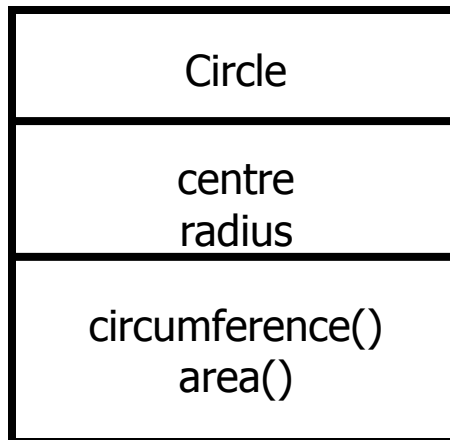


# Classes and Objects in Java

## Basics of Classes in Java

# Class

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.



# Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a class definition:

```
class ClassName  
{  
    //fields declaration  
    //methods declaration  
}
```

# Adding Fields: Class Circle with fields

- Add *fields*

```
class Circle
{
    public double x, y; // centre coordinate
    public double r; // radius of the circle
}
```

- The fields (data) are also called the *instance* variables.

# Adding Methods

- A class with only data fields has no life.
- Methods are declared inside the body of the class.
- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
    Method-body;
}
```

# Adding Methods to Class Circle

```
class Circle {
```

```
    public double x, y; // centre of the circle
```

```
    public double r;    // radius of circle
```

```
    //Methods to return circumference and area
```

```
    public double circumference() {
```

```
        return 2*3.14*r;
```

```
    }
```

```
    public double area() {
```

```
        return 3.14 * r * r;
```

```
    }
```

```
}
```

Method Body



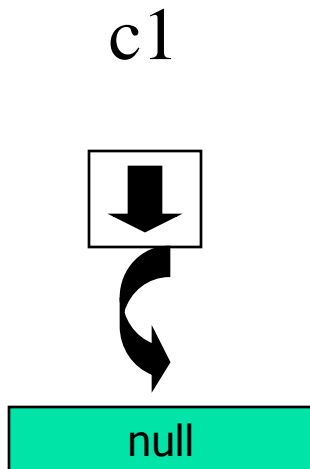
# Data Abstraction

- Declare the Circle class, have created a new data type – Data Abstraction
- Can define variables (objects) of that type:

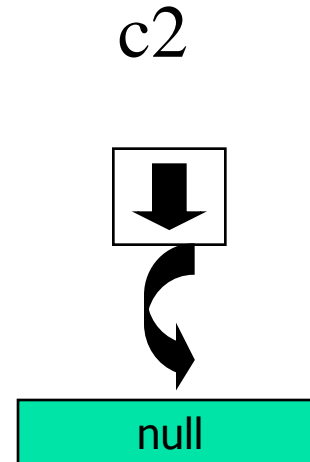
```
Circle c1;  
Circle c2;
```

# Class of Circle cont.

- c1, c2 simply refers to a Circle object, not an object itself.



Points to nothing (Null Reference)



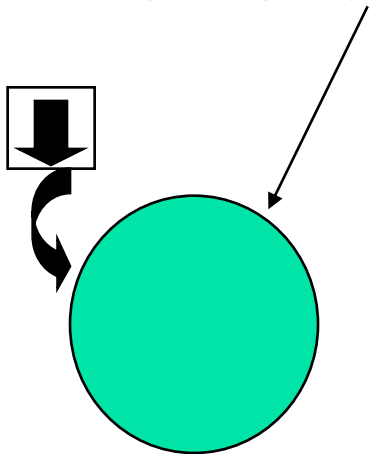
Points to nothing (Null Reference)



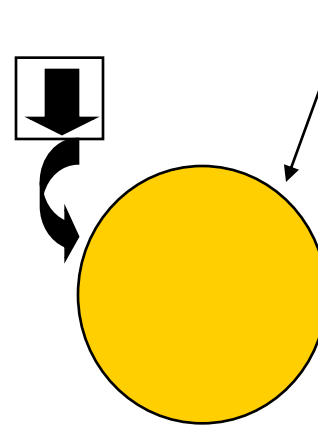
# Creating objects of a class

- Objects are created dynamically using the *new* keyword.
- `c1` and `c2` refer to `Circle` objects

`c1 = new Circle() ;`



`c2 = new Circle() ;`



# Creating objects of a class

```
c1 = new Circle();  
c2 = new Circle() ;
```

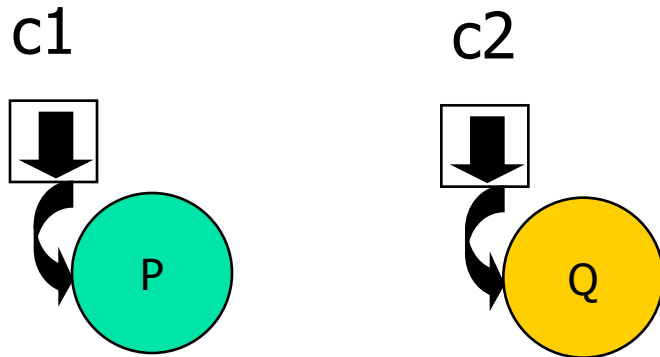
```
c2 = c1;
```

# Creating objects of a class

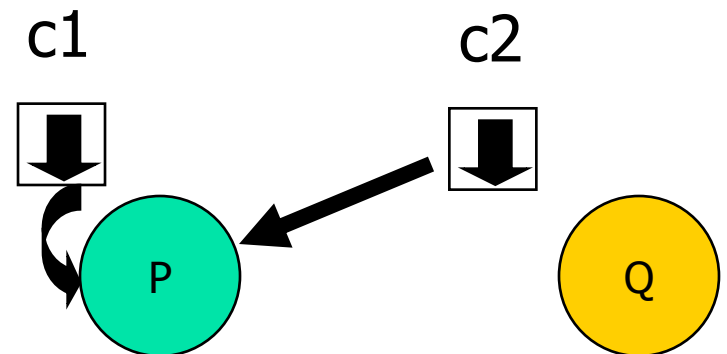
```
c1 = new Circle();  
c2 = new Circle() ;
```

