String is a sequence of characters.
- Java implements strings as objects of type **String.**
- Strings are used for storing text.
- A String variable contains a collection of characters surrounded by double quotes.
- Example

  Create a variable of type String and assign it a value:

String greeting = "Hello";

Dr. Smitha Shekar B

JAVA String

- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- Java has methods to,
  - compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- **String** objects can be constructed in a number of ways, making it easy to obtain a string when needed.
- When you create a **String** object, you are creating a string that cannot be changed.
  - That is, once a **String** object has been created, you cannot change the characters that comprise that string.

Dr. Smitha Shekar B

- For those cases in which a modifiable string is desired,
- Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

- The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically.

- All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations.

- All three implement the **CharSequence** interface.

- To say that the strings within objects of type **String** are unchangeable
  means that the contents of the **String** instance cannot be changed after it has been created.

- However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors

The **String** class supports several constructors.

- To create an empty **String**, call the **default constructor.**

  For example,

  String s = new String();

  will create an instance of **String** with no characters in it.

- To create strings that have initial values,

➢ The **String** class provides a variety of constructors to handle this.

➢ To create a **String** initialized by an array of characters, use the constructor shown here: **String(char *chars*[ ])**

For example

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string "abc".

Dr. Smitha Shekar B

- To specify a **subrange** of a character array as an initializer use the following constructor:

String(char *chars*[ ], int *startIndex*, int *numChars*)

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use.

For example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **cde**.

Construct one String from another

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

String(String *strObj*)

Here, *strObj* is a **String** object.

Example: **// Construct one String from another.**

```
class MakeString {
        public static void main(String args[]) {
                char c[] = {'J', 'a', 'v', 'a'};
                String s1 = new String(c);
                String s2 = new String(s1);
                System.out.println(s1);
                System.out.println(s2);
                }
                }
```

The output from this program is as follows:

```
Java
Java
```


String class provides constructors that initialize a string when given a **byte** array.

Two forms are shown here:

String(byte *asciiChars*[ ])

String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

Here, *asciiChars* specifies the array of bytes.

The second form allows you to specify subrange.

In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

The following program illustrates these constructors.

```
// Construct string from subset of char array.

class SubStringCons {
        public static void main(String args[]) {
        byte ascii[] = {65, 66, 67, 68, 69, 70 };
        String s1 = new String(ascii);
        System.out.println(s1);
        String s2 = new String(ascii, 2, 3);
        System.out.println(s2);
        }
        }
```

This program generates the following output:

```
ABCDEF
CDE
```

Dr. Smitha Shekar B

NOTE

- The contents of the array are copied whenever you create a **String** object from an array.
- If you modify the contents of the array after you have created the string, the **String** will be unchanged.

You can construct a **String** from a **StringBuffer** by using the constructor :

- **String(StringBuffer *strBufObj*)**

You can construct a **String** from a **StringBuilder** by using this constructor:

- **String(StringBuilder *strBuildObj*)**

Dr. Smitha Shekar B

String Length

- A String in Java is actually an object, which contain methods that can perform certain operations on strings.
- For example, the length of a string can be found with the length() method.

Example

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
```

Dr. Smitha Shekar B

length( ) method

- The length of a string is the number of characters that it contains.

- To obtain this value, call the **length( )** method, shown here:

```
int length( )
```

- The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Dr. Smitha Shekar B

Special String Operations

String Literals

String Concatenation

String Concatenation with Other Data Types

String Conversion and toString( )

Dr. Smitha Shekar B

Character Extraction

charAt( )

getChars( )

getBytes( )

toCharArray( )

String Comparison

equals( ) and equalsIgnoreCase( )

regionMatches( )

startsWith( ) and endsWith( )

equals( ) Versus ==

compareTo( )

Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf( )** Searches for the first occurrence of a character or substring.
- **lastIndexOf( )** Searches for the last occurrence of a character or substring.

Dr. Smitha Shekar B

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete.

Sampling of the methods are as follows,

- **substring( ) // There are 2 methods**
- **concat( )**
- **replace( ) // There are 2 methods**
- **trim( )**

Dr. Smitha Shekar B

More String Methods

There are many string methods available, for example **toUpperCase()** and **toLowerCase()**:

Example

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   //   Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   //   Outputs "hello world"
```

Dr. Smitha Shekar B

String Concatenation

The + operator can be used between strings to add them together to make a new string.

This is called **concatenation**.

Example

```
String firstName = “Smitha";
String lastName = “Shekar";
System.out.println(firstName + " " + lastName);
```

Dr. Smitha Shekar B

Sample Program

```
public class MyClass {
  public static void main(String[] args) {
    String firstName = “Smitha";
    String lastName = “Shekar";
    System.out.println(firstName + " " + lastName);
  }
}
```

Smitha Shekar

You can also use the concat() method to concatenate two strings.

Example

```
String firstName = “ABC ";
String lastName = “XYZ";
System.out.println(firstName.concat(lastName));
```

```
public class MyClass {
  public static void main(String[] args) {
    String firstName = “ABC";
    String lastName = “XYZ";

System.out.println(firstName.concat(last
Name));
  }
}
```

Dr. Smitha Shekar B

Java uses the + operator for both addition and concatenation.

Numbers are added.

If you add two numbers, the result will be a number:

Example

```
int x = 10;
int y = 20;
int z = x + y;      // z will be 30 (an integer/number)
```

Strings are concatenated.

If you add two strings, the result will be a string concatenation:

Example

```
String x = "10";
String y = "20";
String z = x + y;   // z will be 1020 (a String)
```

Dr. Smitha Shekar B

NOTE

If you add a number and a string, the result will be a string concatenation.

