

Unit - 3

Inheritance

Interfaces

Inheritance Basics

- What is an Ostrich?
- What is a Duck?
- What is a Penguin?
- Who are you?
 - Student → person → Mammal → Animal
- Sedan → Car → LandVehicle → Vehicle

Inheritance Basics

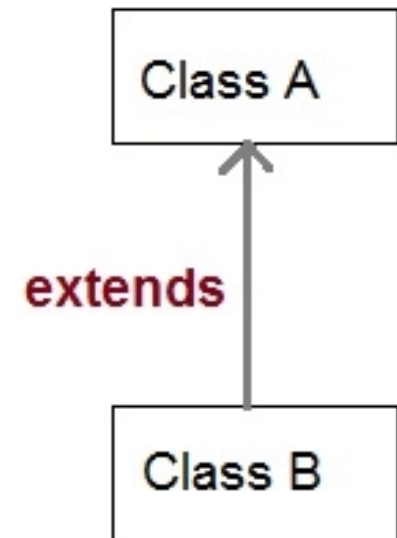
- One of the three fundamental principles of OOP.
- Allows hierarchical classification.
 - Create a general class that defines traits common to a set of related items. (Generalization)
 - This class can then be inherited by other, more specific classes each adding those things that are unique to it. (Specialization)
 - A class that is inherited is called superclass, base class or parent class.
 - The class that does the inheritance is called a subclass, derived class or child class.

- If A is a superclass, the keyword **extends** is used to create a subclass of A.

```
class subclass-name extends superclass-name  
{  
  
}
```



- **Example:**

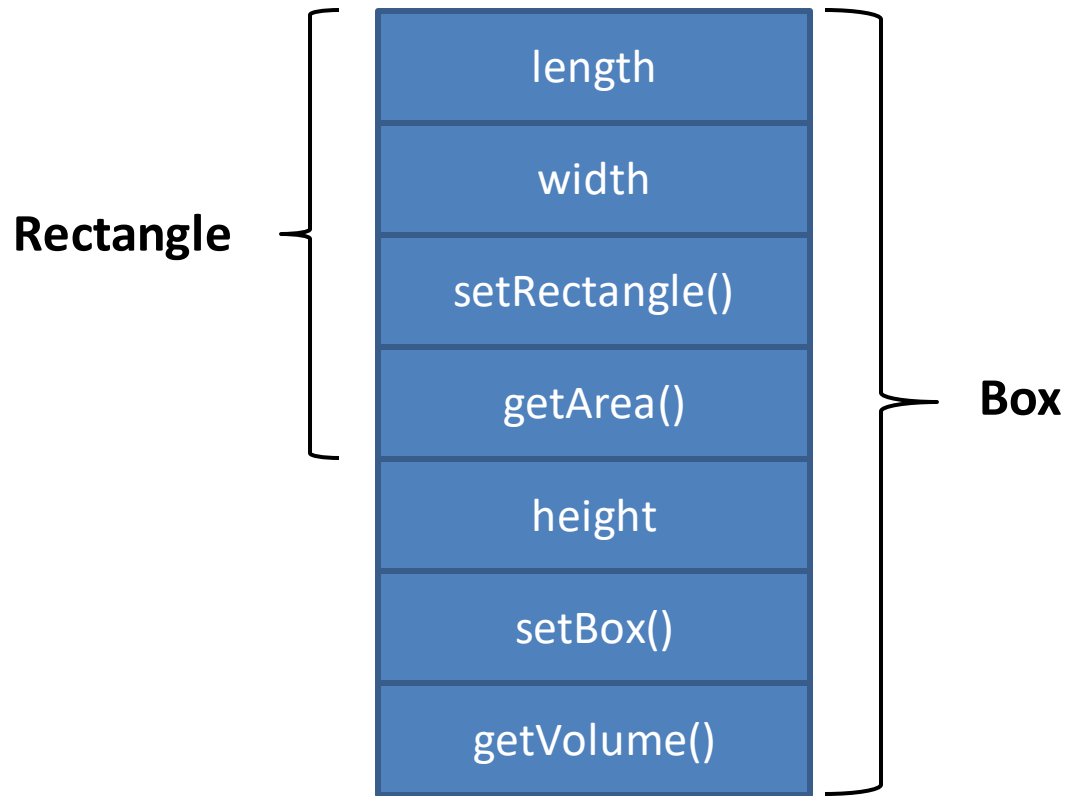
```
class A {  
    //Members and methods declarations  
}  
  
class B extends A {  
    //Members and methods from A are inherited  
    //Members and methods declarations of B  
}
```



Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.
- A class member that has been declared as private will remain private to its class.
- It is not accessible by any code outside its class, including subclasses.

```
class Rectangle {  
    private double length, width;  Cannot be accessed by Box class  
    void setRectangle(double length, double width) {  
        this.length = length;  
        this.width = width;  
    }  
    double getArea() {  
        return length * width;  
    }  
}  
 Getters or Accessor methods are used by the  
subclass and other classes to obtain value of private  
data members  
  
class Box extends Rectangle {  
    private double height;  
    void setBox(double length, double width, double height) {  
        setRectangle(length,width);  
        this.height = height;  
    }  
    double getVolume() {  
        return getArea() * height;  
    }  
}  
  
public class InheritExample {  
    public static void main(String[] args) {  
        Box room = new Box();  
        room.setBox(12.5, 10.5, 9.5);  
        System.out.println("Volume is " + room.getVolume());  
    }  
}
```



Constructors and Inheritance

- It is possible to have constructors for both superclass and subclasses.
 - Which constructor is responsible for building an object of the subclass – the one in superclass, the one in subclass, or both?
 - Answer: The constructor for the superclass constructs the superclass portion of the object, and the constructor for the subclass constructs the subclass part.
- In a class hierarchy, constructors are called in the order of derivation, from superclass to subclass.


```
class Person {
    Person() {
        System.out.println("Constructor of person class");
    }
}

class Employee extends Person {
    Employee() {
        System.out.println("Constructor of employee class");
    }
}

class PermanentEmployee extends Employee {
    PermanentEmployee() {
        System.out.println("Constructor of permanent employee class");
    }
}

class ConstructorInheritanceDemo {
    public static void main(String args()) {
        PermanentEmployee pObj = new PermanentEmployee();
    }
}
```

Output:

Constructor of person class

Constructor of employee class

Constructor of permanent employee class

- When both the superclass and the subclass define constructors, the process is a bit more complicated because both the superclass and subclass constructors must be executed.
- In this case, you must use the keyword `super`.
- `super` has two uses:
 - It can be used to call the superclass constructor.
 - It can be used to access a member of the superclass that has been hidden by a member of the subclass.

