Inheritance and Polymorphism

Edited by: Dr. Nadine Zbib

Objectives

- Inheritance Relationship
- To define a subclass from a superclass.
- To invoke the superclass's constructors and methods using the **super** keyword.
- To override instance methods in the subclass.
- To distinguish differences between overriding and overloading.
- To explore the **toString()** method in the **Object** class.
- To explore the **equals()** method in the **Object** class.
- To discover polymorphism and dynamic binding.
- To describe casting and explain why explicit down casting is necessary.
- To explore the **equals()** method in the **Object** class.

Inheritance Relationship between classes

There are different types of relationships between classes:

- Composition
- Aggregation
- Inheritance

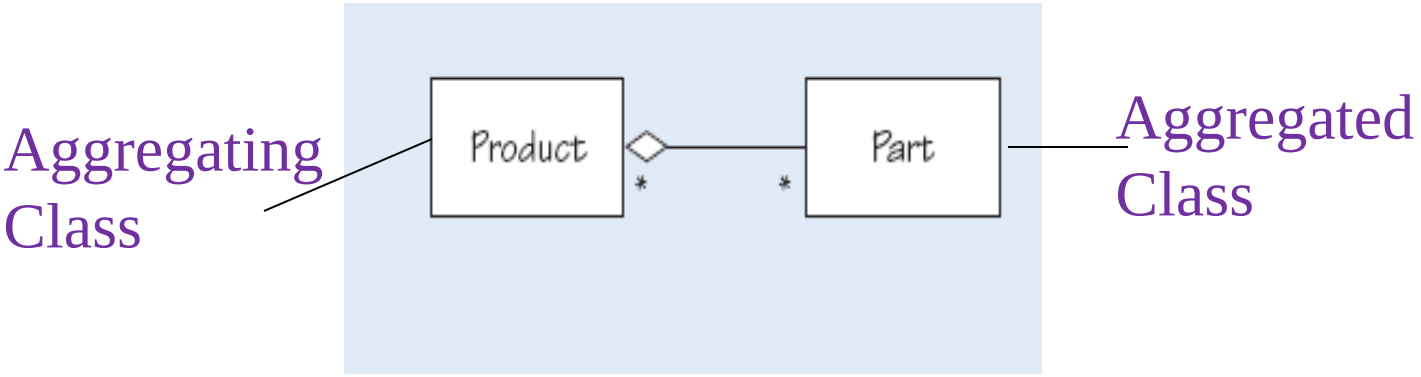
(**has-a** relationship

(**is-a** relationship

In this lecture we will concentrate on the inheritance relationship (**is-a** relationship) and how to implement it in Java O.O. language.

Aggregation Relationships

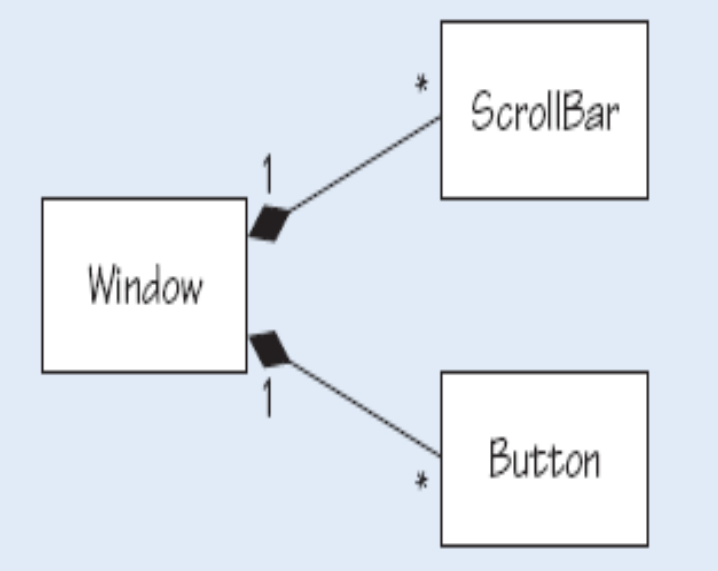
- Aggregations models *has-a* relationships and form a Whole-Part relationship between two classes.
- If you delete the Whole class (aggregating), the Part one (Aggregated) will remain.



Composition Relationships

- Composition is a stronger form of aggregation .
- If the aggregating (composing) object is deleted, all of the aggregated (composed) objects must be deleted.

Consider a Window object containing a ScrollBar and a Button. If the window is destroyed, the scroll bar and button must



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. follows:

```
public class Name {
  ...
}
```

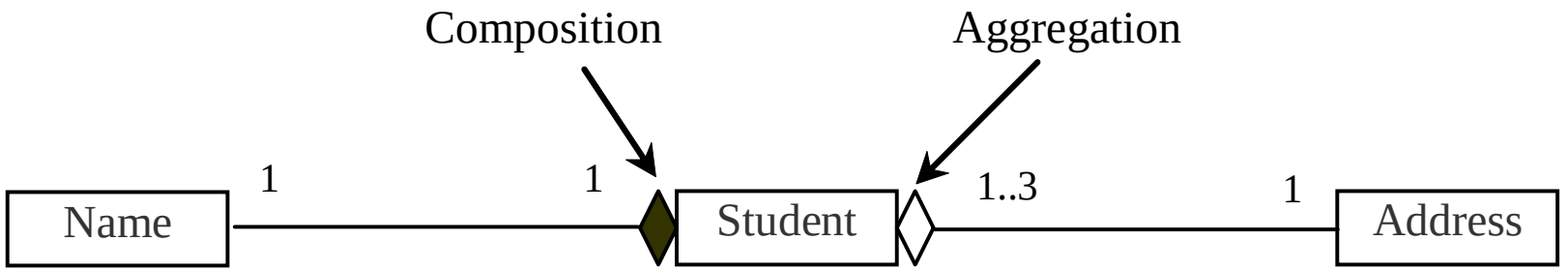
Aggregated class

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

Aggregated class



Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

Inheritance

- Inheritance is an Object oriented feature which allows a class to inherit behavior (methods) and data from other class.
- With the use of inheritance the information is made manageable in a hierarchical order.