Example

```
String x = "10";
int y = 20;
String z = x + y;   // z will be 1020 (a String)
```

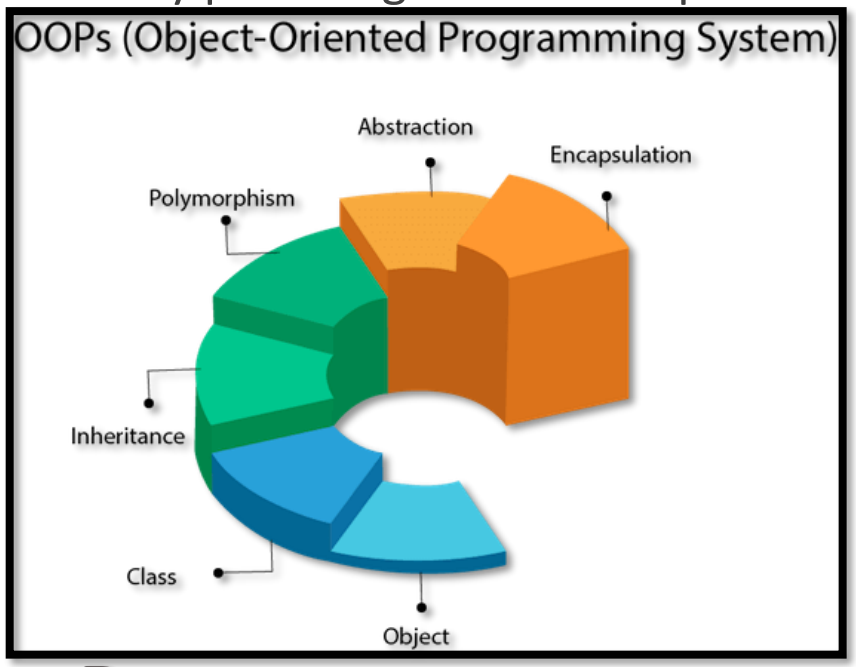
Dr. Smitha Shekar B

Introduction to Classes

Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation



Dr. Smitha Shekar B

Introduction to Classes

Object

Any entity that has state and behavior is known as an object.

For example, a chair, pen, table, keyboard, bike, etc.

It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.
Objects can communicate without knowing the details of each other's data or code.

The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Dr. Smitha Shekar B

Introduction to Classes

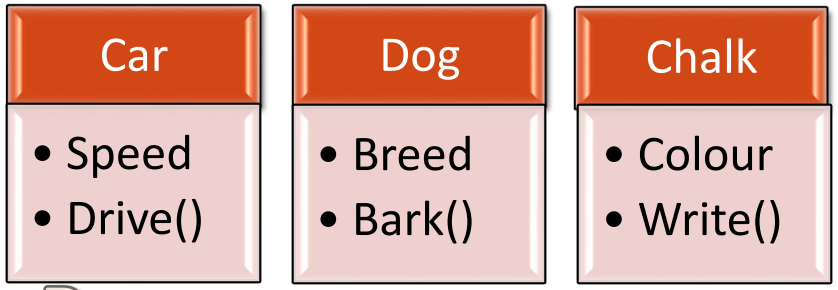
Class

Collection of objects is called class.

It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object.

Class doesn't consume any space.



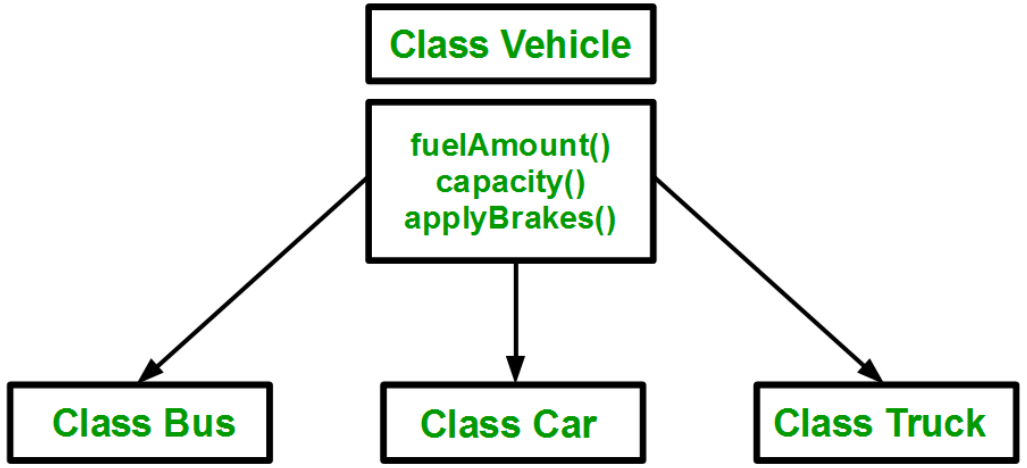
Dr. Smitha Shekar B

Introduction to Classes

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.

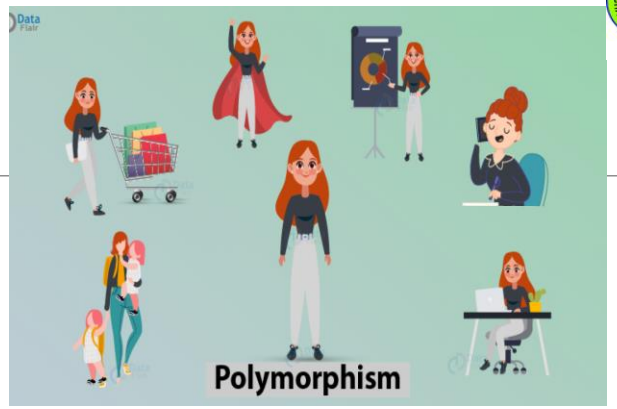
It provides code reusability. It is used to achieve runtime polymorphism.



Dr. Smitha Shekar B

Introduction to Classes

Polymorphism



If one task is performed in different ways, it is known as polymorphism.

For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

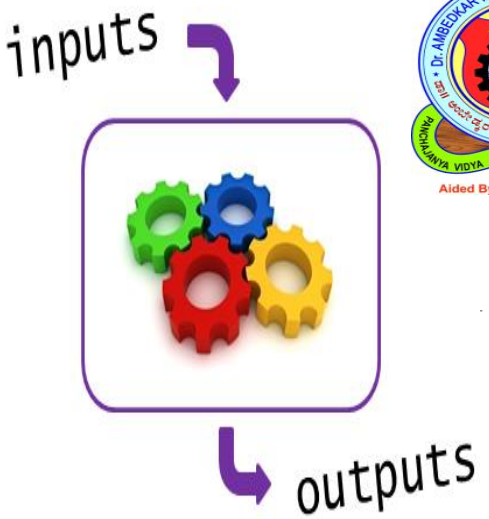
In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Dr. Smitha Shekar B

Introduction to Classes

Abstraction



Hiding internal details and showing functionality is known as abstraction.