Using super to call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of super:
 - `super(parameter_list);`
 - Here `parameter_list` specifies any parameters needed by the constructor of the superclass.
 - `super()` must always be the first statement executed inside a subclass constructor.

```
class Person {
    String FirstName, LastName;
    Person(String fName, String lName) {
        FirstName = fName;
        LastName = lName;
    }
}

class Student extends Person {
    int id;
    String standard, classTeacher;

    Student(String fName, String lName, int sId, String stnd, String cTeacher) {
        super(fName,lName);
        id = sId;
        standard = stnd;
        classTeacher = cTeacher;
    }
}

class SuperKeywordForConstructorDemo {
    public static void main(String args[]) {
        Student sObj = new Student("Amit","Patil",1,"II- B","RMK");
    }
}
```

```

class TwoDShape {
    double width, height;
    TwoDShape() {
        width = height = 0;
    }
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
    TwoDShape(double x) {
        width = height = x;
    }
}

class Triangle extends TwoDShape {
    Triangle() {
        super();
    }
    Triangle(double w, double h) {
        super(w, h);
    }
    Triangle(double x) {
        super(x);
    }
}

```

Any form of constructor defined by the superclass can be called by using the keyword `super()`. The constructor executed will be the one that matches the arguments.

Using super to Access Superclass Members

- `super.member`
 - Here, member can be either a method or an instance variable.
 - This is useful in situations where member names of a subclass hide members by the same name in the superclass.

```
class A {
    int i;
}
class B extends A {
    int i;
    B(int a, int b) {
        super.i= a;
        i = b;
    }
    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
    }
}
```

Output:

i in superclass: 1

i in subclass: 2

Method Overriding

- In a class hierarchy, when a method in a subclass has the **same return type, name and signature** as a method in its superclass, then the method is said to **override** the method in the superclass.
 - When an overridden method is called from within a subclass, it will always refer to the version of the method defined by that subclass.
 - The version of the method defined by the superclass will be hidden.
 - Overriding provides the ability to define a behavior that is specific to the subclass type.

Superclass

```
public int foo() {  
    // default implementation  
}
```



Subclass

```
public int foo() {  
    // new implementation  
}
```

- Method overriding is one of the way by which java achieves Run Time Polymorphism.
 - The version of a method that is executed will be determined by the object that is used to invoke it.
 - If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.
 - In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

Method overriding

```
public class Animal
{
    public void makeSound()
    {
        System.out.println("the animal makes sounds");
    }
}
public class Dog extends Animal
{
    public void makeSound()
    {
        System.out.println("the dog barks");
    }
}
```

Method overriding

```
class Bank
{
    int getRateOfInterest () {return 0;}
}
class SBI extends Bank
{
    int getRateOfInterest () {return 8;}
}
class ICICI extends Bank
{
    int getRateOfInterest () {return 7;}
}
class AXIS extends Bank
{
    int getRateOfInterest () {return 9;}
}
```

Overridden Methods Support Polymorphism

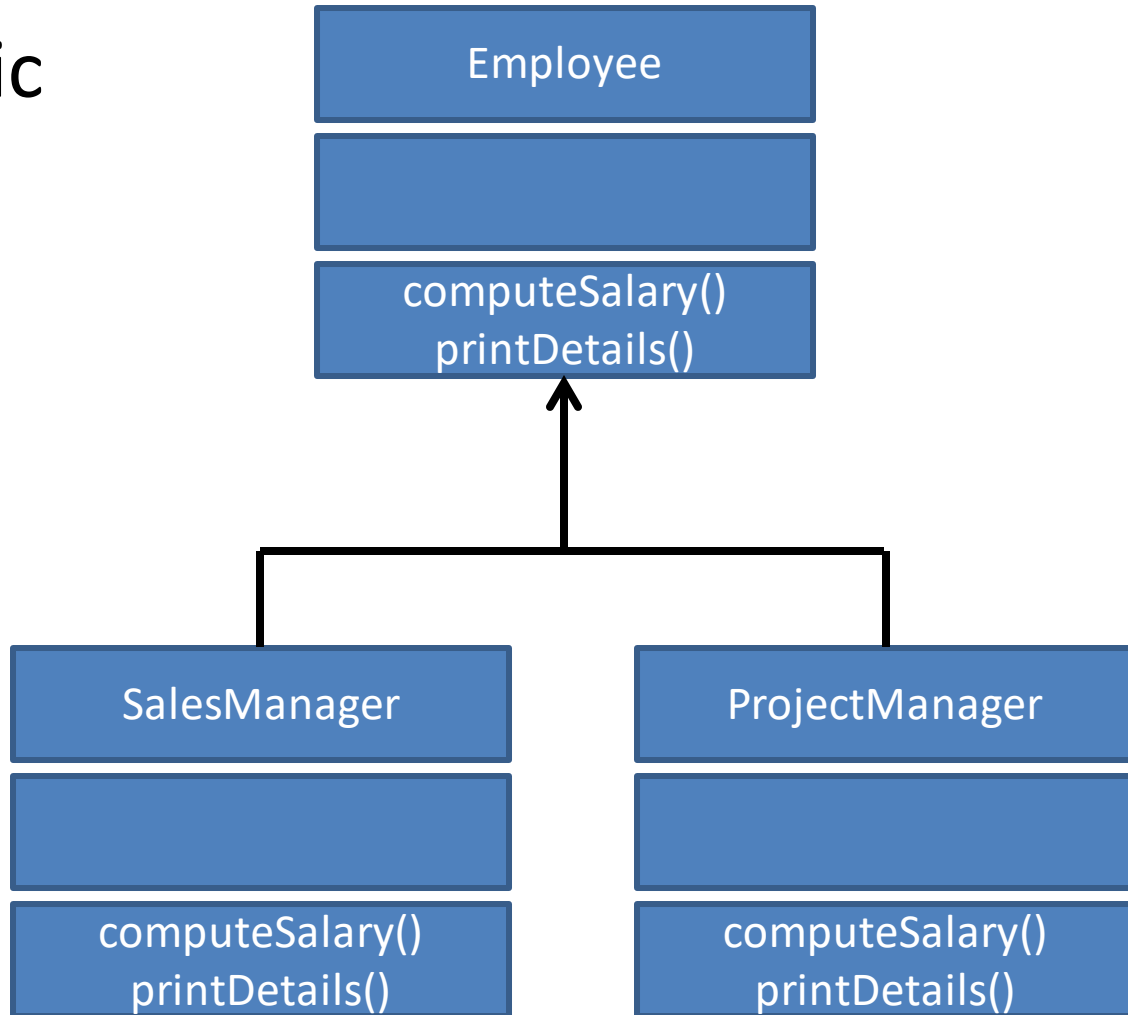
- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch
 - It is a mechanism by which a call to an overridden method is resolved at run time rather than compile time.
 - This is how Java implements run-time polymorphism.