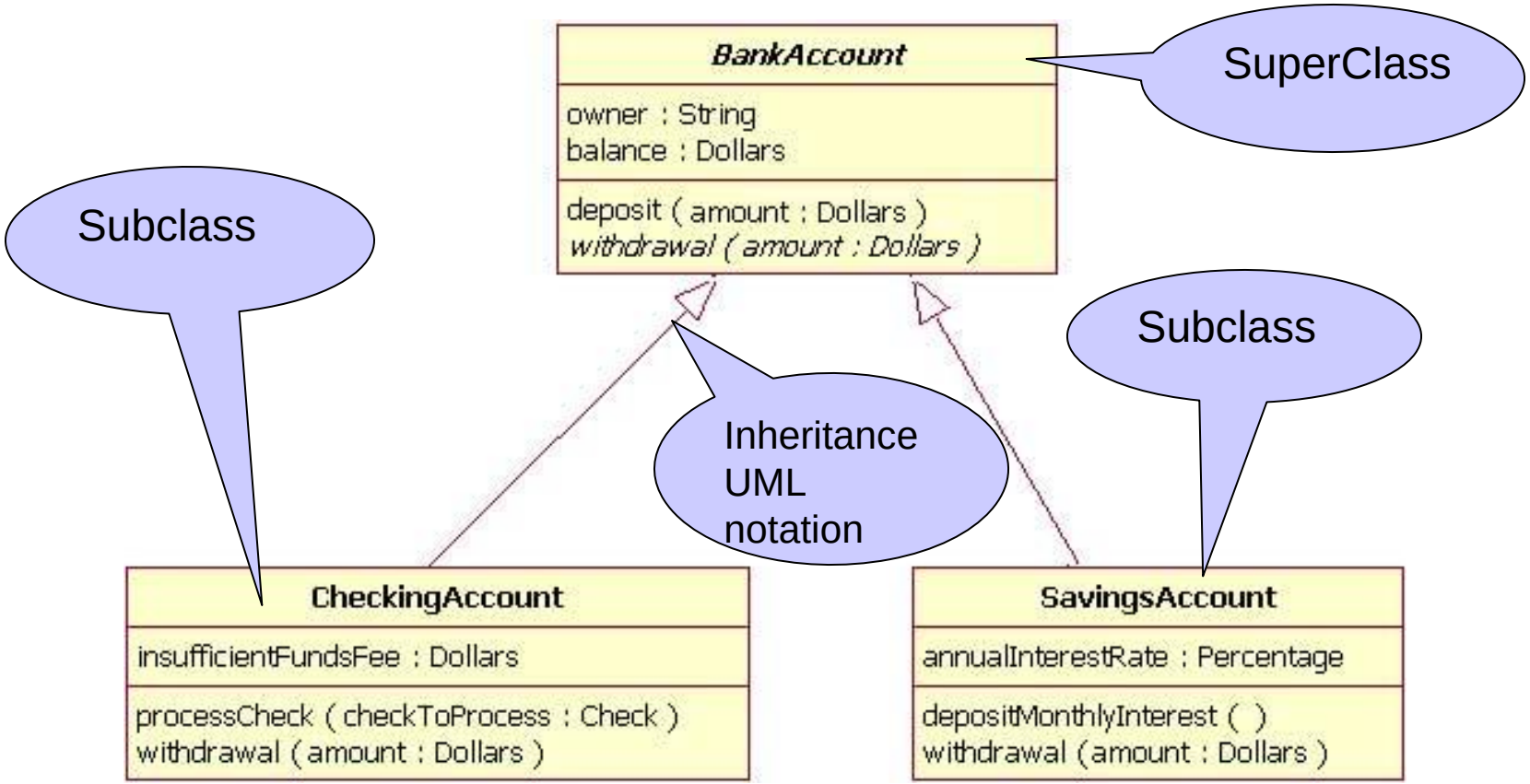
Subclass and SuperClass

- The class which inherits the properties of other is known as subclass (derived class, child class). The subclass can have additional methods and data in addition to inherited ones.

- The class whose properties are inherited is known as superClass (base class, parent class). Superclass holds the common data and methods.

UML Notation for inheritance

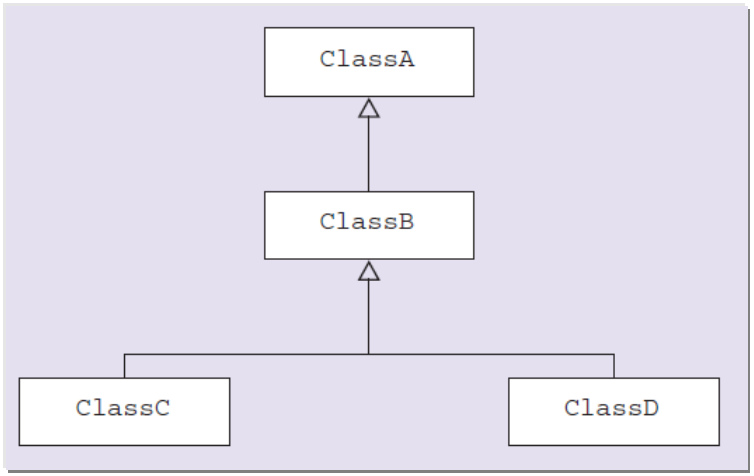
What are the members of each class in the below class diagram?



The Class Heirarchy

- There are two types in Java hierarchy:
  1. user-defined
  2. Java class library.

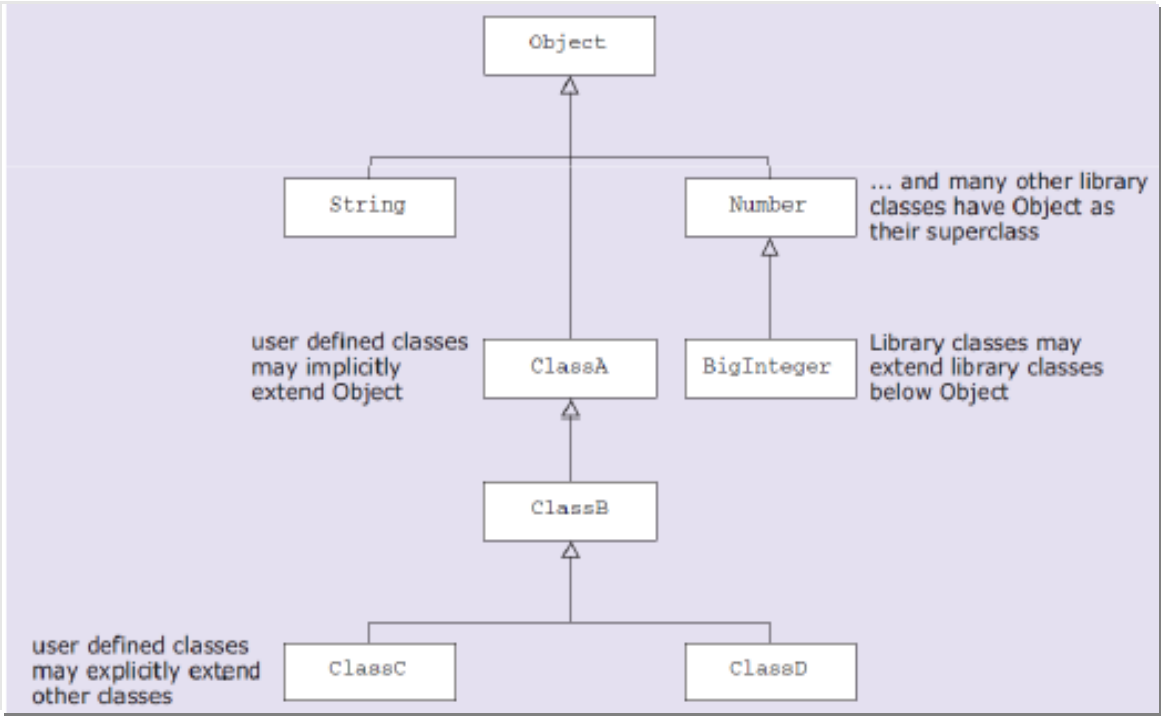
- The figure below shows a user-defined hierarchy.
- class C and class D inheriting from class B which, in turn, inherits from class A.