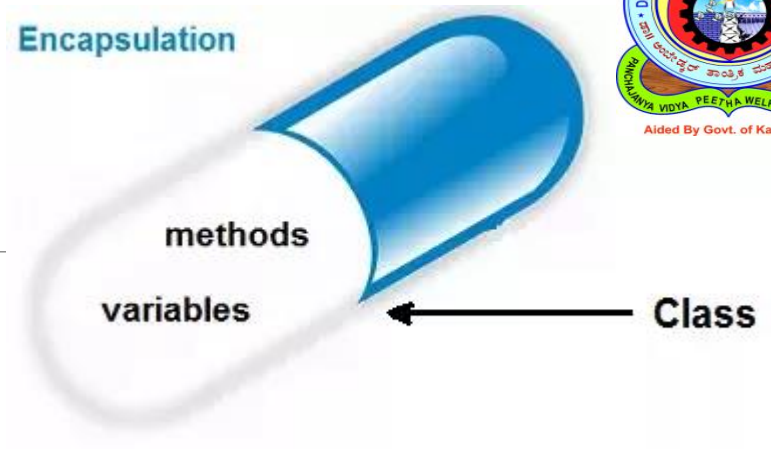
For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Dr. Smitha Shekar B

Introduction to Classes

Encapsulation



Binding (or wrapping) code and data together into a single unit are known as encapsulation.

For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation.

Java bean is the fully encapsulated class because all the data members are private here.

Dr. Smitha Shekar B

Introduction to Classes

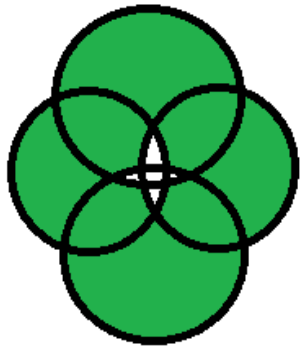
Coupling

Coupling refers to the knowledge or information or dependency of another class.

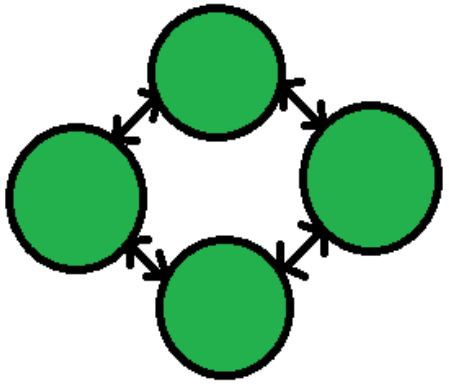
It arises when classes are aware of each other.
If a class has the details information of another class, there is strong coupling.

In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field.

You can use interfaces for the weaker coupling because there is no concrete implementation.



Tight coupling:
1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:
1. Less Interdependency
2. Less coordination
3. Less information flow

Introduction to Classes

Cohesion

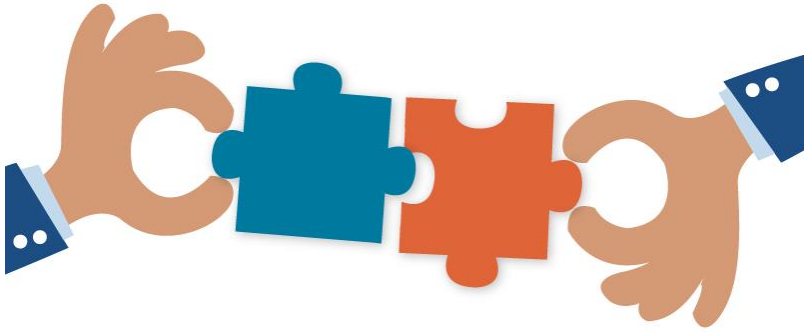
Cohesion refers to the level of a component which performs a single well-defined task.

A single well-defined task is done by a highly cohesive method.

The weakly cohesive method will split the task into separate parts.

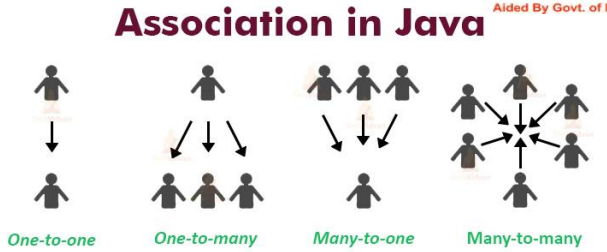
The java.io package is a highly cohesive package because it has I/O related classes and interface.

The java.util package is a weakly cohesive package because it has unrelated classes and interfaces.



Dr. Smitha Shekar B

Introduction to Classes



Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples.

For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be unidirectional or bidirectional.

Introduction to Classes

Composition

The composition is also a way to achieve Association.

The composition represents the relationship where one object contains other objects as a part of its state.

There is a strong relationship between the containing object and the dependent object.

It is the state where containing objects do not have an independent existence.

If you delete the parent object, all the child objects will be deleted automatically.

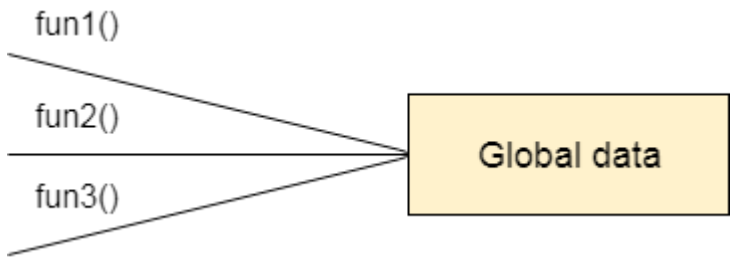
Dr. Smitha Shekar B

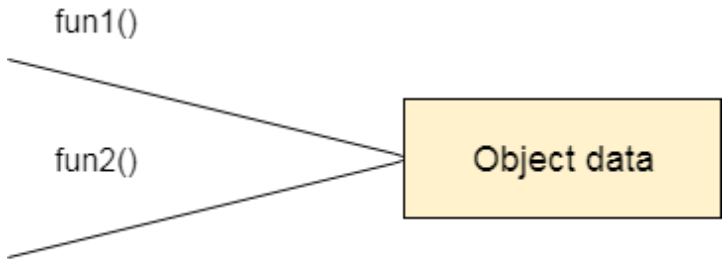
Advantage of OOPs over Procedure-oriented programming language

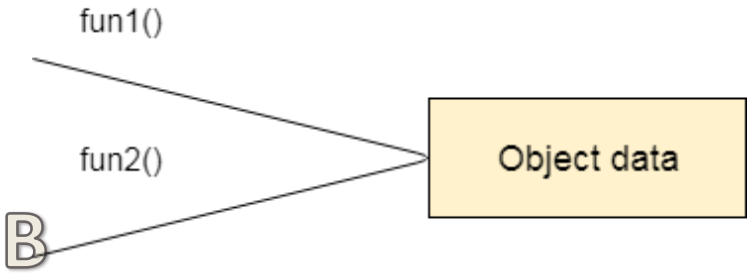
1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.





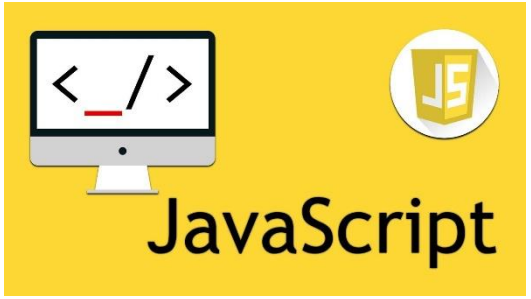


Dr. Smitha Shekar B

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance.

JavaScript and VBScript are examples of object-based programming languages.



Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as **convention** not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention.

If you fail to follow these conventions, it may generate confusion or erroneous code.

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Advantage of naming conventions in Java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers.

Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

The name must not contain any white spaces.
The name should not start with special characters like & (ampersand), $ (dollar), _ (underscore).
Let's see some other rules that should be followed by identifiers.

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Class

1. It should start with the uppercase letter.
2. It should be a noun such as Color, Button, System, Thread, etc.
3. Use appropriate words, instead of acronyms.

•**Example: -**

```
1.public class Employee
2.{
3.//code snippet
4.}
```

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Interface

1. It should start with the uppercase letter.
2. It should be an adjective such as Runnable, Remote, ActionListener.
3. Use appropriate words, instead of acronyms.

•**Example: -**

```
1.interface Printable
2.{
3.//code snippet
4.}
```

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Method

1. It should start with lowercase letter.
2. It should be a verb such as main(), print(), println().
3. If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

•**Example:-**

```
1. class Employee
2.{
3.//method
4.void draw()
5.{
6.//code snippet
7.}
8.}
```

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Variable

1. It should start with a lowercase letter such as id, name.
2. It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
3. If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
4. Avoid using one-character variables such as x, y, z.

•**Example :-**

```
1.  class Employee
2.{
3.//variable
4.int id;
5.//code snippet
6.}
```

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Package

1. It should be a lowercase letter such as java, lang.
2. If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

•**Example :-**

```
1.package com.javaexample; //package
2.class Employee
3.{
4.//code snippet
5.}
```

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

Constant

1. It should be in uppercase letters such as RED, YELLOW.
2. If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
3. It may contain digits but not as the first letter.

•**Example :-**

```
1.class Employee
2.{
3.//constant
4. static final int MIN_AGE = 18;
5.//code snippet
6.}
```

Dr. Smitha Shekar B

JAVA NAMING CONVENTIONS

CamelCase in java naming conventions

1. Java follows camel-case syntax for naming the class, interface, method, and variable.

2. If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Dr. Smitha Shekar B

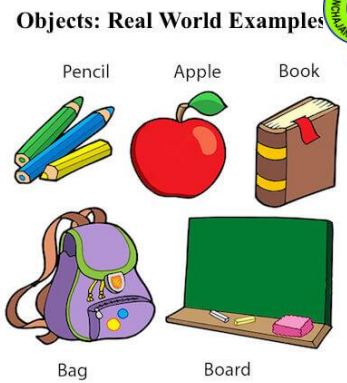
OBJECTS AND CLASSES IN JAVA

In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA



What is an object in Java?

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

1. State: represents the data (value) of an object.
2. Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
3. Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

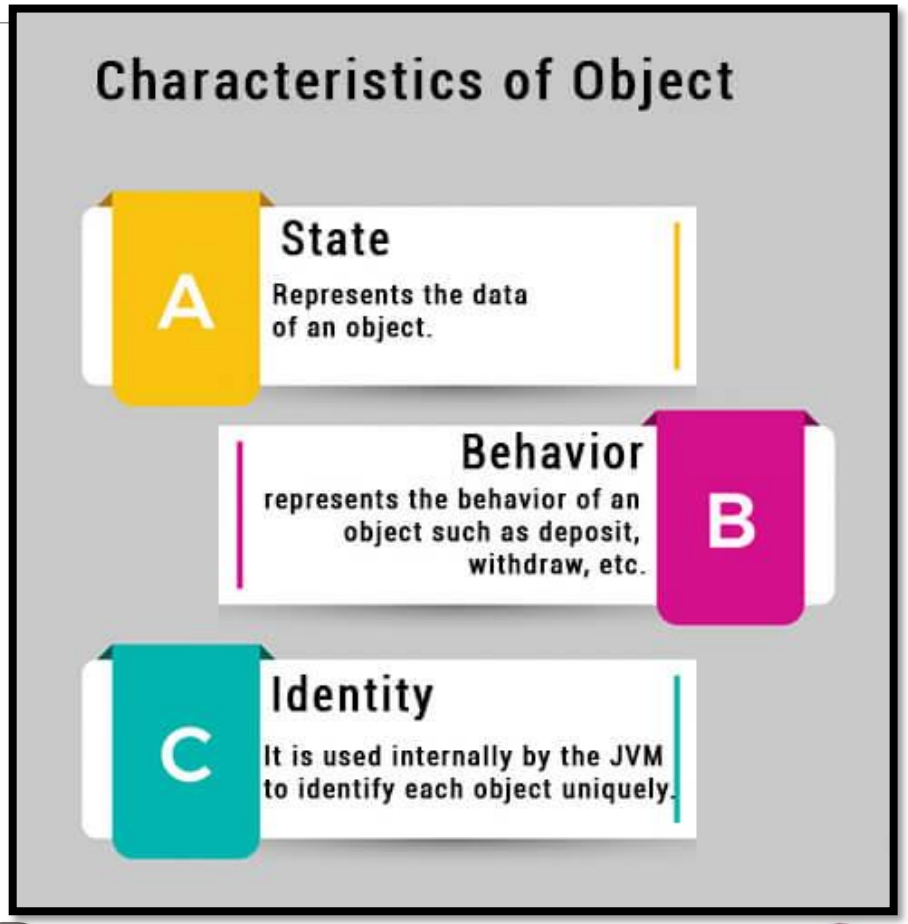
Characteristics of an Object

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

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

Object Definitions:

1. An object is a real-world entity.
2. An object is a runtime entity.
3. The object is an entity which has state and behavior.
4. The object is an instance of a class.



Dr. Smitha Shekar B

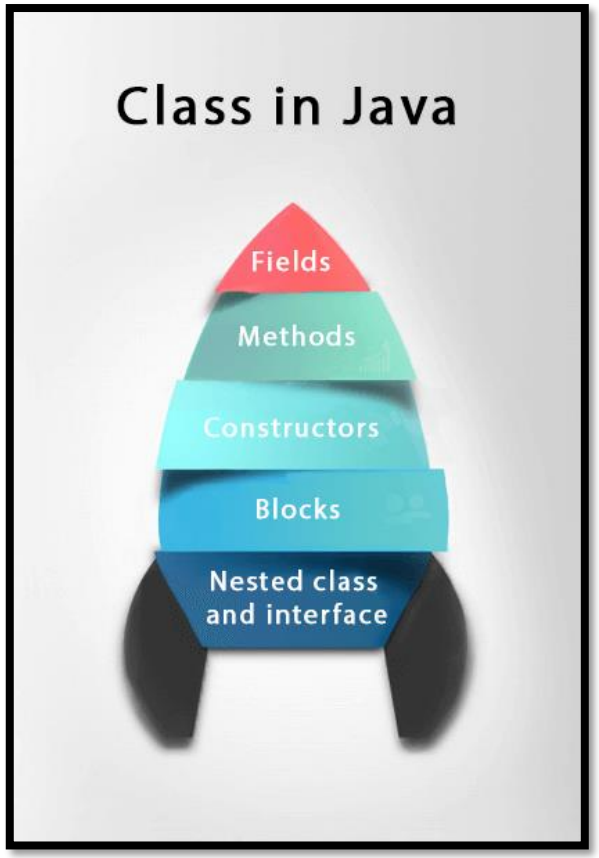
OBJECTS AND CLASSES IN JAVA

What is a class in Java?

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

A class in Java can contain:

1. Fields
2. Methods
3. Constructors
4. Blocks
5. Nested class and interface



Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Syntax to declare a class:

```
1. class <class_name>{
2.     field;
3.     method;
4. }
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable.

Instance variable doesn't get memory at compile time.

It gets memory at runtime when an object or instance is created.

That is why it is known as an instance variable.

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

- A named section of code.
- Code re-use, Write-Once Run it many times.
- In Java, all methods are part of classes.
- Static methods can be called without declaring an object of the class first.

1. Advantage of Method
    1. Code Reusability
    2. Code Optimization

Dr. Smitha Shekar B

Anatomy of a Java Method

Access Modifier Declaration Method-name Parameter

```
public static int sqaure(int number)
```

Return-type

```
{
    int result= number * number;

    return result;
}
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

new keyword in Java

The **new** keyword is used to allocate memory at runtime.

All objects get memory in Heap memory area.

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name.

We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference var
iable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file.
If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

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

```
0
Null
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

3 Ways to initialize object

There are 3 ways to initialize object in Java.

- By reference variable
- By method
- By constructor

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Smitha";
  System.out.println(s1.id+"  "+s1.name);//printing members
                                          with a white space
 }
}
```

Output:

101 Smitha

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

1) Object and Class Example: Initialization through reference

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();

//Initializing objects
  s1.id=101;
  s1.name="Smitha";
  s2.id=102;
  s2.name="Kriti";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

101 Smitha
102 Kriti

OBJECTS AND CLASSES IN JAVA

2) Object and Class Example: Initialization through method

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

File: TestStudent4.java

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

Output:

111 Smitha
222 Aishwarya

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

Bike is created

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.
File: TestEmployee.java

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

Output:

```
101 AAA 45000.0
102 BBB 25000.0
103 CCC 55000.0
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.
File: TestRectangle1.java

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

}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

```
55
45
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object.
It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation();//anonymous object
```

Calling method through a reference:

```
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Anonymous object

Let's see the full example of an anonymous object in Java.

```
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output:

Factorial is 120

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of reference variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Creating multiple objects by one type only

Let's see this example

```
//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
 public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

55
45

Dr. Smitha Shekar B

OBJECTS AND CLASSES IN JAVA

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() meth
ods
class Account{
int acc_no;
String name;
float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
```

```
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created.
  - At the time of calling constructor, memory for the object is allocated in the memory.
  - It is a special type of method which is used to initialize the object.
  - Every time an object is created using the new() keyword, at least one constructor is called.

- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

- There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation.
It is not necessary to write a constructor for a class.
It is because java compiler creates a default constructor if your class doesn't have any.

Dr. Smitha Shekar B

Example

```
class Test{
public ~~void~~ Test()
{
}
public static void main(String[] args)
{
Test t = new Test();
~~t.Test();~~
}
```

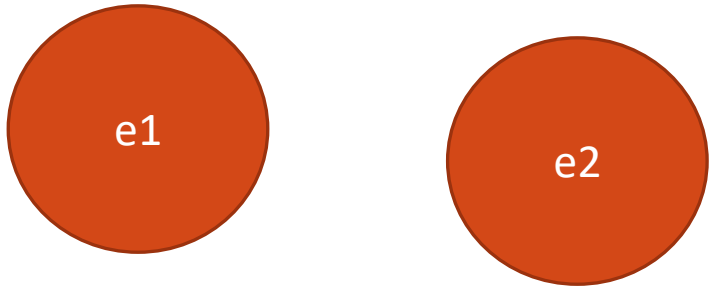
- It is a block(similar to a method) having same name as that of class name.
- Does not have any return type
- The only modifiers applicable for constructors are: public, protected, default & private
- It executes automatically when we create an object

Dr. Smitha Shekar B

Need for Constructor in JAVA

Example

```
class Employee{
String name=“ Smitha”;
int emp-id=101;
public static void main(Strings[] args)
{
Employee e1=new Employee();
e1.name=“Smitha”;
e1.emp-id=101;
Employee e2=new Employee();
}
}
```



Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Rules for creating Java constructor

There are three rules defined for the constructor.

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized
- The only modifiers applicable for constructors are: public, protected,default & private

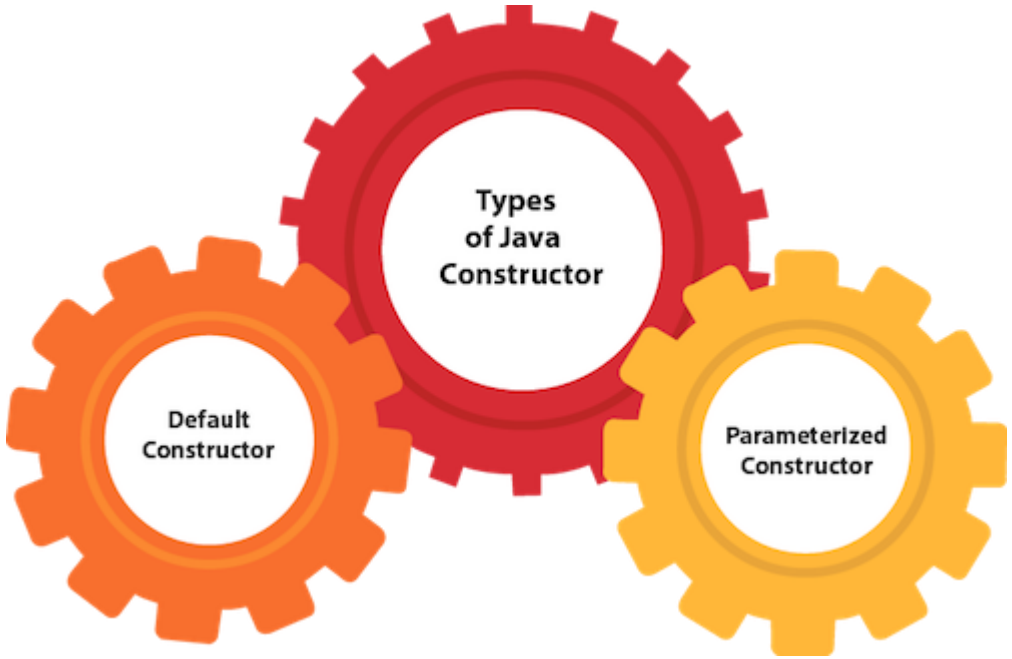
Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.
Syntax of default constructor:

```
<class_name>(){}
```

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation


```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1()
{
System.out.println("Bike is created");
}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

Bike is created

Dr. Smitha Shekar B

Default Constructor/ Parameterized

```
class Test{

int i;

String name;

public static void main(String[] args)

{

Test t=new Test();

System.out.println(t.1 + " " + t.name);

}

}
```

```
class Test{
Test() //User-defined
{
System.out.println("No Arg. Constructor");
}

public static void main(String[] args)
{
Test t=new Test();
}
}
```

```
class Test{
Test(int a)
{
System.out.println("Parameterized Constructor");
}

public static void main(String[] args)
{
Test t=new Test(10);
}
}
```

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

```
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Smitha");
    Student4 s2 = new Student4(222,"Savitha");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

111 Smitha
222 Savitha

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
They are arranged in a way that each constructor performs a different task.

They are differentiated by the compiler by the number of parameters in the list and their types.

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Java");
    Student5 s2 = new Student5(222,"OOPs",25);
    s1.display();
    s2.display();
   }
}
```

Output:

111 Java
222 OOPs 25

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

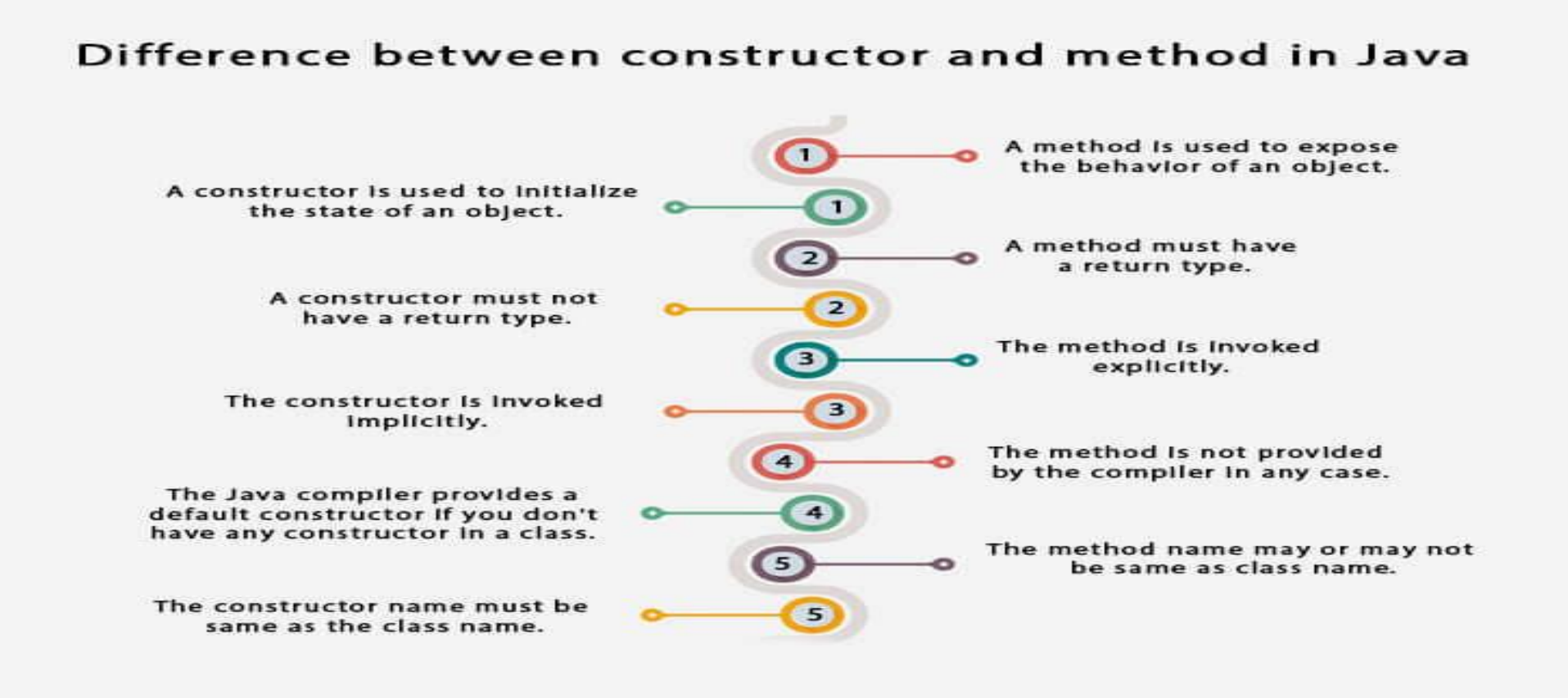
Difference between constructor and method in Java

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Difference between constructor and method in Java



Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.
There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Java Copy Constructor

In this example, we are going to copy the values of one object into another using Java constructor.

```
//Java program to initialize the values from one object to another object.
class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Java");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Java
111 Java
```

Dr. Smitha Shekar B

CONSTRUCTORS IN JAVA

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{
    int id;
    String name;
    Student7(int i,String n){
    id = i;
    name = n;
    }
    Student7(){}
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student7 s1 = new Student7(111,"Java");
    Student7 s2 = new Student7();
    s2.id=s1.id;
    s2.name=s1.name;
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Java
111 Java
```

Dr. Smitha Shekar B

Control Statements in JAVA

- A Java program normally executes from top to bottom, but if we want to control the order of execution of the program, based on logic and values, we use Control Statements.
- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.

Control statements are divided into three groups:

1) **selection (or conditional)statements** allow the program to choose different parts of the execution based on the outcome of an expression

2) **iteration(or looping) statements** enable program execution to repeat one or more statements

3) **jump statements** enable your program to execute in a non-linear fashion

Dr. Smitha Shekar B

Selection(or Conditional) Statements

Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.

Java provides four selection statements:

1) if
2) if-else
3) if-else-if
4) switch

Dr. Smitha Shekar B

Iteration(or looping) Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:
    1) while
    2) do-while
    3) for

Dr. Smitha Shekar B

Jump Statements

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:
    1) break
    2) continue
    3) return
- In addition, Java supports **exception handling** that can also alter the control flow of a program.

Dr. Smitha Shekar B

Introducing Access Control, Understanding static, Introducing final.

- Encapsulation links data with the code that manipulates it.

- However, encapsulation provides another important attribute: ***access control***.
  - Through encapsulation, you can control what parts of a program can access the members of a class.
  - By controlling access, you can prevent misuse.

- For example, allowing access to data only through a well defined set of methods, can prevent the misuse of that data.

- You will be introduced to the mechanism by which you can precisely control access to the various members of a class.

Dr. Smitha Shekar B

Access Control

Access Modifiers

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

Dr. Smitha Shekar B

How a member can be accessed is determined by the *access modif* attached to its declaration.

Java supplies a rich set of access modifiers.

- Some aspects of access control are related mostly to inheritance or packages.
  - (A *package* is, essentially, a grouping of classes.)

- Java's access modifiers are **public**, **private**, and **protected**.
- Java also defines a default access level.
- **protected** applies only when inheritance is involved.

Defining public and private.

- When a member of a class is modified by **public**, 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.
- Now you can understand why **main( )** has always been preceded by the **public** modifier.
  - It is called by code that is outside the program—that is, by the Java run-time system.
- When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

Dr. Smitha Shekar B

To understand the effects of public and private access, consider the following program:

//This program demonstrates the difference between public and private.

```
class Test {
 int a; // default access
 public int b; // public access
 private int c; // private access
 // methods to access c
 void setc(int i) { // set c's value
 c = i;
 }
 int getc() { // get c's value
 return c;
 } }
```

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
}
}
```


OBSERVATIONS

Inside the **Test** class,

---

a uses default access, which for this example is the same as specifying **public**.

b is explicitly specified as **public**.

Member **c** is given private access. This means that it cannot be accessed by code outside of its class.

So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods:

- **setc( )** and **getc( )**.
- If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

Understanding static

- To define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
  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.
- You can declare both methods and variables to be **static**.

Dr. Smitha Shekar B

Understanding static(Variables)

- 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.
- When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Dr. Smitha Shekar B

Understanding static(Methods)

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance).

Dr. Smitha Shekar B

Understanding static(block)

If you need to do computation in order to initialize your **static** variables,

you can declare a **static** block that gets executed exactly once, when the class is first loaded.

The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block.

Dr. Smitha Shekar B

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);
}
}
```

Static block initialized.
x = 42
a = 3
b = 12

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, which prints a message and
- then initializes **b** 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**.

Dr. Smitha Shekar B

Outside of the class in which they are defined, static methods and variables can be used independently of any object.

- To do so, you need only specify the name of their class followed by the dot operator.
- For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

- Here, *classname* is the name of the class in which the **static** method is declared.
- this format is similar to that used to call non-**static** methods through object reference variables.
- A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class.
- This is how Java implements a controlled version of global methods and global variables.

Dr. Smitha Shekar B

Example.

Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
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);
}
}
```

Output of this program:

a = 42

b = 99

Introducing final

A field can be declared as **final**.

Doing so prevents its contents from being modified, making it, essentially, a constant.

This means that you must initialize a **final** field when it is declared.

You can do this in one of two ways:

First, you can give it a value when it is declared.

Second, you can assign it a value within a constructor.

The first approach is the most common.

Dr. Smitha Shekar B

Introducing final

Example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.

It is a common coding convention to choose all uppercase identifiers for **final** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**.

Declaring a parameter **final** prevents it from being changed within the method.

Declaring a local variable **final** prevents it from being assigned a value more than once.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

Dr. Smitha Shekar B

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*.

The scope of a nested class is bounded by the scope of its enclosing class.
Thus, if class B is defined within class A, then B does not exist independently of A.

A nested class has access to the members, including private members, of the class in which it is nested.

However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.

It is also possible to declare a nested class that is local to a block.

Dr. Smitha Shekar B

Introducing Nested and Inner Classes

There are two types of nested classes: *static* and *non-static*.

A static nested class is one that has the **static** modifier applied.
Because it is static, it must access the non-static members of its enclosing class through an object.

That is, it cannot refer to non-static members of its enclosing class directly.
Because of this restriction, static nested classes are seldom used.

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 and may refer to them directly in the same way that other non-static members of the outer class do.

Dr. Smitha Shekar B

This program illustrates how to define and use an inner class.

The class named **Outer,**

- has one instance variable named **outer_x**,
- one instance method named **test( )**, and
- defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

display: outer_x = 100

Dr. Smitha Shekar B

Defining a Package

- A package is both a naming and a visibility control mechanism:

  1) divides the name space into disjoint subsets.

     It is possible to define classes within a package that are not accessible by code outside the package.

  2) controls the visibility of classes and their members.

     It is possible to define class members that are only exposed to other members of the same package.

- Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages

Creating a Package

A package statement inserted as the first line of the source file:

```
package myPackage;
  class MyClass1 { … }
  class MyClass2 { … }
```

This means that all classes in this file belong to the myPackage package.

The package statement creates a name space where such classes are stored.

When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

Other files may include the same package instruction:

1. package myPackage;
   class MyClass1 { … }
   class MyClass2 { … }
2. package myPackage;
   class MyClass3{ … }

A package may be distributed through several source files

Dr. Smitha Shekar B

Interfaces (Differences between classes and interfaces)

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- One class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.
- Interfaces are developed to support multiple inheritance.
- The methods present in interfaces r pure abstract..

Dr. Smitha Shekar B

- The access specifiers public, private,protected are possible with classes, but the interface uses only one specifier public.
- interfaces contains only the method declarations.... no definitions.
- An interface defines, which method a class has to implement.
- Another important point about interfaces is that a class can implement multiple interfaces.

Defining an interface

- Using interface, we specify what a class must do, but not how it does this.
- An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.
- An interface is defined with an interface keyword.

Dr. Smitha Shekar B

Defining an Interface

- An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body.

- For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3


{
        // constant declarations double E = 2.718282;
      //method signatures
 void doSomething (int i, double x);
int doSomethingElse(String s);
}
```

Dr. Smitha Shekar B

- The public access specifier indicates that the interface can be used by any class in any package.
- If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.
- An interface can extend other interfaces, just as a class can extend or subclass another class.
- A class can extend only one other class, an interface can extend any number of interfaces.
- The interface declaration includes a comma-separated list of all the interfaces that it extends.

Implementing interface

General format:

```
access interface name {
type method-name1(parameter-list);
type method-name2(parameter-list);
…
type var-name1 = value1;
type var-nameM = valueM;
…
}
```

Dr. Smitha Shekar B

Two types of access:

1) public – interface may be used anywhere in a program

2) default – interface may be used in the current package only

Interface methods have no bodies – they end with the semicolon after the parameter list.

They are essentially abstract methods.

An interface may include variables, but they must be final, static and initialized with a constant value.

In a public interface, all members are implicitly public.

Interface Implementation

A class implements an interface if it provides a complete set of methods defined by this interface.

1) any number of classes may implement an interface

2) one class may implement any number of interfaces

Each class is free to determine the details of its implementation.

Implementation relation is written with the implements keyword.

Dr. Smitha Shekar B

Implementation Format

General format of a class that includes the implements clause:

Syntax:

access class name extends super-class implements interface1, interface2, …, interfaceN {

…

}

Access is public or default.

Dr. Smitha Shekar B

Implementation Comments

If a class implements several interfaces, they are separated with a comma.

If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.

The methods that implement an interface must be declared public.

The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: Interface

Declaration of the Callback interface:

```
interface Callback {
void callback(int param);
}
```

Client class implements the Callback interface:

```
class Client implements Callback {
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Dr. Smitha Shekar B

More Methods in Implementation

An implementing class may also declare its own methods:

```
class Client implements Callback {
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement “ +
“interfaces may also define ” + “other members, too.");
}
}
```

Dr. Smitha Shekar B

Applying interfaces

- **A Java *interface* declares a set of method signatures i.e., says what behavior exists**
- **Does not say how the behavior is implemented**

  i.e., does not give code for the methods
- **Does not describe any state (but may include “final” constants)**

Dr. Smitha Shekar B

- A concrete class that implements an interface Contains "implements *InterfaceName*" in the class declaration
- Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface
- An abstract class can also implement an interface
- Can optionally have implementations of some or all interface methods

- Interfaces and Extends both describe an “is- a” relation
- If B *implements* interface A, then B inherits the (abstract) method signatures in A
- If B *extends* class A, then B inherits everything in A,
  - which can include method code and instance variables as well as abstract method signatures
- Inheritance” is sometimes used to talk about the superclass/subclass “extends” relation only

Variables in interface

- Variables declared in an interface must be constants.
- A technique to import shared constants into multiple classes:

1) declare an interface with variables initialized to the desired values

2) include that interface in a class through implementation

As no methods are included in the interface, the class does not implement anything except importing the variables as constants.

Dr. Smitha Shekar B

Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;

interface SharedConstants {

int NO = 0;

int YES = 1;

int MAYBE = 2;

int LATER = 3;

int SOON = 4;

int NEVER = 5;

}
```

Dr. Smitha Shekar B

Example: Interface Variables 2

Question implements SharedConstants, including all its constants.

Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30) return NO;
else if (prob < 60) return YES;
else if (prob < 75) return LATER;
else if (prob < 98) return SOON;
else return NEVER;
}
}
```

Dr. Smitha Shekar B

Example: Interface Variables 3

AskMe includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO: System.out.println("No"); break;
case YES: System.out.println("Yes"); break;
case MAYBE: System.out.println("Maybe"); break;
case LATER: System.out.println("Later"); break;
case SOON: System.out.println("Soon"); break;
case NEVER: System.out.println("Never"); break;
}
}
```


Example: Interface Variables 4

- The testing function relies on the fact that both ask and answer methods,
- defined in different classes, rely on the same constants:

```
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```


extending interfaces

One interface may inherit another interface.

The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
void myMethod1(…) ;
}
interface MyInterface2 extends MyInterface1 {
void myMethod2(…) ;
}
```

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example: Interface Inheritance 1

Consider interfaces A and B.

```
interface A {
void meth1();
void meth2();
}
```

B extends A:

```
interface B extends A {
void meth3();
}
```

Dr. Smitha Shekar B

Example: Interface Inheritance 2

MyClass must implement all of A and B methods:

```
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
} }
```


Example: Interface Inheritance 3

Create a new MyClass object, then invoke all interface methods on it:

```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```


Package java.io

Provides for system input and output through data streams, serialization and the file system.

Interface Summary

DataInput The DataInput interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.

DataOutputThe DataOutput interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream

Externalizable Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.

SerializableSerializability of a class is enabled by the class implementing the java.io.Serializable interface.

Dr. Smitha Shekar B

Thank You