When an overridden method is called through superclass reference.....

- A superclass reference variable can refer to a subclass object.
 - Java uses this fact to resolve calls to overridden methods at run time.
 - Java determines which version of that method to execute based on the type of the object being referred to at the time of the call.
 - This determination is done at run-time.
 - It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

Example for Dynamic Method Dispatch

```
class Employee {  
    static double basic = 20000;  
    String empName;  
    int empID;  
    Employee(String name, int id) {  
        empName = name;  
        empID = id;  
    }  
    void computeSalary() {  
    }  
    void printDetails() {  
        System.out.println("Name:" + empName);  
        System.out.println("Employee ID: " + empID);  
    }  
}
```



```
class SalesManager extends Employee {  
    static double da = 0.65, hra = 0.85;  
    static int comm = 10000;  
    double salary;  
    SalesManager(String name, int id) {  
        super(name, id);  
    }  
    void computeSalary() {  
        salary = basic + basic * da + basic * hra + comm;  
    }  
    void printDetails() {  
        super.printDetails();  
        System.out.println("Salary: " + salary);  
    }  
}
```

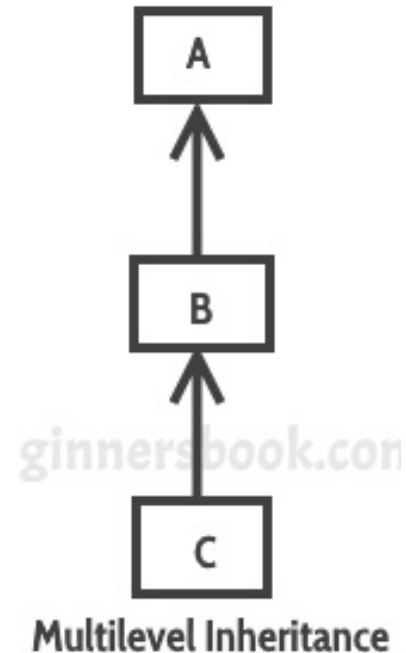


```
class ProjectManager extends Employee {  
    static double da = 0.70, hra = 0.9;  
    static int allowance = 15000;  
    double salary;  
    ProjectManager(String name, int id) {  
        super(name, id);  
    }  
    void computeSalary() {  
        salary = basic + basic * da + basic * hra + allowance;  
    }  
    void printDetails() {  
        super.printDetails();  
        System.out.println("Salary: " + salary);  
    }  
}
```

```
public class OverridingDemo {  
    public static void main(String[] args) {  
        Employee e1;  
        SalesManager s1 = new SalesManager("Sachin T", 101);  
        ProjectManager p1 = new ProjectManager("Ramesh", 201);  
        s1.computeSalary();  
        s1.printDetails();  
        p1.computeSalary();  
        p1.printDetails();  
        e1 = s1;  
        e1.computeSalary();  
        e1 = p1;  
        e1.computeSalary();  
    }  
}
```

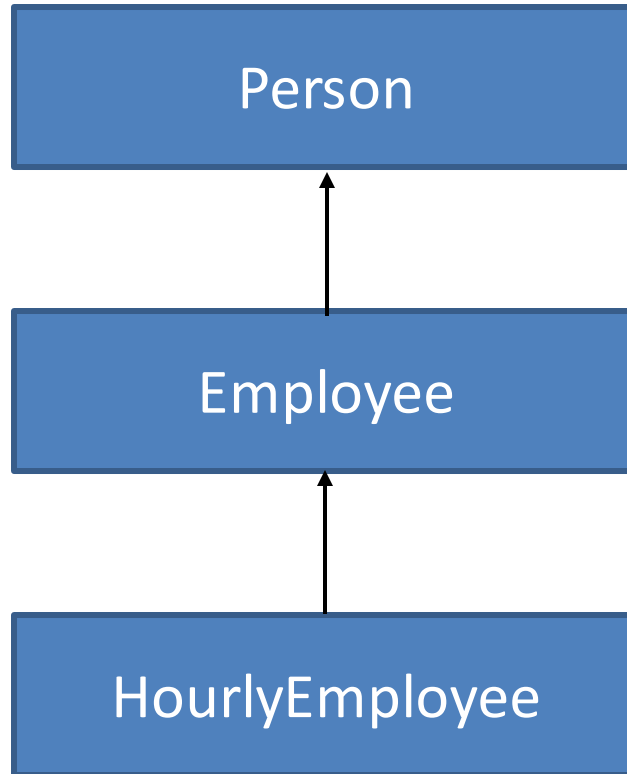
Creating a Multilevel Hierarchy

- The relationships of objects or classes through inheritance give rise to a hierarchy.
- In multilevel inheritance, a subclass is inherited from another subclass.
 - It is not uncommon that a class is derived from another derived class as shown in the figure.
 - The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.
 - The class B is known as intermediate base class since it provides a link for the inheritance between A and C.
 - In such a hierarchy, `super()` always refers to the constructor in the closest superclass.