Java inheritance hierarchy

The figure below shows part of the Java inheritance hierarchy, including the root of the hierarchy, which is the class whose name is Object.

The class Object has no superclass.



Why Inheritance?

- The main benefit behind the inheritance is the reusability feature, where a new class (subclass) can be created using an existing one which save effort, time, avoid redundancy and reduce time of testing.

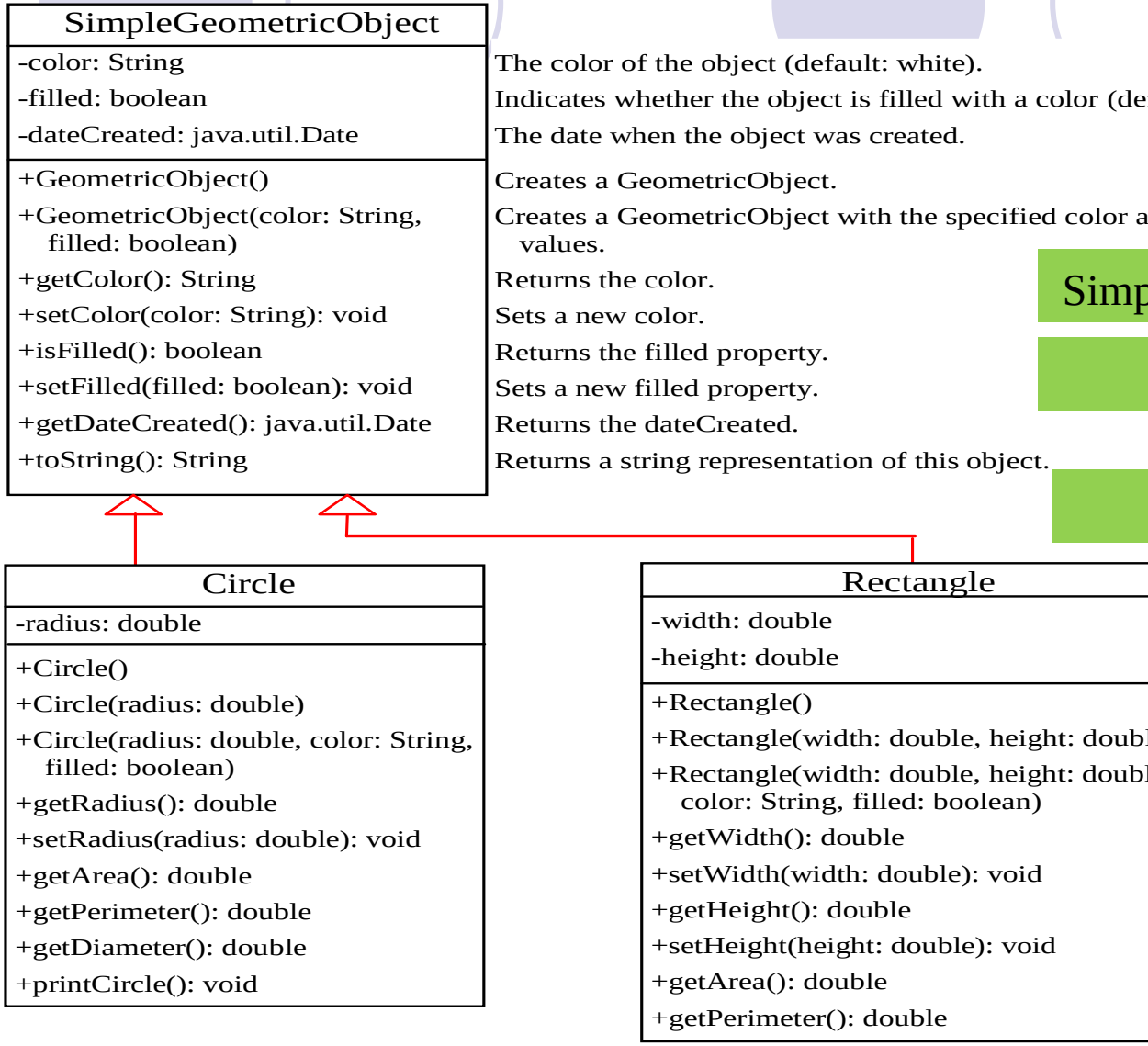
Syntax of Java Inheritance

- The extends keyword is used to represent inheritance relationship in Java language.

- The general format of subclass is:

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

Superclasses and Subclasses



SimpleGeometricObject

Circle

Rectangle

TestCircleRectangle

Run

Example: SimpleGeometricObject Class

```
import java.util.Date;
public class SimpleGeometricObject {
  private String color = "white";
  private boolean filled;
  private Date dateCreated;

    public SimpleGeometricObject() {
    dateCreated = new java.util.Date();
  }
 public SimpleGeometricObject(String color,
boolean filled) {
    dateCreated = new Date();
    this.color = color;
    this.filled = filled;
  }
 public String getColor() { return color; }

 public void setColor(String color)
      {this.color = color;}
```

```
public boolean isFilled()
{    return filled;      }

public void setFilled(boolean filled)
{   this.filled = filled;}

  public Date getDateCreated()
  {
    return dateCreated;
  }

  public String toString() {
    return "created on " +
              dateCreated +
             "\ncolor: " + color +
              " and filled: " +
               filled;
  }

}
```

Continue: Circle class

```
public class Circle extends
SimpleGeometricObject {
  private double radius;

  public Circle() { }

  public Circle(double radius)
  { this.radius = radius; }

  public Circle(double radius, String
                color, boolean filled)
  { this.radius = radius;
    setColor(color);
    setFilled(filled); }

  public double getRadius()
  { return radius; }

  public void setRadius(double radius)
  { this.radius = radius; }

  public double getArea()
   { return radius * radius *
       Math.PI; }

  public double getDiameter()
  { return 2 * radius; }

  public double getPerimeter()
  { return 2 * radius * Math.PI; }

  public void printCircle() {
    System.out.println("The circle
    is created " + getDateCreated()
    + " and the radius is " +
    radius);
  }
}
```

