CS23312 OBJECT ORIENTED PROGRAMMING

COURSE OBJECTIVES

- To introduce the foundational concepts of object-oriented programming including classes, objects, and inheritance in Java.
- To expose exception handling, Java streams, and Java-Doc comments for code documentation.
- To emphasize the concepts of generics and multi-threading in Java, including synchronization techniques and the use of executors and synchronizers.
- To give insight into Java networking and JDBC, including socket programming, RMI, and database connectivity.
- To train in GUI programming using Swing and JavaFX, including the implementation of the Model-View-Controller design pattern.

SYLLABUS

UNIT I PARADIGMS & BASIC CONSTRUCTS 9

- Object oriented programming concepts – objects – classes – methods and messages – abstraction and encapsulation – inheritance – abstract classes – polymorphism.- Objects and classes in Java – defining classes – methods - access specifiers – static members – constructors – finalize method

UNIT II EXCEPTION HANDLING & STREAMS 9

- Arrays – Strings - Packages – Java-Doc comments –- Inheritance – class hierarchy – polymorphism – dynamic binding – final keyword – abstract classes-Exception handling – exception hierarchy – throwing and catching exceptions-The Object class – Reflection – interfaces – object cloning – inner classes – proxies - I/O Streams

UNIT III GENERICS & MULTI THREADING 9

- Motivation for generic programming – generic classes – generic methods – generic code and virtual machine – inheritance and generics – reflection and generics - Multi threaded programming – interrupting threads – thread states – thread properties – thread synchronization – Executors – synchronizers.

UNIT IV JAVA NETWORKING & JDBC 9

- Socket programming in Java-InetAddress and URL classes-TCP and UDP protocols in Java-ServerSocket and Socket classes-Multi-threaded servers-Handling multiple client connections-Introduction to RMI-Creating RMI servers and clients-RMI registry-RMI and object serialization-Overview of JDBC-JDBC drivers-Connecting to databases Executing SQL queries

UNIT V GUI PROGRAMMING 9

- Introduction to Swing – Model-View-Controller design pattern – layout management – Swing Components -Introduction to JavaFX - JavaFX components.

COURSE OUTCOMES

At the end of the course, students will be able to

- CO1: Implement Java classes and objects, abstraction, encapsulation, inheritance, and polymorphism by applying object-oriented programming principles.
- CO2: Handle exceptions effectively in Java programs, Java streams, and Java-Doc comments for effective documentation.
- CO3: Execute generic programming techniques, multi-threaded programming, and synchronization mechanisms in Java applications.
- CO4: Develop Java applications for network communication using socket programming, RMI, and JDBC, including the execution of SQL queries for database operations.
- CO5: Establish interactive graphical user interfaces using Swing and JavaFX, separation of concerns, and efficient layout management by applying the Model-View-Controller design pattern.

What is Java?

- Java is a **programming language** and a **platform**.
- Java is a high level, robust, object-oriented and secure programming language.
- Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995.
- *James Gosling* is known as the father of Java.
- Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.
- **Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Application

- According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:
- Desktop Applications such as acrobat reader, media player, antivirus, etc.
- Web Applications such as irctc.co.in etc.
- Enterprise Applications such as banking applications.
- Mobile
- Embedded System
- Smart Card
- Robotics
- Games, etc.

Types of Java Applications

- There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

   – Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

   – An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

   – An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

   – An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

- There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

   – It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

   – It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3) Java ME (Java Micro Edition)

   – It is a micro platform that is dedicated to mobile applications.

4) JavaFX

   – It is used to develop rich internet applications. It uses a lightweight user interface API.

History of Java

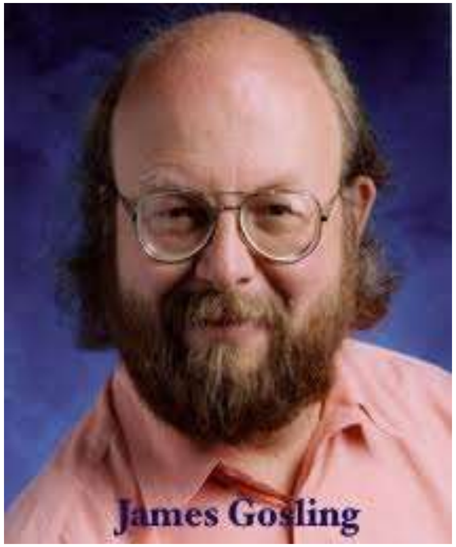
- The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.
- The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.
- The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".
- Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

1) James Gosling, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

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

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

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



Why Java was named as "Oak"?

5) **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.

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

Why Java Programming named "Java"?

7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

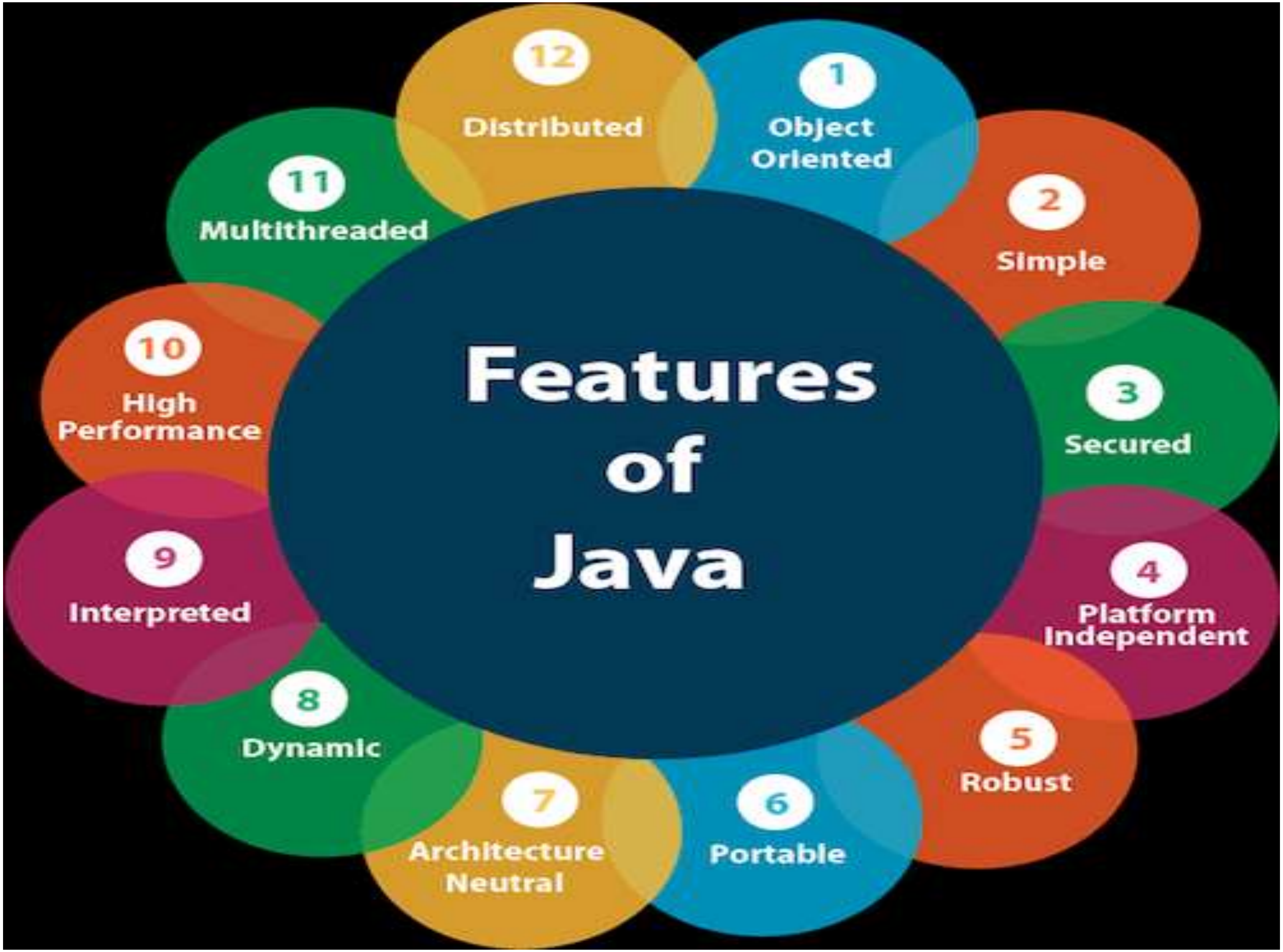
12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10. 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)
- Java SE 11 (September 2018)
- Java SE 12 (March 2019)
- Java SE 13 (September 2019)
- Java SE 14 (Mar 2020)
- Java SE 15 (September 2020)
- Java SE 16 (Mar 2021)
- Java SE 17 (September 2021)
- Java SE 18 (to be released by March 2022)
- Since Java SE 8 release, the Oracle corporation follows a pattern in which every even version is release in March month and an odd version released in September month.

Features of Java



OOPs (Object-Oriented Programming System)

- **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

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.

Class

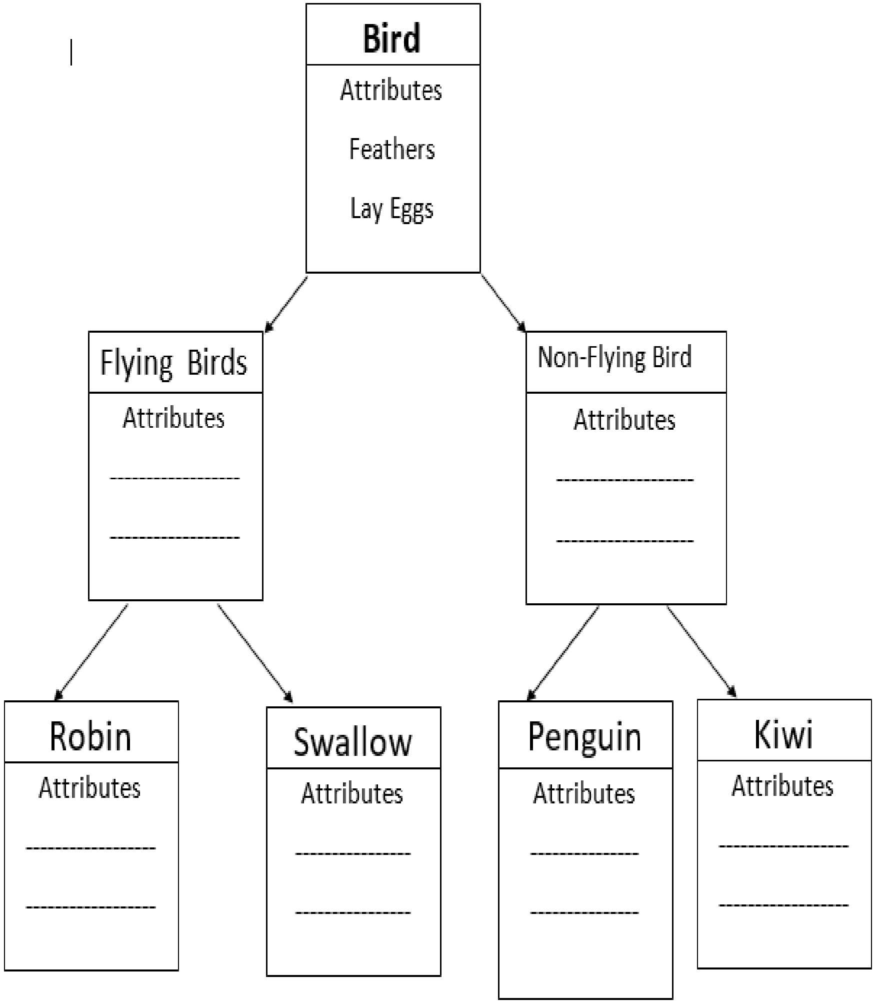
- *Collection of objects* is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class doesn't consume any space.
- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
  - **Fields**
  - **Methods**
  - **Constructors**
  - **Blocks**
  - **Nested class and interface**

- Syntax to declare a class:

```
class <class_name>
{
   field;
   method;
}
```

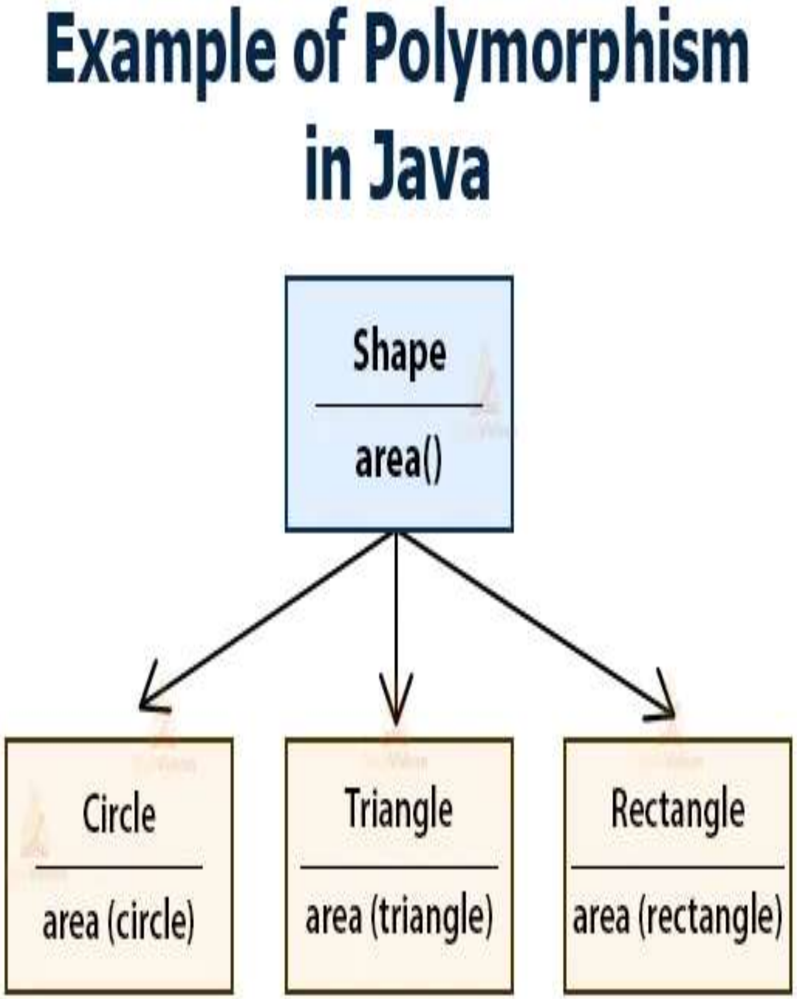
Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.



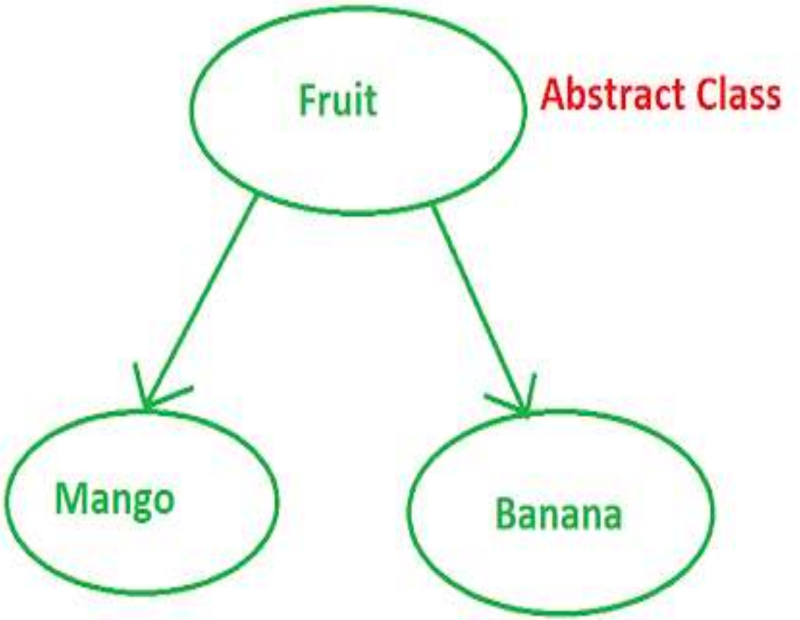
Polymorphism

- 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.



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



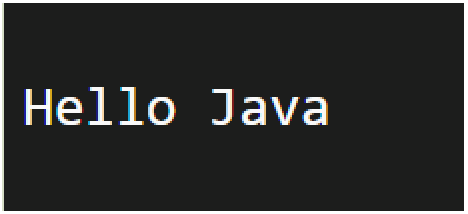
Encapsulation

- *Binding (or wrapping) code and data together into a single unit 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.

First Java Program | Hello World Example

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

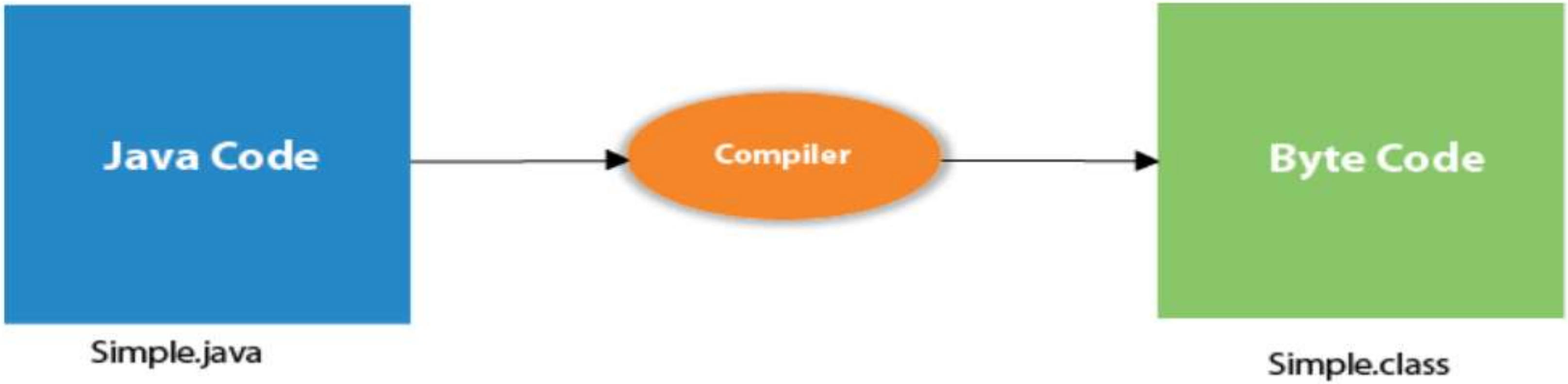
- **To compile:**
  - javac Simple.java
- **To execute:**
  - java Simple
- **output**



```
Hello Java
```

Compilation Flow:

- When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



Parameters used in First Java Program
Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

Java If-else Statement

- The Java *if statement* is used to test the condition.
- It checks boolean condition: *true* or *false*.
- There are various types of if statement in Java.
  - if statement
  - if-else statement
  - if-else-if ladder
  - nested if statement

Java if Statement

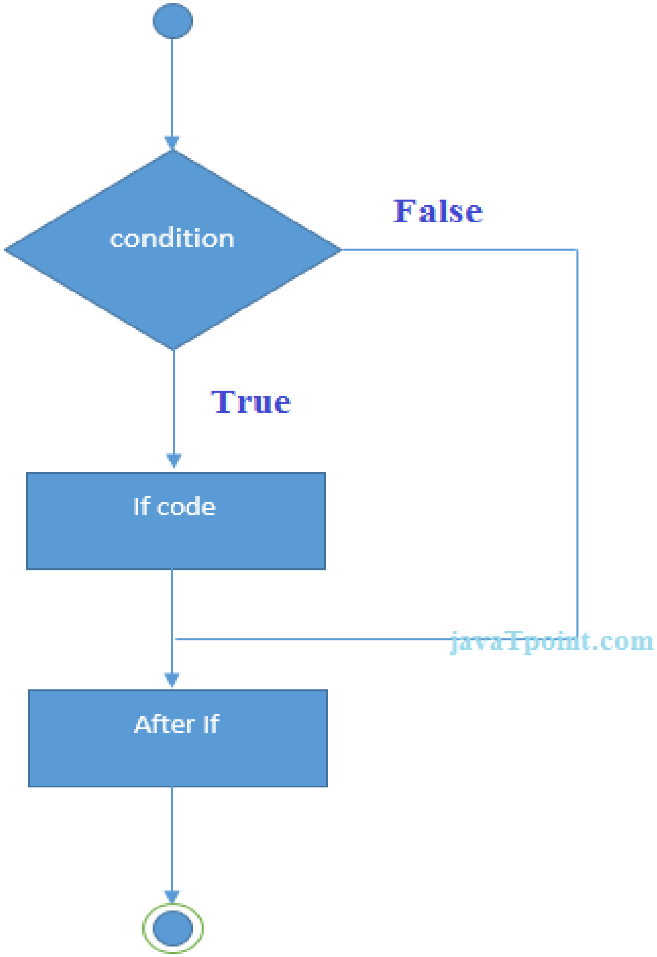
- The Java if statement tests the condition. It executes the *if block* if condition is true.
- **Syntax:**

if(condition)

{

//code to be executed

}



//Java Program to demonstate the use of if statement.

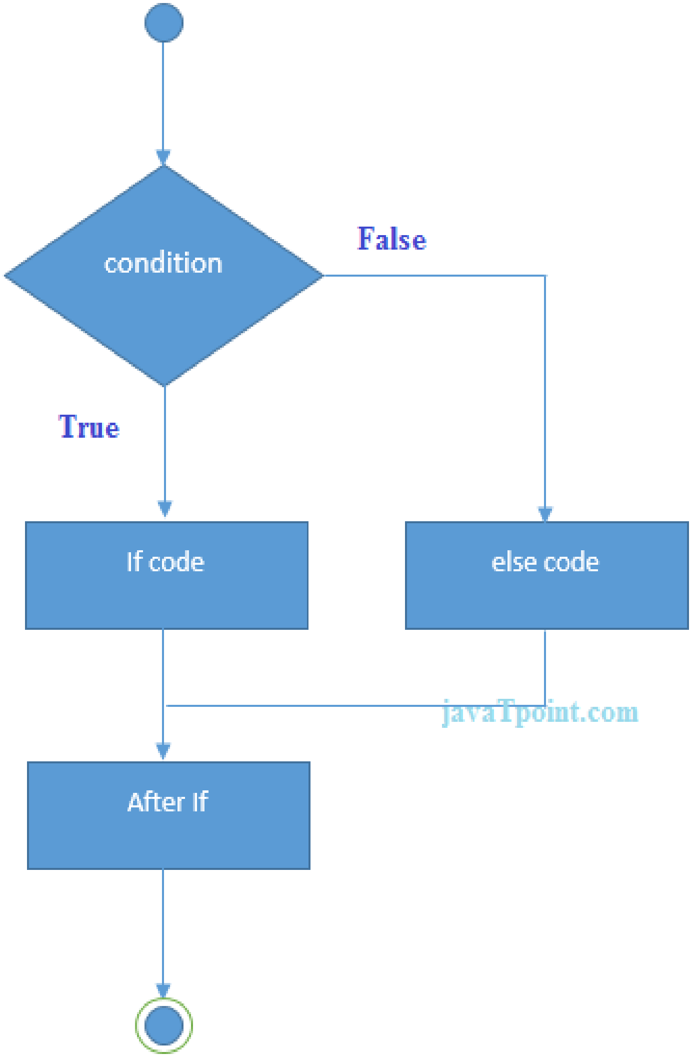
```
public class IfExample
{
public static void main(String[] args)
   {
  //defining an 'age' variable
  int age=20;
  //checking the age
  if(age>18)
   {
     System.out.print("Age is greater than 18");
  }
}
```

- Output:
  - Age is greater than 18

Java if-else Statement

- The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.
- **Syntax:**

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



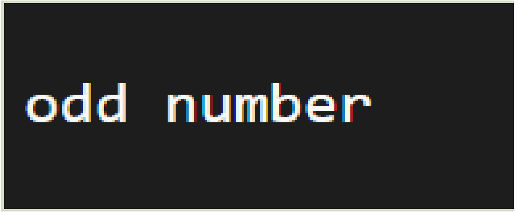
Example:

//A Java Program to demonstrate the use of if-else statement.

//It is a program of odd and even number.

```
public class IfElseExample {
public static void main(String[] args) {
   //defining a variable
   int number=13;
   //Check if the number is divisible by 2 or not
   if(number%2==0)
    {
      System.out.println("even number");
     }
    else
    {
      System.out.println("odd number");
     }
}
}
```

Output:



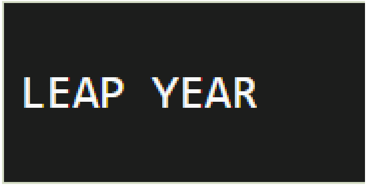
```
odd number
```

Leap Year Example

A year is leap, if it is divisible by 4 and 400. But, not by 100.:

```
public class LeapYearExample
{
public static void main(String[] args)
{
   int year=2020;
   if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0))
    {
      System.out.println("LEAP YEAR");
    }
   else
    {
      System.out.println("COMMON YEAR");
     }
}
}
```

output



```
LEAP YEAR
```

Java if-else-if ladder Statement

- The if-else-if ladder statement executes one condition from multiple statements.
- **Syntax:**

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions
 are false
}
```

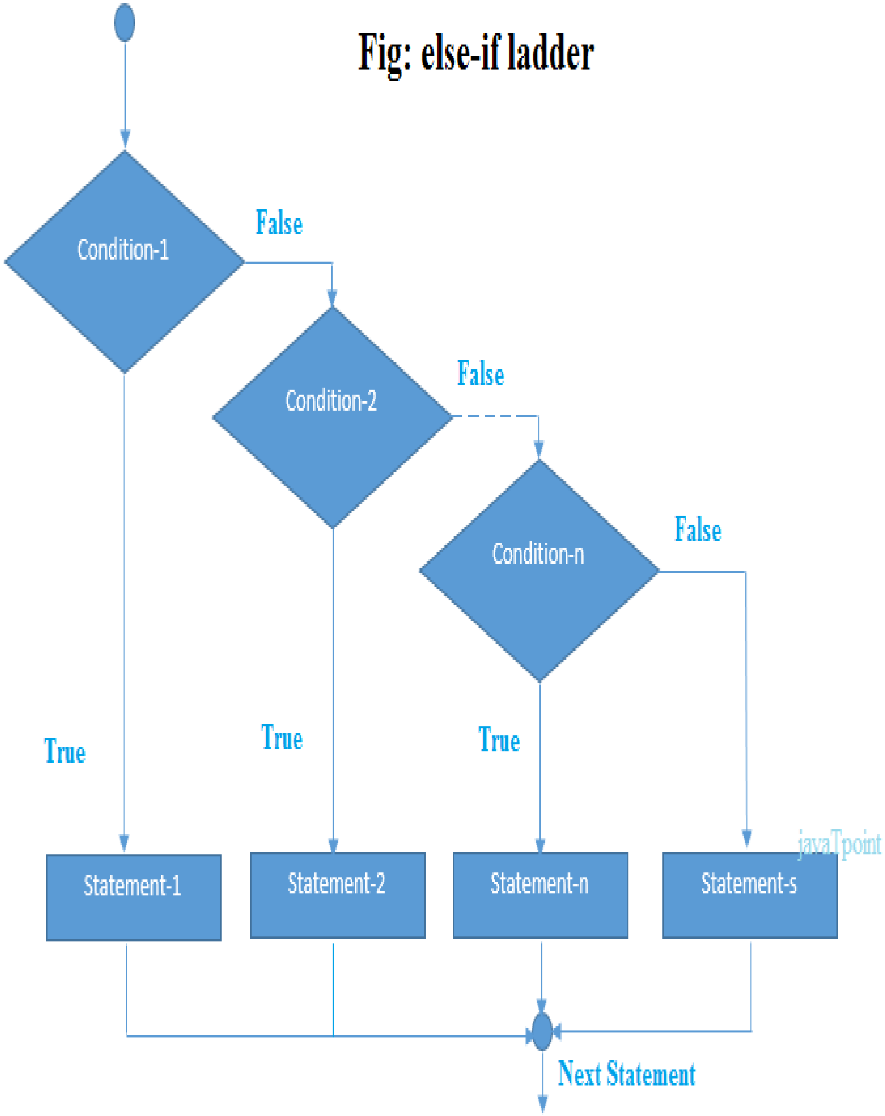


Fig: else-if ladder

//Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+

```
public class IfElseIfExample
 {
public static void main(String[] args) {
   int marks=65;

   if(marks<50)
     {
      System.out.println("fail");
      }
   else if(marks>=50 && marks<60)
     {
      System.out.println("D grade");
     }
   else if(marks>=60 && marks<70)
     {
      System.out.println("C grade");
     }
else if(marks>=70 && marks<80)
      {
       System.out.println("B grade");
      }
else if(marks>=80 && marks<90)
      {
       System.out.println("A grade");
      }
else if(marks>=90 && marks<100)
      {
       System.out.println("A+ grade");
      }
Else
      {
       System.out.println("Invalid!");
      }
}  }
```

```
Output:  C grade
```

Program to check POSITIVE, NEGATIVE or ZERO:

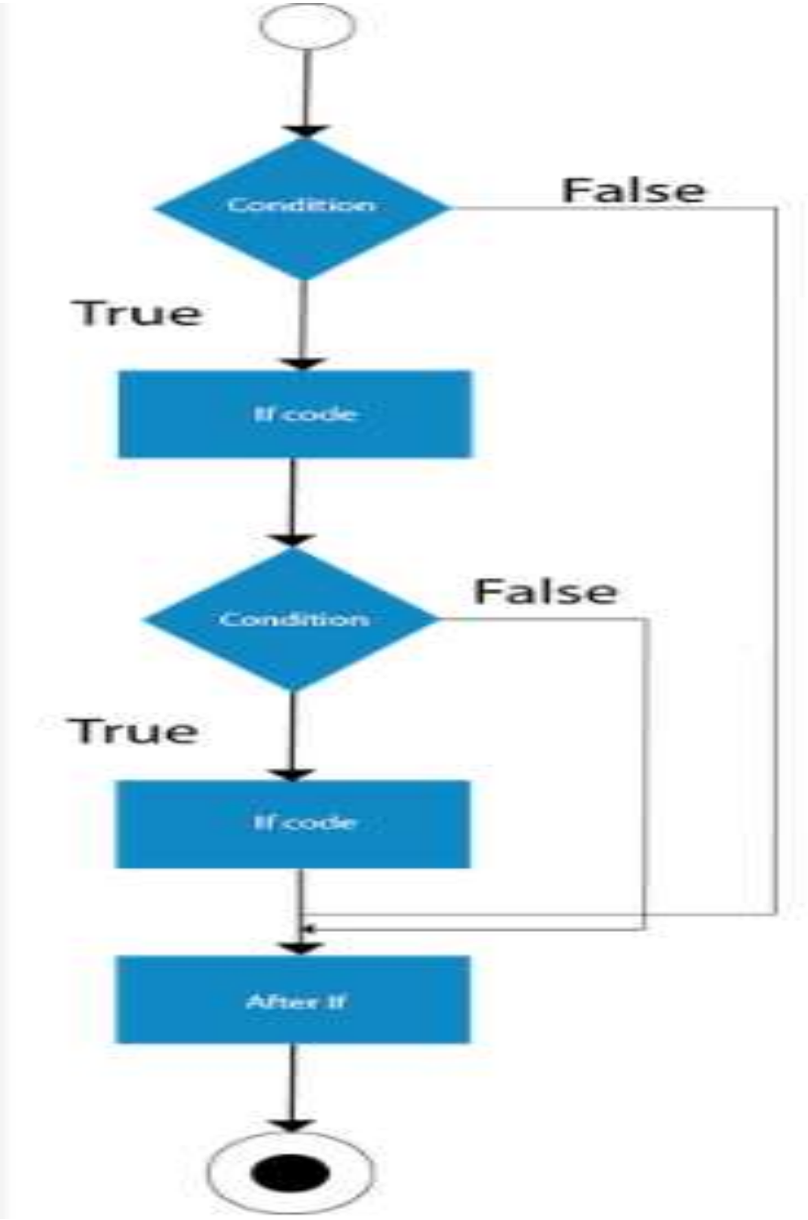
```
public class PositiveNegativeExample {
public static void main(String[] args) {
    int number=-13;
    if(number>0)
      {
    System.out.println("POSITIVE");
      }
   else if(number<0)
      {
    System.out.println("NEGATIVE");
      }
   else
      {
    System.out.println("ZERO");
      }
}
}
```

- Output: NEGATIVE

Java Nested if statement

- The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.
- **Syntax:**

```
if(condition){
    //code to be executed
        if(condition){
          //code to be executed
    }
 }
```



//Java Program to demonstrate the use of Nested If Statement.

```
public class JavaNestedIfExample {
public static void main(String[] args) {
   //Creating two variables for age and weight
   int age=20;
   int weight=80;
   //applying condition on age and weight
   if(age>=18)
    {
      if(weight>50)
           {
         System.out.println("You are eligible to donate blood");
           }
    }
}}
Output:  You are eligible to donate blood
```

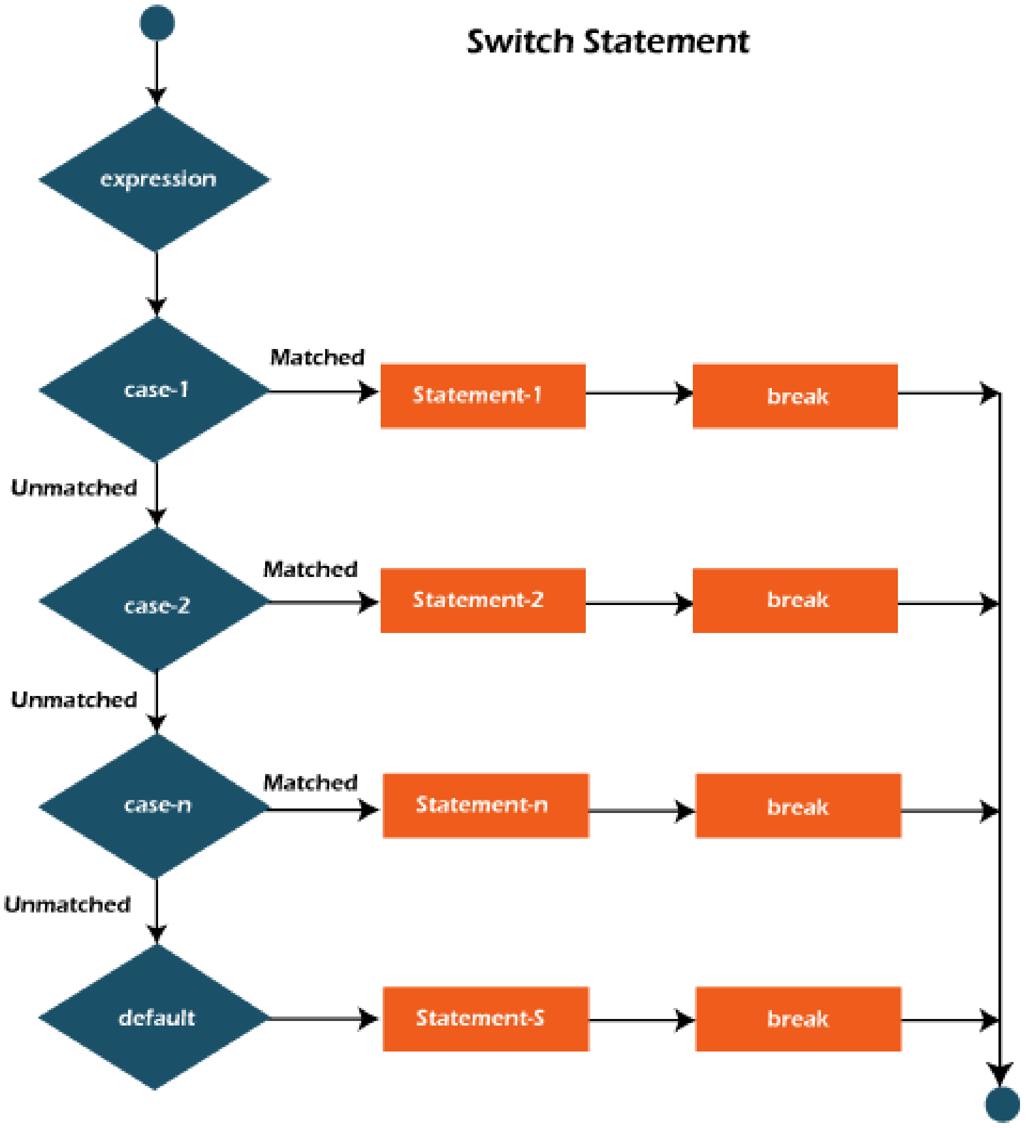
Java Switch Statement

- the Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.
- In other words, the switch statement tests the equality of a variable against multiple values.
- Points to Remember
  - There can be *one or N number of case values* for a switch expression.
  - The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
  - The case values must be *unique*. In case of duplicate value, it renders compile-time error.
  - The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums*
  - *and string*.
  - Each case statement can have a *break statement* which is optional. When control reaches to the break statement , it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
  - The case value can have a *default label* which is optional.

Syntax: Switch Statement

switch(expression){
case value1:
 //code to be executed;
 break; //optional
case value2:
 //code to be executed;
 break; //optional
......

default:
code to be executed **if** all cases are not matched;
}



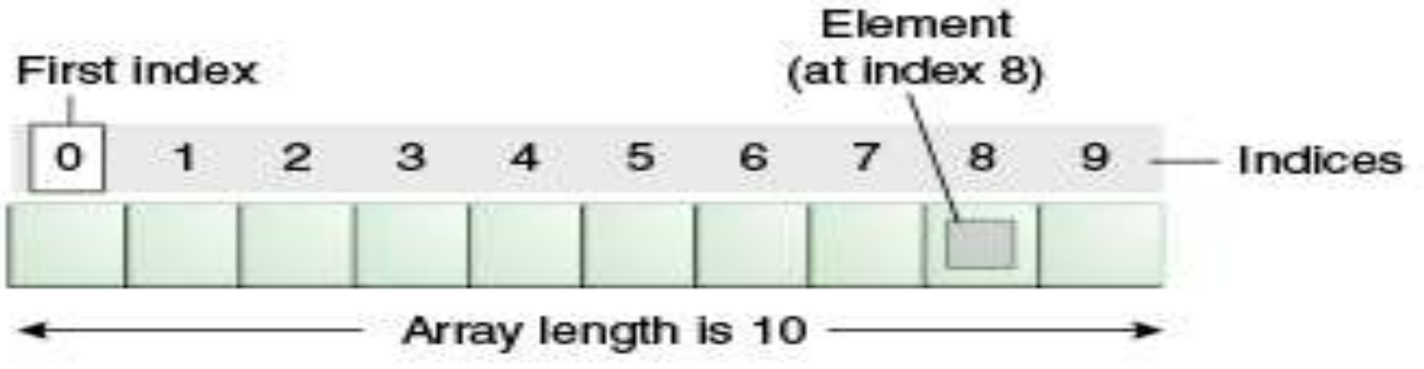
SwitchExample.java

```
public class SwitchExample {
public static void main(String[] args)
    {
  //Declaring a variable for switch
   expression
  int number=20;
  //Switch expression
  switch(number){
  //Case statements
  case 10: System.out.println("10");

  break;
case 20: System.out.println("20");

  break;
  case 30: System.out.println("30
   ");
  break;
  //Default case statement
  default:System.out.println("Not
    in 10, 20 or 30");
  }
}
}
```

- **Output:** 20

Java Arrays

- Normally, an array is a collection of similar type of elements which has contiguous memory location.
- **Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
- Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.



Java Arrays

Advantages

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

Disadvantages

- **Size Limit:** We can store only the 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 which grows automatically.

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

  ```
  arrayRefVar=new datatype[size];
  ```

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

- 10 20 70 40 50

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]);
}}
```

Output:

- 33 3 4 5

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();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Constructor in java

- In Java, a **constructor** is a special method used to initialize objects.
- It is called automatically when an object of a class is created.
- Constructors are primarily used to set up the initial state of an object by assigning values to its attributes or performing other setup tasks.

Key Features of Constructors:

- **Same Name as the Class**: A constructor must have the same name as the class it belongs to.
- **No Return Type**: Constructors do not have a return type, not even void.
- **Called Automatically**: They are invoked automatically when an object is created using the new keyword.

Types of Constructors:

- Default Constructor
- Parameterized Constructor

Types of Constructors:

- Default Constructor
- Parameterized Constructor

Default Constructor

- A constructor with no parameters.
- If no constructor is explicitly defined, Java provides a default constructor.

```
class Example {
    int value;

    // Default Constructor
    Example() {
        value = 0;
        System.out.println("Default Constructor Called");
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example(); // Default constructor is
     called
    }
}
```

Parameterized Constructor

- A constructor that accepts parameters to initialize an object with specific values.

```
class Example
{
    int value;

    // Parameterized Constructor
    Example(int val) {
        value = val;
        System.out.println("Value is: " + value);
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example(42); // Passes 42 to the constructor
    }
}
```

Constructor Overloading

- Multiple constructors can be defined in a class with different parameter lists.

```
class Example {
    int value;

    // Default Constructor
    Example() {
        value = 0;
    }

    // Parameterized Constructor
    Example(int val) {
        value = val;
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj1 = new Example();     // Calls default constructor
        Example obj2 = new Example(100);  // Calls parameterized constructor
    }
}
```

Java static keyword

- The **static keyword** in Java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.
- The static can be:
  - Variable (also known as a class variable)
  - Method (also known as a class method)
  - Block

Java static variable

- If you declare any variable as static, it is known as a static variable
- The static variable can be used to refer to the common property of all objects
    - which is not unique for each object, for example, the company name of employees, college name of students, etc.
    - The static variable gets memory only once in the class area at the time of class loading.

Understanding the problem without static variable

- Advantages of static variable
  - It makes your program **memory efficient** (i.e., it saves memory).

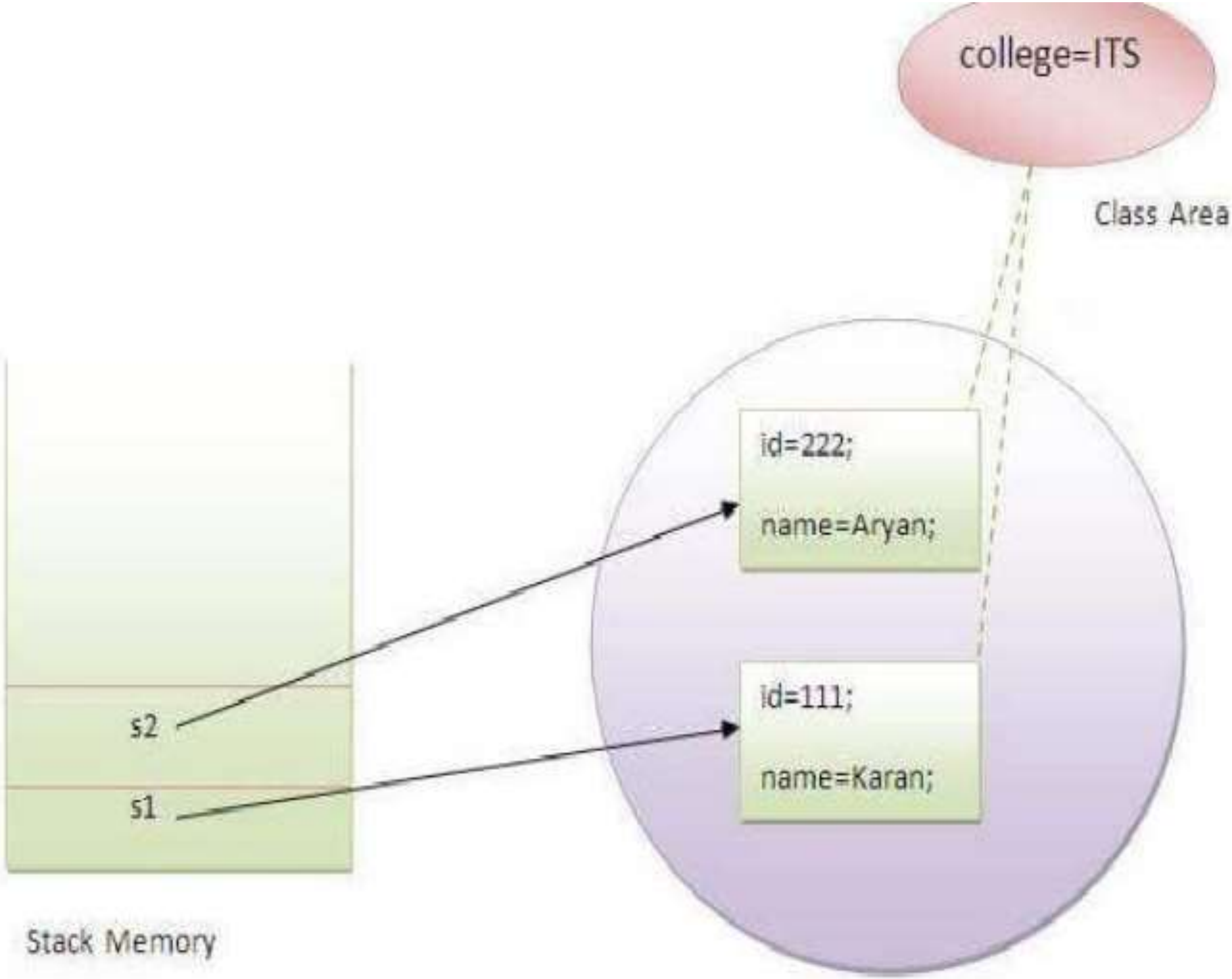
```
class Student
{
    int rollno;
    String name;
    String college="ITS";
}
```

```
class Student
{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n)
    {
   rollno = r;
   name = n;
    }
   //method to display the values
   void display ()
     {
      System.out.println(rollno+" "+name+" "+college);
     }
}
```

```
public class TestStaticVariable1
{
 public static void main(String args[])
{
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

- Output

```
111 Karan ITS
222 Aryan ITS
```


college=ITS
Class Area
id=222;
name=Aryan;
id=111;
name=Karan;
s2
s1
Stack Memory

Static Methods

- A static method can be called without creating an object of the class.
- It can only access static variables and other static methods directly.

Static Methods example

```
class Example {
    static void displayMessage() {
        System.out.println("This is a static method.");
    }

    public static void main(String[] args) {
        Example.displayMessage(); // No object needed
    }
}
```

Static Blocks

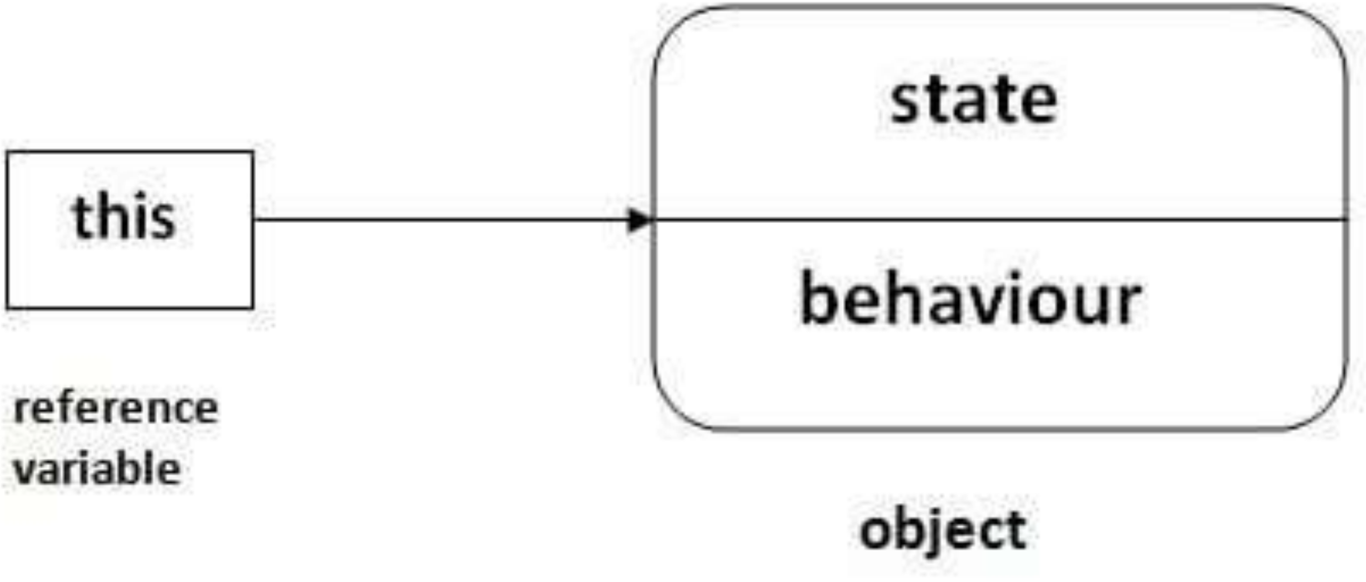
- A static block is used to initialize static variables.
- It runs once when the class is loaded into memory.

Static Blocks example

```
class Example {
    static int value;

    static {
        value = 42; // Static block
        System.out.println("Static block executed.");
    }

    public static void main(String[] args) {
        System.out.println("Value: " + value); // Output: Value: 42
    }
}
```

this keyword in java

- There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object



Usage of java this keyword

- Here is given the 6 usage of java this keyword.
  - this can be used to refer current class instance variable.
  - this can be used to invoke current class method (implicitly)
  - this() can be used to invoke current class constructor.
  - this can be passed as an argument in the method call.
  - this can be passed as argument in the constructor call.
  - this can be used to return the current class instance from the method

this: to refer current class instance variable

- The this keyword can be used to refer current class instance variable.
- If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```java
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

this: to invoke current class method

- You may invoke the method of the current class by using the this keyword.
- If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.
- Let's see the example

```
class A
{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

finalize() in Java

- The finalize() method in Java is a special method provided by the Object class.
- It is invoked by the Garbage Collector just before an object is destroyed, allowing the object to perform cleanup operations (like releasing resources) before it is removed from memory.

Key Points:

- Purpose: It is used to clean up resources (e.g., closing files, releasing memory, etc.) before an object is garbage collected.
- Declaration:

```
protected void finalize() throws Throwable
```

- It is protected and can be overridden in your class.
- Invocation: The Garbage Collector calls finalize() automatically, but there is no guarantee when or if it will be called.

```
class Example {
   @Override
   protected void finalize() throws Throwable {
      System.out.println("Finalize method called");
      // Perform cleanup tasks here
   }
}

public class Main {
   public static void main(String[] args) {
      Example obj = new Example();
      obj = null; // Make the object eligible for garbage collection
      System.gc(); // Request garbage collection
   }
}
```

- Output:
- Finalize method called (if the Garbage Collector runs).

Important Notes:

- Deprecated: The finalize() method was deprecated in Java 9 and removed in Java 18. It is no longer recommended for use in modern applications.
- Alternatives:
- Use try-with-resources for managing resources like files or streams.
- Implement the AutoCloseable or Closeable interfaces for custom cleanup logic.

Java Comments

- The Java comments are the statements that are not executed by the compiler and interpreter.
- The comments can be used to provide information or explanation about the variable, method, class or any statement.
- It can also be used to hide program code.

Types of Java Comments

There are three types of comments in java.

- Single Line Comment
- Multi Line Comment
- Documentation Comment

1) Java Single Line Comment

- The single line comment is used to comment only one line.

Syntax:

```
//This is single line comment
```

```
public class CommentExample1 {
public static void main(String[] args) {
   int i=10;//Here, i is a variable
   System.out.println(i);
}
}
```

2) Java Multi Line Comment

- The multi line comment is used to comment multiple lines of code.
- **Syntax:**
- /*
- This
- is
- multi line
- comment
- */

example

- **public class** CommentExample2 {
- **public static void** main(String[] args) {
- /* Let's declare and
- print variable in java. */
- **int** i=10;
- System.out.println(i);
- }
- }

3) Java Documentation Comment

- The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.
- **Syntax:**

```
/**
This
is
documentation
comment
*/
```

Example-JavaDoc comments

- /** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
- **public class** Calculator {
- /** The add() method returns addition of given n umbers.*/
- **public static int** add(**int** a, **int** b){**return** a+b;}
- /** The sub() method returns subtraction of give n numbers.*/
- **public static int** sub(**int** a, **int** b){**return** a-b;}
- }

Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, you can add new methods and fields in your current class also.

Why use inheritance in java

- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusabilit

Terms used in Inheritance

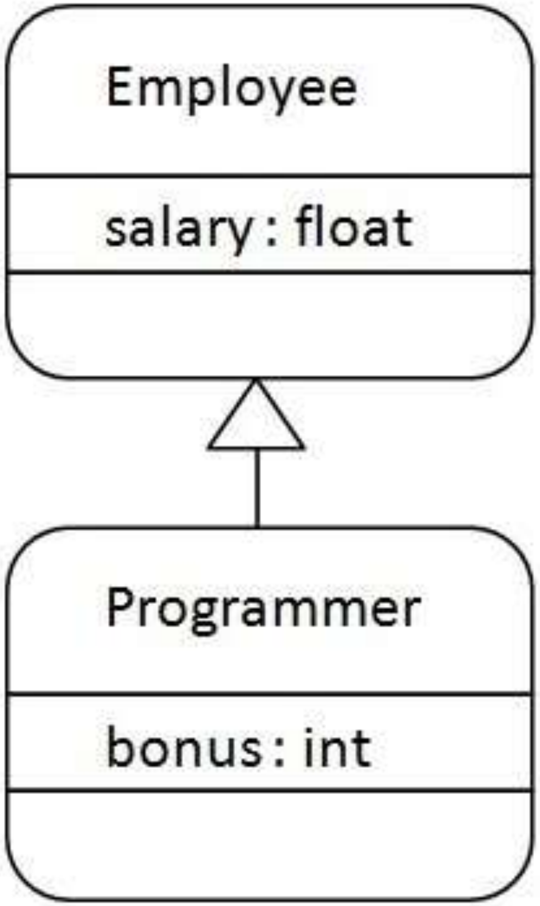
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class

The 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. The meaning of "extends" is to increase the functionality.

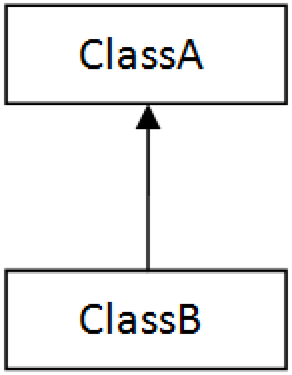
Java Inheritance Example



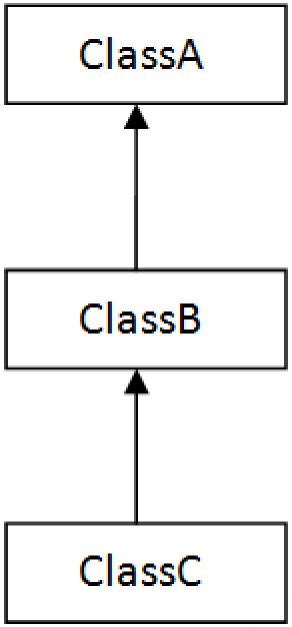
```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of inheritance in java

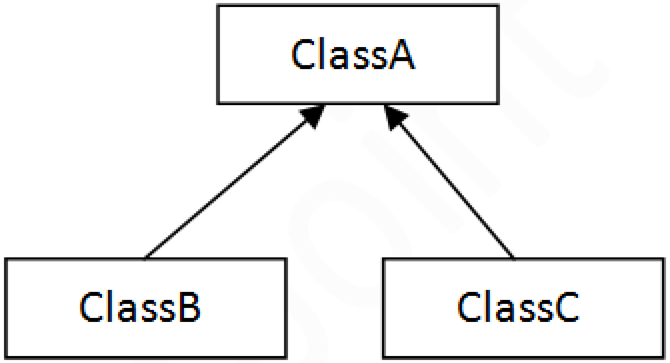
- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



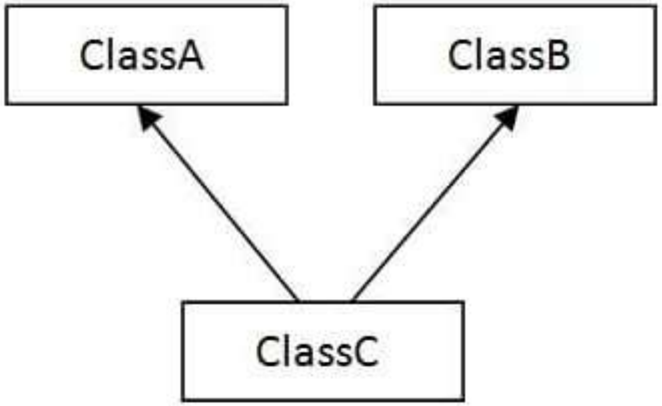
1) Single



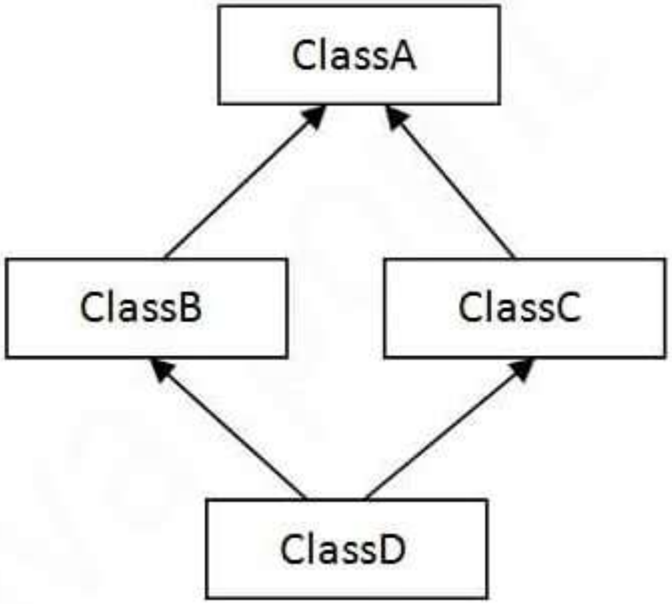
2) Multilevel



3) Hierarchical



4) Multiple



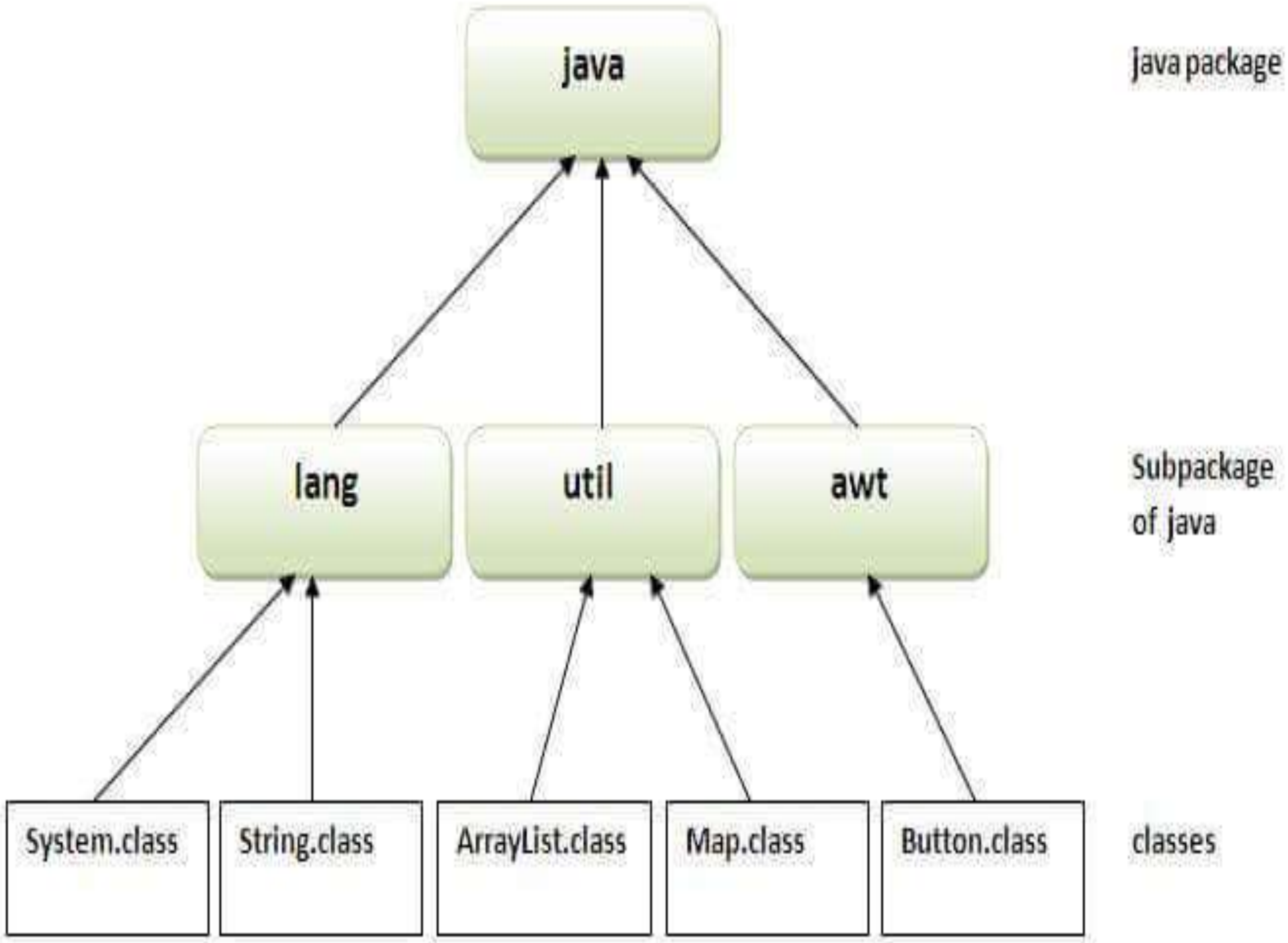
5) Hybrid

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form
  - built-in package
  - user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
-

Advantage of Java Package

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


java
java package
lang
util
awt
Subpackage of java
System.class
String.class
ArrayList.class
Map.class
Button.class
classes

Simple example of java package

- The **package keyword** is used to create a package in java.

```
package mypack;
public class Simple
{
 public static void main(String args[])
{
    System.out.println("Welcome to package");

   }
}
```

How to compile java package

- javac -d directory javafilename

For **example**

```
javac -d . Simple.java
javac -d d:/abc Simple.java
```

- The -d switch specifies the destination where to put the generated class file.
- You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc.
- If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

- You need to use fully qualified name e.g. myp
- **To Compile:**

  javac -d . Simple.java
- **To Run :**

  java mypack.Simpleack.Simple etc to run the class.
- Output: Welcome to package
- The -d is a switch that tells the compiler where to put the class file i.e. it represents destination.
- The . represents the current folder.

How to access package from another package?

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

1. import package.*;
2. import package.classname;
3. 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.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

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

Example of package that import the packagename.*

```
//save by B.java
package mypack;
import pack.*;

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

2) Using packagename.classname

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

```
//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.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

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

Abstract class in Java

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract and non-abstract methods (method with the body).
- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details
- for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class in Java

- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.
- **Example of abstract class**
- **abstract class** A{}

Abstract Method in Java

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- **Example of abstract method**

abstract void printStatus();//no method body an d abstract

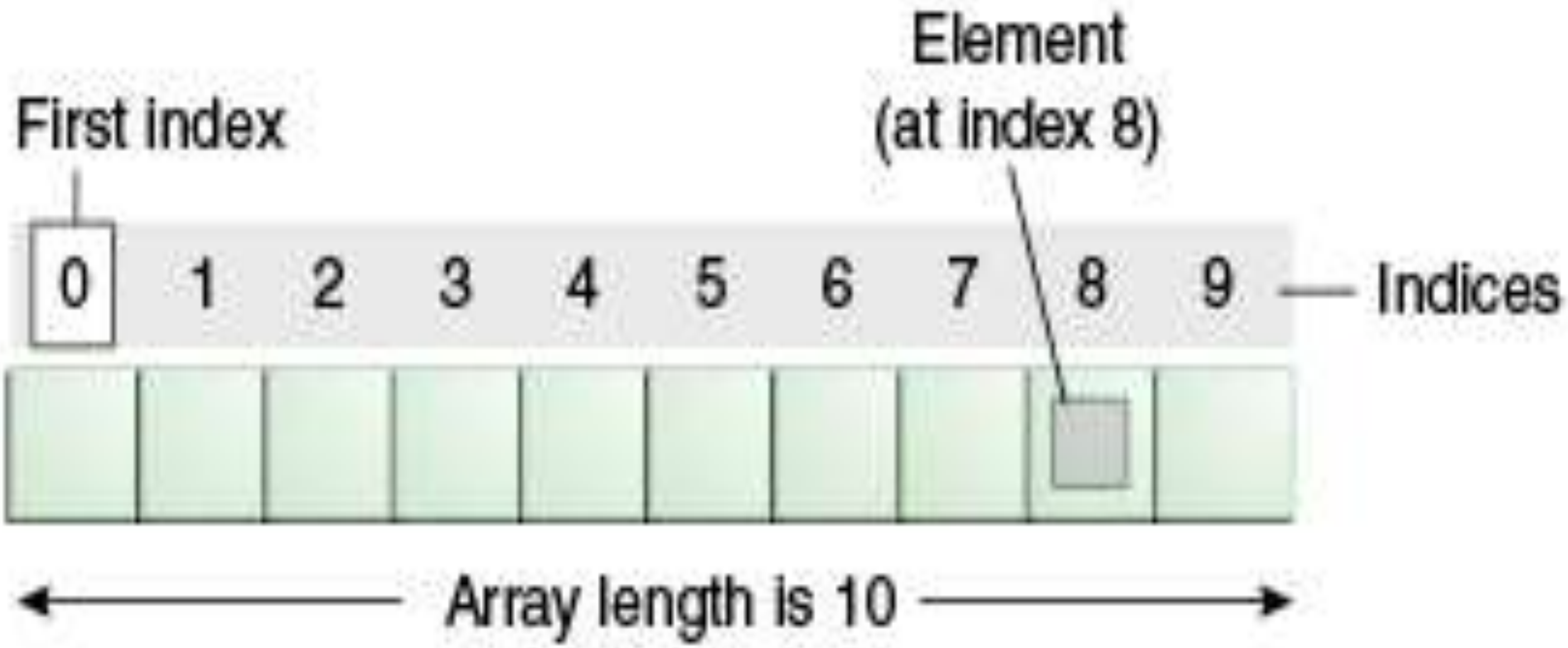
Example of Abstract class that has an abstract method

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Java Arrays

- Normally, an array is a collection of similar type of elements which has contiguous memory location.
- **Java array** is an object which contains elements of a similar data type.
- Additionally, The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements.
- We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

- Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.
- In Java, array is an object of a dynamically generated class.
- Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces.
- We can store primitive values or objects in an array in Java.
- Like C/C++, we can also create single dimentional or multidimentional arrays in Java.
- Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.


Element
(at index 8)
First index
0
1
2
3
4
5
6
7
8
9
Indices
Array length is 10

Advantages

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

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

arrayRefVar=**new** datatype[size];

Example of Java Array

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
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]);
}}
```

For-each Loop for Java Array

- We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one.
- It holds an array element in a variable, then executes the body of the loop.
- The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

```java
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Passing Array to a Method in Java

- We can pass the java array to method so that we can reuse the same logic on any array.

```
class Testarray2{
//creating a method which receives an array as a parameter
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};//declaring and initializing an array
min(a);//passing array to method
}}
```

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

1) Java final variable

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

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

2) 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();
  }
}
```

3) Java final class

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

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

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

Interface in Java

- An **interface in Java** is a blueprint of a class.
- It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables.
- It cannot have a method body.

Why use Java interface?

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

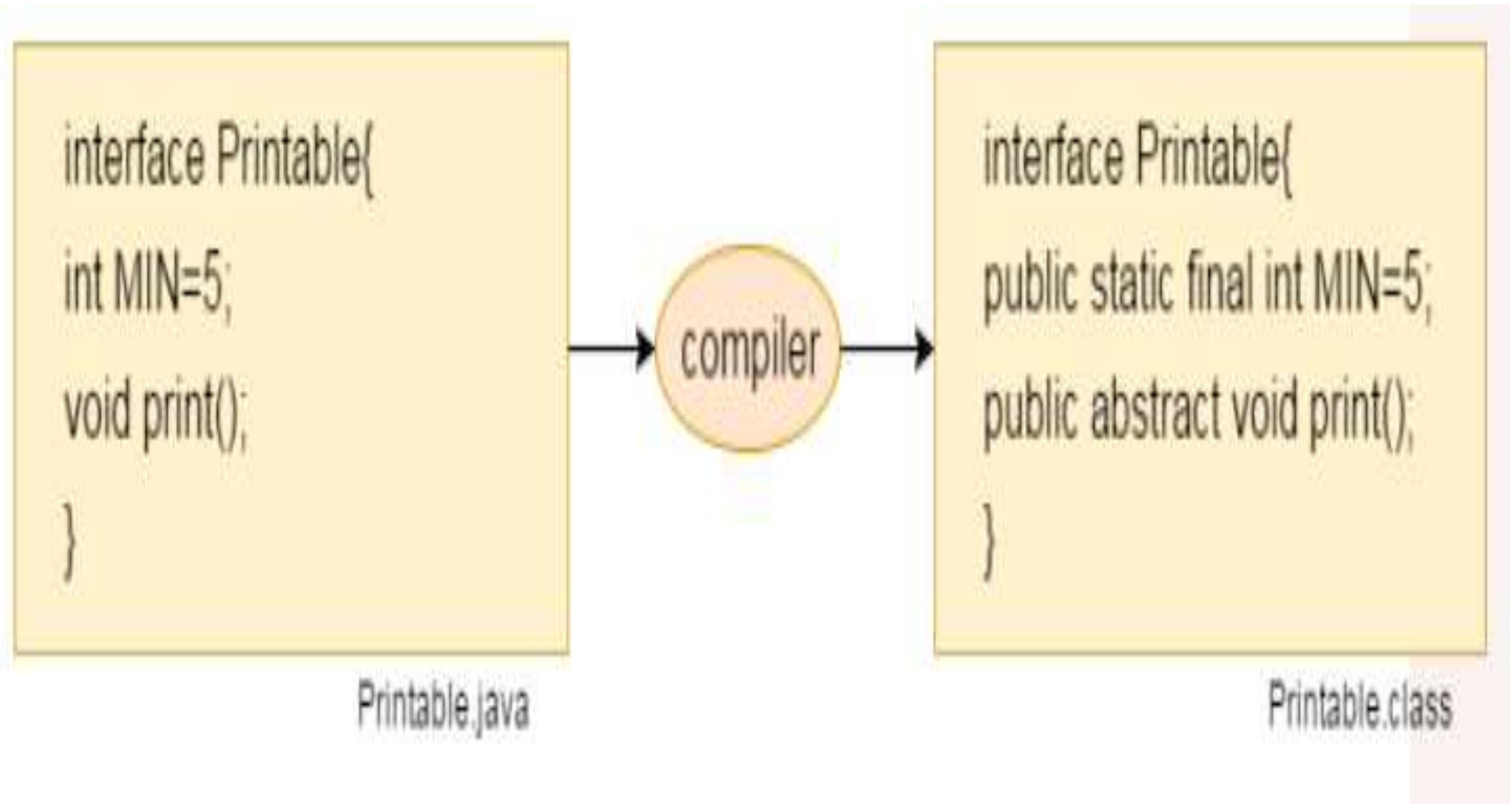
How to declare an interface?

- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

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

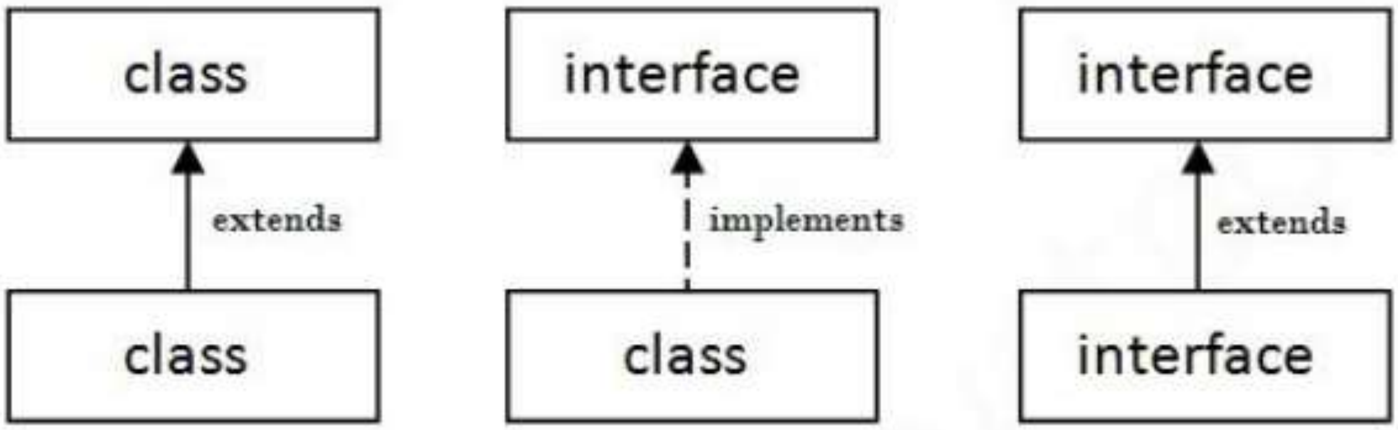


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

a class extends another class, an interface extends another interface, but a **class implements an interface**.



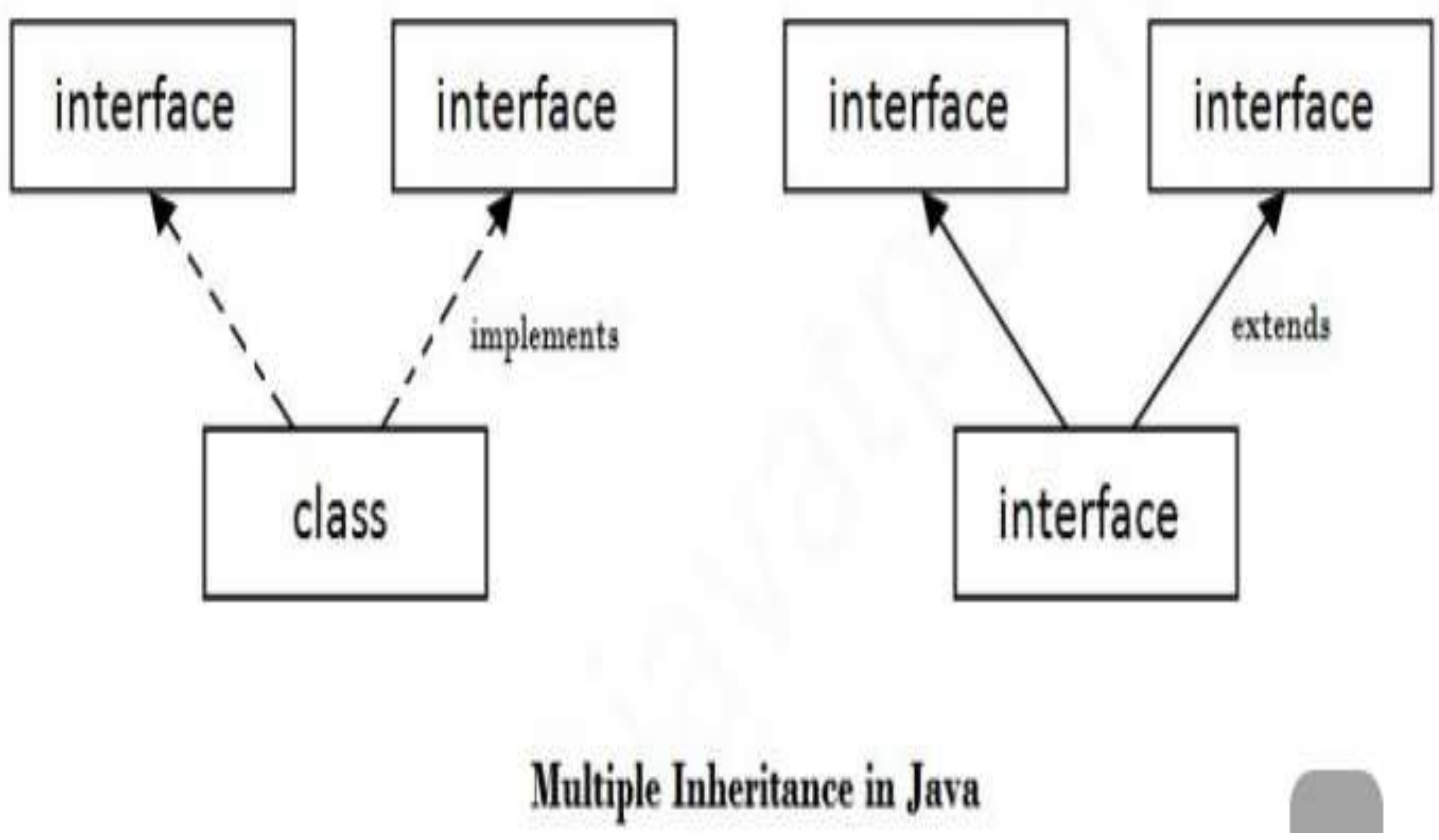
Java Interface Example

```
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();
 }
}
```

```
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by meth
    od e.g. getDrawable()
d.draw();
}}
```

```java
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

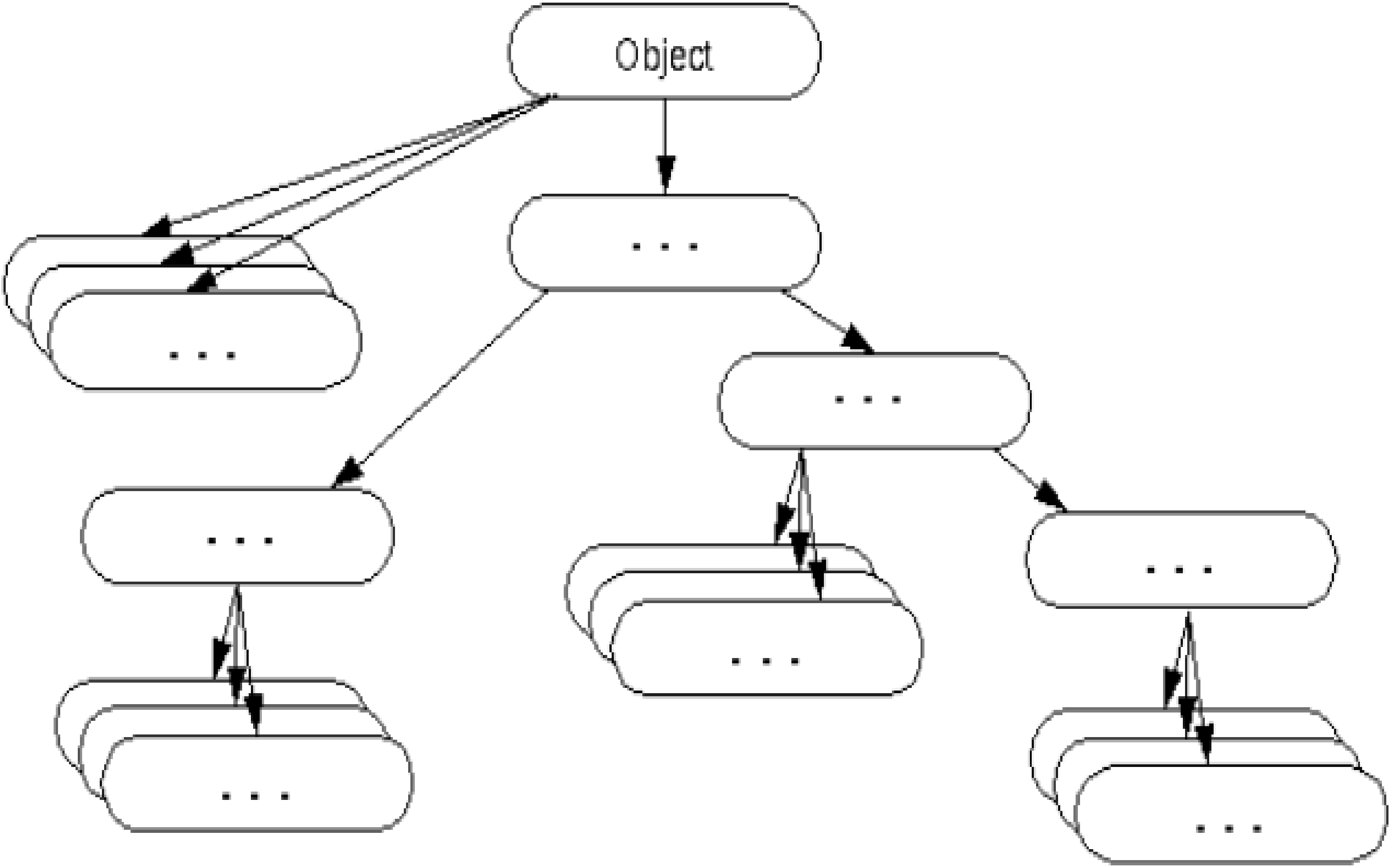
Multiple inheritance in Java by interface



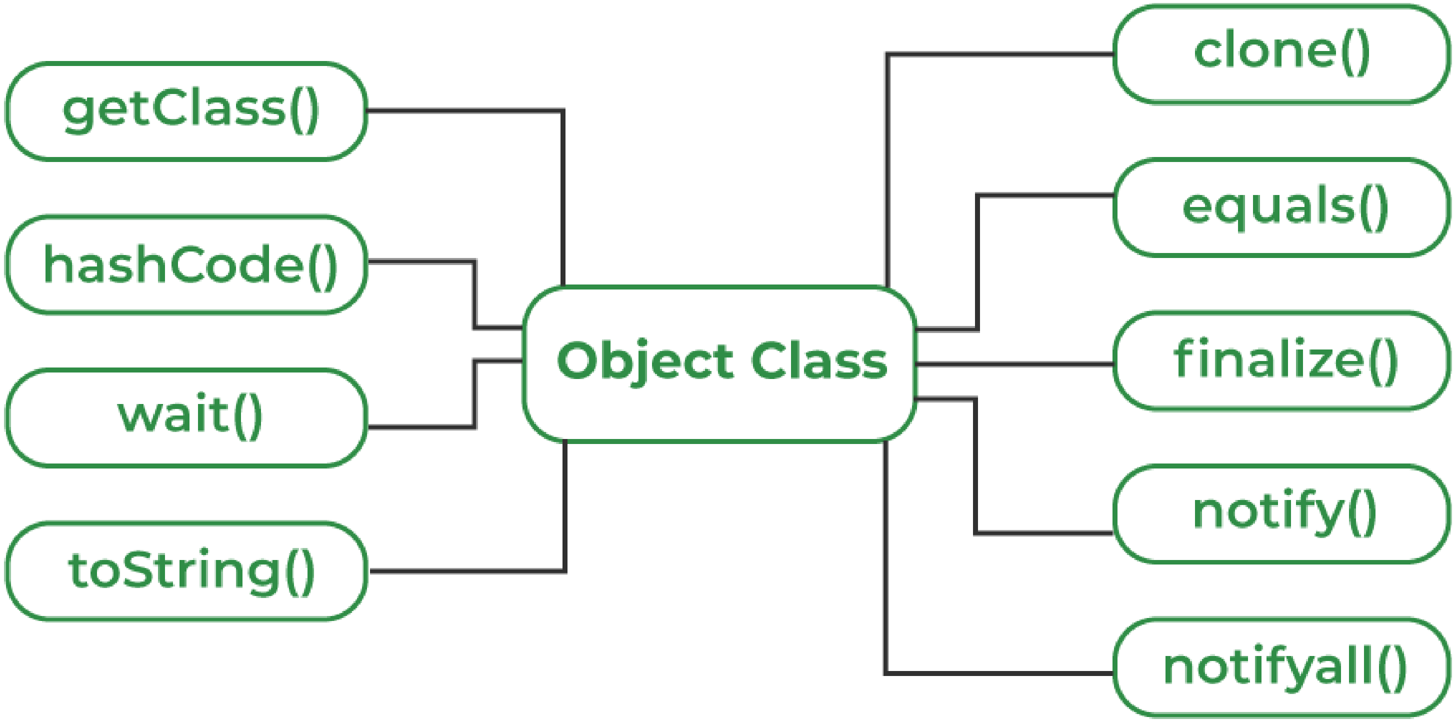
```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

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

Object class in Java

- The **Object class** is the parent class of all the classes in java by default.
- In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.
- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Object
. . .
. . .
. . .
. . .
. . .
. . .
. . .
. . .


getClass()
hashCode()
wait()
toString()
Object Class
clone()
equals()
finalize()
notify()
notifyall()

Methods of Object class

- public final Class getClass()

  returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
- public int hashCode()

  returns the hashcode number for this object.
- public boolean equals(Object obj)

  compares the given object to this object.
- protected Object clone() throws CloneNotSupportedException

  creates and returns the exact copy (clone) of this object.