Syntax of Multilevel Inheritance

```
class A {  
    //Do something  
}  
class B extends A {  
    //Do something  
}  
class C extends B {  
    //Do something  
}
```



```
class person {  
    private String name;  
    person(String s) {  
        setName(s);  
    }  
    public void setName(String s) {  
        name = s;  
    }  
    public String getName() {  
        return name;  
    }  
    public void display() {  
        System.out.println("Name = " + name);  
    }  
}
```

```
class Employee extends person {  
    private int empid;  
    Employee(String sname,int id) {  
        super(sname);  
        setEmpid(id);  
    }  
    public void setEmpid(int id) {  
        empid = id;  
    }  
    public int getEmpid() {  
        return empid;  
    }  
    public void display() {  
        super.display();  
        System.out.println("Empid = " + empid);  
    }  
}
```

```
class HourlyEmployee extends Employee {  
    private double hourlyRate;  
    private int hoursWorked;  
    HourlyEmployee(String sname,int id,double hr,int hw) {  
        super(sname,id);  
        hourlyRate = hr;  
        hoursWorked = hw;  
    }  
    public double GetGrosspay() {  
        return (hourlyRate * hoursWorked);  
    }  
    public void display() {  
        super.display();  
        System.out.println("Hourly Rate = " + hourlyRate);  
        System.out.println("Hours Worked = " + hoursWorked);  
        System.out.println("Gross pay = " + GetGrosspay());  
    }  
}
```



```
class MultilevelInheritance {  
    public static void main(String[] args) {  
        HourlyEmployee emp = new HourlyEmployee(  
                                                    "Kapil Sharma", 1, 1000, 28);  
        emp.display();  
    }  
}
```

When are Constructors Executed?

- In a class hierarchy, constructors are called in the order of derivation, from superclass to subclass.

Why is that so?

- Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and a prerequisite to any initialization performed by the subclass.
- Since `super()` must be the first statement executed in a subclass' constructor, this order is the same whether or not `super()` is used.
- If `super()` is not used, then the default constructor of each superclass will be executed.

```
class A
{
    A()
    { System.out.println("Inside A's constructor."); }
}
class B extends A
{
    B()
    { System.out.println("Inside B's constructor."); }
}
class C extends B
{
    C() { System.out.println("Inside C's constructor."); }
}
class CallingCons
{
    public static void main(String args[])
    {
        C c = new C();
    }
}
```

Superclass References and Subclass Objects

- In Java, type compatibility is strictly enforced. Hence, a reference variable for one class type cannot refer to an object of another class type.
- However, an object reference variable can refer to objects of its type.

```
class A {  
    int a;  
    A(int i) { a= i; }  
}  
class B {  
    int a;  
    B(int i) { a= i; }  
}  
class Incompatible {  
    public static void main(String [] args) {  
        A ob1 = new A(10);  
        A ob2;
```

```
        B ob3 = new B(10);  
        ob2 = ob1; // OK, both are same type  
        ob2 = ob3; // Error, not of same type
```

Even though class A and class B are physically same, it is not possible to assign to an A variable a reference to a B object because they have different types.

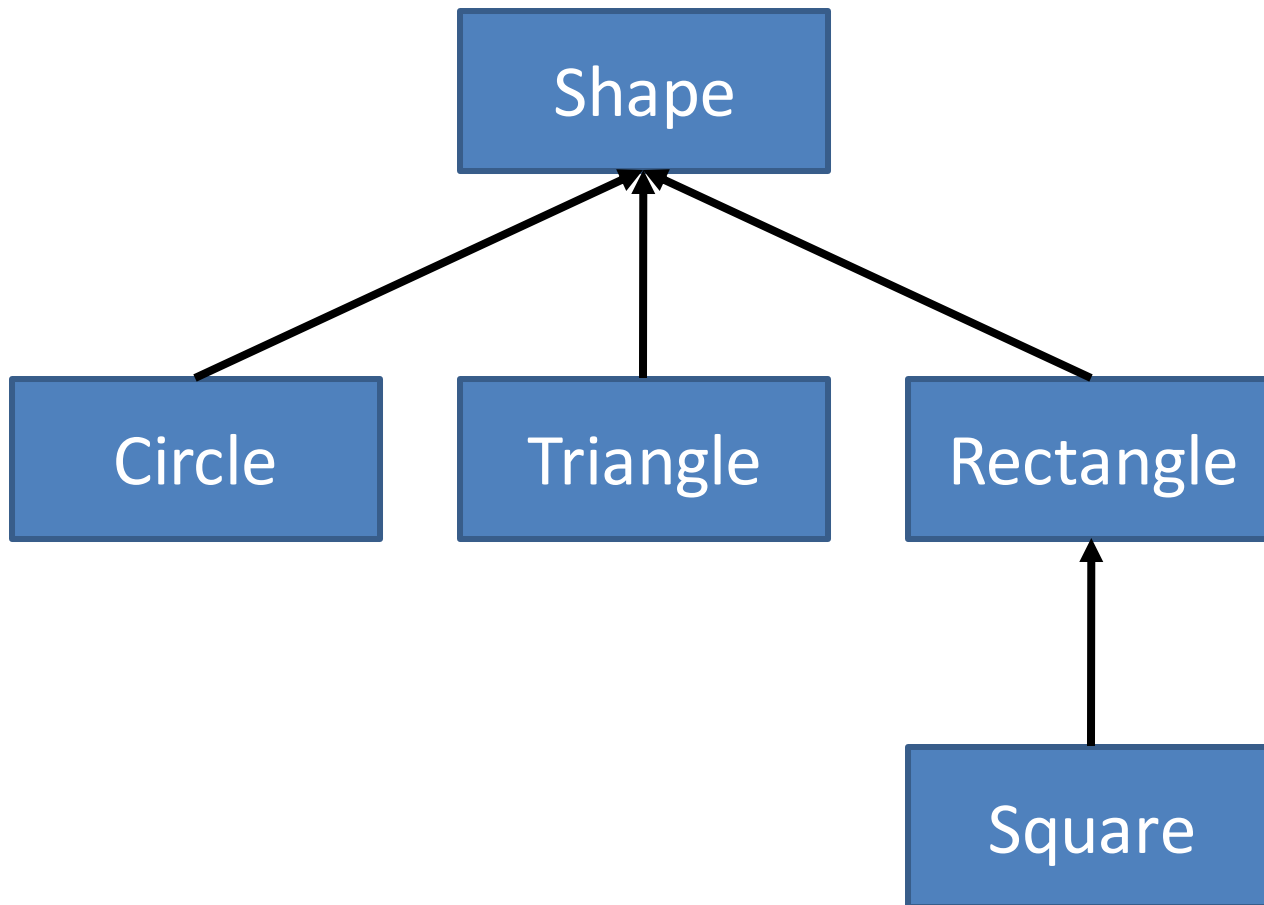
Four ways to assign superclass and subclass references to variables of superclass and subclass types