Continue: Class Rectangle

```
public class Rectangle extends
SimpleGeometricObject {
  private double width, height;

  public Rectangle() { }

  public Rectangle( double w, double h)
  { this.width = w; this.height = h; }

  public Rectangle( double w, double h,
       String color, boolean filled)
  { this.width = w; this.height = h;
    setColor(color); setFilled(filled);
  }
  public double getWidth()
  { return width; }
```

```
  public void setWidth(double width)
   { this.width = width;     }
  public double getHeight()
   { return height; }

  public void setHeight(double height)
  { this.height = height; }

  public double getArea()
  { return width * height; }

  public double getPerimeter()
  { return 2 * (width + height); }
}
```

Continue: Class Test

```
public class TestCircleRectangle {
  public static void main(String[] args) {
    Circle circle = new Circle(1);
    System.out.println("A circle " + circle.toString());
    System.out.println("The color is " + circle.getColor());
    System.out.println("The radius is " + circle.getRadius());
    System.out.println("The area is " + circle.getArea());
    System.out.println("The diameter is " + circle.getDiameter());

    Rectangle rectangle = new 2, 4);
    System.out.println("\n A rectangle " + rectangle.toString());
    System.out.println("The area is " + rectangle.getArea());
    System.out.println("The perimeter is " + rectangle.getPerimeter());
  }
}
```

Are superclass's Constructor Inherited?

- No. They are not inherited.
- They are invoked explicitly or implicitly.
- Explicitly using the super() keyword.
  - A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass.
  - They can only be invoked from the subclasses' constructors, using the keyword super().
  - *If the keyword super() is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor: super()
- To call a superclass method: super.method()

CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```
public class Apple extends Fruit {
}

public class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Defining a Subclass

A subclass inherits from a superclass. A subclass can also:

- Have new properties
- Have new methods
- Override the methods of the superclass

Calling Superclass Methods

You could rewrite the `printCircle()` method in the `Circle` class as follows:

```
public void printCircle() {
  System.out.println("The circle is
  created " + super.getDateCreated() +
  " and the radius is " + radius);
}
```

Overriding Methods in the Superclass

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*.

```
public class Circle extends GeometricObject {
  // Other methods are omitted

  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

NOTE

- An instance method can be overridden only if it is accessible.
- Thus a private method cannot be overridden, because it is not accessible outside its own class.
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

NOTE

- Like an instance method, a static method can be inherited. However, a static method cannot be overridden.
- If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

Overloading VS. Overriding

- **Overloading:** is the process of having more than one method in the same class, or inherited from some superclass, with the same name but a different method signature, i.e. different types and/or numbers or arguments.
- **Overriding:** is the process of having more than one methods with same name and same signature in super and sub class but with different implementation. Wherein a subclass redefines or replaces the inherited method.

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Overriding

```
public class Parent{
  public void method(int i)
  { System.out.println("parent:"+i);}
}
public class Child extends Parent {
  public void method(int i)
  { System.out.println("child:"+i*2);}
}
```

```
public class Test{
  public static void main()
  {  Parent p = new Parent();
     Child c = new Child();
     Parent k = new Child();
     p.method(2);
     c.metod(2);
     k.method(2);
   }
}
```

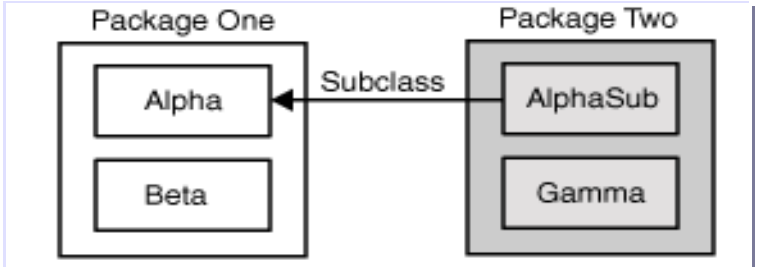
The output is:
parent 2
child 4
child 4

Access modifiers

There are three keywords associated with controlling levels of access to class members in Java: **public, protected & private. These are known as access modifiers.**

There is actually a fourth level of access, 'default'. There is no 'default' keyword; default access arises when none of the access modifiers is specified.

The following example illustrates the accessibility:



Example:

Visibility for a ref. var. defined in Alpha

Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- **`public`**

  The class, data, or method is visible to any class in any package using qualified name (a reference and a dot notation).

- **`private`**

  The data or methods can be accessed only by the declaring class. The get and set methods are used to read and modify private properties.

- **`Protected`**

  The data or methods are accessible inside class where they are declared, and inside subclasses. All class in the same package can access protected member using qualified name.

- **No modifier: default members** can be accessed by all classes in the same package using qualified name

```java
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```java
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```java
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