- public String toString()

  returns the string representation of this object.
- public final void notify()

  wakes up single thread, waiting on this object's monitor.
- public final void notifyAll()

  wakes up all the threads, waiting on this object's monitor.
- public final void wait(long timeout)throws InterruptedException

  causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).

- public final void wait(long timeout,int nanos)throws InterruptedException

  causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).

- public final void wait()throws InterruptedException

  causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).

- protected void finalize()throws

  Throwableis invoked by the garbage collector before object is being garbage collected.

Object Cloning in Java

- the **Cloneable interface** that indicates that a class has provided a safe **clone() method**.
- To understand what cloning means recall what happens when you make a copy of a variable holding an object reference.
- The original and the copy are references to the same object. This means the original and the copy are mutually dependent, i.e., a change in one causes a change in the other as well.
- If we would like a copy to be a new object that begins its life being identical to the original but whose state can change over time we must use the **clone() method**.

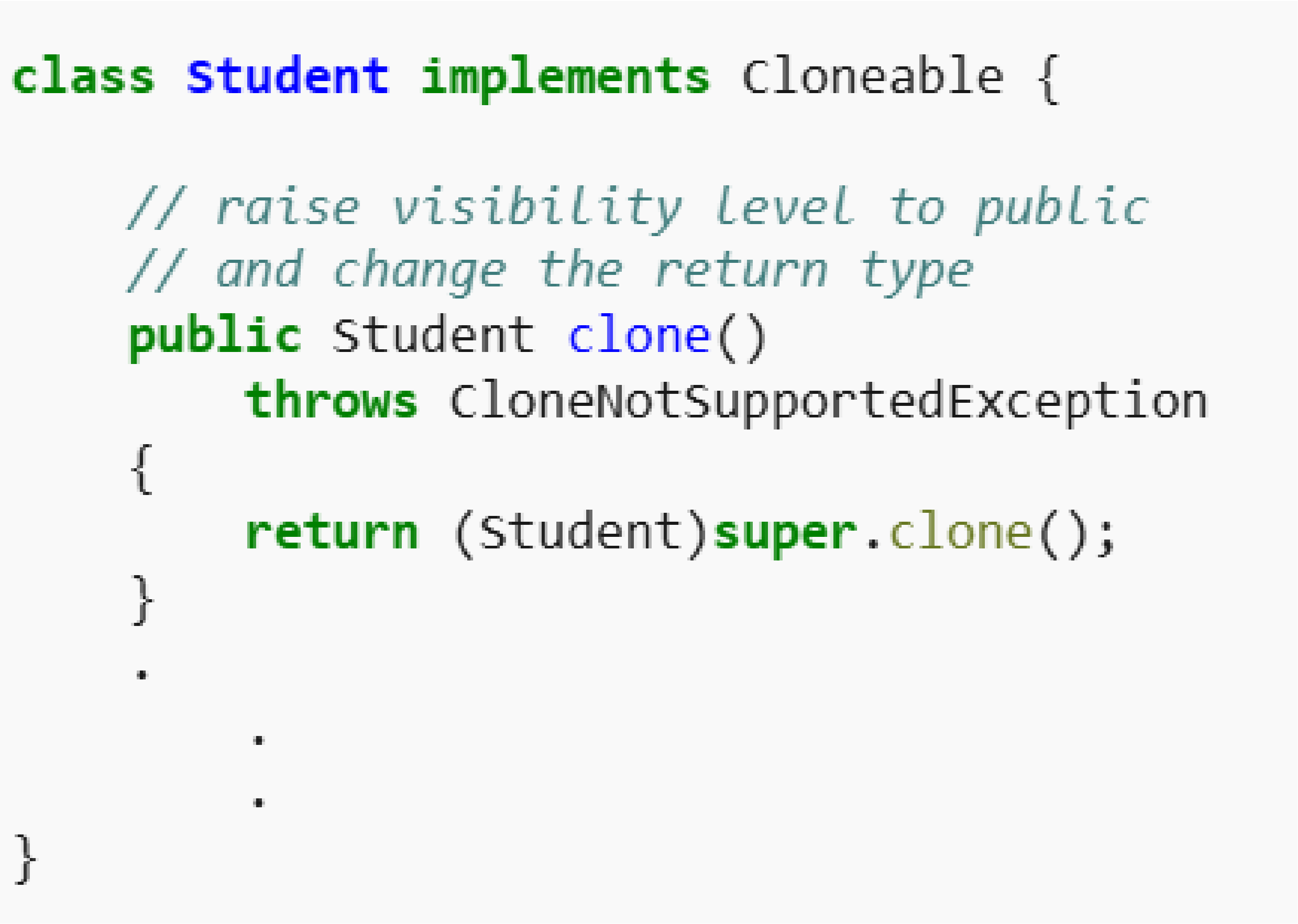
- The **clone() method** is declared **protected** in the **Object class**, so our code can't simply call **obj.clone()**.
- Now we might ask, but aren't protected methods accessible from any subclass and isn't every class a subclass of **Object**.
- Fortunately the rules for protected access are much more subtle.
- A subclass can call protected **clone() method** to clone its own objects.
- We must redefine clone to be **public** to be accessed by any methods.

- Even though the default implementation of **clone** is adequate, you still need to implement the **Cloneable** interface, redefine **clone() method** to be **public**, and call **public**

Key Points About Object Cloning

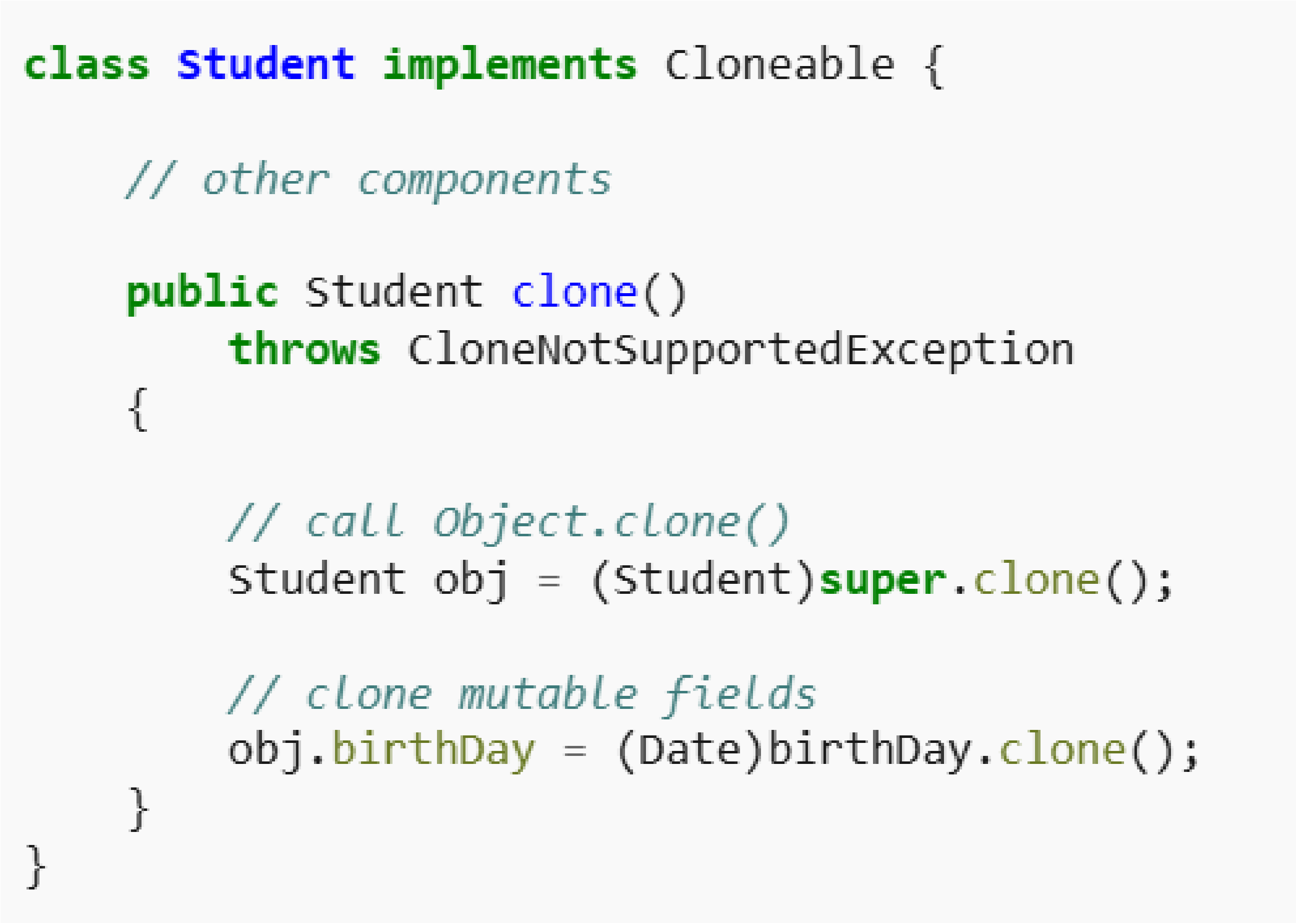
- Object cloning in Java refers to creating an exact copy of an object. Java provides the `clone()` method in the `Object` class to achieve this. Cloning can be categorized into **shallow cloning** and **deep cloning**, depending on how the object's fields are copied
- **clone() Method**:
  - Defined in the Object class.
  - Creates and returns a copy of the object.
  - The class must implement the Cloneable interface to use this method; otherwise, it throws a CloneNotSupportedException.
- **Shallow Copy**:
  - Copies the object and its primitive fields.
  - For reference fields, it copies the references, not the actual objects. Changes to the referenced objects affect both the original and the clone.
- **Deep Copy**:
  - Copies the object and all objects referenced by it recursively.
  - Ensures that changes to the referenced objects do not affect the original object.

Shallow Copy:



```
class Student implements Cloneable {

    // raise visibility level to public
    // and change the return type
    public Student clone()
        throws CloneNotSupportedException
    {
        return (Student)super.clone();
    }
    .
        .
        .
}
```

Deep Copy:



```
class Student implements Cloneable {

    // other components

    public Student clone()
        throws CloneNotSupportedException
    {

        // call Object.clone()
        Student obj = (Student)super.clone();

        // clone mutable fields
        obj.birthDay = (Date)birthDay.clone();
    }
}
```

```
class Person implements Cloneable {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Overriding the clone() method
    @Override
    protected Object clone() throws
CloneNotSupportedException {
        return super.clone();
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "',
age=" + age + "}";
    }
}
```

```
public class ShallowCloningExample {
    public static void main(String[] args) {
        try {
            // Original object
            Person person1 = new Person("John", 25);

            // Cloning the object
            Person person2 = (Person) person1.clone();

            // Displaying both objects
            System.out.println("Original: " + person1);
            System.out.println("Clone: " + person2);

            // Modifying the clone
            person2.name = "Doe";

            // Changes in the clone do not affect the original
            System.out.println("After modification:");
            System.out.println("Original: " + person1);
            System.out.println("Clone: " + person2);

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

- Original: Person{name='John', age=25}
- Clone: Person{name='John', age=25}
- After modification:
- Original: Person{name='John', age=25}
- Clone: Person{name='Doe', age=25}

```
import java.util.Objects;

class Address implements Cloneable {
    String city;
    String state;

    public Address(String city, String state) {
        this.city = city;
        this.state = state;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
    public String toString() {
        return "Address{city='" + city + "', state='" + state + "'}";
    }
}

class Employee implements Cloneable {
    String name;
    Address address;

    public Employee(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // Perform deep cloning
        Employee cloned = (Employee) super.clone();
        cloned.address = (Address) address.clone();
        return cloned;
    }
```

```
 @Override
    public String toString() {
        return "Employee{name='" + name + "', address=" +
address + "}";
    }
}

public class DeepCloningExample {
    public static void main(String[] args) {
        try {
            // Original object
            Address address = new Address("Chennai", "Tamil
Nadu");
            Employee emp1 = new Employee("Alice", address);

            // Cloning the object
            Employee emp2 = (Employee) emp1.clone();

            // Displaying both objects
            System.out.println("Original: " + emp1);
            System.out.println("Clone: " + emp2);

            // Modifying the clone's address
            emp2.address.city = "Bangalore";

            // Changes in the clone do not affect the original
            System.out.println("After modification:");
            System.out.println("Original: " + emp1);
            System.out.println("Clone: " + emp2);

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```