# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 1

## Lecture 1

- Why Java
- History of Java
- OOPs Concept
- Features of Java

## Why Java

### 1. Java is Easy to Learn

Java is beginner-friendly and one of the most popular programming languages among new developers. It has a syntax similar to English and enables you to write, debug, compile, and learn java programming fast.

### 2. Java is Versatile

Java follows the ‘write once and run anywhere’ principle and can be used for programming applications using different platforms.

## 3. Java is Object-Oriented

Java is an object-oriented programming language and this makes it scalable and flexible. Since it uses the syntax of an object-oriented programming language, the developers can create modular programs.

## 4. Java is Scalable

Java is used everywhere, including desktops, mobile, applications, and so on. It can effectively run on any operating system and is ideal for building applications. This scalability and versatility have made Java a game-changing language across multiple sectors and devices.

## 5. Java is Platform-Independent

Java has the ability to easily move across platforms and can be run similarly on different systems. This critical nature of being platform-independent at the source and binary levels makes Java an essential language to learn for developers.

## 6. Java Has a Rich API

Java has a rich Application Programming Interface (API) system that includes packages, interfaces, and classes, along with their methods and fields. This enables developers to integrate various websites and applications.

## 7. Java is Open Source

Most of Java’s features are open-source; this makes building applications cheap and easy. Java has the support of libraries like Google Guava, Maven, JHipster, and Apache Commons, allowing developers a wide choice to work with.

## 8. Java is Free of Cost

Java is a free-to-download software on Oracle Binary Code License (BCL), enabling beginners to develop applications easily and learn Java programming effectively.

## History of Java

- Java is an Object-Oriented programming language developed by **James Gosling** in the early 1990s.
- The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc.
- Originally C++ was considered to be used in the project but the idea was rejected for several reasons.
- James Gosling and his team called their project "**Greentalk**" and its file extension was **.gt** and later became to known as "**OAK**".

## Why “Oak”?

- The name **Oak** was used by **Gosling** after an **oak tree** that remained outside his office.
- But they had to later rename it as “**JAVA**” as it was already a trademark by **Oak Technologies**.
- **Java** name was decided after much discussion since it was so unique.
- Gosling came up with this name while having a coffee near his office.

- Java was created on the principles like **Robust, Portable, Platform Independent, High Performance, Multithread, etc.** and was called one of the **Ten Best Products of 1995** by the **TIME MAGAZINE**.

## History of various Java versions

| VERSION | RELEASE DATE |
|---|---|
| JDK Beta | 1995 |
| JDK 1.0 | January 1996 |
| JDK 1.1 | February 1997 |
| J2SE 1.2 | December 1998 |
| J2SE 1.3 | May 2000 |
| J2SE 1.4 | February 2002 |
| J2SE 5.0 | September 2004 |
| JAVA SE 6 | December 2006 |
| JAVA SE 7 | July 2011 |
| JAVA SE 8 | March 2014 |
| JAVA SE 9 | September 2017 |
| JAVA SE 10 | March 2018 |
| JAVA SE 11 | September 2018 |
| JAVA SE 12 | March 2019 |

## Latest Versions of Java

- Java SE 21 (LTS)  September, 19th 2023
- Java 22 was released on March 19, 2024.

## OOPs (Object Oriented Programming System)

- **Object** means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:
- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

**Object -** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviours -wagging, barking, eating. An object is an instance of a class.

**Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support

***Inheritance***

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

***Polymorphism***

- When **one task is performed by different ways** i.e. known as polymorphism. For example. shape or rectangle etc.
- In java, we use method overloading and method overriding to achieve polymorphism.

***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.

***Encapsulation***

- **Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: 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.

## Features of Java

- **Object Oriented :** In java everything is an Object.
- **Platform independent:** Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code.
- **Simple :**Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.
- **Secure :** With Java's secure feature it enables to develop virus-free, tamper-free systems.

- **Architectural- neutral :**Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors.
- **Portable :**being architectural neutral and having no implementation dependent aspects of the specification makes Java portable.
- **Robust :**Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multi-threaded :** With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously.

- **Interpreted :**Java byte code is translated on the fly to native machine instructions and is not stored anywhere.
- **Distributed :**Java is designed for the distributed environment of the internet.

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 2

## Lecture 2

- JVM, JRE, Java Environment
- Java Source File Structure
- Compilation

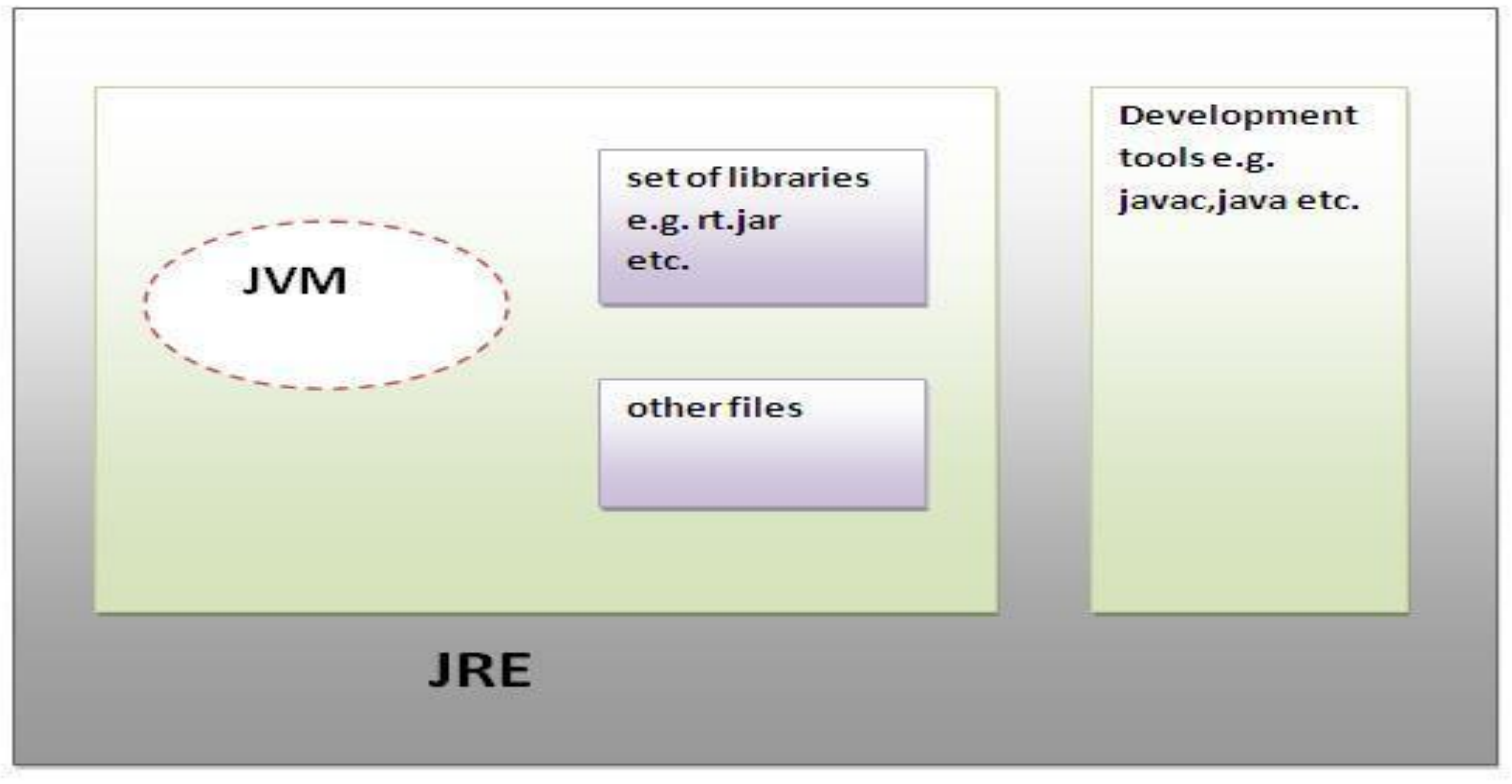
## JDK

The Java Development Kit (JDK) is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications.

It is a core package used in Java, along with the JVM (Java Virtual Machine) and the JRE (Java Runtime Environment).