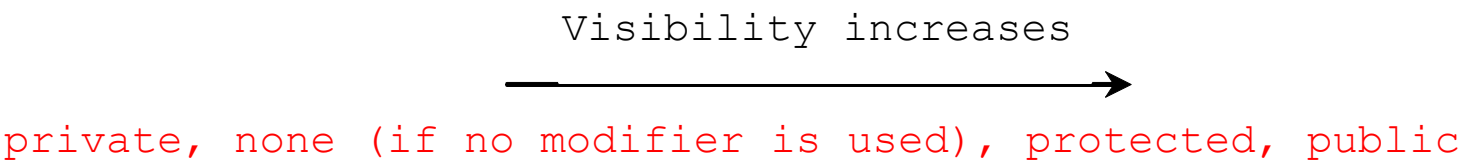
```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

The default modifier on a class restricts access to within a package, and the public modifier enables unrestricted access.

The `protected` Modifier

- The **`protected`** modifier can be applied on data and methods in a class. A protected data 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 a different package.

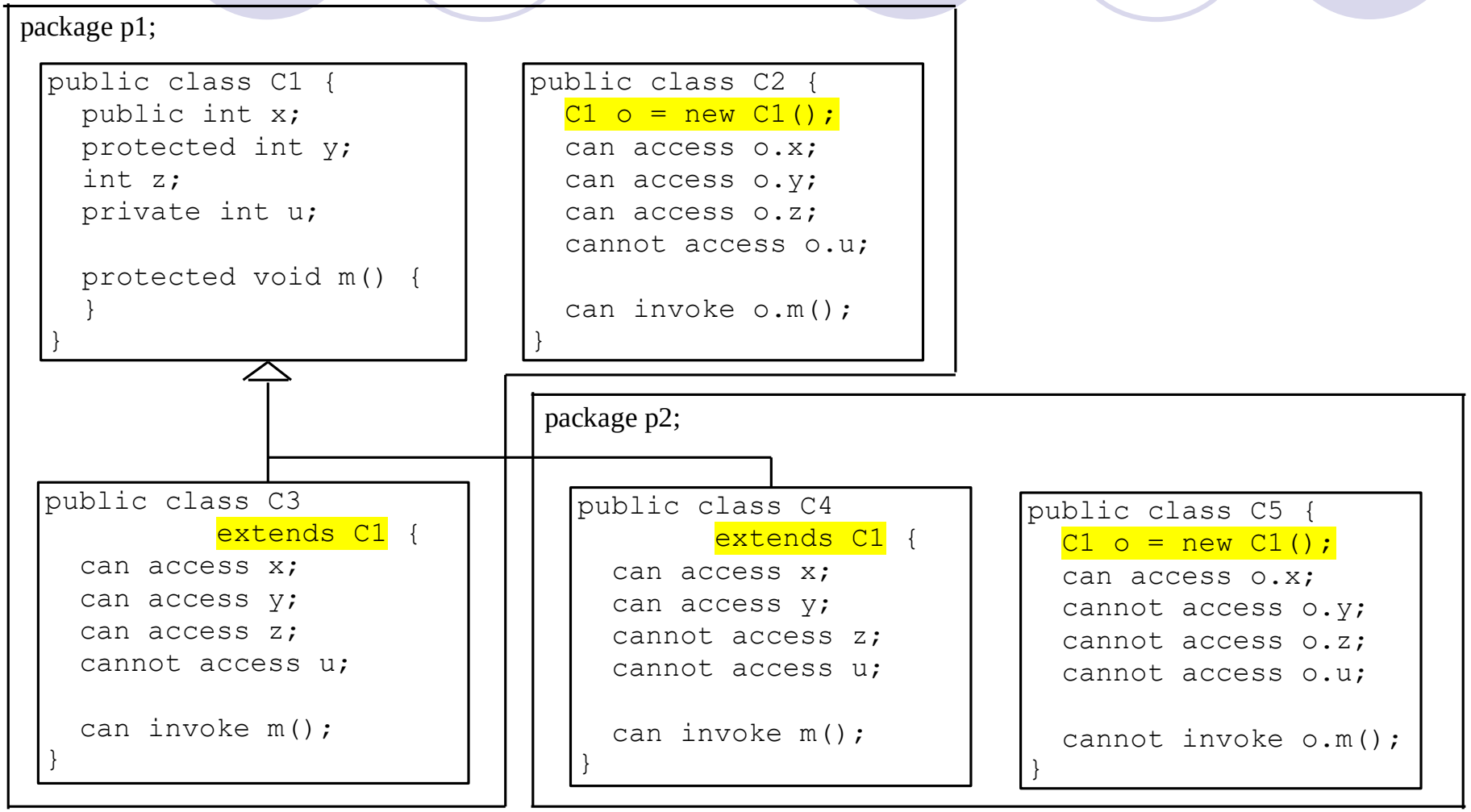
- private, default, protected, public



Accessibility Summary

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	✓	–	–	–

Visibility Modifiers



A Subclass Cannot Weaken the Accessibility

- 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.

NOTE

- The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method.
- A final local variable is a constant inside a method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
   ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

The Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```

The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

The `equals` Method

The **`equals()`** method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return this == obj;
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

NOTE

- The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references.
- The **equals** method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects.
- The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

The `instanceof` Operator

Use the **`instanceof`** operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance
  of Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is "
  +
    ((Circle)myObject).getDiameter());
  ...
}
```

Exercise

Use the `GemotricObject` class and its subclassess `Cirlce` and `Rectangle` to do the following:

1. Override Object's toString() method in Circle class to returnthe status of Circle object.
2. Override Object's toString() method in Rectangle class to return the status of Rectangle object.
3. Overide Object's equals() method to compare two circle objects.
4. Overide Object's equals() method to compare two Rectangle objects.
5. Test the above statements in the main method.

Polymorphism

Polymorphism: the ability for a word or symbol to mean different things in different contexts.

Types of Polymorphism:

- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Having Methods with the same name in different classes with relationship or without. These methods may differ in arguments, implementation or no change at all. Overriding and overloading are types of polymorphism.

Polymorphism in Java

Polymorphism means that a variable of a supertype can refer to a subtype object.

- A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.

- Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

PolymorphismDemo Run

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

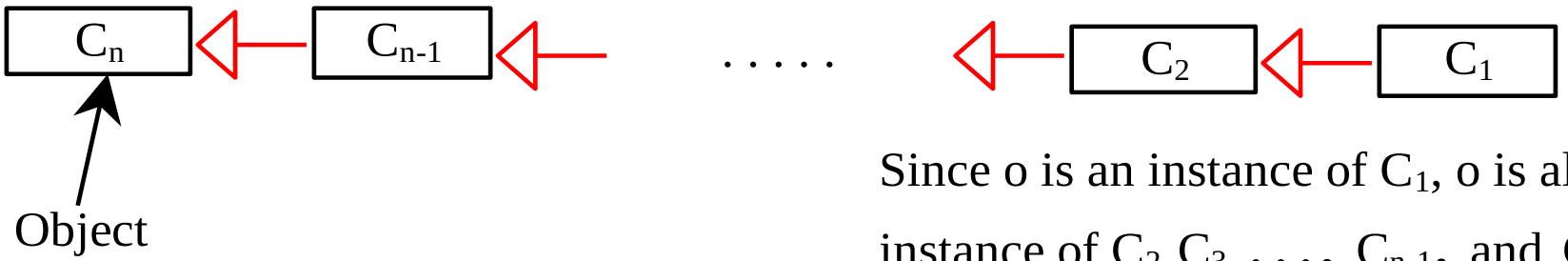
DynamicBindingDemo

Run

Output:
Student
Student
Person
java.lang.Object@b8bef7

Dynamic Binding

- Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class.
- In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues.

- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
- A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

- Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming.
- If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String).
- When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy.

In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student();   // Implicit casting
m(o);
```

The statement Object o = new Student(), known as **implicit casting**, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference **o** to a variable of the Student type using the following statement:

```
Student b = o;  //A compiler error would occur
```

Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't?

This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.

Even though you can see that **o** is really a Student object, the compiler is not so clever to know it.

To tell the compiler that **o** is a Student object, use an explicit casting.

The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o;   // Explicit casting
```

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)Fruit;   //Fruit is superclass

Orange x = (Orange)Fruit;
```

Abstract method in Abstract class

- An abstract class defines a common set of methods and a common set of instance variables for its subclasses. An abstract class could contain abstract methods.
- An abstract method: a method header defined in a class without implementation.
- **One cannot create instances of abstract classes.**
- Abstract classes are specified in the class header using the Java keyword abstract.

```
public abstract class Class_Name{
        ……….
        ……….
        public abstract return_data_type method_name();
}
```

Abstract Class & Concrete class

- A concrete sub class is needed to extend an abstract class, and the sub class should implement the abstract method using the same method header (overridden).
- Concrete class: is a fully implemented class, with no abstract methods.
- Concrete methods: a fully implemented methods.
- The concrete subclass that is extended from an abstract class, should implement all the abstract methods, even if they are not used in the subclass.

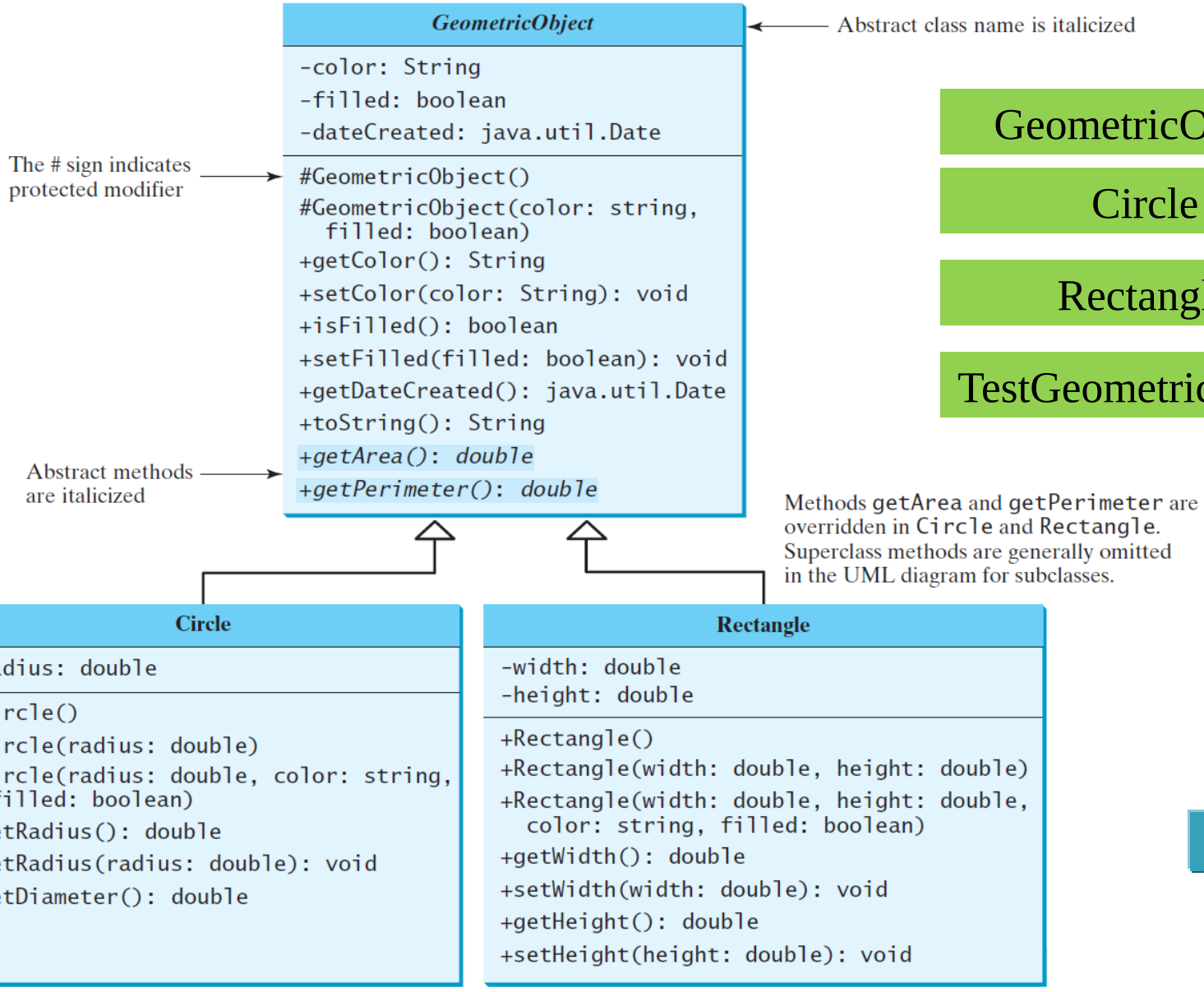
Abstract Class

- Abstract classes usually define abstract methods, that are then overridden in the concrete classes derived from them.
- It is possible to declare variables of an abstract class type.
- It is not possible to create objects of an abstract class.

Abstract Class & abstract methods

- When to use abstract classes?
- Usually when you have many classes that have common features:
  1. Common attributes,
  2. Common methods, that has the same signature and implementation.
  3. Common methods that has the same signature but different implementation.
- In this case you create an abstract class (Super class) that has all the common features, wherein the common methods that have same signature but different implementation are represented as abstract methods.
- And you will need concrete subclasses to complete the implementation of abstract methods.

Abstract Classes and Abstract Methods



GeometricObject

Circle

Rectangle

TestGeometricObject

Run

Example: SimpleGeometricObject Class

```
import java.util.Date;
public class GeometricObject {
  private String color = "white";
  private boolean filled;
  private Date dateCreated;

  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }
  protected GeometricObject(String
                color, boolean filled) {
    dateCreated = new Date();
    this.color = color;
    this.filled = filled;
  }
 public String getColor() { return color; }

 public void setColor(String color)
      {this.color = color;}
```

```
public boolean isFilled()
{    return filled;      }

public void setFilled(boolean filled)
{   this.filled = filled;}

 public Date getDateCreated()
 {
    return dateCreated;
 }

 public String toString() {
    return "created on " +
              dateCreated +
             "\ncolor: " + color +
              " and filled: " +
               filled;
  }

 public abstract double getArea();
 public abstract double getPremieter();

}
```

Continue: Circle class

```
public class Circle extends GeometricObject
{
  private double radius;

  public Circle() { }

  public Circle(double radius)
  { this.radius = radius; }

  public Circle(double radius, String
                color, boolean filled)
  { this.radius = radius;
    setColor(color);
    setFilled(filled); }

  public double getRadius()
  { return radius; }

  public void setRadius(double radius)
  { this.radius = radius; }
```

```
public double getDiameter()
 { return 2 * radius; }
public double getArea()
  { return radius * radius *
      Math.PI; }
 public double getPerimeter()
 { return 2 * radius * Math.PI; }
 public String toString() {
   return ("The circle is " +
         super.toString()+
         " and the radius is " +
          radius);
  }
}
```

Continue: Class Rectangle

```
public class Rectangle extends
SimpleGeometricObject {
  private double width, height;

  public Rectangle() { }

  public Rectangle( double w, double h)
  { this.width = w; this.height = h; }

  public Rectangle( double w, double h,
       String color, boolean filled)
  { this.width = w; this.height = h;
    setColor(color); setFilled(filled);
  }
  public double getWidth()
  { return width; }
 public void setWidth(double width)
  { this.width = width;     }
 public double getHeight()
  { return height; }
```

```
public void setHeight(double height)
 { this.height = height; }

 public double getArea()
 { return width * height; }

 public double getPerimeter()
 { return 2 * (width + height); }
 public String toString() {
   return "The rectangle is " +
         super.toString()+
         " the height = " + height+
         "width =" +  width;

  }
}
```

Abstract class without abstract method

A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

Concrete method overridden to be abstract

A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

Abstract class as type

You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new    GeometricObject[10];
```

NOTE: We are not creating Gemoetric objects; only Geometric references.

Continue

- We can then store references to objects of any subclass of GeometricObjects in the array, e.g.
- geo[0] = new Circle(5);
- geo[1] = new Rectangle(3,4);
- We can use geo[] to call any method in GeometricObject class, in case of abstract methods the overridden concrete ones will be invokes based on the type.
- System.out.println(geo[0].getArea());
- This will invoke getArea() of Circle Class.
- However, you could not call geo[0].getRadius() because geo[] is of type GemotricObject class and getRadius() is not a method there.
-

Interfaces

- What is an interface?
- Why is an interface useful?
- How do you define an interface?
- How do you use an interface?

What is an interface? Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods.

An interface specifies a list of methods that a group of unrelated classes should implement, so that their instances can interact together by responding to a common subset of messages.

Interfaces

- In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.

- For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Interfaces

- Interfaces are “like” classes, but they may ONLY have:

1. Abstract methods, i.e. method header but NO method code.
   - All methods in an interface are automatically **public.**
   - No need to writ public in the method header.

2. Constants, but NO instance variables.
   - All constants in an interface are automatically **public static final.**
   - Note that an interface is NOT a class and will have NO instances of its own, and so it has NO attributes and NO constructor.

Why is an interface useful?

- To have unrelated classes implement similar methods (behaviors)
- To model multiple inheritance (i.e. to “inherit” from multiple interfaces) – you want to impose multiple sets of behaviors to your class.
- Some Java methods can be applied to objects only if implement certain interfaces

Interface is a Special Class

- An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.
- Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class.
- For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName
{
  constant declarations;
  abstract method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

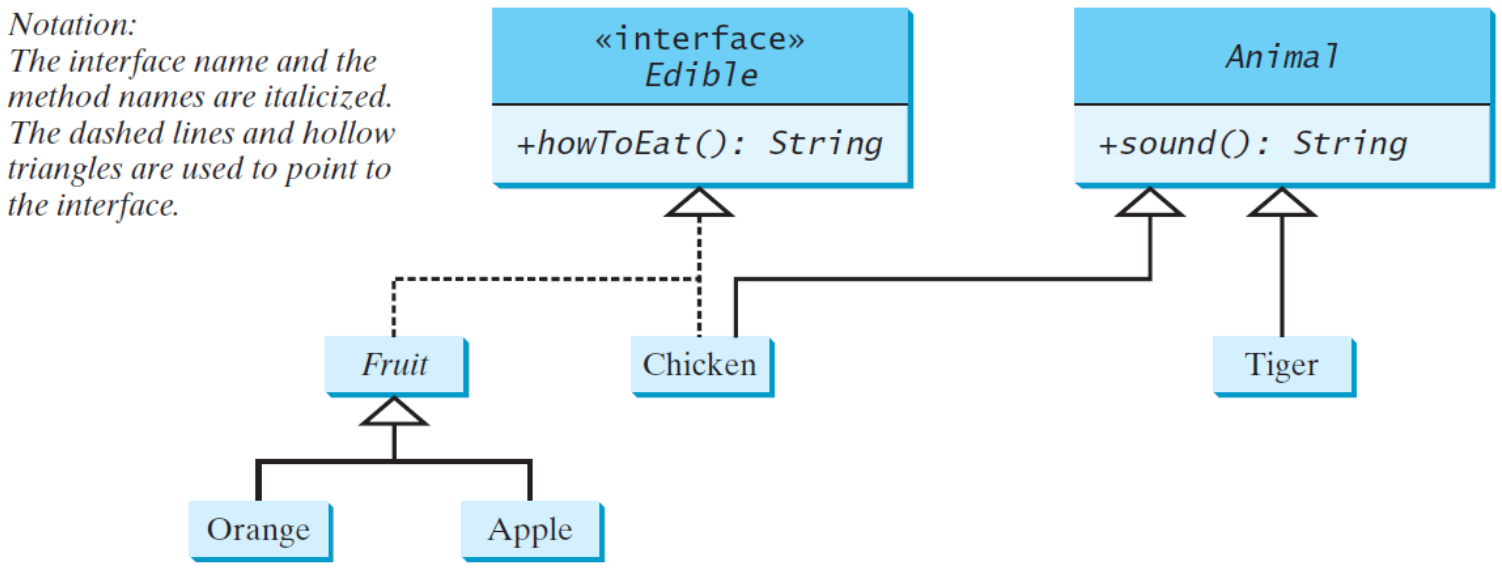
How to use an Itnerface?

- To make use of an interface we need a class to **implement the interface.**
- The class does this by providing the code for any abstract methods in the interface.

```
public class Chicken implements Edible{
……
public String howToEat() {    }
  }
```

Example

You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).



Edible TestEdible Run

Edible Interface

```
public interface Edible {
    public abstract String howToEat();
}
```

```
public class Chicken extends Animal implements Edible {

   public String howToEat() { return "Chicken: Fry it"; }

   public String sound() { return "Chicken: cock-a-doodle-doo"; }

}
```

Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface.

For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E>
{
  public int compareTo(E o);

}
```

Example

```
1  System.out.println(new Integer(3).compareTo(new Integer(5)));
2  System.out.println("ABC".compareTo("ABE"));
3  java.util.Date date1 = new java.util.Date(2013, 1, 1);
4  java.util.Date date2 = new java.util.Date(2012, 1, 1);
5  System.out.println(date1.compareTo(date2));
```

The Comparable Interface

- This interface allows comparing or sorting objects.
- Any class that implements the **Comparable interface** must have a met
- The type that the comparison will involve is added in <>.
- The compareTo method allows the class to define an appropriate measure of comparisonhod **compareTo.**

Example

In class below, comparison is based on salary.

```
class Employee implements Comparable <Employee>
{
    private String name;
    private String address;
    private float salary;
    ...
    public int compareTo (Employee otherEmployee)
    {
        if (salary > otherEmployee.salary) return 1;
        if (salary < otherEmployee.salary) return -1;
        return 0;
    }
```

How to use the 'comparable' Employee class?

Usage1:

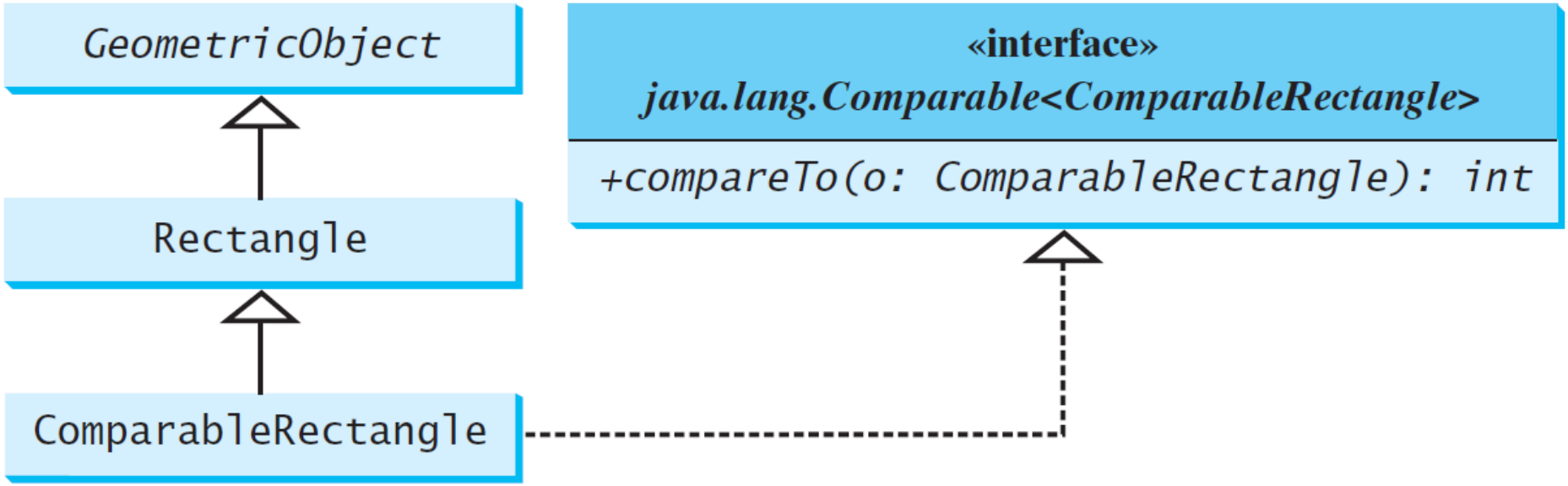
```
Employee e1 = new Employee("ali", "kw", 1000);
Employee e2 = new Employee("hmd", "kw", 2000);
System.out.println(e1.compareTo(e2));
```

Usage2:

- **Arrays** is a special class in **java.util**, with a static method called **sort**, which can be used to arrange the objects of an array in order, **given that** the objects have **CompareTo** method.

```
Employee [ ] staff = new Employee [STAFF_COUNT] ;
...
Arrays.sort(staff);
```

Defining Classes to Implement Comparable



ComparableRectangle SortRectangles Run

ComparableRectangl Class

- public class ComparableRectangle extends Rectangle implements Comparable<ComparableRectangle> {
- public ComparableRectangle(double width, double height) { super(width, height); }
- public int compareTo(ComparableRectangle o) {
- if (getArea() > o.getArea())
- return 1;
- else if (getArea() < o.getArea())
- return -1;
- else
- return 0; }
- public String toString() {
- return "Width: " + getWidth() + " Height: " + getHeight() + " Area: "
- + getArea();}
- }

Multiple inheritance in Java is not allowed

- Java avoids multiple inheritance problems by the use of interfaces.
- Class can implement more than one interface, and extend one class ONLY– this gives an alternative to multiple inheritance.
- public class MonthlyEmployee
- extends Employee implements Comparable, Serializable { }

Exercise

Assume `Employee` and `Parent` are classes; `Comparable` and `Serializable` are standard Java interfaces. Are these declarations valid? If not, why not?

(a)
```
class MonthlyEmployee extends Employee, Parent
implements Comparable
```

(b)
```
class MonthlyEmployee extends Comparable
```

(c)
```
class MonthlyEmployee extends Employee implements Parent
```

(d)
```
class MonthlyEmployee implements Comparable, Serializable
```

(e)
```
public interface Sortable extends Comparable
```

(f)
```
public interface Sortable extends Parent
```

(g)
```
Comparable employee1 = new Comparable();
```

ANSWER.....

(a) Invalid – a class cannot inherit from more than one superclass.

(b) Invalid – a class cannot extend an interface, although it can implement it.

(c) Invalid – a class cannot implement another class: we cannot get multiple inheritance this way!

(d) Valid – a class can implement more than one interface.

(e) Valid – an interface can inherit from another interface.

(f) Invalid – an interface cannot inherit from a class.

(g) Invalid – you can define a variable (like `employee1`) of an interface type but you cannot create an object of an interface type.

Interfaces vs. Abstract Classes

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

- In an interface, the data must be constants; an abstract class can have all types of data.
- Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.
- All classes share a single root, the Object class, but there is no single root for interfaces.

Interfaces vs. Abstract Classes

- **DIFFERENCES**
- **Interface:**
  - An interface is not a class.
  - All its methods are abstract
  - Only constant data is allowed.
  - An interface can be implemented by any number of unrelated classes, which declare this using the implements keyword. A class may implement any number of interfaces.
- **Abstract class:**
  - An abstract class uses the keyword abstract in the class header.
  - It normally has at least one abstract method, either defined within the class or inherited from a superclass, and its abstract methods must be explicitly declared abstract.
  - It can also have concrete methods (that is, it may be fully implemented)
  - It can also have instance variables, unlike an interface.
  - It can only constrain its own concrete subclasses, by requiring them to implement its abstract methods.
  - All classes share a single root, the Object class, but there is no single root for interfaces.

Interfaces vs. Abstract Classes

- **SIMILARITIES**
- They can both place requirements on objects of other classes.
- Both can have abstract methods (although neither need have abstract methods).
- You cannot create objects of an abstract class type or an interface type.
- You can create reference variables of either type. These are normally used to refer to an object of a subclass of the abstract type or to an object of a class that implements the interface, respectively.
- Both can inherit: the abstract class from another class; the interface only from another interface.

Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features.

How do you decide whether to use an interface or a class?

- In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person.

- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces.

- For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface.

Guidelines for Designing a Class

- (Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.

- You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

Designing a Class, cont.

(Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities.

The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities.

- The String class deals with immutable strings.
- The StringBuilder class is for creating mutable strings.
- The StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

Designing a Class, cont.

- Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it. To do this follow the below instructions:

Designing a Class, cont.

- Design the properties (attributes) to ensure that the user can set properties in any order, with any combination of values,
- Design methods to function independently of their order of occurrence.
- Provide a public no-arg constructor.
- override the equals() method and the toString() method defined in the Object class whenever possible.
- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and initialize variables to avoid programming errors.

Using Visibility Modifiers

- