Object-Oriented Programming

Chapter Three: Inheritance

Contents

- At the end of this chapter ,the student will be able to get the following concept

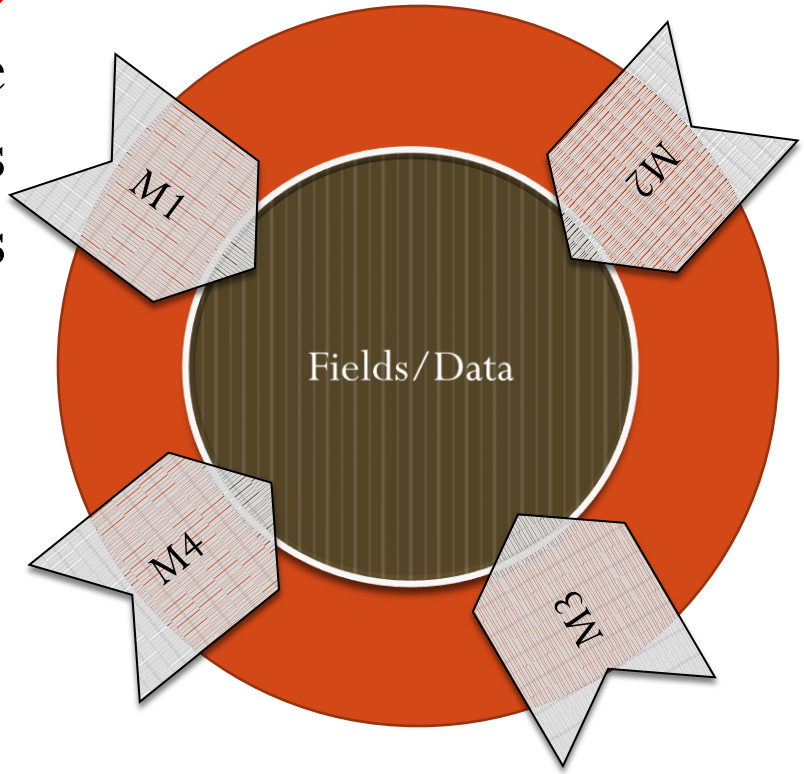
- Concept of encapsulation and data abstraction
- Concept of inheritance
- Super classes and subclasses
- Protected members
- Overriding methods
- Using this() and super()
- Use of final with inheritance
- Constructors in subclasses

Introduction

- All object-oriented programming languages provide mechanisms that help you implement the object-oriented model.
- The three fundamental object oriented programming principles are:
  - Encapsulation (data hiding) and Data abstraction
  - Inheritance
  - Polymorphism
- Main purpose of these principles is to manage software system complexity by improving software quality factors.

Encapsulation

- Encapsulation refers to the combining of fields and methods together in a class such that the methods operate on the data, as opposed to users of the class accessing the fields directly.
- It is a the technique which involves making the fields in a class private and providing access to the fields via public methods.



Encapsulation

- If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.
- For this reason, encapsulation is also referred to as data hiding.
- Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.
- Access to the data and code is tightly controlled by an interface.
- The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code.
- With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Encapsulation Example

```
public class EncapTest{
        private String name;
        private String idNum;
        private int age;
        public int getAge(){
            return age;
        }
        public String getName(){
             return name;
        }
public String getIdNum(){
    return idNum;
  }
```

```
    public void setAge( int
newAge){
      age = newAge;
   }
    public void setName(String
newName){
      name = newName;
   }
    public void setIdNum( String
newId){
      idNum = newId;
   }
}
```

Encapsulation, example

- The public methods are the access points to this class's fields from the outside java world.
- Normally these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters.
- The variables of the EncapTest class can be accessed as below:

```
public class RunEncap{
   public static void main(String args[]){
     EncapTest encap = new EncapTest();
     encap.setName("James");
     encap.setAge(20);
     encap.setIdNum("12343ms");
      System.out.print("Name : " + encap.getName()+ " Age : "+ encap.getAge());
   }
}
```

Out Put:

Name : James Age : 20

Data Abstraction

- Classes normally hide the details of their implementation from their clients.
  - This is called **information hiding**.
- The client cares about what functionality a class offers, not about how that functionality is implemented.
  - This concept is referred to as **data abstraction**.

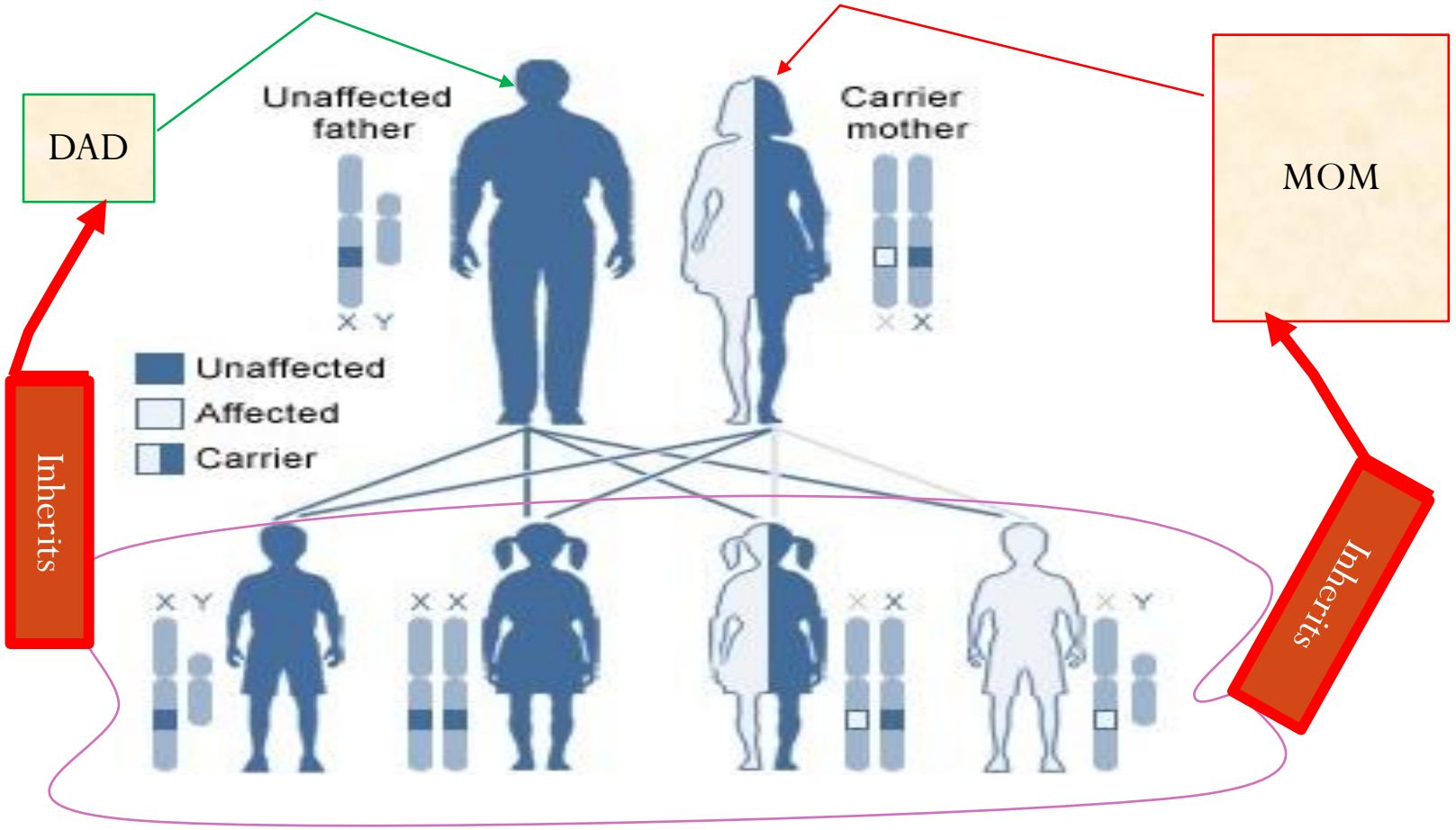
```
public void getFirstName(String keyWord)
{
    String temp="";
    for (int i=0;i<=records.length;i++)
    {
        temp=search(keyWord);
        if(temp!="")
        {
            ...
        }
    }
}
```

Inheritance

- In object-oriented programming (OOP)
  - **Inheritance** is the capability of a class (subclass)to use the properties (data fields) and methods of another class (Super Classes) while adding its own functionality.
    - I.e. **Inheritance** is a way to form new classes using classes that have already been defined.
  - Inheritance is employed to help reuse existing code with little or no modification.
  - The new classes, known as **derived classes**, inherit attributes and behavior of the pre-existing classes, which are referred to as **base classes** (or **ancestor classes**).
  - The inheritance relationship of sub- and superclasses gives rise to a hierarchy.
  - With the use of inheritance the information is made manageable in a hierarchical order.
  - Elements to be inherited form parent class are protected and public Members.

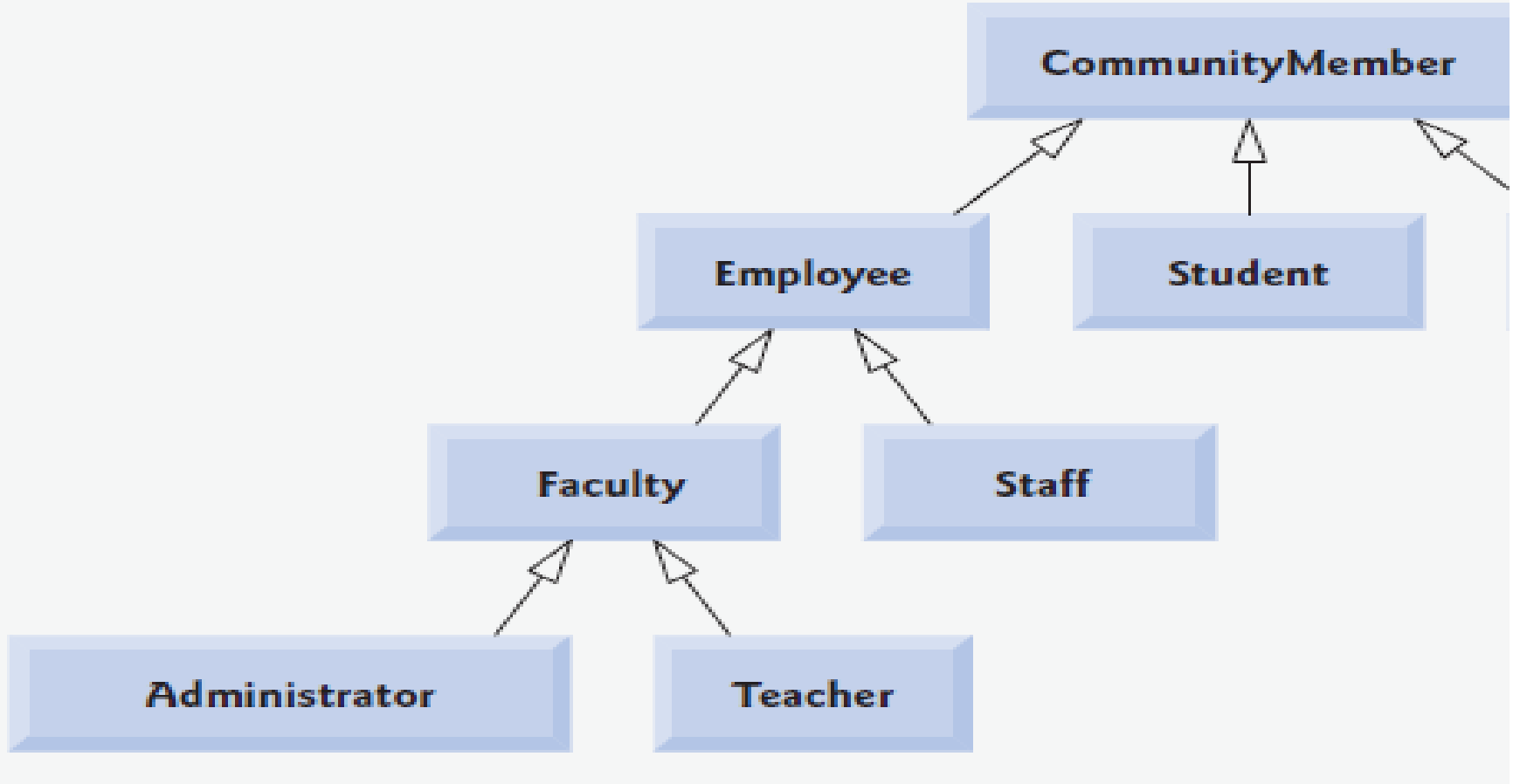
Inheritance Cont.

- Inheritance in Real World



Inheritance Cont.

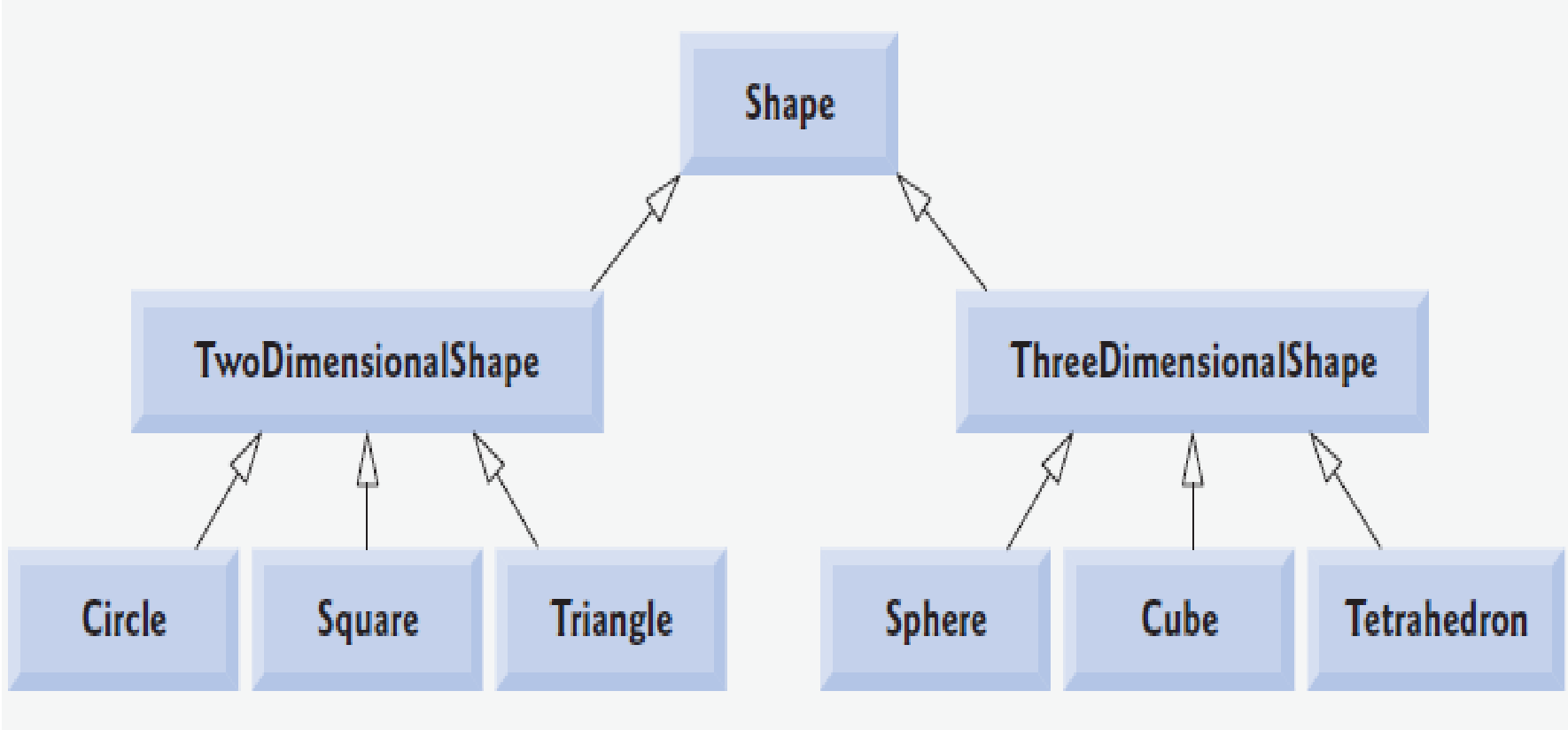
- Inheritance in OOP



Inheritance hierarchy for university CommunityMembers.

Inheritance Cont.

- Inheritance in OOP



Inheritance Hierarchy for Shapes

Inheritance Cont.

- **Applications of inheritance**
  - **Specialization**
    - the new class or object has data or behavior aspects that are not part of the inherited class.
  - **Overriding**
    - permit a class or object to replace the implementation of an aspect—typically a behavior—that it has inherited
  - **Code re-use**
    - re-use of code which already existed in another class.

Super classes and subclasses

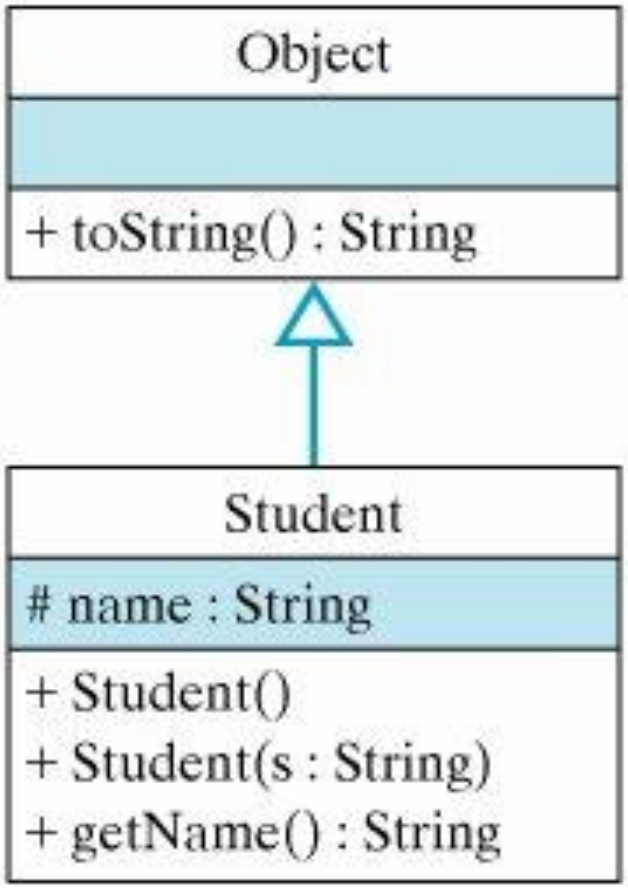
- a class C1 extended from another class C2 is called a subclass, and C2 is called a superclass.
- A superclass is also referred to as a supertype, a parent class, or a base class, and a subclass as a subtype, a child class, an extended class, or a derived class.
- **A subclass inherits** accessible data fields and methods from its superclass, and may also add new data fields and methods.

Super classes and subclasses Cont.

- When we talk about inheritance in java the most commonly used key words would be **extends** and **implements**.
- These words would determine whether one object IS-A type of another.
- With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.
- IS-A defines relationship between a superclass and its subclasses.
  - This means that an object of a subclass can be used wherever an object of the superclass can be used.

Super classes and subclasses Cont.

```
public class Student {
    protected String name;
    public Student() {
        name = "";
        }
    public Student(String s) {
        name = s;
        }
    public String getName() {
        return name;
        }
}
```



Note: Object is the super class of all Java Classes.

Super classes and subclasses Cont.

```
public class Animal{ ...}
public class Mammal extends Animal{...}
public class Reptile extends Animal{...}
public class Dog extends Mammal{...}
```

- Now based on the above example, In Object Oriented terms following are true:
  - Animal is the superclass of Mammal and Reptile classes.
  - Mammal and Reptile are subclasses of Animal class.
  - Dog is the subclass of both Mammal and Animal classes.
- A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore the following is illegal:

```
public class extends Animal, Mammal{...}
```

Super classes and subclasses Cont.

- We can assure that Mammal is actually an Animal with the use of the instance operator.

```
public class Dog extends Mammal{
   public static void main(String args[]){
      Animal a = new Animal();
      Mammal m = new Mammal();//m instanceof Animal
      Dog d = new Dog();/*d instanceof Mammal and d
instanceof Animal*/
}
}
```

Inheritance Example

```
class Vehicle {

        // Instance fields
        int noOfTyres; // no of tyres
        private boolean accessories; // check if accessorees present or not
        protected String brand; // Brand of the car
        // Static fields
        private static int counter; // No of Vehicle objects created
        // Constructor
        Vehicle() {
                System.out.println("Constructor of the Super class called");
                noOfTyres = 5;
                accessories = true;
                brand = "X";
                counter++;
        }
```

Inheritance Example Cont.

```
        // Instance methods
        public void switchOn() {
                accessories = true;
        }
        public void switchOff() {
                accessories = false;
        }
        public boolean isPresent() {
                return accessories;
        }
        private void getBrand() {
                System.out.println("Vehicle Brand: " + brand);
        }
        // Static methods
        public static void getNoOfVehicles() {
                System.out.println("Number of Vehicles: " + counter);
        }
}
```

Inheritance Example Cont.

```
class Car extends Vehicle {


        private int carNo = 10;
        public void printCarInfo() {
                System.out.println("Car number: " + carNo);
                System.out.println("No of Tyres: " + noOfTyres); // Inherited.
                //  System.out.println("accessories: "    + accessories); // Not Inherited.
                System.out.println("accessories: " + isPresent()); // Inherited.
                //        System.out.println("Brand: "     + getBrand());  // Not Inherited.
                System.out.println("Brand: " + brand); // Inherited.
                //  System.out.println("Counter: "    + counter);     // Not Inherited.
                getNoOfVehicles(); // Inherited.
        }
}
```

Inheritance Example Cont.

```
public class VehicleDetails { // (3)

        public static void main(String[] args) {
                new Car().printCarInfo();
        }
}
```

Output

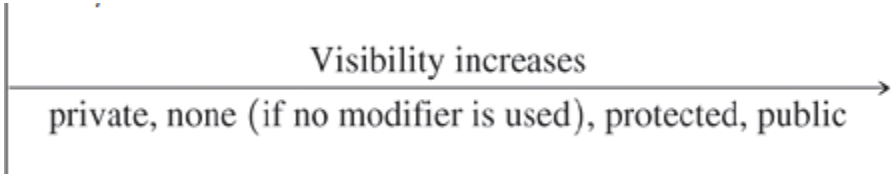
```
Constructor of the Super class called
Car number: 10
No of Tyres: 5
accessories: true
Brand: X
Number of Vehicles: 1
```

Protected members

- The modifier protected can be applied to data and methods in a class.
- A protected datum or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in different packages.

- The modifiers private, protected, and public are known as visibility or accessibility modifiers because they specify how class and class members are accessed.
- The visibility of these modifiers increases in this order:

Protected members Cont.

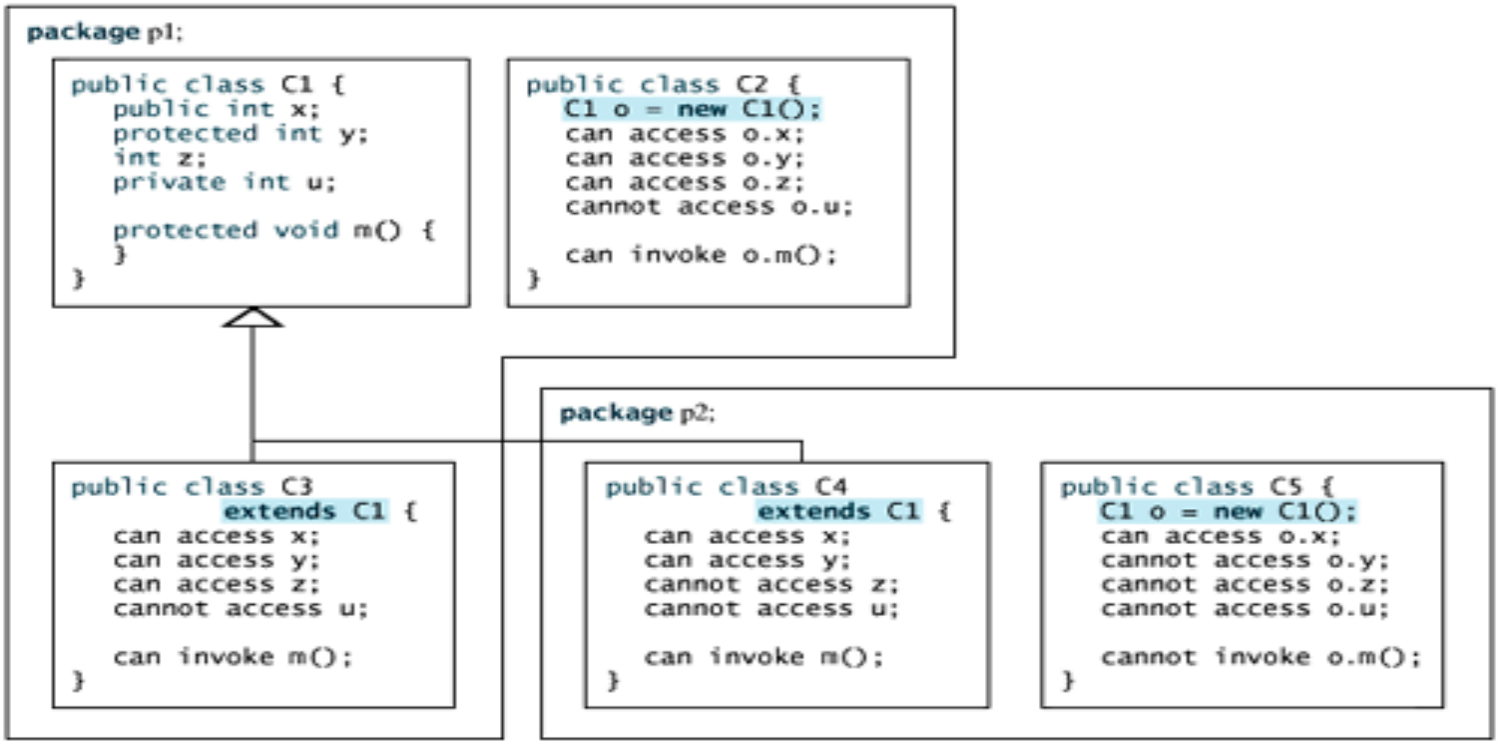


- Table 3.1 summarizes the accessibility of the members in a class. Next page figure illustrates how a public, protected, default, and private datum or method in class C1 can be accessed from a class C2 in the same package, from a subclass C3 in the same package, from a subclass C4 in a different package, and from a class C5 in a different package.

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
(default)	✓	✓	–	–
private	✓	–	–	–

Table 3.1 Data and Methods Visibility

Protected members Cont.



- **Figure 3.1Visibility modifiers are used to control how data and methods are accessed.**

Protected members Cont.

- Use the private modifier to hide the members of the class completely so that they cannot be accessed directly from outside the class.
- Use no modifiers to allow the members of the class to be accessed directly from any class within the same package but not from other packages.
- Use the protected modifier to enable the members of the class to be accessed by the subclasses in any package or classes in the same package.
- Use the public modifier to enable the members of the class to be accessed by any class.

Protected members Cont.

- The private and protected modifiers can only be used for members of the class.
- The public modifier and the default modifier (i.e., no modifier) can be used on members of the class as well on the class.
- A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.

Note

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Overriding methods

- A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Method overriding Rules

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level. For example: if the super class method is declared public then the overridding method in the sub class cannot be either private or protected. However the access level can be less restrictive than the overridden method's access level.
- Instance methods can be overridden only if they are inherited by the subclass.
- Constructors cannot be overridden.

Method overriding Rules…

- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.

Overriding methods Cont.

A real example of Java Method Overriding

- Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, CBE, Abay and Bunna banks could provide 7.25, 7.5, 7.75 rate of interest respectively . And any bank minimum rate of interest must have 7. The java program is shown below :

Overriding methods Cont.

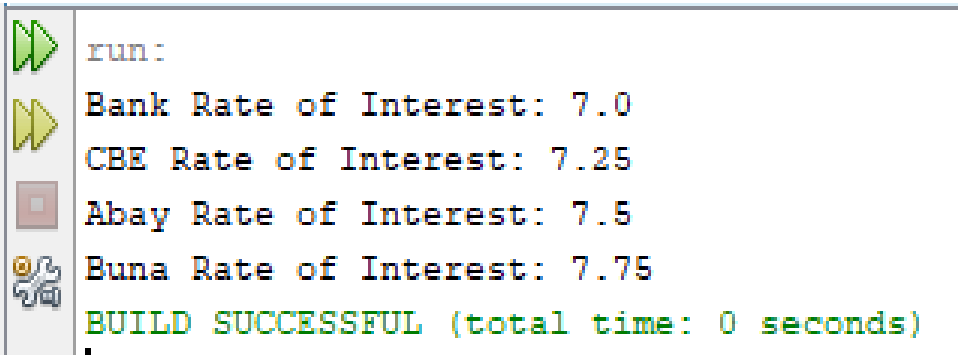
```
1   package overridtest;
    class Bank{
        public double getRateOfInterest(){
            return 7;

        }
    }
    class CBE extends Bank{
        @Override
            public double getRateOfInterest(){
            return 7.25;
        }
            }

            class Abay extends Bank{
                @Override
            public double getRateOfInterest(){
                    return 7.5;
                }
            }
    class Bunna extends Bank{
        @Override
            public double getRateOfInterest(){
            return 7.75;
        }
    }
```

Overriding methods Cont.

```
28  public class OverridTest {
29      public static void main(String[] args) {
30          Bank bnk=new Bank();
31         CBE c=new CBE();
32          Abay a=new Abay();
33          Bunna b=new Bunna();
34          System.out.println("Bank Rate of Interest: "+bnk.getRateOfInterest());
35          System.out.println("CBE Rate of Interest: "+c.getRateOfInterest());
36          System.out.println("Abay Rate of Interest: "+a.getRateOfInterest());
37          System.out.println("Buna Rate of Interest: "+b.getRateOfInterest());
38      }
39  }
```



Overriding methods Cont.

- Can we override static method?
  - No, a static method cannot be overridden. It can be proved by runtime polymorphism
- Why can we not override static method?
  - It is because the static method is bound with class whereas instance method is bound with an object.
- Can we override java main method?
  - No, because the main is a static method.

Using this() and super()

- First of all, this super() is different from the super keyword in Java.
  - The super keyword in Java is used to access the parent members of a class.
  - super(), with parenthesis, is used to call the parent class constructor.

```
1   package supertest;
    class Parent{
3        int x=10,y=30;
4   }
5   class Child extends Parent{
        int x=5,y=6;
7       public void print(){
8           System.out.println("x="+super.x+"y="+super.y);
9           System.out.println("x="+x+"y="+y);
10      }
11  }
12  public class SuperTest {
13      public static void main(String[] args) {
14          Child c=new Child();
15          c.print();
16      }
17  }
```

```
run:
x=10y=30
x=5y=6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Using this() and super() Cont.

- When extending a class constructor you can reuse the immediate super class constructor and overridden superclass methods by using the reserved word super.
- Note that
  - You cannot override final methods, methods in final classes, private methods or static methods.
- **Example**

```
class A{
 int a, b, c;
  A(int p, int q, int r){
      a=p;
      b=q;
      c=r;
 }
 void Show(){
    System.out.println("a = " + a);
     System.out.println("b = " + b);
     System.out.println("c = " + c);
 }
}
```

Using this() and super() Cont.

```
public class B extends A{
      int d;
      B(int l, int m, int n, int o){
        super(l, m, n);
        d=o;
       }
      void Show(){
          super.show()
          System.out.println("d = " + d);
      }
      public static void main(String args[]){
        B b = new B(4,3,8,7);
        b.Show();
      }
    }
```

Output

a = 4
b = 3
c = 8
d = 7

Using this() and super() Cont.

- super() can only be used in the first statement of the constructor
- this() with parenthesis is different from the this keyword.
- The this keyword is used to access the members of the current class. It can also be used to access the members of the parent class that are accessible from the child class, e.g., inheritance.

Using this() and super() Cont.

```
1   package thistest;
2   class Parent{
3       int length=20;
4       public void printLength()
5       {
6           System.out.println("Length:"+this.length);
7       }
8   }
9   class Child extends Parent{
10      int width=5;
11      int getArea()
12      {
13          return width*this.length;
14      }
15  }
16  public class ThisTest {
17      public static void main(String[] args) {
18
19          Child c=new Child();
20          c.printLength();
21          System.out.println("Area"+c.getArea());
22
23      }
24  }
```

```
run:
Length:20
Area100
BUILD SUCCESSFUL (total time: 0 seconds)
```

Using this() and super() Cont.

- this() with parenthesis is used to call the *current class constructor*. It should also be used in the first line of the constructor.

```
1   package thismethodexample;
2   class parent{
3     parent(){
4         this(10,20);
5     }
6     parent(int x){
7       System.out.println("x="+x);
8     }
9     parent(int a,int b){
10        this(5);
11      System.out.println("a="+a+"   b="+b);
12    }
13  }
14  public class ThisMethodExample {
15      public static void main(String[] args) {
16          parent obj = new parent();
17      }
18  }
```

```
run:
x=5
a=10   b=20
BUILD SUCCESSFUL (total time: 0 seconds)
```

Using this() and super() Cont.

- Difference between super() and this() in java
- super and this keyword super() as well as this() keyword both are used to make constructor calls.
- super() is used to call Base class's constructor(i.e, Parent's class) while this() is used to call the current class's constructor.
- Important points about this() and super()
  - we can use either super() or this() as first statement inside constructor and not both.
  - Recursive constructor call not allowed
-

Using this() and super() Cont.

super() keyword	this() keyword
super() calls the parent constructor	this() can be used to invoke the current class constructor
It can be used to call methods from the parent.	It can be passed as an argument in the method call.
It is returned with no arguments.	It can be passed as an argument in the constructor call.
It can be used with instance members.	It is used to return the current class instance from the method.

Use of final with inheritance

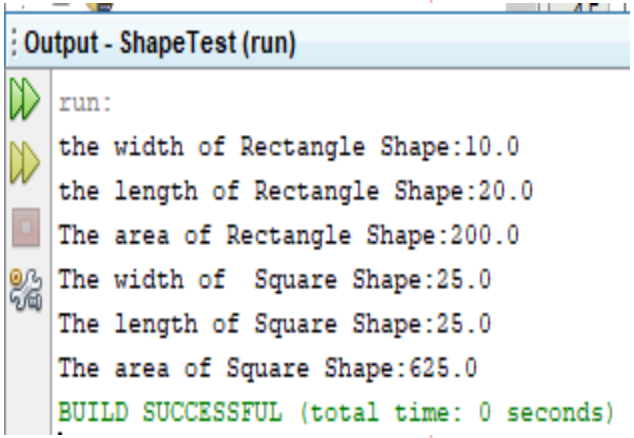
- final keyword with inheritance in java
  - The final keyword is final that is we cannot change.
  - We can use final keywords for variables, methods, and class.
  - If we use the final keyword for the inheritance that is if we declare any method with the final keyword in the base class so the implementation of the final method will be the same as in derived class.
  - We can declare the final method in any subclass for which we want that if any other class extends this subclass.

Use of final with inheritance Cont.

```
1   package shapetest;
    abstract class Shape{
3       double width,length;
4       public Shape(double width,double length)
5       {
6           this.width=width;
7           this.length=length;
8       }
9       public final double getWidth(){
10          return width;
11      }
12      public final double getLength()
13      {
14          return length;
15      }
        abstract double getArea();
17  }
18  class Rectangle extends Shape{
19      public Rectangle(double width, double length)
20      {
21          super(width,length);
22      }
        @Override
24      public final double getArea(){
25          return this.getWidth()*this.getLength();
26      }
27  }
```

Use of final with inheritance Cont.

```
28  class Square extends Shape{
29      public Square(double length)
30      {
31          super(length,length);
32      }
        @Override
34      public final double getArea(){
35          return this.getWidth()*this.getLength();
36      }
37  }
38
39  public class ShapeTest {
40      public static void main(String[] args) {
41          Shape rectangleShape=new Rectangle(10,20);
42          Shape squareShape=new Square(25);
43          System.out.println("the width of Rectangle Shape:"+rectangleShape.getWidth());
44          System.out.println("the length of Rectangle Shape:"+rectangleShape.getLength());
45          System.out.println("The area of Rectangle Shape:"+rectangleShape.getArea());
46          System.out.println("The width of  Square Shape:"+squareShape.getWidth());
47          System.out.println("The length of Square Shape:"+squareShape.getLength());
48          System.out.println("The area of Square Shape:"+squareShape.getArea());
49      }
50  }
```



Use of final with inheritance Cont.

- **Using final to Prevent Inheritance**

➢ When a class is declared as final then it cannot be subclassed i.e. no other class can extend it.

- The following fragment illustrates the final keyword with a class:

```
final class A {
// methods and fields
 } //The following class is illegal.
class B extends A {
// ERROR! Can't subclass A
}
```

Use of final with inheritance Cont.

- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final

- Using final to Prevent Overriding
  - When a method is declared as final then it cannot be overridden by subclasses.
  - The Object class does this
    - a number of its methods are final.
  - The following fragment illustrates the final keyword with a method:

Use of final with inheritance Cont.

```
class A
{
   final void m1()
   {
      System.out.println("This is a final method.");
   }
}

class B extends A
{
   void m1()
   {
      // ERROR! Can't override.
      System.out.println("Illegal!");
   }
}
```

Constructors in subclasses

- What is the order of execution of constructor in Java inheritance?
  - While implementing inheritance in a Java program, every class has its own constructor.
  - Therefore the execution of the constructors starts after the object initialization.
  - It follows a certain sequence according to the class hierarchy.
  - There can be different orders of execution depending on the type of inheritance.

1. Order of execution of constructor in Single inheritance

- In single level inheritance, the constructor of the base class is executed first.

Constructors in subclasses Cont.

```
1   package orderofconstractorexcution;
2   class ParentClass {
3       ParentClass()
4       {
5           System.out.println("this is parent class constractor!!!");
6       }
7   }
8   class ChildClass extends ParentClass{
9       ChildClass(){
10      System.out.println("this is Sub(Child) class constractor!!!");
11      }
12  }
13  public class OrderOfConstractorExcution {
14      public static void main(String[] args) {
15          System.out.println("The order of constractors Excution");
16          new ChildClass();
17      }
18  }
```

```
Output - OrderOfConstractorExcution (run)
run:
The order of constractors Excution
this is parent class constractor!!!
this is Sub(Child) class constractor!!!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Constructors in subclasses Cont.

- In the previous code, after creating an instance of ChildClass the ParentClass constructor is invoked first and then the ChildClass.

2. Order of execution of constructor in Multilevel inheritance

- In multilevel inheritance, all the upper class constructors are executed when an instance of bottom most child class is created.

Constructors in subclasses Cont.

```
1   package multipleconstractororder;
2   class University{
3       University()
4       {
5           System.out.println("This is University class constractor!!!");
6       }
7   }
8   class College extends University{
9       College(){
10      System.out.println("This is College class constractor!!!");
11     }
12  }
13      class Department extends College{
14          Department(){
15
16          System.out.println("This is College class constractor!!!");
17          }
18      }
19  class Student extends Department{
20      Student ()
21      {
22          System.out.println("This is Student class constractor!!!");
23      }
24  }
```

Constructors in subclasses Cont.

```
25  public class MultipleConstractorOrder {
26      public static void main(String[] args) {
27  System.out.println("This is multilevel Inheritance constractor excution order!!!");
            new Student();
29      }
30  }
```

Output - MultipleConstractorOrder (run)

```
run:
This is multilevel Inheritance constractor excution order!!!
This is University class constractor!!!
This is College class constractor!!!
This is College class constractor!!!
This is Student class constractor!!!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Abstraction

- An abstract class is one that cannot be instantiated. Abstraction refers to the ability to make a class abstract in OOP.
- All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner.
- Classes that can be instantiated are known as concrete classes.
- A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.
- Use the **abstract** keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the class keyword.

Abstraction of Method

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.
- The abstract keyword is also used to declare a method as abstract.
- An abstract methods consist of a method signature, but no method body - its signature is followed by a semicolon, not curly braces as usual.
- Declaring a method as abstract has two results:
  - The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
  - Any child class must either **override the abstract method** or **declare itself abstract.**

Ways to achieve Abstraction

- There are two ways to achieve abstraction in java
  - Abstract class (0 to 100%)
  - Interface (100%)
- **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.

Abstract class and methods Example

```
1   package abstractclasstest;
2   abstract class Shape{//abstract class
3       double width,height;
4       public Shape(double x,double y){
5           width=x;
6           height=y;
7       }
8      double getWidth()
9       {
10          return width;
11      }
12      double getLength()
13      {
14          return height;
15      }
16      abstract double getArea();//abstract method
17      abstract double getPerimeter(); //abstract method
18  }
```

```
19   class Rectangle extends Shape{
20  Rectangle(double x,double y)
21  {
22  super(x,y);
23  }
    @Override
25      public double getArea() {
26          return  getWidth()*getLength();
27      }
        @Override
29      public double getPerimeter() {
30          return 2*(getWidth()+getLength());
31      }
32  }
```

Abstract class and methods Example

```
33      class Square extends Shape{
34          public Square(double x)
35          {
36              super(x,x);
37          }
            @Override
39          double getArea() {
40              return getWidth()*getLength();
41          }
            @Override
43          double getPerimeter() {
44              return 4*getWidth();
45          }
46      }
47  public class AbstractClassTest {
48      public static void main(String[] args) {
49          Shape shape=new Rectangle(3,4);
50          Square sq=new Square(4.0);
51          System.out.println("the width of rectangle:"+shape.getWidth());
52          System.out.println("the length of Rectangle:"+shape.getLength());
53          System.out.println("Area of Rectangle:"+shape.getArea());
54          System.out.println("perimeter  of Rectangle:"+shape.getPerimeter());
55          System.out.println("The width of  Square:"+sq.getWidth());
56          System.out.println("The length of  Square:"+sq.getLength());
57          System.out.println("Area   of Square:"+sq.getArea());
58          System.out.println("perimeter of Square:"+sq.getPerimeter());
59      }
60  }
```

Output - AbstractClassTest (run)

```
run:
the width of rectangle:3.0
the length of Rectangle:4.0
Area of Rectangle:12.0
perimeter  of Rectangle:14.0
The width of  Square:4.0
The length of  Square:4.0
Area   of Square:16.0
perimeter of Square:16.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Interfaces

- Interfaces are similar to abstract classes but all methods are abstract and all data fields are static final.
  - That is an interface is a collection of abstract methods and static final variables.
- Interfaces can be inherited (i.e. you can have a sub-interface). As with classes the extends keyword is used for inheritance.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Interfaces Cont.

- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.
- Java does not allow multiple inheritance for classes (i.e. a subclass being the extension of more than one superclass).
- An interface is similar to a class in the following ways:
  - An interface can contain any number of methods.
  - An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
  - The bytecode of an interface appears in a **.class** file.
  - Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Interfaces Cont.

- However, an interface is different from a class in several ways, including:
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

Declaring Interfaces

- The **interface** keyword is used to declare an interface.

```
public interface NameOfInterface
{
  //Any number of final, static fields
  //Any number of abstract method declarations\
}
```

- Interfaces have the following properties:
  - An interface is implicitly abstract. You do not need to use the abstract keyword when declaring an interface.
  - Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  - Methods in an interface are implicitly public.

Implementing Interfaces

- A class uses the **implements** keyword to implement an interface.
- The implements keyword appears in the class declaration following the extends portion of the declaration.

```
1   package mammalint;
    interface Animal{
        public void eat();
        public void travel();
5   }
6   public class MammalInt implements Animal {
        @Override
8       public void eat() {
9           System.out.println("Mammals eats");
10      }
        @Override
12      public void travel() {
13          System.out.println("Mammals travels");
14      }
15      public static void main(String[] args) {
16          MammalInt m=new MammalInt();
17          m.eat();
18          m.travel();
19      }
20  }
```

```
Output - MammalInt (run)
run:
Mammals eats
Mammals travels
BUILD SUCCESSFUL (total time: 0 seconds)
```

Extending Interfaces

- An interface can extend another interface, similarly to the way that a class can extend another class.
- The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

```
1   package gameinterface;
     interface Sport{
       public void setSportName(String name);
       public void setTeamName(String name1,String name2);
        public void getSportName();
       public void getTeamName();
7   }
     interface visitor extends Sport{
       public void setVisitor(int numberOfAudiance);
       public void getVisitor();
11  }
     interface Score extends Sport{
          public void setGoalScored(int teamOneGoal,int teamTwoGoal);
          public void getScored();
          public void endOfPeriod(int period);
          public void overtimePeriod(int ot);
          public int getNormalTime();
          public int getOvertimePeriod();
19  }
```

Extending Interfaces Cont.

```
20  class SportInformation implements visitor,Score{
21  int audiance;
22  String sportName;
23  String teamName1,teamName2;
24  int teamOneNoOfGoal,teamTwoNoOfGoal;
25  int teamOneScore,teamTwoScore;
26  int normalTime,adationalTime;
        @Override
28      public void setVisitor(int numberOfAudiance) {
29          audiance=numberOfAudiance;
30      }
        @Override
32      public void getVisitor() {
33          System.out.println("Number of Audiance:"+audiance);
34      }
        @Override
36      public void setSportName(String name) {
37          sportName=name;
38      }
        @Override
40      public void setTeamName(String name1, String name2) {
41          teamName1=name1;
42          teamName2=name2;
43      }
```

Extending Interfaces Cont.

```
        @Override
45      public void getSportName() {
46          System.out.println("Sport Name:"+sportName);
47      }
        @Override
49      public void getTeamName() {
50  System.out.println("Team One Name:"+teamName1+"\nTeam Two Name "+teamName2);
51      }
        @Override
53      public void setGoalScored(int teamOneGoal,int teamTwoGoal) {
54          teamOneNoOfGoal=teamOneGoal;
55          teamTwoNoOfGoal=teamTwoGoal;
56      }
        @Override
58      public void getScored() {
59          if(teamOneNoOfGoal==teamTwoNoOfGoal)
60          {
61             teamOneScore=1;
62             teamTwoScore=1;
63          }
64          else if(teamOneNoOfGoal>teamTwoNoOfGoal)
65          {
66              teamOneScore=3;
67             teamTwoScore=0;
68          }
69          else
70          {
```

Extending Interfaces Cont.

```
71              teamOneScore=0;
72             teamTwoScore=3;
73          }
74          System.out.println("Team One Score:"+teamOneScore+"\nTeame Two Score:"+teamTwoScore);
75      }
        @Override
77      public void endOfPeriod(int period) {
78          normalTime=period;
79      }
        @Override
81      public void overtimePeriod(int ot) {
82          adationalTime=ot;
83      }
84
        @Override
86      public int getNormalTime(){
87        return normalTime;
88      }
        @Override
90      public int getOvertimePeriod() {
91                  return adationalTime;
92      }
93  }
```

Extending Interfaces Cont.

```
94  public class GameInterface {
95      public static void main(String[] args) {
96        SportInformation gi=new SportInformation ();
97        gi.endOfPeriod(90);
98        gi.setGoalScored(3,5);
99        gi.setTeamName("adama","Fasil");
100       gi.setSportName("FootBall");
101       gi.setVisitor(300);
102       gi.overtimePeriod(5);
103       gi.getSportName();
104       gi.getTeamName();
105       gi.getVisitor();
106       System.out.println("Normal Time:"+gi.getNormalTime()+"\nOverTime:"+gi.getOvertimePeriod());
107       gi.getScored();
108
109 }
110 }
```

```
Output - GameInterface (run)
run:
Sport Name:FootBall
Team One Name:adama
Team Two Name Fasil
Number of Audiance:300
Normal Time:90
OverTime:5
Team One Score:0
Teame Two Score:3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Extending Interfaces Cont.

- The visitor interface has two methods, but it inherits four from Sport; thus, a class that implements visitor needs to implement all six methods.
- Similarly, a class that implements Score needs to define the six methods from score and the two methods from Sports.
- Note that an interface can extend more than one parent interface
  - public interface Hockey extends Sports, Event

Abstract Classes & Interfaces

- what's the difference between an interface and an abstract class?

➢ An interface cannot implement any methods, whereas an abstract class can A class can implement many interfaces but can have only one superclass (abstract or not)

➢ An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

- Syntax:

➢ abstract class:

public class Apple extends Food { … }

➢ interface:

public class Person implements Student, Athlete, Chef { … }

Abstract Classes & Interfaces Cont.

- Why are they useful?

Answer : By leaving certain methods undefined, these methods can be implemented by several different classes, each in its own way.

Example: Chess Playing Program

- an abstract class called ChessPlayer can have an abstract method makeMove(), extended differently by different subclasses.
- an interface called ChessInterface can have a method called makeMove(), implemented differently by different classes

Abstract Classes & Interfaces Cont.

- What about inheritance?
- Answer : Follow these simple rules:
  - An abstract class can't inherit from more than one other class.
  - Interfaces can inherit from other interfaces, and a single interface can inherit from multiple other interfaces

Abstract Classes & Interfaces Cont.

Example:

```
interface Singer  {

    void sing();
    void warmUpVoice();
}

interface Dancer  {

    void dance();
    void stretchLegs();
}

interface Talented extends Singer, Dancer  {
    // can sing and dance. Wowwee.
}
```

Q & A