1. Assigning a superclass reference to a superclass variable is straightforward.
2. Assigning a subclass reference to a subclass variable is straightforward.
3. Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an object of its superclass*.
 - The superclass variable can be used to refer only to superclass members.
 - If this code refers to subclass-only members through the superclass variable, the compiler reports errors.
4. Attempting to assign a superclass reference to a subclass variable is a compilation error.

```
class Person {  
    String name;  
    int age;  
    Person(String n, int a) {  
        name = n;  
        age = a;  
    }  
    void showDetails() {  
        System.out.println("Name:" + name);  
        System.out.println("Age:" + age);  
    }  
}
```

```
class Student extends Person {  
    int rollNo;  
    int phy, chem, mat;  
    Student(String n, int a, int rno, int m1, int m2, int m3) {  
        super(n, a);  
        rollNo = rno;  
        phy = m1;  
        chem = m2;  
        mat = m3;  
    }  
    void showDetails() {  
        super.showDetails();  
        System.out.println("Roll Number: " + rollNo);  
        System.out.println("Physics: " + phy);  
        System.out.println("Chemistry: " + chem);  
        System.out.println("Maths: " + mat);  
    }  
    void showAvg() {  
        float avg = (float) (phy + chem + mat) / 2.0f;  
        System.out.println("Average: " + avg);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student s1 = new Student("Ramesh", 21, 1, 23, 24, 21);  
        Person p1 = s1;  
        p1.showDetails();  
        p1.showAvg(); //ERROR: cannot find symbol  
        ((Student)p1).showAvg();  
        s1.showAvg();  
        Person p2 = new Person("Sachin", 32);  
        Student s2 = p2; //ERROR: incompatible types  
        Student s3 = (Student) p2; // OK.  
        s2.showAvg();  
        s2.showDetails();  
    }  
}
```

Base class - Shape

```
class Shape
{
    float area, perimeter;
    String type;
    void display()
    {
        System.out.println("I am a " + type);
        System.out.println("Area is " + area);
        System.out.println("Perimeter is " +
                               perimeter);
    }
}
```

Derived class - Circle

```
class Circle extends Shape
{
    float radius;
    Circle(float rad)
    {
        type = "Circle";
        radius = rad;
    }
    void computeArea()
    {
        area = 3.142f * radius * radius;
    }
    void computePerimeter()
    {
        perimeter = 2.0f * 3.142f * radius;
    }
}
```

Derived class - Triangle

```
class Triangle extends Shape
{
    float a, b, c;
    Triangle(float s1, float s2, float s3)
    {
        type = "Triangle";
        a = s1; b = s2; c = s3;
    }
    void computeArea()
    {
        float s = (a + b + c) / 2.0f;
        area = (float) Math.sqrt(s*(s - a)*(s - b)*(s - c));
    }
    void computePerimeter()
    {
        perimeter = a + b + c;
    }
}
```

Derived class - Rectangle

```
class Rectangle extends Shape
{
    float length, width;
    Rectangle(float len, float wid)
    {
        type = "Rectangle";
        length = len;
        width = wid;
    }
    void computeArea()
    {
        area = length * width;
    }
    void computePerimeter()
    {
        perimeter = 2.0f * (length + width);
    }
}
```

Using abstract classes

- A class that is declared using “**abstract**” keyword is known as abstract class.
- It can have abstract methods(methods without body) as well as concrete methods (regular methods with body).
- A normal class(non-abstract class) cannot have abstract methods.
- An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it.

Abstract class declaration

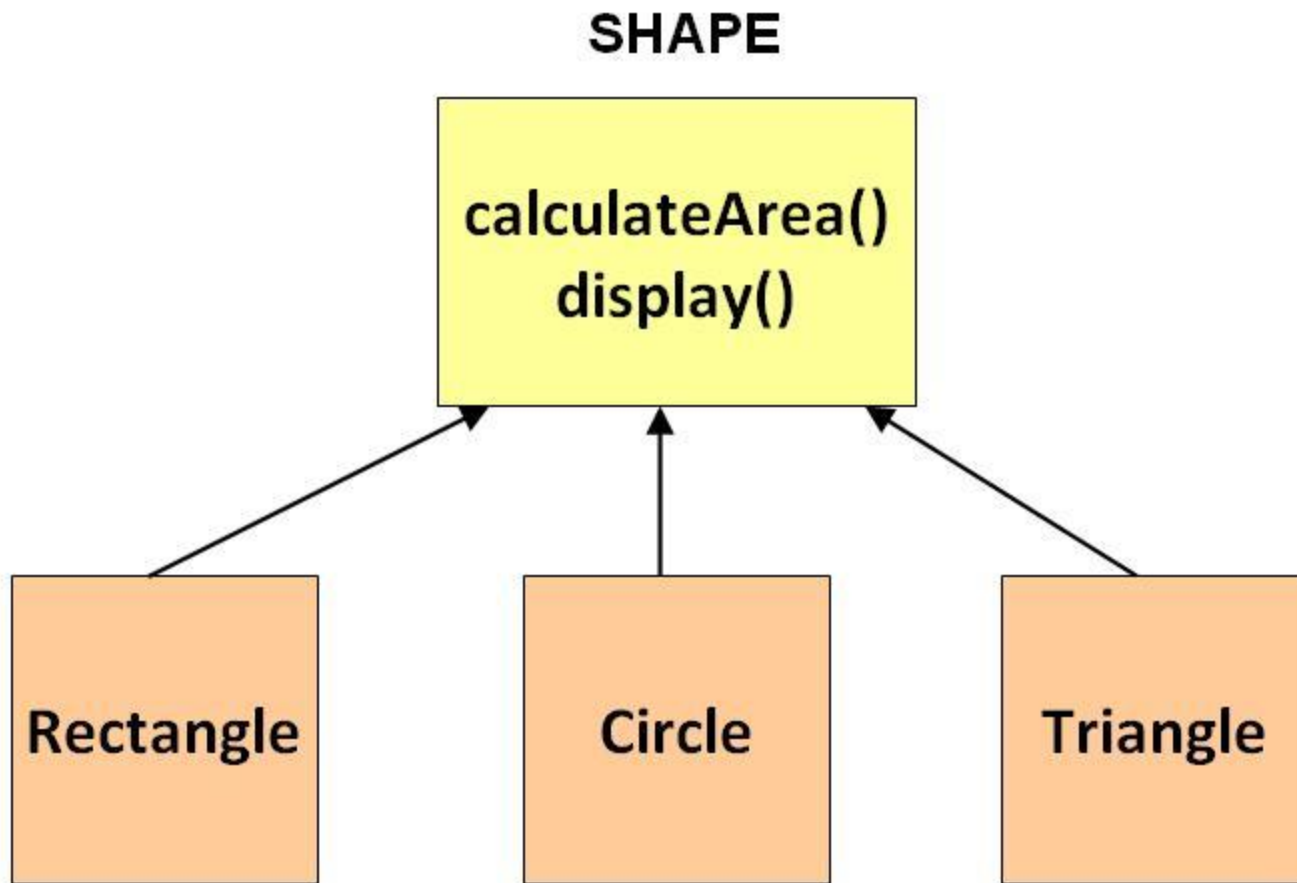
An abstract class outlines the methods but not necessarily implements all the methods.

```
// Declaration using abstract keyword
abstract class A
{
    // This is abstract method
    abstract void myMethod();

    // This is concrete method with body
    void anotherMethod()
    {
        //Does something
    }
}
```

Rules for abstract classes

- A class derived from the abstract class **must implement** all those methods that are declared as abstract in the base class.
- Abstract class cannot be instantiated.
- If a derived class does not implement all the abstract methods of abstract base class, then the **derived class must need to be declared abstract as well.**

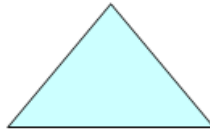


- How these **objects** will look in a practical world?

Rectangle obj = new Rectangle();



Triangle obj = new Triangle();



Shape obj = new Shape();



???

Observe that the Shape class serves our goal of achieving **inheritance and polymorphism**. But it was not built to be instantiated. Such classes are labeled **abstract**.

Shape class Revisited...

```
abstract class Shape
```

```
{  
    float area, perimeter;  
    String type;  
    void display()  
    {  
        System.out.println("I am a " + type);  
        System.out.println("Area is " + area);  
        System.out.println("Perimeter is " + perimeter);  
    }  
}
```

```
abstract void computeArea();
```

```
abstract void computePerimeter();
```

```
}
```

**subclasses must
implement these
two methods**

- Reference of an abstract class can point to objects of its subclasses thereby achieving **run-time polymorphism**.
 - Shape obj = new Rectangle(4, 5); OR
 - Shape obj;
 - Rectangle r1 = new Rectangle(4, 5);
 - obj = r1;
 - obj1.computeArea();
 - obj1.computePerimeter();
 - obj1.display();

Using final with Inheritance

- The keyword final has three uses.
 - It can be used to create the equivalent of a named constant. We cannot change the value of final variable.

```
class Bike
{
    final int SPEEDLIMIT=90;
}
```

2. It can be used to disallow a method from being overridden.

- A final method is inherited but you cannot override it.

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

Can we declare a class as both abstract and final?

3. It can be used to prevent a class from being inherited.
Declaring a class as final implicitly declares all of its methods as final, too.

```
final class Bike
{
    ... ..
}
class Honda1 extends Bike // Error!
{
    void run()
    {
        System.out.println("running at 100kmph");
    }
}
```

The Object class

- Mother of all Java classes!!
 - Java defines one special class called Object that is an implicit superclass of all other classes.
 - A reference variable of type Object can refer to an object of any other class.
 - Since arrays are implemented as classes, a variable of type Object can also refer to any array.

Methods of Object class

- `Object clone()`
 - Creates a new object that is the same as the object being cloned.
- `boolean equals(Object ob)`
 - Determines whether one object is equal to another.
- `void finalize()`
 - Called before an unused object is recycled.
- `class<?> getClass()`
 - Obtains the class of an object at run time.
- `String toString()`
 - Returns a string that describes the object
- `int hashCode()`
 - Returns the hash code associated with the invoking object.
- `wait()`, `notify()` are used multithreading.

finalize

- This method is called just before an object is garbage collected.
- It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object.
- We should override `finalize()` method to dispose system resources, perform clean-up activities and minimize memory leaks.
- For example before destroying Servlet objects web container, always called `finalize` method to perform clean-up activities of the session.

finalize()

```
public class Test {  
    public static void main(String[] args) {  
        Test t = new Test();  
        System.out.println(t.hashCode());  
        t = null;  
        // calling garbage collector  
        System.gc();  
        System.out.println("end");  
    }  
    @Override  
    protected void finalize() {  
        System.out.println("finalize method called");  
    }  
}
```

clone()

- **clone()** – returns a new object that is exactly the same as this object
- **equals()** - Compares the given object to “this” object

class Rectangle implements Cloneable

```
{
    float length, width;
    Rectangle(float len, float wid)
    {
        type = "Rectangle";
        length = len;
        width = wid;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
    public static void main(String[] args) throws CloneNotSupportedException {
        r1 = Rectangle(4, 5);
        Rectangle r2 = (Rectangle)r1.clone(); // Different objects
        Rectangle r3 = r1; // r3 holds the same object reference and hence has the same object
        if(r2.equals(r1)) // False
        if(r3.equals(r1)) // True
    }
}
```

getClass()

- Returns the class object of “this” object

```
{
```

```
    Object obj = new String("Hello World!");
```

```
    Class c = obj.getClass();
```

```
    Rectangle r1 = new Rectangle(4, 5);
```

```
    Class d = r1.getClass();
```

```
    System.out.println("Class of Object obj is : " + c.getName());
```

```
    System.out.println("Class of Object obj is : " + d.getName());
```

```
}
```

Output:

Class of Object obj is : java.lang.String

Class of Object obj is : shapeclass.Rectangle

hashCode()

- Returns a hash value that is used to search object in a collection.
 - For every object, JVM generates a unique number which is hashCode. It returns distinct integers for distinct objects (not the address of object!)

```
{  
    ShapeClass t = new ShapeClass();  
    System.out.println(t.hashCode());  
}
```

toString()

- Provides String representation of an Object and used to convert an object to String.
 - Returns class name, then @, then unsigned hexadecimal representation of the hash code of the object
 - `System.out.println(r1.toString());` returns
 - `shapeclass.Rectangle@7852e922`
 - Whenever we try to print any Object reference, then internally `toString()` method is called.
 - `Student s = new Student();`
 - `// Below two statements are equivalent`
 - `System.out.println(s);`
 - `System.out.println(s.toString());`

Interfaces

Interface Fundamentals

- Interface defines a set of methods that will be implemented by a class.
 - Syntactically, interfaces are similar to abstract classes, except that no method can include a body.
 - Interface does not provide implementation of the method it defines.
 - Therefore, an interface simply specifies what must be done, but not how.
 - Interface is a construct that describes functionality without specifying implementation.

- Once an interface is defined, any number of classes can implement it.
 - This makes it possible for two or more classes to provide the same functionality even though they might do it in different ways.
 - Furthermore, one class can implement any number of interfaces.

Difference between Class and Interface

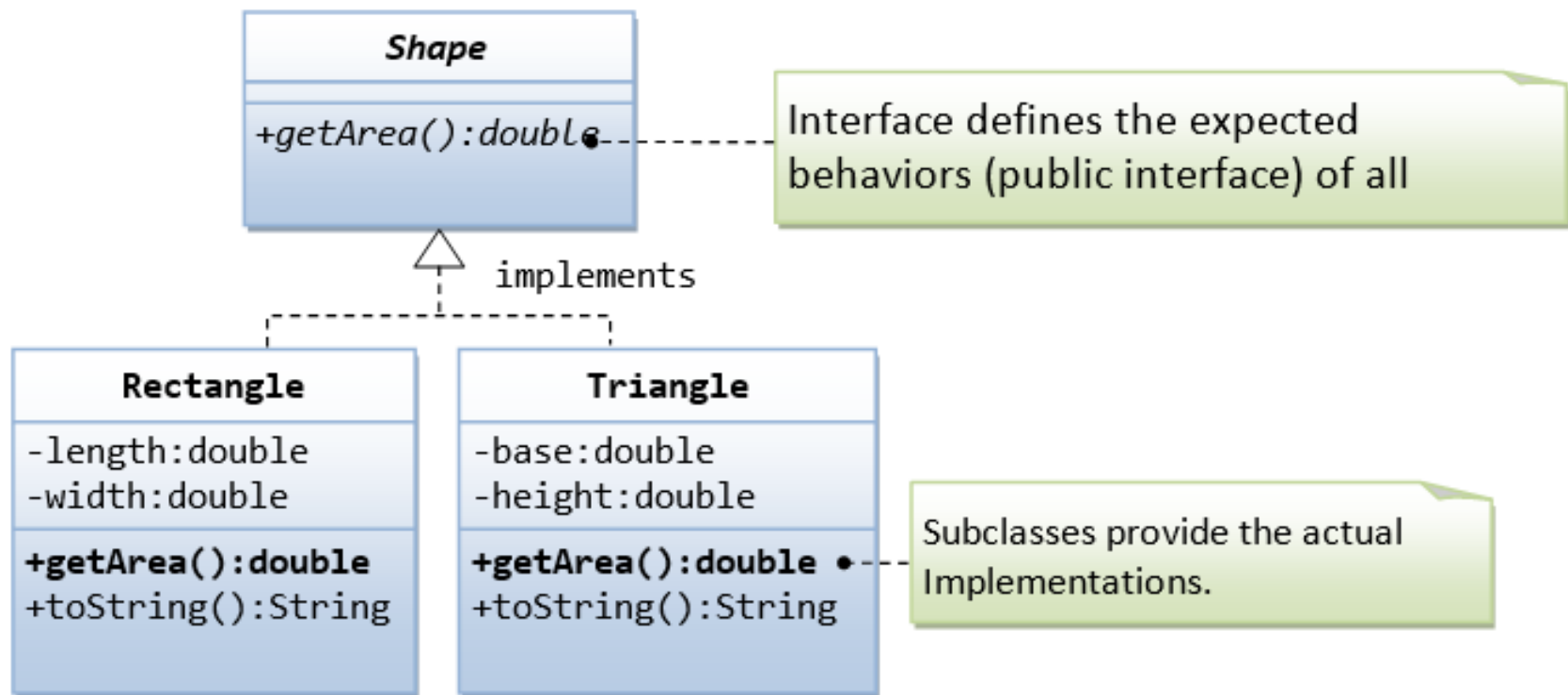
- In class, you can instantiate variable and create an object.
- Class can contain concrete(with implementation) methods.
- The access specifiers used with classes are private, protected and public.
- In an interface, you can't instantiate variable and create an object.
- The interface cannot contain concrete(with implementation) methods.
- In Interface only one specifier is used- Public.

General form of Interface Declaration

- *access* interface *name* {
 - *ret_type method_name(param_list);*
 - *ret_type method_name(param_list);*
 - *// ...*
 - *ret_type method_name(param_list);*
- }
 - Here *access* is either public (commonly) or not used (default).
 - *name* can be any valid identifier.
 - Methods inside the interface are essentially abstract methods.
 - No method can have an implementation.
 - It is the responsibility of each class that includes an interface to provide the implementation.
 - In an interface methods are implicitly public.

Creating an Interface

```
interface Animal {  
    public void animalSound(); // interface method (does not have a body)  
    public void sleep(); // interface method (does not have a body)  
}  
  
class Duck implements Animal {  
    public void animalSound() {  
        // The body of animalSound() is provided here  
        System.out.println("The Duck says: Quack Quack");  
    }  
    public void sleep() {  
        // The body of sleep() is provided here  
        System.out.println("Zzz");  
    }  
}  
  
class MyMainClass {  
    public static void main(String[] args) {  
        Duck d1 = new Duck(); // Create a Duck object  
        d1.animalSound();  
        d1.sleep();  
    }  
}
```



```
interface Shape {  
    double getArea();  
}  
class Rectangle implements Shape {  
    private int length;  
    private int width;  
    public Rectangle(int length, int width) {  
        this.length = length;  
        this.width = width;  
    }  
    @Override  
    public String toString() {  
        return "Rectangle[length=" + length + ",width=" + width + "];"  
    }  
    @Override  
    public double getArea() {  
        return length * width;  
    }  
}
```

```
class Triangle implements Shape { // More than one class can
                                //implement an interface
```

```
    private int base;
```

```
    private int height;
```

```
    public Triangle(int base, int height) {
```

```
        this.base = base;
```

```
        this.height = height;
```

```
    }
```

```
    @Override
```

```
    public String toString() { // We can define new methods
```

```
        return "Triangle[base=" + base + ",height=" + height + "];"
```

```
    }
```

```
    @Override
```

```
    public double getArea() {
```

```
        return 0.5 * base * height;
```

```
    }
```

```
    void display() {
```

```
        System.out.println("Hello World");
```

```
    }
```

```
}
```


- IPrime is an interface
 - isPrime() abstract method
- PrimeTester class implements IPrime
- ImprovedPrimeTester
- FurtherImprovedPrimeTester
- FastestPrimeTester -- >Fermat's method

Using Interface References

- When we define a class, we are creating a new reference type.
- An interface declaration also creates a new reference type.
 - When a class implements an interface, it is adding that interface's type to its type.
 - As a result, an instance of a class that implements an interface is also an instance of that interface type.
 - An interface reference variable can refer to any instance of its type.
 - When we call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed. (Similar to using a superclass reference to access a subclass object)

```
public class InterfaceTest {  
    public static void main(String[] args) {  
        Shape s1;  
        s1 = new Rectangle(1, 2); // upcast  
        System.out.println(s1);  
        System.out.println("Area is " + s1.getArea());  
        Triangle t1 = new Triangle(3, 4);  
        t1.display();  
        Shape s2 = t1;  
        s2.display();  
        System.out.println(s2);  
        System.out.println("Area is " + s2.getArea());  
        // Cannot create instance of an interface  
        //Shape s3 = new Shape("green"); // Compilation Error!!  
    }  
}
```

Note: An interface reference variable has knowledge only of the methods declared by its interface declaration. It cannot be used to access any other variables or methods that might be provided by an implementing class.

Implementing Multiple Interfaces

```
public interface GPS {  
    public void getCoordinates();  
}  
  
public interface Radio {  
    public void startRadio();  
    public void stopRadio();  
}  
  
public class Smartphone implements GPS, Radio {  
    public void getCoordinates() {  
        // return some coordinates  
    }  
    public void startRadio() {  
        // start Radio  
    }  
    public void stopRadio() {  
        // stop Radio  
    }  
}
```

If a Class implements multiple Interfaces, then there could be a chance of method signature overlap. Since Java does not allow multiple methods of the exact same signature, this can lead to ambiguity.

Constants in Interfaces

- Apart from declaring methods, an interface can also include variables.
 - However, such “variables” are not instance variables. Instead, they are implicitly public, final and static and must be initialized.
 - Thus, they are essentially constants.
 - A large program often makes use of several constant values such as array size, limits, special values, that could be used by several classes in that program.
 - An interface can be used to make such constants available to each class.

- interface Status {
 - int OPEN = 1;
 - int CLOSED = 2;
- }
- However, the constant interface pattern is a poor use of interfaces. Instead use,
- public final class Status {
 - public static final int OPEN = 1;
 - public static final int CLOSED = 2;
- }

Interfaces can be Extended

- One interface can inherit another by use of the keyword `extends`.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Nested Interfaces

- An interface can be declared a member of another interface or of a class.
- Such an interface is called a member interface or a nested interface.
- An interface nested in a class can use any access modifier. An interface nested in another interface is implicitly public.
 - When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.
 - Thus, outside of the interface or class in which a nested interface is declared, its name must be fully qualified.


```
interface A {  
    public interface NestedINF {  
        boolean isNegative(int x);  
    }  
    void doSomething();  
}  
class B implements A.NestedINF {  
    public boolean isNegative(int x) {  
        return x < 0 ? true : false;  
    }  
}
```

Fully qualified



Note: As class B implements A.NestedINF, it need not have to implemet doSomething()

```
public class NestedInterface {  
    public static void main(String[] args) {  
        A.NestedINF nif = new B(); // Legal as B implements A.NestedINF  
        if(nif.isNegative(25))  
            System.out.println("The number is negative");  
        else  
            System.out.println("The number is positive");  
    }  
}
```

End of Unit - 3