```
c2 = c1;
```

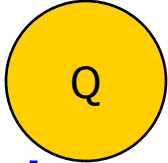
Before Assignment



After Assignment



# Automatic garbage collection

- The object  does not have a reference and cannot be used in future.
- The object becomes a candidate for automatic **garbage collection**.
- Java automatically collects garbage periodically and releases the memory used to be used in the future.

# Using Circle Class

```
class Driver
{
    public static void main(String args[])
    {
        Circle c1; // creating reference
        c1 = new Circle(); // creating object
        c1.x = 10; // assigning value to data field
        c1.y = 20;
        c1.r = 5;
        double area = c1.area(); // invoking method
        double circumf = c1.circumference();
        System.out.println("Radius="+c1.r+" Area="+area);
        System.out.println("Radius="+c1.r+" Circumference =" +circumf);
    }
}
```

```
javac Circle.java
javac Driver.java
java Driver
Radius=5.0 Area=78.5
Radius=5.0 Circumference =31.400000000000002
```

# Summary

- Classes, objects, and methods are the basic components used in Java programming.
- We have discussed:
  - How to define a class
  - How to create objects
  - How to add data fields and methods to classes
  - How to access data fields and methods to classes

# Platform Independence

- Source Code is converted to Intermediate code known as **BYTE CODE**.
- This Byte Code is read by JVM (Java Virtual Machine).
- JVM converts the Byte code to Machine Code.
- The machine code so created can be read by the machine on which it was formed.

# Main method in Java

- main method is defined as  

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

  - public as it will be called by the JVM outside the package.
  - static as it will be called by JVM without making the object of the class.
  - void as it does not return any value.
  - String args[] for receiving command line arguments as array of Strings.



# Introduction to Java

- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995.

# Introduction to Java

- The primary motivation was the need for a platform-independent (that is, architecture-neutral) language
- Need to create software to be embedded in various consumer electronic devices,
- Java was then developed as portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.

# Introduction to Java

- Emergence of the World Wide Web:
  - Web demanded portable programs.
  - Java was propelled to the forefront of computer language design.
- So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

# Java's Magic: The Bytecode

- The output of a Java compiler is Bytecode that makes Java as Platform-independent.
- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- JVM will read the Bytecode and will convert to the executable code which is compatible to the machine on which it is converted.

# Just-In-Time (JIT) compiler

- When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis.
- It is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time.

# Just-In-Time (JIT) compiler

- Instead, a JIT compiler compiles code as it is needed, during execution.
- The remaining code is simply interpreted.
- However, the just-in-time approach still yields a significant performance boost.

# The Java Buzzwords

- The key considerations were summed up by the Java team in the following list of buzzwords:

- **Simple**
- **Secure**
- **Portable**
- **Object-oriented**
- **Robust**
- **Multithreaded**
- **Architecture-neutral**
- **Interpreted**
- **High performance**
- **Distributed**
- **Dynamic**

# 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

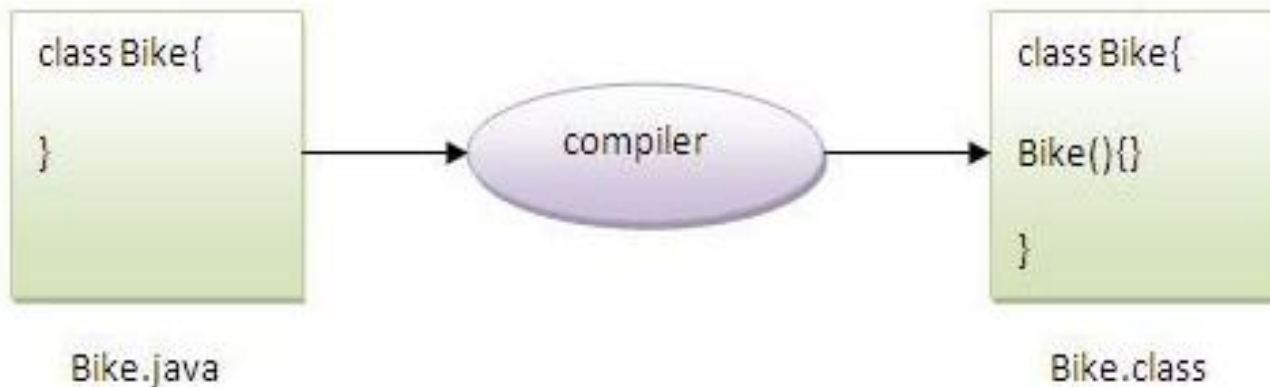
- There are basically two rules defined for the constructor.
  - Constructor name must be same as its class name
  - Constructor must have no explicit return type

# Types of java constructors

- There are two types of constructors:
  - Default constructor (no-argument constructor)
  - Parameterized constructor

# Java Default Constructor

- A constructor that have no parameter is known as default constructor.
- If there is no constructor in a class, compiler automatically creates a default constructor.



# Example of Default Constructor

```
class Bike1{  
int maxspeed;  
Bike1()  
{  
maxspeed=60;  
} //End of default constructor  
public static void main(String args[]) {  
Bike1 b=new Bike1(); //Constructor is called here  
} //End of main method  
} //End of class
```

# Java parameterized constructor

- A constructor that have parameters is known as parameterized constructor.
- Parameterized constructor is used to provide different values to the distinct objects.

# Parameterized Constructors Example

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

# Parameterized Constructors Example

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

# Parameterized Constructors Example

```
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
        double vol;  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```



# 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.

# Example of Constructor Overloading

```
class Student
```

```
{
```

```
    int id; String name, mobile_number;
```

```
    Student (int i, String n) {
```

```
        id = i;
```

```
        name = n;
```

```
    }
```

```
    Student (int i, String n, String a) {
```

```
        id = i;
```

```
        name = n;
```

```
        mobile_number=a;
```

```
}
```

# Example of Constructor Overloading

```
void display()
```

```
{
```

```
System.out.println(id+" "+name+" "+mobile_number);
```

```
}
```

```
public static void main(String args[]){
```

```
    Student s1 = new Student(111,"Karan");
```

```
    Student s2 = new Student(222,"Aryan","9123456789");
```

```
    s1.display();
```

```
    s2.display();
```

```
}
```

```
}
```