# JDK

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


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

JDK

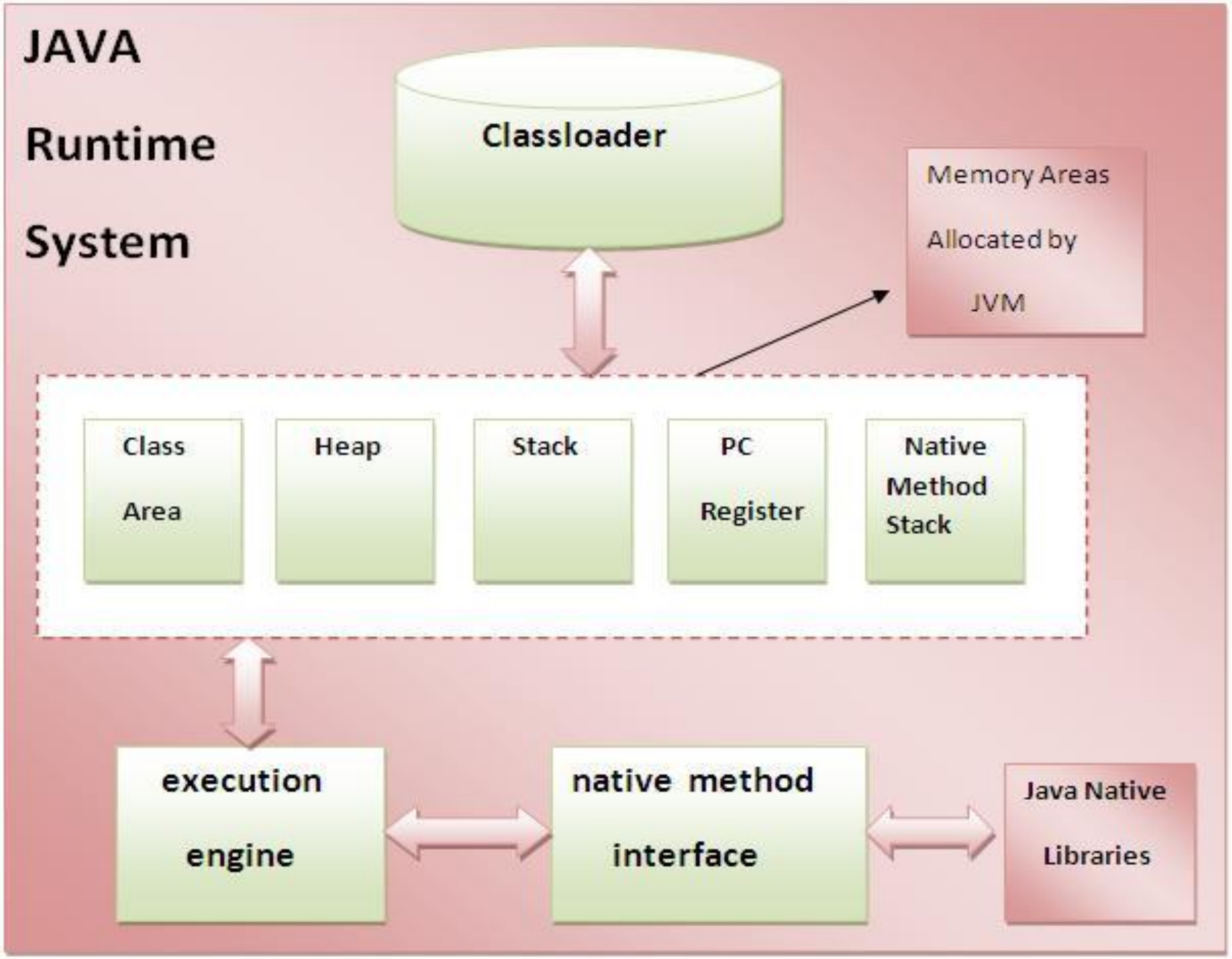
## JVM

- JVM (Java Virtual Machine) is an abstract machine.
- It is a specification that provides runtime environment in which java byte code can be executed.
- JVMs are available for many hardware and software platforms.
- JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

## The JVM performs following main tasks

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

# Internal Architecture of JVM



1) **Classloader**

Classloader is a subsystem of JVM that is used to load class files.

**2) Class(Method) Area**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

**3) Heap**

It is the runtime data area in which objects are allocated.

**4) Stack**

It holds local variables and partial results, and plays a part in method invocation and return.

**5) Program Counter Register**

It contains the address of the Java virtual machine instruction currently being executed.

**6) Native Method Stack**

It contains all the native methods used in the application.

**7) Execution Engine**

It contains:

**i) A virtual processor**

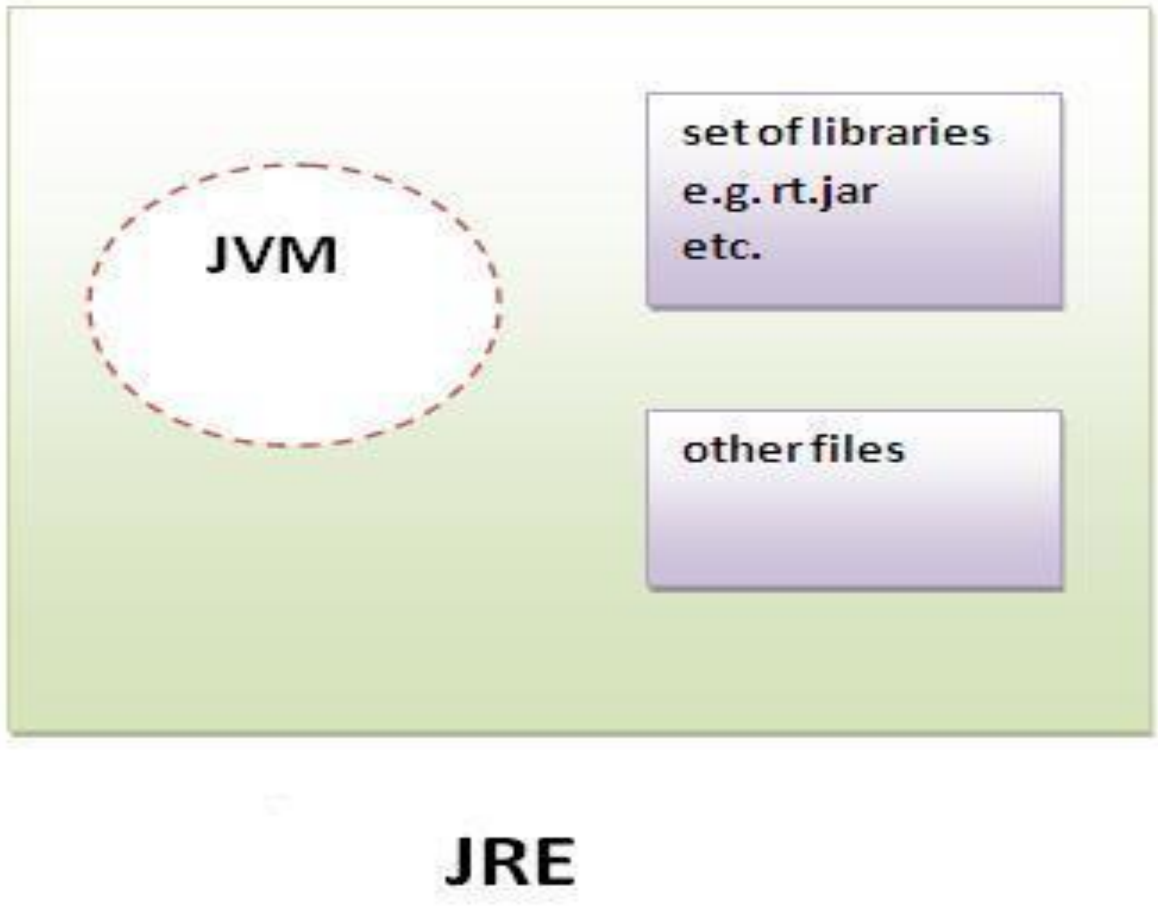
**ii) Interpreter:** Read bytecode stream then execute the instructions.

**iii) Just-In-Time(JIT) compiler:** It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed.

## JRE

- JRE is an acronym for Java Runtime Environment.
- It is used to provide runtime environment. It is the implementation of JVM.
- It physically exists.
- It contains set of libraries + other files that JVM uses at runtime.

## JRE



# Java Source File Structure

- In Java, a source file typically follows a specific structure to define classes, interfaces, and other elements of the program.
- Package Declaration (Optional):

  The package declaration is used to organize related classes and interfaces into a package. It is the first non-comment line in the file and is optional.

  For example: `package com.example.myapp;`

- **Import Statements (Optional):** Import statements are used to bring in classes or entire packages from other packages to use in the current source file. They appear after the package declaration (if present) and before the class declaration.

  For example:

  ```
  import java.util.ArrayList;
  import java.util.List;
  ```

- **Class Declaration:** A Java source file can contain one public class (with the same name as the file) and any number of non-public classes. The class declaration consists of the class keyword followed by the class name and optional modifiers (e.g., public, abstract, final). For example:

- **Interface Declaration (Optional):** Similar to classes, a Java source file can also contain interfaces. The interface declaration consists of the interface keyword followed by the interface name and optional modifiers.

  For example:

  ```
  public interface MyInterface {
  // interface body
  }
  ```

- **Class or Interface Body:** The body of a class or interface contains fields, methods, constructors, and nested classes or interfaces. It is enclosed in curly braces {}.

  For example:

```
public class MyClass {
    private int myField;
    public MyClass(int value) {
        myField = value;
    }
    public void myMethod() {
        System.out.println("Hello, world!");
    }
    // nested class
    private class NestedClass {
        // nested class body
    }
}
```

- **Comments:** Java supports single-line comments (//) and multi-line comments (/* */) for adding explanations or documentation to the code.

## How to Run Java Program

- Write Your Java Program: Create a Java source file with a .java extension.
- For example, let's say you have a simple program called HelloWorld.java

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

- **Compile Your Java Program**:

Open a command prompt and navigate to the directory containing your Java source file.

Use the javac command to compile your program.

```
javac HelloWorld.java
```

- **Run Your Java Program:** Use the java command to run your compiled Java program.

```
java HelloWorld
```

This will execute your HelloWorld class, and you should see the output Hello, world! printed to the console.

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 3

## Lecture 3

- Tokens of Java:
- Operators
- Variables
- Data types
- Comments

## Java Tokens

In Java, **Tokens** are the smallest elements of a program that is meaningful to the compiler. They are also known as the fundamental building blocks of the program.

Tokens can be classified as follows:

- Keywords
- Identifiers
- Constants
- Special Symbols
- Operators
- Comments
- Separators

## 1. Keyword:

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can’t be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed.

| | | |
|---|---|---|
| abstract | assert | boolean |
| break | byte | case |
| catch | char | class |
| const | continue | default |

## 2. Identifiers:

Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore (_) as a first character. Identifier names must differ in spelling and case from any keywords.

### Examples of valid identifiers:

MyVariable

MYVARIABLE

myvariable

x

i

x1

i1

## 3. Constants/Literals:

Constants are also like normal variables. But the only difference is, their values cannot be modified by the program once they are defined.

final data_type variable_name;

## 4. Special Symbols:

The following special symbols are used in Java having some special meaning and thus, cannot be used for some other purpose.

[] () {}, ; * =

## 5. Operators:

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-

- Arithmetic Operators
- Unary Operators
- Assignment Operator
- Relational Operators
- Logical Operators
- Ternary Operator
- Bitwise Operators
- Shift Operators

## 6. Comments:

In Java, Comments are the part of the program which are ignored by the compiler while compiling the Program. They are useful as they can be used to describe the operation or methods in the program.

The Comments are classified as follows:

- Single Line Comments
- Multiline Comments

```
// This is a Single Line Comment
/*
This is a Multiline Comment
*/
```

## 7. Separators:

Separators are used to separate different parts of the codes. It tells the compiler about completion of a statement in the program. The most commonly and frequently used separator in java is semicolon (;).

# Operators in java

**Operator** in java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

## Java Unary Operator Example: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21
 }}
```

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

## Java AND Operator Example: Logical && and Bitwise &

- The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.
- The bitwise & operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not
    checked
System.out.println(a<b&a++<c);//false & true = false
System.out.println(a);//11 because second condition is chec
    ked
}}
```

## Java OR Operator Example: Logical || and Bitwise |

- The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.
- The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is no
    t checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is ch
    ecked
}}
```

## Java Ternary Operator

```
class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

## Java Assignment Operator

```
class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3  a=a+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

## Java Shift Operator Example: Left Shift

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240

}}
```

## Java Shift Operator Example: Right Shift

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

## Java Variable Types

Variable is name of reserved area allocated in memory.

There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/static variables

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

## Local variables :

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

## Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object’s state that must be present through out the class.

## Class/static variables :

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 4

## Lecture 4

- Encapsulation
- Defining Classes in Java
- Control Flow

## Encapsulation

- **Encapsulation** is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- It is a protective shield that prevents the data from being accessed by the code outside this shield.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.

- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

## Advantages of Encapsulation

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.
- **Freedom to programmer in implementing the details of the system.**

## Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especially if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

# Data Encapsulation in Java

```
class Area {
   int length;
   int breadth;
    // constructor to initialize values
   Area(int length, int breadth)
   {
      this.length = length;
      this.breadth = breadth;
   }
    // method to calculate area
   public void getArea()
   {
      int area = length * breadth;
      System.out.println("Area: " + area);
   }
}
```

```
class Main {
    public static void main(String[] args)
    {
        Area rectangle = new Area(2, 16);
        rectangle.getArea();
    }
}
```

## Class Definition in Java

- In object-oriented programming, a class is a basic building block.
- It can be defined as template that describes the data and behavior associated with the class instantiation.
- Instantiating is a class is to create an object (variable) of that class that can be used to access the member variables and methods of the class.
- A class can also be called a logical template to create the objects that share common properties and methods.

- Java provides a reserved keyword class to define a class.
- The keyword must be followed by the class name.
- Inside the class, we declare methods and variables.

## class declaration includes the following in the order as it appears:

- Modifiers: A class can be public or has default access.
- class keyword: The class keyword is used to create a class.
- Class name: The name must begin with an initial letter (capitalized by convention).
- Superclass (if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- Interfaces (if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- Body: The class body surrounded by braces, { }.

```
// class definition
public class Calculate {
   // instance variables
   int a;
   int b;
   // constructor to instantiate
   Calculate (int x, int y) {
      this.a = x;
      this.b = y;
   }
     // method to add numbers
   public int add () {
      int res = a + b;
      return res;
   }
}
```

```
class MyMain
{
public static void main(String[] args)
{
     // creating object of Class
     Calculate c1 = new Calculate(45, 4);
     // calling the methods of Calculate class
     System.out.println("Addition is :" + c1.add());
    }
}
```

## Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear.

However, Java provides statements that can be used to control the flow of Java code.

Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

## Java provides three types of control flow statements.

1. Decision Making statements
    1. if statements
    2. switch statement
2. Loop statements
    1. do while loop
    2. while loop
    3. for loop
    4. for-each loop
3. Jump statements
    1. break statement
    2. continue statement

# Java for loop

```
for(initialization, condition, increment/decrement) {
//block of statements
}
int sum = 0;
for(int j = 1; j<=10; j++)
{
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " +
sum);
```

# Java while loop

```
while(condition){
//looping statements
}
while(i<=10)
{
System.out.println(i);
i = i + 2;
}
```

## Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable.

The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

```
public class Calculation {
public static void main(String[] args)
{
String[] names =
{"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array
names:\n");
for(String name:names) {
System.out.println(name);
} }
}
```

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 5

## Lecture 5

- Arrays
- Strings

## Java Array

- Array is a collection of similar type of elements that have contiguous memory location.
- **Java array** is an object that contains elements of similar data type.
- It is a data structure where we store similar elements.
- We can store only fixed set of elements in a java array.
- Array in java is index based, first element of the array is stored at 0 index.

## Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

## Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

## Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

## Single Dimensional Array in java

Syntax to Declare an Array in java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in java

Array RefVar=**new** datatype[size];

Example

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

## Example of single dimensional java array

```
class Testarray{
public static void main(String args[]){
 int a[]=new int[5];
a[0]=10;
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
for(int i=0;i<a.length;i++)
System.out.println(a[i]);
 }}
```

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

**int** a[]={33,3,4,5};//declaration, instantiation and initialization

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

# Passing Array to method in java

```
class Testarray2{
static void min(int arr[])
{
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
  System.out.println(min);
}
  public static void main(String args[]){
  int a[]={33,3,4,5};
min(a);//passing array to method
  }}
```

## Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java.

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

## Example to instantiate Multidimensional Array in java

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

Example to initialize Multidimensional Array in java

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

## Example of Multidimensional java array

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

```
class Testarray5{
public static void main(String args[]){
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
int c[][]=new int[2][3];
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}}
```

## Example of Multidimensional java array

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

```
class Testarray5{
public static void main(String args[]){
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
int c[][]=new int[2][3];
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}}
```

# Final arrays in Java

```
class Test
{
    public static void main(String args[])
    {
      final int arr[] = {1, 2, 3, 4, 5};  // Note: arr is final
      for (int i = 0; i < arr.length; i++)
      {
         arr[i] = arr[i]*10;
         System.out.println(arr[i]);
      }
    }
}
```

The array *arr* is declared as final, but the elements of array are changed without any problem.

Arrays are objects and object variables are always references in Java.

So, when we declare an object variable as final, it means that the variable cannot be changed to refer to anything else.

## Jagged Array in Java

Jagged array is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D arrays but with variable number of columns in each row. These type of arrays are also known as Jagged arrays.

```
class Main
{
  public static void main(String[] args)
  {
              int arr[][] = new int[2][];
             arr[0] = new int[3];
             arr[1] = new int[2];
             int count = 0;
      for (int i=0; i<arr.length; i++)
         for(int j=0; j<arr[i].length; j++)
            arr[i][j] = count++;
       System.out.println("Contents of 2D Jagged Array");
      for (int i=0; i<arr.length; i++)
      {
         for (int j=0; j<arr[i].length; j++)
            System.out.print(arr[i][j] + " ");
         System.out.println();
      }
   }}
```

## Java String

- **Java String** provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.
- In java, string is basically an object that represents sequence of char values.
- An array of characters works same as java string.

For example:

```
char[] ch={'j','a','v','a'};
String s=new String(ch);
```

is same as:

```
String s="java";
```

- The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.
- The java String is immutable i.e. it cannot be changed but a new instance is created.
- For mutable class, you can use StringBuffer and StringBuilder class.

There are two ways to create String object:

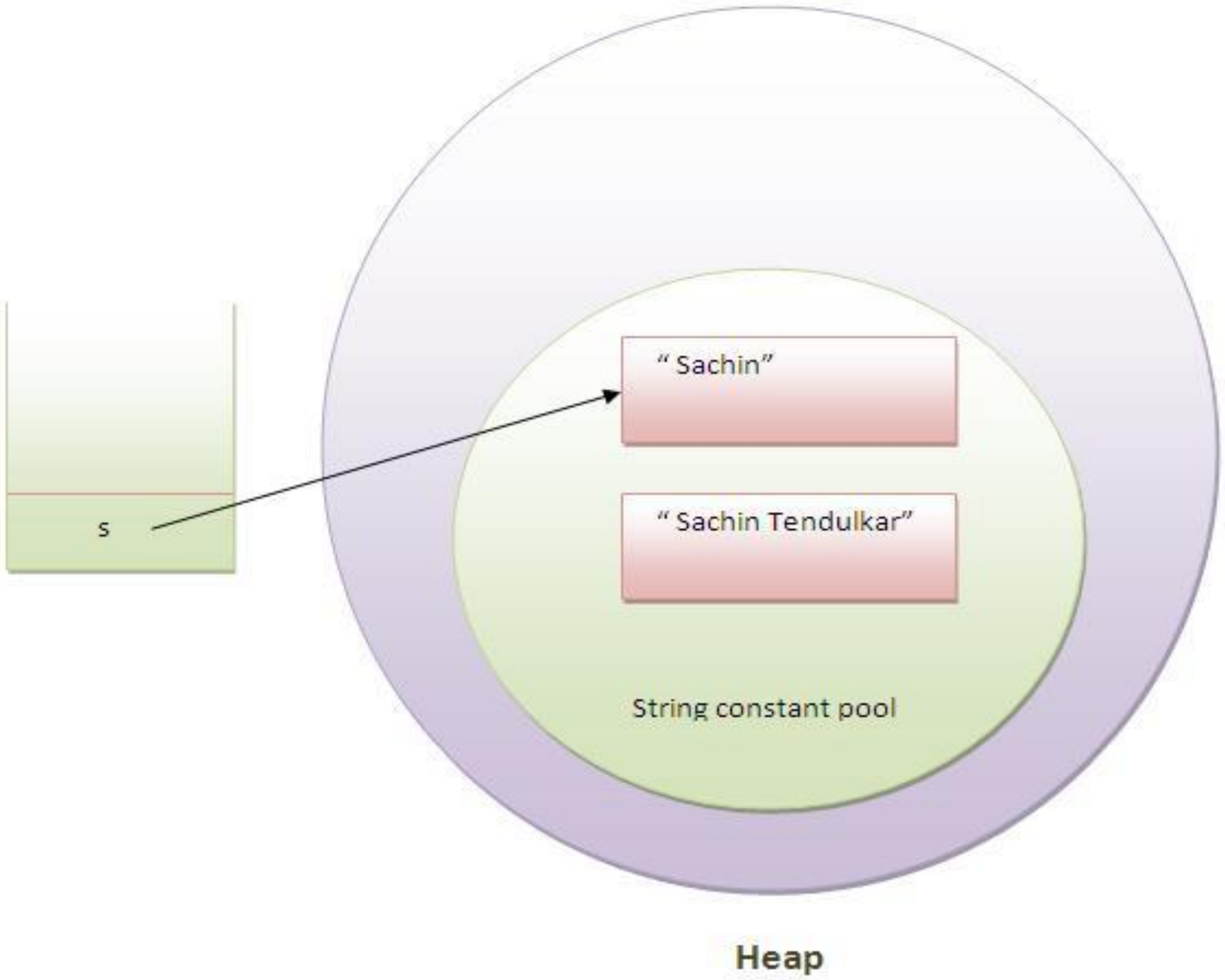
- By string literal
- By new keyword

| Method | Description |
|---|---|
| char charAt(int index) | returns char value for the particular index |
| int length() | returns string length |
| String substring(int beginIndex) | returns substring for given begin index |
| String substring(int beginIndex, int endIndex) | returns substring for given begin index and end index |
| boolean contains(CharSequence s) | returns true or false after matching the sequence of char value |
| boolean equals(Object another) | checks the equality of string with object |
| boolean isEmpty() | checks if string is empty |
| String concat(String str) | concatinates specified string |
| String replace(char old, char new) | replaces all occurrences of specified char value |
| String replace(CharSequence old, CharSequence new) | replaces all occurrences of specified CharSequence |
| String trim() | returns trimmed string omitting leading and trailing spaces |
| String toUpperCase() | returns string in uppercase. |
| String toUpperCase(Locale l) | returns string in uppercase using specified locale. |
| int indexOf(int ch) | returns specified char value index |
| int indexOf(int ch, int fromIndex) | returns specified char value index starting with given index |

# Immutable String in Java

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the stri
    ng at the end
   System.out.println(s);//will print Sachin because strings ar
    e immutable objects
 }
}
```


s
“ Sachin”
“ Sachin Tendulkar”
String constant pool
Heap

## 1) String compare by equals() method

- The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:
- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   String s4="Saurav";
   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
 } }
Output: true      true      false
```

```
class Teststringcomparison2{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="SACHIN";


   System.out.println(s1.equals(s2));//false
  System.out.println(s1.equalsIgnoreCase(s2));//true
 }
}
```

Output: false       true

## 2) String compare by == operator

The = = operator compares references not values.

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);//true
                                   (because both refer to same instance)
  System.out.println(s1==s3);
                        //false(because s3 refers to instance created in non
    pool)
 }
}
```

## 3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

**s1 == s2** :0

**s1 > s2** :positive value

**s1 < s2** :negative value

```
class Teststringcomparison4{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(because s1>s3)
   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 } }
```

```
Output:0      1      -1
```

# String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings.

There are two ways to concat string in java:

By + (string concatenation) operator

By concat() method

1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Sachin"+" Tendulkar";
   System.out.println(s);//Sachin Tendulkar
 } }
```

Output:Sachin Tendulkar

## Java String split

```
public class SplitExample{
public static void main(String args[]){
String s1="java string split method by javatpoint";
String[] words=s1.split(“ ");//splits the string based on whitespace
for(String w:words){
System.out.println(w);
}
}}
```

## Java String endsWith

```
public class EndsWithExample{
public static void main(String args[]){
String s1="java by javatpoint";
System.out.println(s1.endsWith("t"));
System.out.println(s1.endsWith("point"));
}}
```

Output:

```
true true
```

## substring() method

```
public class SubstringExample{
public static void main(String args[]){
String s1="javatpoint";
System.out.println(s1.substring(2,4));//returns va
System.out.println(s1.substring(2));//returns vatpoint
}}
```

# Java String join() method example

```
public class StringJoinExample{
public static void main(String args[]){
String joinString1=String.join("-
   ","welcome","to","javatpoint");
System.out.println(joinString1);
}}
```

```
welcome-to-javatpoint
```

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 6

## Lecture 6

- Class
- Object
- Constructors
- Methods

## Class in Java

- Class is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

## Syntax to declare a class:

```
class <class_name>
{
   data member;
   method;
}
```

## Object in Java

- **Object is an instance of a class.** Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

# Constructor in Java

- **Constructor in java** is a *special type of method* that is used to initialize the object.
- Java constructor is *invoked at the time of object creation*.
- It constructs the values i.e. provides data for the object that is why it is known as constructor.

## Rules for creating java constructor

There are basically two rules defined for the constructor.

- Constructor name must be same as its class name
- Constructor must have no explicit return type

## Types of java constructors

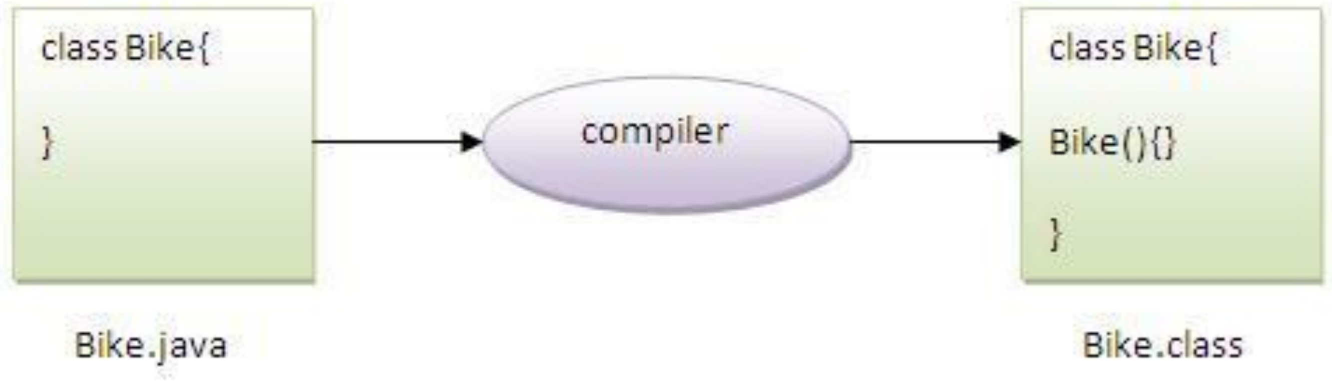
There are two types of constructors:

- Default constructor (no-arg constructor)
- Parameterized constructor

## 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.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}}
```



```
class Bike{

}
```

```
class Bike{

Bike(){}

}
```

**Rule: If there is no constructor in a class, compiler automatically creates a default constructor.**

Example of parameterized constructor

```
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

## Constructor Overloading in Java

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

```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    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,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

## Java Copy Constructor

- There is no copy constructor in java. But, we can copy the values of 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

```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
    id = i;
    name = n;
    }
    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,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

# Methods in Java

Methods of Java is a collection of statements that perform some specific task and return the result to the caller.

1. A method is like a function i.e. used to expose the behavior of an object.
2. It is a set of codes that perform a particular task.

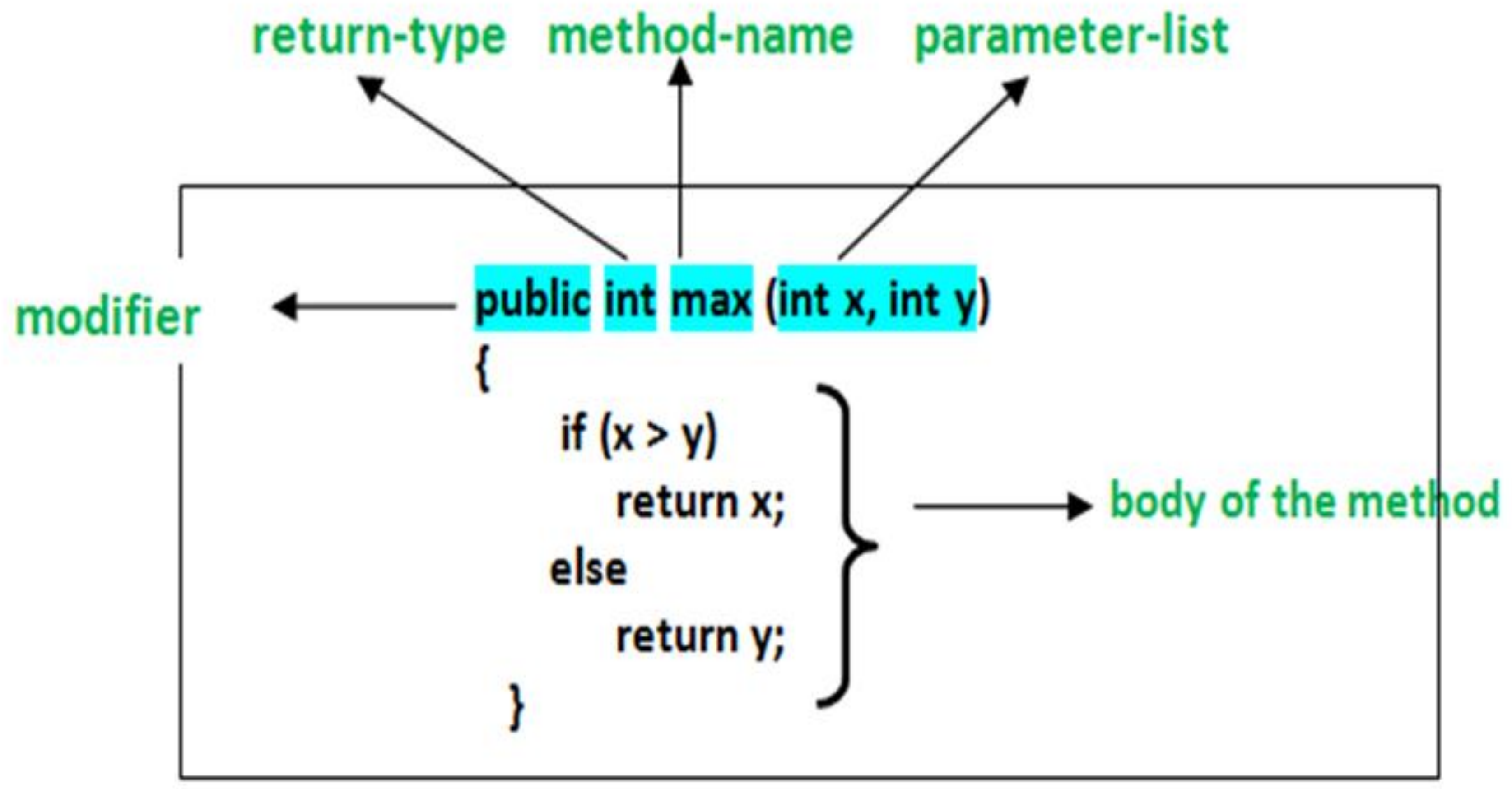
## Syntax of Method

```
<access_modifier> <return_type> <method_name>(
list_of_parameters)
{
   //body
}
```

## Advantage of Method

- Code Reusability
- Code Optimization

# Method Declaration



```
public int max (int x, int y)
{
      if (x > y)
            return x;
      else
            return y;
 }
```

## Method Declaration

In general, method declarations have 6 components:

1.Modifier: It defines the access type of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.

- public: It is accessible in all classes in your application.
- protected: It is accessible within the class in which it is defined and in its subclass/es
- private: It is accessible only within the class in which it is defined.
- default: It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

Note: It is Optional in syntax.

2. The return type: The data type of the value returned by the method or void if does not return a value. It is Mandatory in syntax.

3. Method Name: the rules for field names apply to method names as well, but the convention is a little different. It is Mandatory in syntax.

4. Parameter list: Comma-separated list of the input parameters is defined, preceded by their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses (). It is Optional in syntax.

5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s). It is Optional in syntax.

6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations. It is Optional in syntax.

## Example

```
class Addition {
    // Initially taking sum as 0
    // as we have not started computation
    int sum = 0;
      // Method
    // To add two numbers
    public int addTwoInt(int a, int b)
    {
          // Adding two integer value
        sum = a + b;

        // Returning summation of two values
        return sum;
    }
}
```

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 7

## Lecture 7

- **Static Member**
- **Final Member**

## Static Keyword in Java

- The static keyword in Java is mainly used for memory management.
- The static keyword in Java is used to share the same variable or method of a given class.
- We can apply static keywords with variables, methods, blocks, and nested classes.
- The static keyword belongs to the class than an instance of the class.

**The *static* keyword is a non-access modifier in Java that is applicable for the following:**

1. Blocks
2. Variables
3. Methods
4. Classes

***Note:*** *To create a static member(block, variable, method, nested class), precede its declaration with the keyword static.*

## Characteristics of static keyword:

- **Shared memory allocation**: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.
- **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.

- **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.
- **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.

- **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

```
class Test
{
   // static method
   static void m1()
   {
      System.out.println("from m1");
   }
   public static void main(String[] args)
   {
        // calling m1 without creating
        // any object of class Test
         m1();
   }
}
```

## Static blocks

- If you need to do the 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.

```
class Test
{
    // static variable
    static int a = 10;
    static int b;
    // static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
     public static void main(String[] args)
    {
      System.out.println("from main");
      System.out.println("Value of a : "+a);
      System.out.println("Value of b : "+b);
    }
}
```

## Static variables

- When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level.
- Static variables are, essentially, global variables.
- All instances of the class share the same static variable.

## Important points for static variables:

- We can create static variables at the class level only.
- static block and static variables are executed in the order they are present in a program.

## Static methods

When a method is declared with the static keyword, it is known as the static method. The most common example of a static method is the main( ) method.

Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.

## Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

## Static Classes

- A class can be made static only if it is a nested class.
- We cannot declare a top-level class with a static modifier but can declare nested classes as static.
- Such types of classes are called Nested static classes.

```
class OuterClass
{
      private static String msg = "ABES Engineering College";
      public static class NestedStaticClass
       {
          public void printMessage()
          {
                    System.out.println("Message " + msg);
          }
       }
}
```

```
class MyMain {
            public static void main(String args[])
            {
OuterClass.NestedStaticClass printer= new
OuterClass.NestedStaticClass();
                        printer.printMessage();
            }
}
```

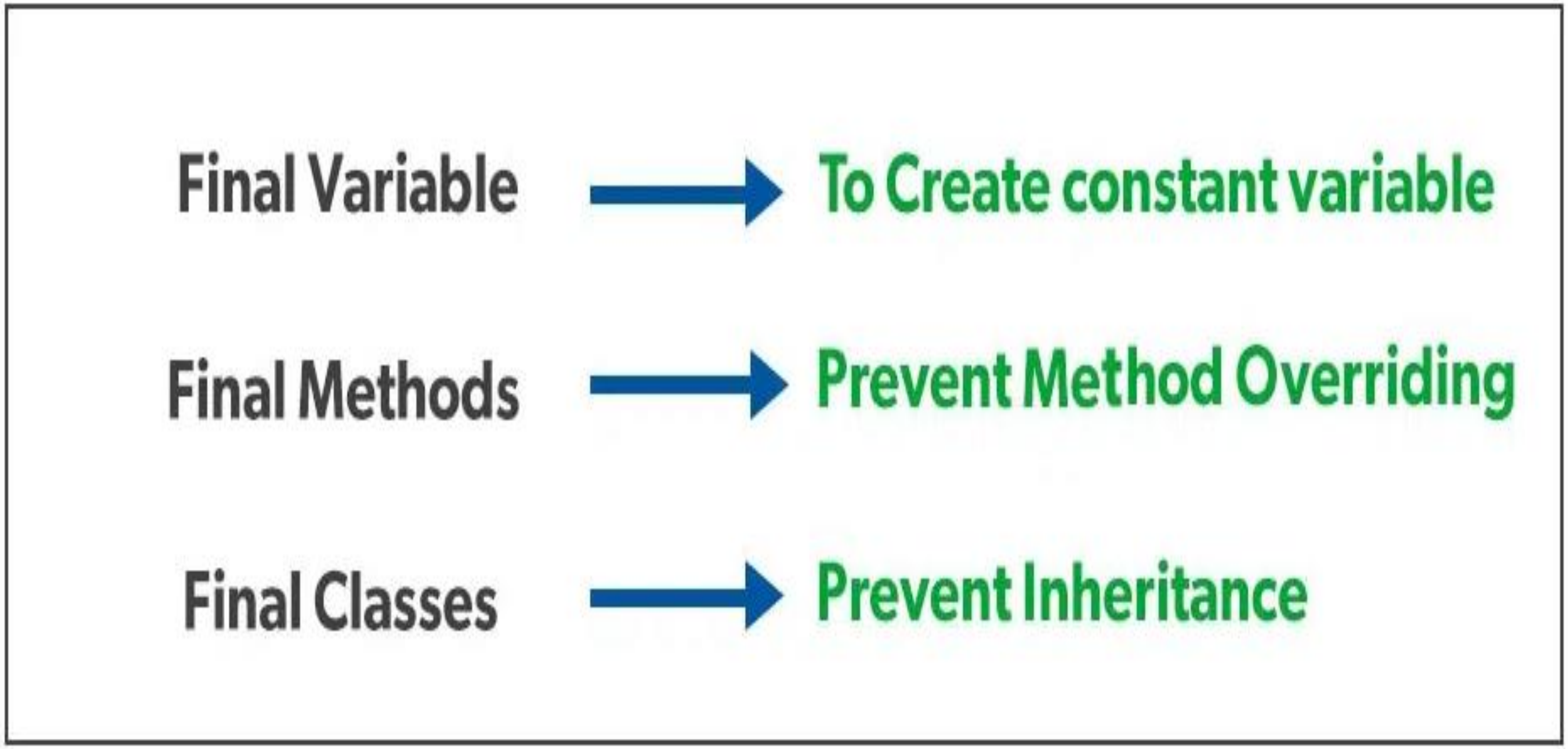
## Final Keyword In Java

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

Final can be:

- variable
- method
- class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.
- It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only.


Final Variable
To Create constant variable
Final Methods
Prevent Method Overriding
Final Classes
Prevent Inheritance

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

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

Output:Compile Time Error

## Java final method

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

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

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

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

Output:Compile Time Error

## Java final class

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

```
final class Bike{}

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

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

Output:Compile Time Error

- A final variable that is not initialized at the time of declaration is known as blank final variable.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful.

## Can we initialize blank final variable?

Yes, but only in constructor. For example:

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

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

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

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 8

## Lecture 8

- Inheritance
- Super Class
- Sub Class
- Access Specifies

## Inheritance in Java

- Inheritance is an important pillar of OOP(Object-Oriented Programming).
- It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.
- In Java, Inheritance means creating new classes based on existing ones.
- A class that inherits from another class can reuse the methods and fields of that class.
- In addition, you can add new fields and methods to your current class as well.

## Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

## Syntax of Java Inheritance

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

The **extends keyword** indicates that you are making a new class that derives from an existing class.

## Super Class/Parent Class:

The class whose features are inherited is known as a superclass(or a base class or a parent class).

## Sub Class/Child Class:

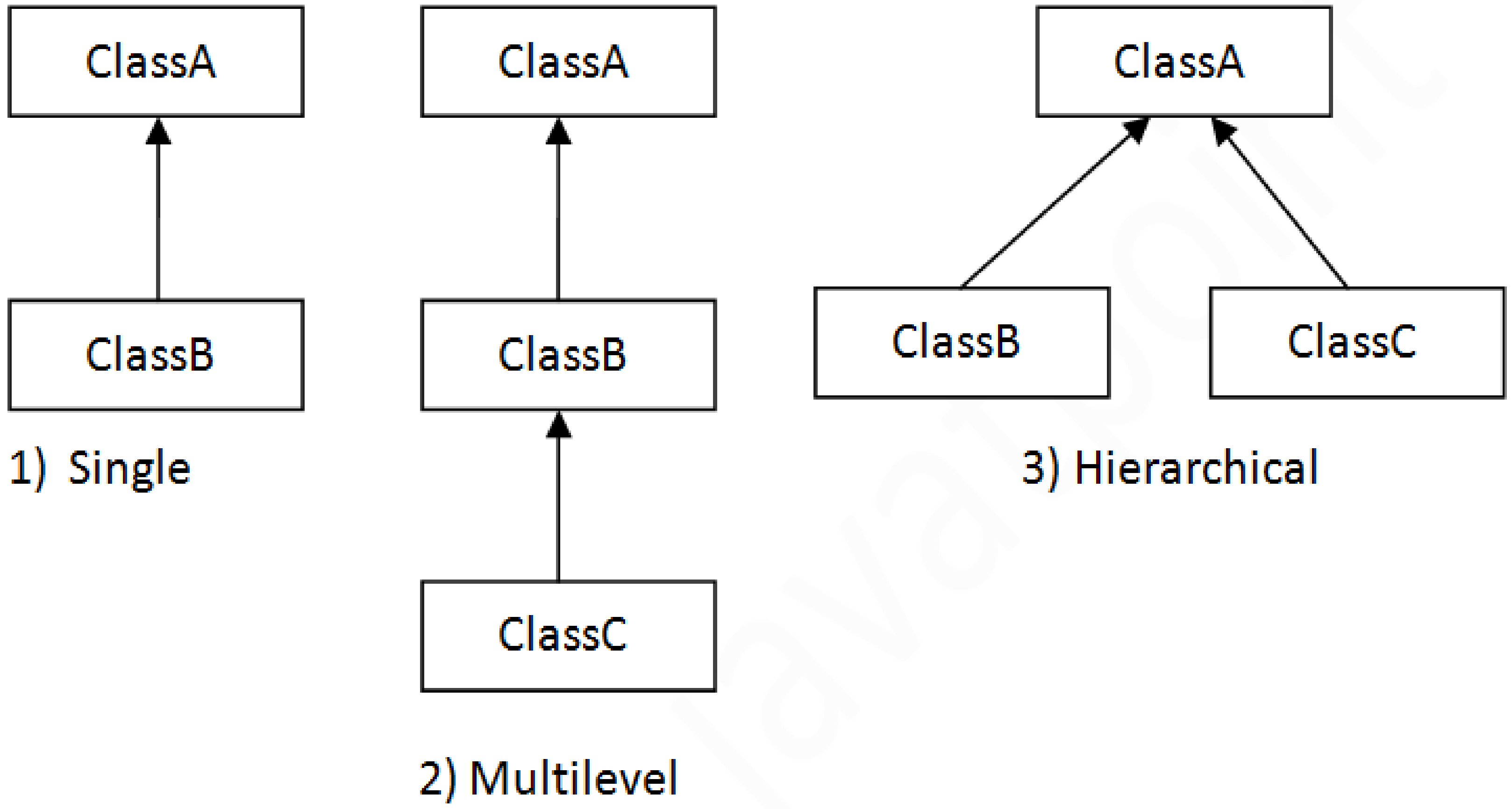
The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

```
import java.io.*;
// Base or Super Class
class Employee {
    int salary = 60000;
}
// Inherited or Sub Class
class Engineer extends Employee {
    int benefits = 10000;
}
```

```
class MyMain {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary
                         + "Benefits : " + E1.benefits);
    }
}
```

```
Salary : 60000
Benefits : 10000
```

# Types of inheritance in java



## Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 Public Static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

**Compile Time Error**

## Java Modifier Types

Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

- Java Access Modifiers
- Non Access Modifiers

## Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors.

The four access levels are:

- Visible to the package. the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

## Private access modifier

The private access modifier is accessible only within class.

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

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

- If you make any class constructor private, you cannot create the instance of that class from outside the class.

**Note: A class cannot be private or protected except nested class.**

## default access modifier

- If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

```
package pack;
class A{
 void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
 public static void main(String args[]){
  A obj = new A();//Compile Time Error
  obj.msg();//Compile Time Error
 }
```

## protected access modifier

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

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

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

## public access modifier

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

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

## Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 9

## Lecture 9

- Polymorphism
- Overriding
- Overloading

## Polymorphism

- Polymorphism is considered one of the important features of Object-Oriented Programming.
- Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
- The word “poly” means many and “morphs” means forms, So it means many forms.

## Types of Java Polymorphism

In Java Polymorphism is mainly divided into two types

- Compile-time Polymorphism
- Runtime Polymorphism

## Compile-Time Polymorphism in Java

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

***Note:*** *Java doesn’t support the Operator Overloading.*

## Method Overloading

When there are multiple functions with the same name but different parameters then these functions are said to be overloaded.

Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

```
class  Overloading{
   // Method with 2 integer parameters
   static int Multiply(int a, int b)
   {
      // Returns product of integer numbers
      return a * b;
   }
    // Method 2
   // With same name but with 2 double parameters
   static double Multiply(double a, double b)
   {
      // Returns product of double numbers
      return a * b;
   }
}
```

# Java Method Overloading

```
class OverloadingExample
{
static int add(int a,int b)
{
return a+b;
}
static int add(int a,int b,int c)
{
return a+b+c;
}
}
```

## Runtime Polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

## Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.
- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism.

## Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

```
class Vehicle{
 void run(){System.out.println("Vehicle is runnin
   g");}
}
class Bike extends Vehicle{  void run(){System.o
   ut.println(“Bike  is running");}


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

| No. | Method Overloading | Method Overriding |
|---|---|---|
| 1) | *Method overloading is used* to increase the readability *of the program.* | *Method overriding is used* to provide the specific implementation *of the method that is already provided by its super class.* |
| 2) | *Method overloading is performed* within class. | *Method overriding occurs* in two classes *that have IS-A (inheritance) relationship.* |
| 3) | *In case of method overloading,* parameter must be different. | *In case of method overriding,* parameter must be same. |
| 4) | *Method overloading is the example of* compile time polymorphism. | *Method overriding is the example of* run time polymorphism. |
| 5) | *In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.* | *Return type must be same or covariant in method overriding.* |

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 10

## Lecture 10

- **Abstraction**
- **Interfaces**
- **Abstract Class**

# Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

## Example to understand Abstraction:

Television remote control is an excellent example of abstraction. It simplifies the interaction with a TV by hiding the complexity behind simple buttons and symbols, making it easy without needing to understand the technical details of how the TV functions.

## Abstract class in Java

- A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

## **Ways to achieve Abstaction**

There are two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

## Abstract class in Java

- A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

**Example**

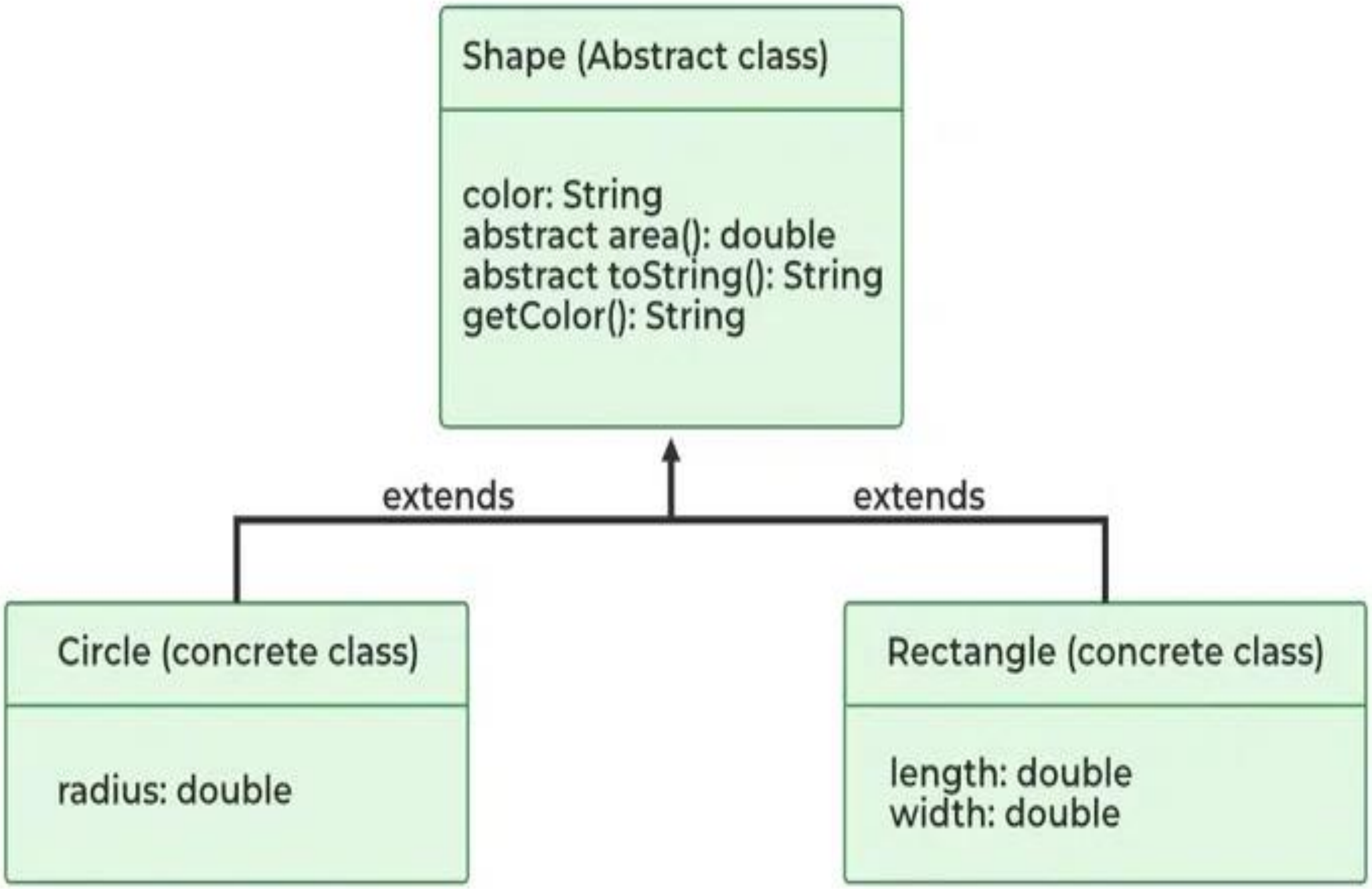
**abstract class** A{}

## Abstract method

- A method that is declared as abstract and does not have implementation is known as abstract method.

**Example**

**abstract void** printStatus();//no body and abstract


Shape (Abstract class)
color: String
abstract area(): double
abstract toString(): String
getColor(): String
extends
extends
Circle (concrete class)
radius: double
Rectangle (concrete class)
length: double
width: double

## Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
abstract class Bike{
 abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely..");}

public static void main(String args[]){
 Honda4 obj = new Honda4();
 obj.run();
}
}
```

```
abstract class Bank{
abstract double getRateOfInterest();
}

class SBI extends Bank{
double getRateOfInterest(){return 7.5;}
}
class PNB extends Bank{
double getRateOfInterest(){return 7;}
}

class TestBank{
public static void main(String args[]){
SBI b=new SBI();//if object is PNB, method of PNB will be invoked
int interest=b.getRateOfInterest();
System.out.println("Rate of Interest is: "+interest+" %");
}}
```

- Rate of Interest is: 7.5 %

An abstract class can have data member, abstract method, method body, constructor and even main() method.

*File: TestAbstraction2.java*

```
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }  }
```

- **Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.**
- **Rule: If there is any abstract method in a class, that class must be abstract.**

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

```
interface A{
void a();
void b();
void c();
void d();
}

abstract class B implements A{
public void c(){System.out.println("I am C");}
}
```

```
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

class Test5{
public static void main(String args[]){
M a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

## Interface in Java

- An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.
- The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.
- Java Interface also **represents IS-A relationship**.
- It cannot be instantiated just like abstract class.

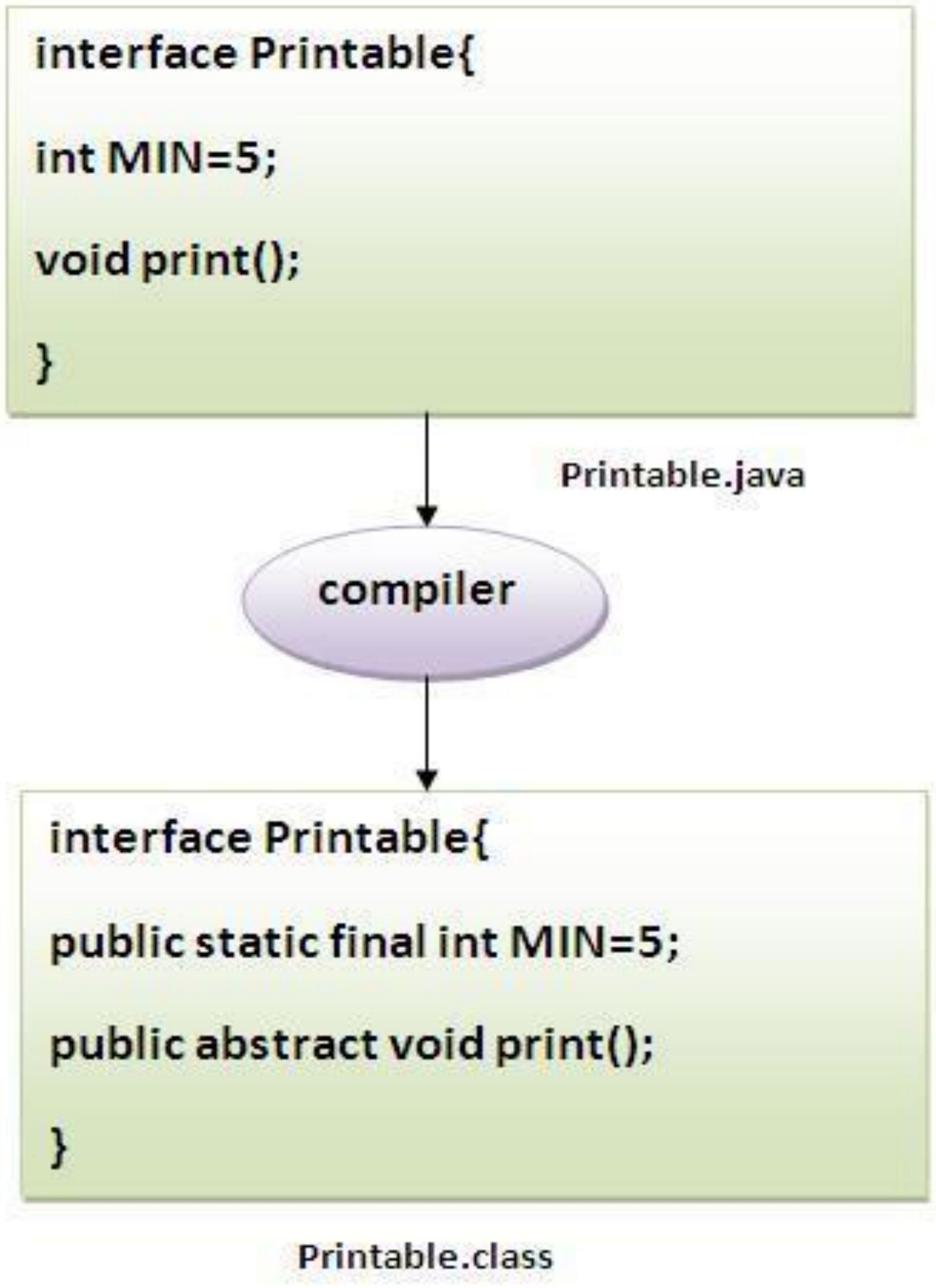
## Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

**The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.**

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



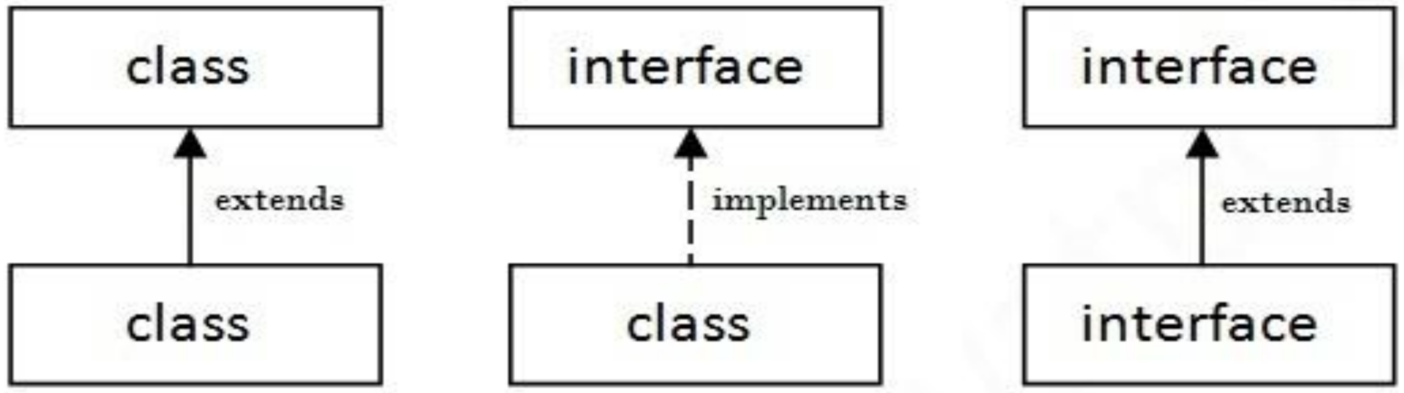
```
interface Printable{

int MIN=5;

void print();

}
```

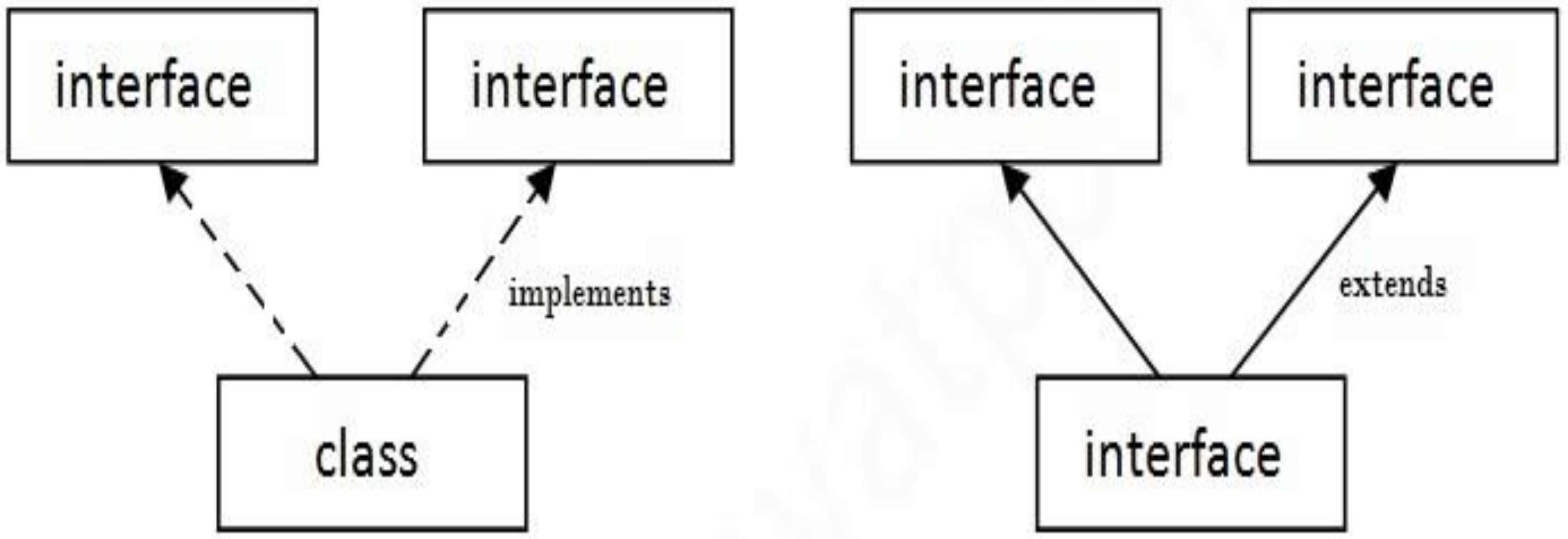
```
interface Printable{

public static final int MIN=5;

public abstract void print();

}
```

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

## Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.





Multiple Inheritance in Java

```
interface Printable{
void print();
}

interface Showable{
void show();
}

class A7 implements Printable,Showable{

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

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

## Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{

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

public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 } }
```

| Abstract class | Interface |
|---|---|
| 1) Abstract class can **have abstract and non-abstract**methods. | Interface can have **only abstract** methods. |
| 2) Abstract class **doesn't support multiple inheritance.** | Interface **supports multiple inheritance.** |
| 3) Abstract class **can have final, non-final, static and non-static variables.** | Interface has **only static and final variables.** |
| 4) Abstract class **can have static methods, main method and constructor.** | Interface **can't have static methods, main method or constructor.** |
| 5) Abstract class **can provide the implementation of interface.** | Interface **can't provide the implementation of abstract class.** |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| ) **Example:**<br>public abstract class Shape{<br>public abstract void draw();<br>} | **Example:**<br>public interface Drawable{<br>void draw();<br>} |

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 11

## Lecture 11

- Defining Package
- CLASSPATH Setting for Packages

# Java Package

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces.

Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cs.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier

- Providing controlled access: protected and default have package level access control.
- A protected member is accessible by classes in the same package and its subclasses.
- A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

## Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

## How packages work

Package names and directory structure are closely related.

For example if a package name is college.staff.cs, then there are three directories, college, staff and cs such that cs is present in staff and staff is present inside college.

Also, the directory college is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate

## Sub packages

Packages that are inside another package are the **sub packages**.

These are not imported by default, they have to imported explicitly.

Also, members of a sub package have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example :

import java.util.*;

util is a subpackage created inside java package.

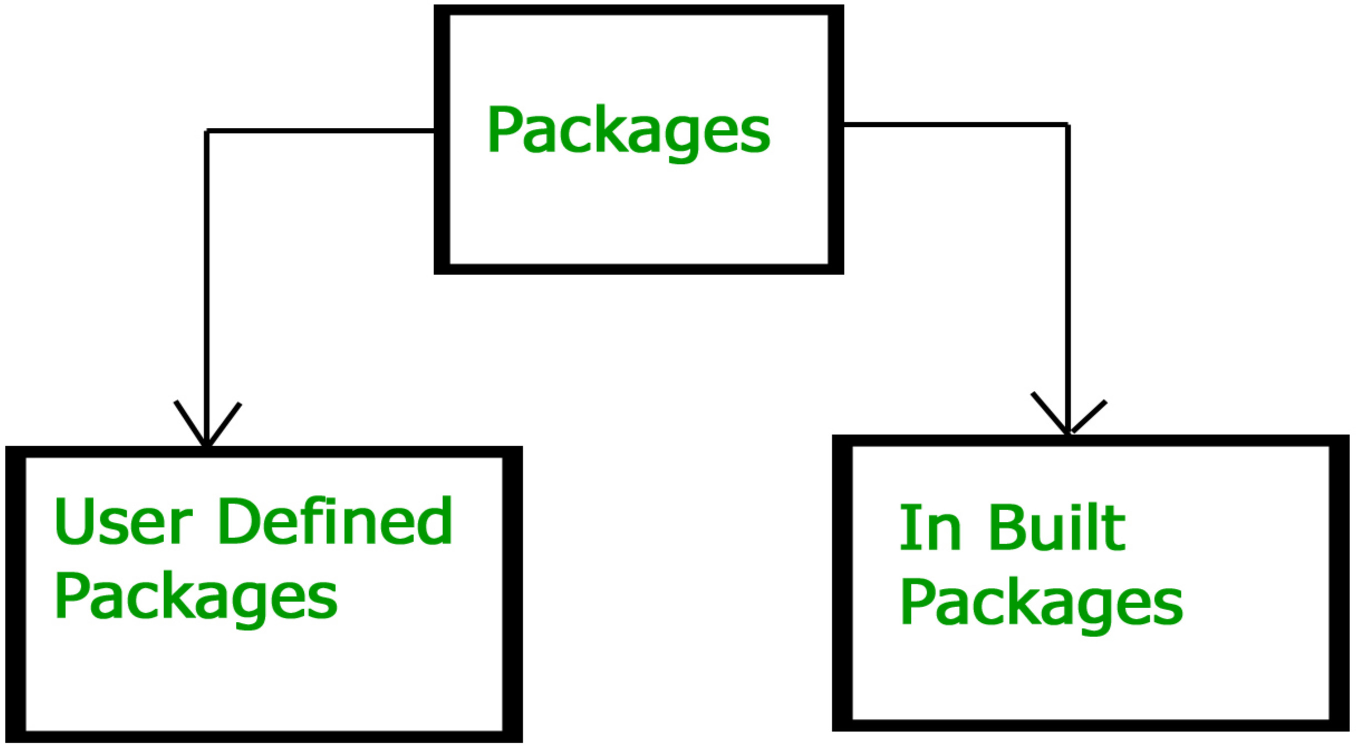
## Accessing classes inside a package

import the Vector class from util package.

```
import java.util.vector;
```

```
// import all the classes from util package
import java.util.*;
```

# Types of Packages



## Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1) java.lang: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) java.io: Contains classed for supporting input / output operations.

3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) java.applet: Contains classes for creating Applets.

5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) java.net: Contain classes for supporting networking operations.

## User-defined packages

These are the packages that are defined by the user.

First we create a directory myPackage (name should be same as the name of the package).

Then create the MyClass inside the directory with the first statement being the package names.

```
// Name of the package must be same as the
directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

```
import myPackage.MyClass;
public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = “ABES Engineering College";
      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();
      obj.getNames(name);
   }
}
```

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

- import package.*;
- import package.classname;
- fully qualified name.

## 1) Using packagename.*

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

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

## 2) Using packagename.classname

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

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

## 3) Using fully qualified name

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

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

## Setting CLASSPATH

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment: In Windows.
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
- > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar

# Object Oriented Programming with Java (Subject Code: BCS-403)

## Unit 1

## Lecture 12

## Lecture 12

- Import and Static Import
- Naming Convention for Packages
- Making JAR Files for Library Packages

## Static import in Java

- In Java, static import concept is introduced in 1.5 version.
- With the help of static import, we can access the static members of a class directly without class name or any object.
- For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly.

## With Static import

```
import static java.lang.Math.*;
class Test2 {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

## Ambiguity in static import

If two static members of the same name are imported from multiple different classes, the compiler will throw an error, as it will not be able to determine which member to use in the absence of class name qualification

## Ambiguity in case of static import

```
package MyPackage;
import static java.lang.Integer.*;
import static java.lang.Byte.*;
public class MyMain {
    public static void main(String[] args)
    {
        System.out.println(MAX_VALUE);
    }
}
```

Error: Reference to MAX_VALUE is ambiguous

## Difference between import and static import:

- With the help of import, we are able to access classes and interfaces which are present in any package. But using static import, we can access all the static members (variables and methods) of a class directly without explicitly calling class name.
- The main difference is Readability, ClassName.dataMember (System.out) is less readable when compared to dataMember(out)
- static import can make your program more readable

## Naming Conventions

- Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
- Companies use their reversed Internet domain name to begin their package names.

For example, com.example.mypackage for a package named mypackage created by a programmer at example.com.

- Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name.

for example, com.example.region.mypackage).

## Making Jar Files

- In Java, JAR stands for Java Archive, whose format is based on the zip format.
- The JAR files format is mainly used to aggregate a collection of files into a single one.
- It is a single cross-platform archive format that handles images, audio, and class files. With the existing applet code, it is backward-compatible.
- In Java, Jar files are completely written in the Java programming language.
- We can either download the JAR files from the browser or can write our own JAR files using Eclipse IDE.