# Constructor V/s method in java

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

# Inheritance in Java

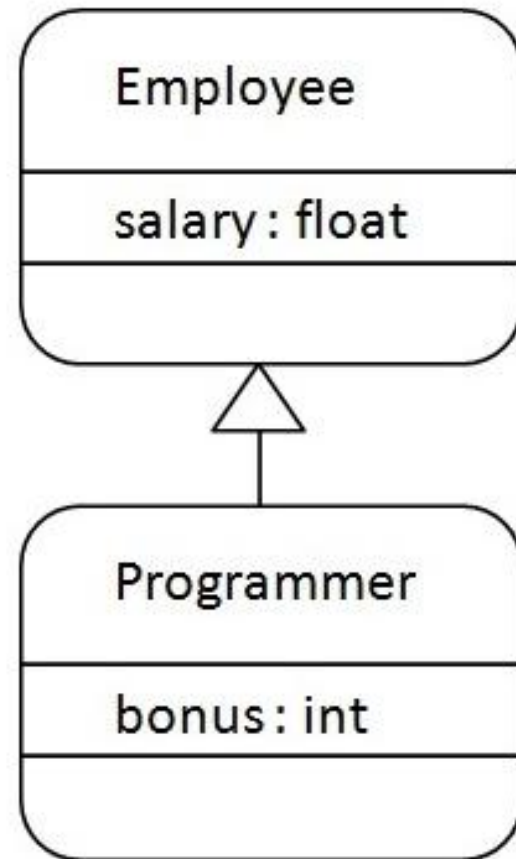
- **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

# Why use inheritance in java

- For Method Overriding
- For Code Reusability.

The **extends keyword** is used for making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.



# Example of inheritance

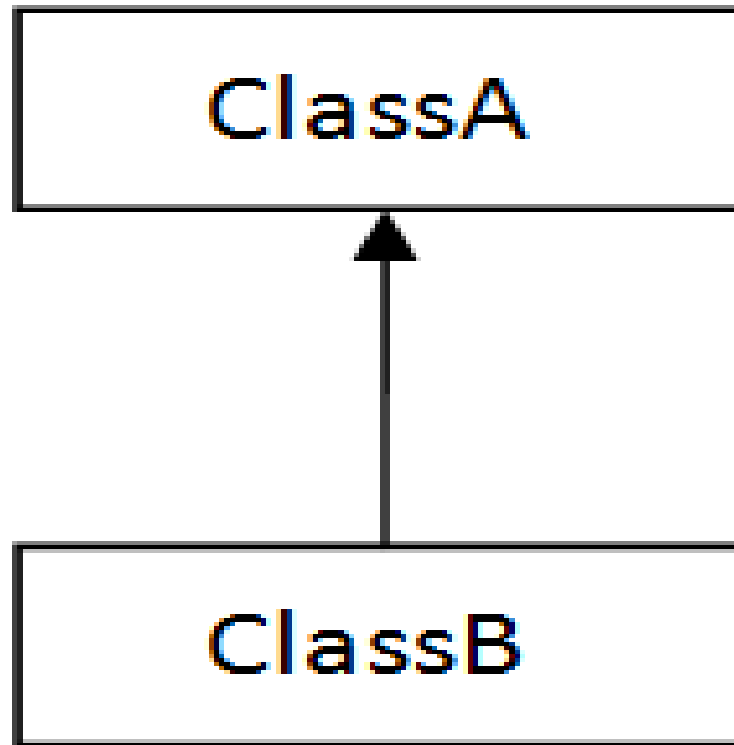
```
class Employee{  
    int salary=40000;  
}  
class Programmer extends Employee {  
    int bonus=10000;  
    void display() {  
        int totalSalary=salary + bonus;  
        System.out.println("Total Salary is:"+totalSalary);  
    }  
    public static void main(String args[]) {  
       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

# Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java:
  - Single
  - Multilevel
  - Hierarchical



# Single Inheritance



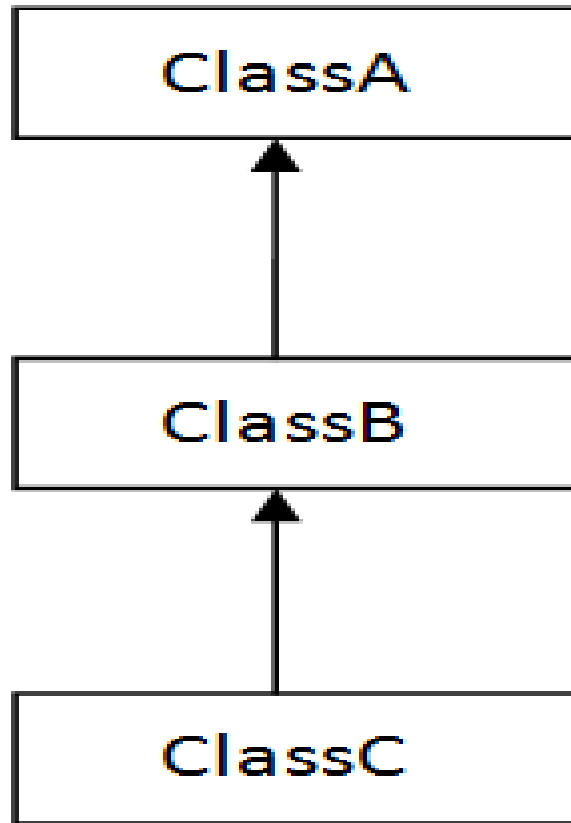
# Example of Single Inheritance

```
class Person{  
String name;  
void displayName() {System.out.println(name);}  
}  
class Student extends Person {  
int p;  
void displayPercentage() {  
System.out.println(p);  
}  
void displayAllDetails() {  
displayName();  
displayPercentage(); }  
}
```

# Example of Single Inheritance

```
class TestInheritance{  
public static void main(String args[]){  
    Student s=new Student();  
    s.displayName();  
    s.displayPercentage();  
    s.displayAllDetails();  
}  
}
```

# Multi-level Inheritance



# Example of Multilevel Inheritance

```
class Animal{  
void eat() {System.out.println("I am eating...");}  
}
```

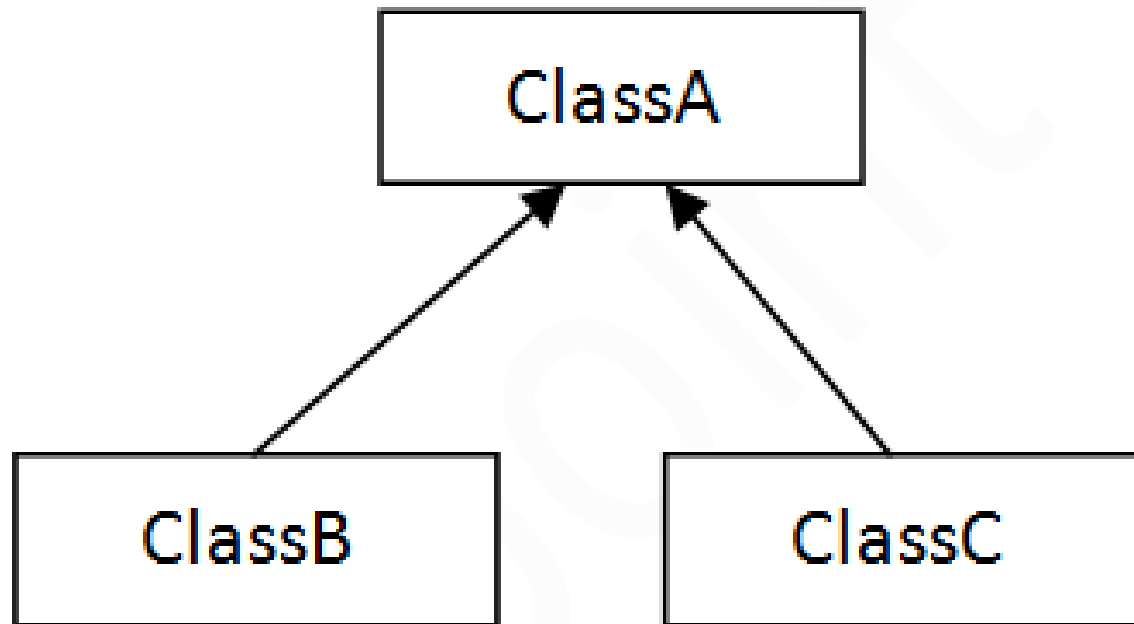
```
class Dog extends Animal{  
void bark() {System.out.println("I am barking...");}  
}
```

```
class BabyDog extends Dog{  
void weep() {System.out.println("I am weeping...");}  
}
```

# Example of Multilevel Inheritance

```
class TestInheritance{  
public static void main(String args[]){  
    BabyDog d=new BabyDog();  
    d.weep();  
    d.bark();  
    d.eat();  
}}
```

# Hierarchical Inheritance



# Example of hierachical Inheritance

```
class Animal{  
void eat() {System.out.println("I am eating...");}  
}  
class Dog extends Animal{  
void bark(){System.out.println("I am barking...");}  
}  
class Cat extends Animal{  
void meow(){System.out.println("I am meowing...");}  
}
```



# Example of Hierarchical Inheritance

```
class TestInheritance{  
public static void main(String args[]){  
    Cat c=new Cat();  
    c.meow();  
    c.eat();  
    //c.bark();//C.T.Error  
}  
}
```

# Abstraction in Java

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.
- 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 with abstract keyword, is known as abstract class in java.
- It can have abstract methods (methods without body) and non-abstract methods (method with body).
- It needs to be extended and its method implemented.
- It cannot be instantiated.

# Abstract method

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

```
abstract class Shape{  
    abstract double calArea();  
}
```

```
class Rectangle extends Shape{  
    double length , breadth;  
    double calArea()  
    {  
        return length*breadth;  
    }  
}
```

# Example of abstract class

```
public abstract class Shape
{
    double area, perimeter;
    public abstract double calArea();
    public abstract double calPerimeter();
}
```

**Save this file as Shape.java**  
**Why?**

# Example of abstract class

```
public class Rectangle extends Shape {  
    private final double width, height;  
    public Rectangle(double width, double height){  
        this.width = width;  
        this.height = height;  
    }  
    public double calArea() {  
        area=width * height;  
        return area;  
    }  
    public double calPerimeter() {  
        perimeter= 2 * (width + height);  
        return perimeter;  
    }  
}
```

# Example of abstract class

```
public class Circle extends Shape {  
    private final double radius;  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    public double calArea() {  
        area=Math.PI * Math.pow(radius, 2);  
        return area;  
    }  
    public double calPerimeter() {  
        perimeter= 2 * Math.PI * radius;  
        return perimeter;  
    }  
}
```

# Example of abstract class

```
public class Triangle extends Shape {  
    private final double a, b, c;  
    public Triangle(double a, double b, double c) {  
        this.a = a; this.b = b; this.c = c;  
    }  
    public double calArea() {  
        double s = calPerimeter() / 2;  
        area= Math.sqrt(s * (s - a) * (s - b) * (s - c));  
        return area;  
    }  
    public double calPerimeter() {  
        perimeter= a + b + c;  
        return perimeter;  
    }  
}
```



# Example of abstract class

```
public class Hexagon extends Shape {  
    private final double a;  
    public Hexagon(double a) {  
        this.a = a;  
    }  
    public double calPerimeter() {  
        return 6*a;  
    }  
}
```

ANYTHING MISSING???  
WHAT IS THE SOLUTION?

*Solution: Either create a method  
calArea() or declare the class as  
abstract*

# Example of abstract class

```
abstract public class Hexagon extends Shape {  
    private final double a;  
    public Hexagon(double a) {  
        this.a = a;  
    }  
    public double perimeter() {  
        return 6*a;  
    }  
}
```

# Example of abstract class

```
public class TestShape {  
    public static void main(String[] args) {  
        Shape rectangle = new Rectangle(5, 6);  
        System.out.println("Rectangle area: " + rectangle.calArea() +  
            "\nRectangle perimeter: " + rectangle.calPerimeter() + "\n");  
        Shape circle = new Circle(3.5);  
        System.out.println("Circle Area: " + circle.calArea() +  
            "\nCircle  
        Perimeter: " + circle.calPerimeter() + "\n");  
        Shape triangle = new Triangle(3,4,5);  
        System.out.println("Triangle Area: " + triangle.calArea() +  
            "\nTriangle Perimeter: " + triangle.calPerimeter() + "\n");  
    }  
}
```

# this keyword

- **this** is a **reference variable** that refers to the current object.
- The 6 usages of java this keyword are:
  - i. to refer current class instance variable.
  - ii. to invoke current class method (implicitly)
  - iii. to invoke current class constructor.
  - iv. this can be passed as an argument to a method.
  - v. this can be passed as an argument to the constructor
  - vi. this can be used to return the current class instance from the method.

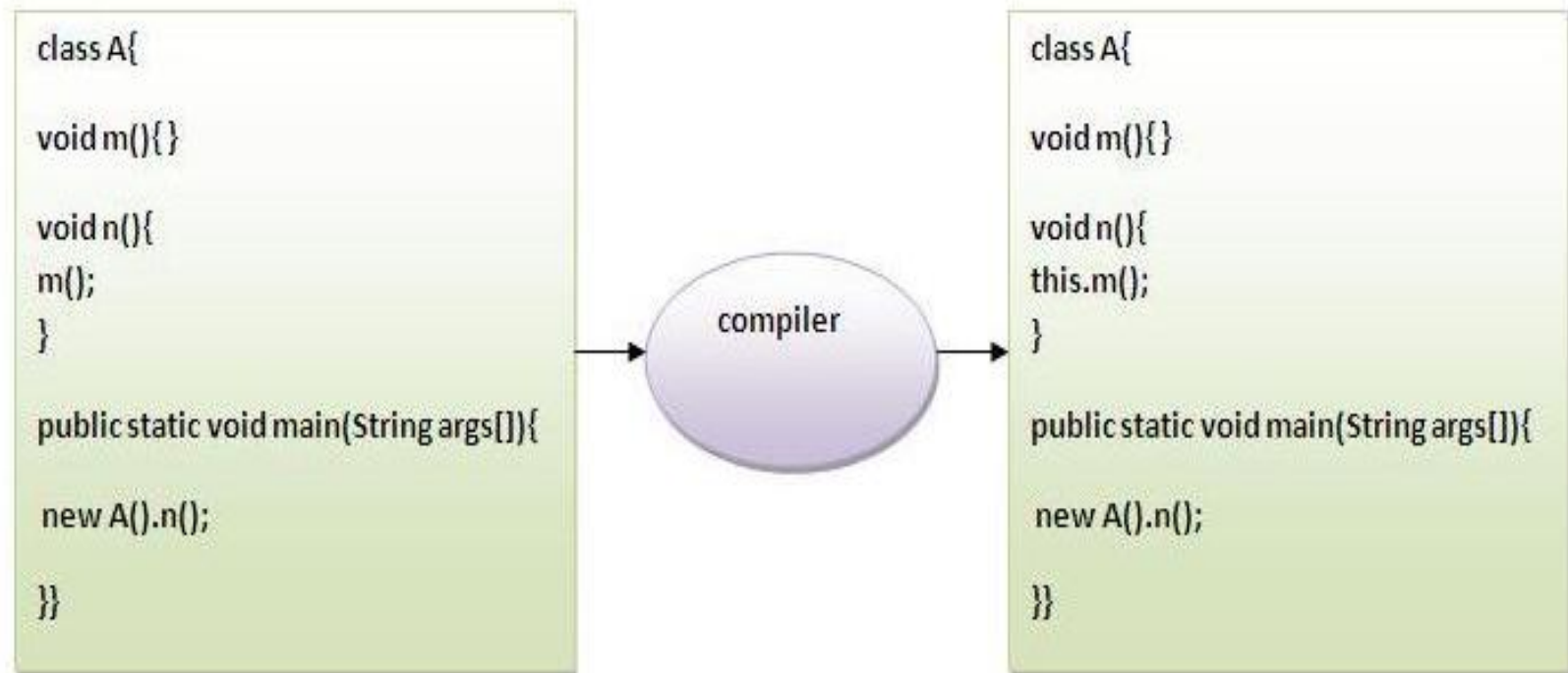
# this keyword

- The need of this keyword to refer current class instance variable is when the local variable and the instance variable have same name(s).

```
class X
{
    int a , b;
    X(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
    public void display()
    {
        System.out.println(" a= " + a + " \n b = " + b );
    }
}
```

# this keyword

- It is used to invoke the method of the current class by using this keyword. If not, the compiler automatically adds this keyword while invoking the method.



# this keyword

- It is also used to invoke current class constructor.
- Example:

```
class A{  
A()  
{  
System.out.println("Hi");  
}  
A(int x)  
{  
this();  
System.out.println(x);  
}  
}
```

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

# this keyword

```
class Student{  
    int rollno;  
    String name, course;  
    float fee;  
    Student(int rollno, String name, String course){  
        this.rollno=rollno;  
        this.name=name;  
        this.course=course;  
    }  
    Student (int rollno, String name, String course, float fee){  
        this(rollno, name, course);//reusing constructor  
        this.fee=fee;  
    }  
}
```



# this keyword

```
void display(){  
    System.out.println(rollno+" "+name+" "+course+" "+fee);  
}  
} // End of Student class
```

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

# Proving this keyword

```
class A5{  
void m(){  
    System.out.println(this); //prints same reference ID  
}  
public static void main(String args[]){  
    A5 obj=new A5();  
    System.out.println(obj); //prints the reference ID  
    obj.m();  
}  
}
```

Output:  
A5@22b3ea59  
A5@22b3ea59

# 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 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.

# Java final variable

- If you make any variable as final, you cannot change the value of final variable
- In other words, it will be a constant.

# Java final variable

```
class Student{  
    final int rollNumber=151500090;//final variable  
    void changeRollNumber(){  
        rollNumber=151500100;  
    }  
    public static void main(String args[]){  
        Student obj=new Student();  
        //obj.changeRollNumber(); Compile Time Error  
    }
```

# Java final method

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

# Java final method

```
class Arithmetic{  
int x; //instance variable  
Arithmetic(int x){  
this.x=x;  
}  
final void printTable(){  
int result; //local variable  
for(int y=1;y<=10;y++){  
result=x*y; //Calculating value in each iteration  
System.out.println(x + " * " + y + " = " + result);  
} //End of for loop  
} // End of printTable method
```

# Java final method

```
void printFactorial(){  
    int f=1, y=1;  
    while(y<=x)  
    {  
        f=f*y; //Calculating value at each iteration  
        y++;  
    }  
    System.out.println("Factorial is" + f);  
}  
} //End of class Arithmetic
```

Note that in the above class, the child class can override printFactorial() method, but cannot override printTable() method.



# Java final method

```
class A extends Arithmetic{  
    void printFactorial(){  
        int f=1; //local variable  
        for(int y=2; y<=x; y++){  
            f=f*y; //Calculating value in each iteration  
        } //End of for loop  
        System.out.println("Factorial is" + f);  
    } // End of printFactorial method  
} //End of class A
```

# Java final class

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

Q) Is final method inherited?

Yes, final method is inherited but you cannot override it.

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable. It can be initialized only in constructor.

# Java static keyword

- The **static keyword** in java is used for memory management mainly. The static keyword belongs to the class than instance of the class.
- The static can be:
  - variable (also known as class variable)
  - method (also known as class method)
  - package
  - block
  - nested class

# Java static variable

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

# Java static variable

```
class Student{  
    int rollno;  
    String name;  
    static String college = "GLA University"; //Class variable  
    Student(int r,String n){ rollno = r; name = n; }  
    void display (){  
        System.out.println(rollno+" "+name+" "+college);  
    }  
    public static void main(String args[]){  
        Student s1 = new Student(111,"Karan");  
        Student s2 = new Student(222,"Aryan");  
        s1.display();  
        s2.display();  
    }  
}
```

# Program for Practice

Q. Develop a program in java that prints the number of objects created so far as soon as a new object is created.

# 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.

# Java static method

```
class Student{  
    int rollno;  
    String name;  
    static String college = "GLA University";  
    static void change(String col){  
        college = col;  
    }  
    Student(int r, String n){ rollno = r; name = n; }  
    void display (){  
        System.out.println(rollno+" "+name+" "+college);  
    }  
}
```



# Java static method

```
public static void main(String args[]){  
    Student s1 = new Student (111,"Karan");  
    Student s2 = new Student (222,"Aryan");  
    Student s3 = new Student (333,"Sonoo");  
    s1.change("GLA University, Mathura");  
    s1.display();  
    s2.display();  
    s3.display();  
    }  
}
```

# Java static method

```
class Calculate{  
    static int cube(int x){  
        return x*x*x;  
    }  
    public static void main(String args[]){  
        int result1=Calculate.cube(5);  
        System.out.println(result1);  
        int result2=Calculate.cube(6);  
        System.out.println(result2);  
    }  
}
```

# Restrictions for static method

- There are two main restrictions for the static method. They are:
  - The static method can not use non static data member or call non-static method directly.
  - this and super cannot be used in static context.

# Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of class loading.

```
class A{  
static{  
    System.out.println("Printing in static block");  
} //End of static block  
public static void main(String args[]){  
    System.out.println("Printing in main method");  
}  
}
```

# Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- The signature of the methods with the same name should be different. It can be made possible by:
  - Changing the number of arguments
  - Changing the data types of arguments
  - Changing the order of data types of arguments

# Method Overloading in Java

```
public class Adder{  
static int add(int a, int b)  
{  
return a + b;  
}  
static int add(int a, int b, int c)  
{  
return a + b + c;  
}  
} //End of Adder class
```

# Method Overloading in Java

```
class Test{  
public static void main(String[] args){  
  
    System.out.println(Adder.add(11,11));  
    System.out.println(Adder.add(11,11,11));  
}  
}
```

Output:

22

33

# Method Overloading in Java

Q. Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only.



# Method Overloading in Java

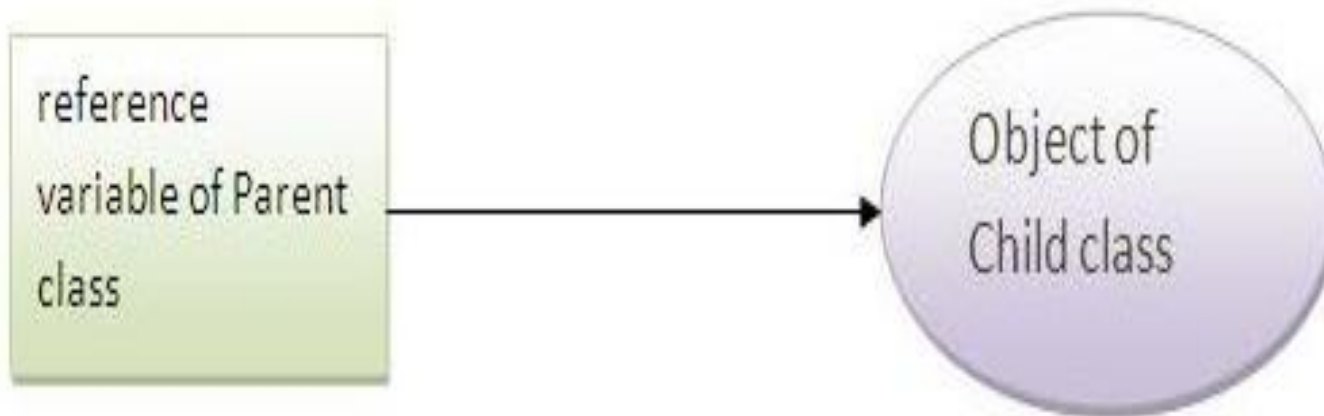
```
class Test{  
public static void main(String[] args){  
    System.out.println("main with String[]");  
}  
public static void main(String args){  
    System.out.println("main with String");  
}  
public static void main(){  
    System.out.println("main without args");  
}  
} //End of class
```

# Runtime Polymorphism in Java

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.

# Runtime Polymorphism in Java

- When reference variable of Parent class refers to the object of Child class, it is known as upcasting.



# Runtime Polymorphism in Java

```
class Bank{  
float getRateOfInterest(){return 0;}  
}  
class SBI extends Bank{  
float getRateOfInterest(){return 8.4f;}  
}  
class ICICI extends Bank{  
float getRateOfInterest(){return 7.3f;}  
}  
class AXIS extends Bank{  
float getRateOfInterest(){return 9.7f;}  
}
```

# Runtime Polymorphism in Java

```
class Test{  
public static void main(String args[]){  
    Bank b;  
    b=new SBI();  
    System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());  
  
    b=new ICICI();  
    System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());  
  
    b=new AXIS();  
    System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());  
}  
}
```

# Static class in Java

- Java allows us to define a class within another class.
- Such a class is called a nested class.
- The class which enclosed nested class is known as Outer class.
- In java, we can't make Top level class static.
- ***Only nested classes can be static.***
- Non-static nested class is also called Inner Class.

```
class OuterClass{
    private static String msg1 = "We Love Java";
    private String msg2="We Love Programming";
    public static class NestedStaticClass{
        public void printMessage() {
            System.out.println(msg1);
        }//End of method
    }//End of static class
    public class InnerClass{
        public void display(){
            System.out.println(msg1 + " " + msg2);
        }//End of method
    }//End of inner class
}//End of outer class
```

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

OuterClass outer = new OuterClass();
OuterClass.InnerClass inner1 = outer.new InnerClass();
inner1.display();

OuterClass.InnerClass inner2= new OuterClass().new InnerClass();
inner2display();
}
}
```



# Summary: Static class in Java

- In java, we can't make Top level class static.
- ***Only nested classes can be static.***
- Non-static nested class is also called Inner Class.
- Only static members of Outer class is directly accessible in nested static class.
- Both static and non-static members of Outer class are accessible in Inner class.
- Static inner class can contain static and non-static methods.
- Non Static inner Class can not contain static method. Static method can be declared in top level class.

# 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.
- 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 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.
- It adds **public, static and final** keywords before data members.

# interface Example

```
interface printable{  
    void print();  
} //End of interface  
class A implements printable{  
    public void print(){  
        System.out.println("Hello");  
    }  
    public static void main(String args[]){  
        A obj = new A();  
        obj.print();  
    }  
}
```

## Example 2

(Also Refer to slide 53-54)

```
interface ThreeDimension{  
double calVolume();  
}
```

## Example 2

(Also Refer to slide 53-54)

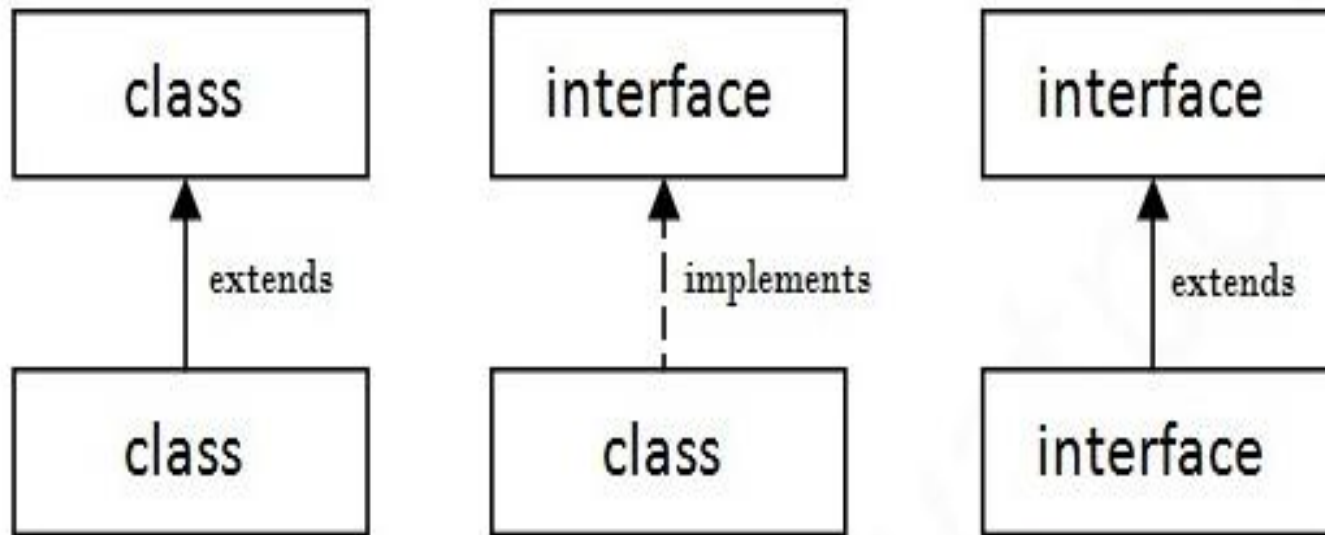
```
class Cuboid extends Rectangle implements ThreeDimension{  
double length, volume;  
Cuboid(double width, double height, double length)  
{  
super(width,height);  
this.length=length;  
}  
public double calVolume(){  
volume=length*width*height;  
return volume;  
}  
} //End of Cuboid class
```

## Example 2

(Also Refer to slide 53-54)

```
class Test{  
public static void main(String args[])  
{  
Cuboid obj = new Cuboid(10,20,30.2);  
System.out.println("Area of base="+ obj.calArea());  
System.out.println("Volume of Cuboid=" + obj.calVolume());  
System.out.println("Perimeter of base="+ obj.calPerimeter());  
}  
} //End of Test class
```

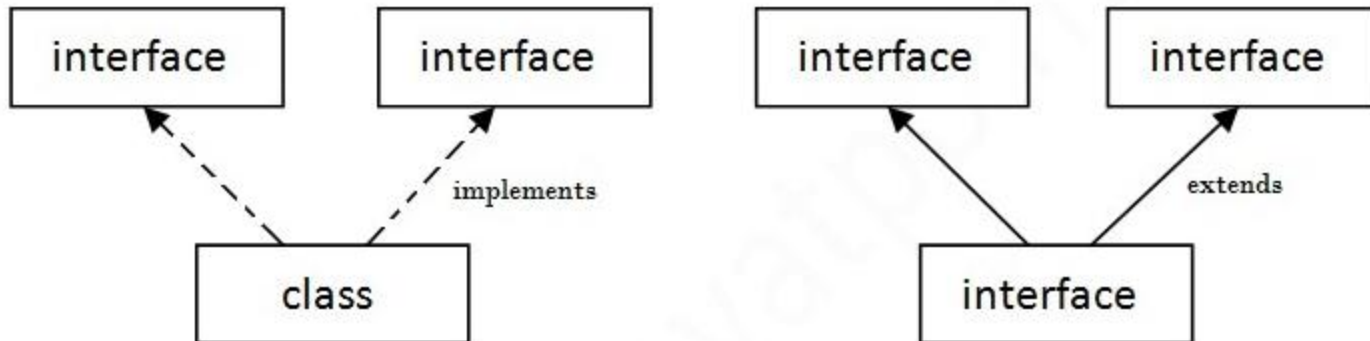
# Relationship between classes and interfaces





# Multiple inheritance in Java by interface

- A class can inherit a single class and one or more interfaces, thus implementing multiple inheritance in Java.
- A class can never inherit two classes.
- Hence, multiple inheritance is not possible without implementing an interface.



**Multiple Inheritance in Java**

# Understanding the concept

Question: Multiple inheritance is not supported through class in java but it is possible by interface, why?

Answer:

Multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

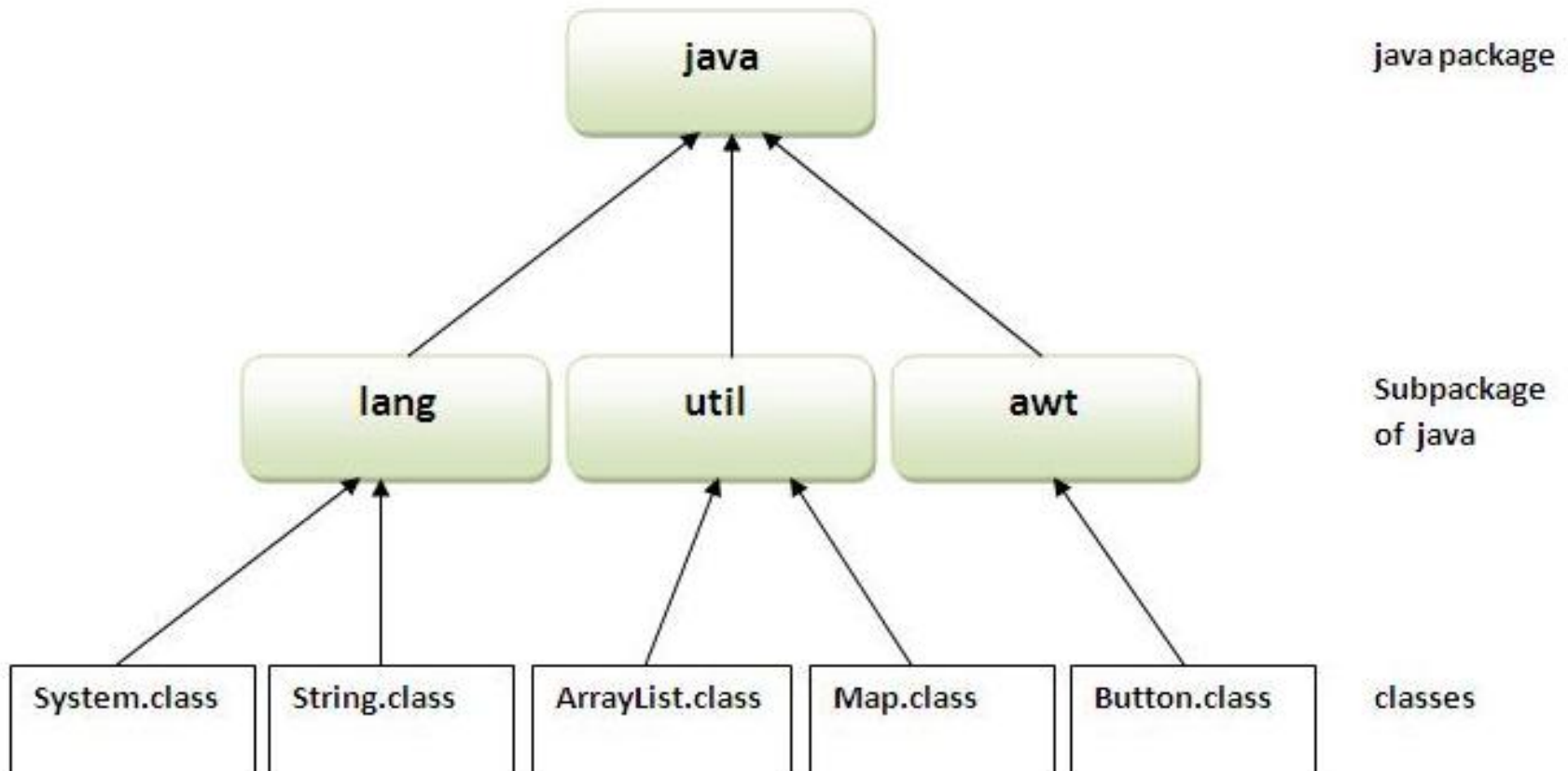
# Abstract class v/s Interface

Abstract class	Interface
1) Abstract class can <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. It can have <b>static methods</b> also.
2) Abstract class <b>doesn't support multiple inheritance.</b>	Interface <b>supports multiple inheritance.</b>
3) Abstract class <b>can have final, non-final, static and non-static variables.</b>	Interface has <b>only static and final variables.</b>
4) Abstract class <b>can provide the implementation of interface.</b>	Interface <b>can't provide the implementation of abstract class.</b>
5) The <b>abstract keyword</b> is used to declare abstract class.	The <b>interface keyword</b> is used to declare interface.

# 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 and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Advantages:
  - 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 Package



# Creating a Java Package

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

# Compiling and Executing a Java Package

```
javac -d directory javafilename  
java packagename.classname
```

For example:

**To Compile:** `javac -d . Simple.java`

**To Run:** `java mypack.Simple`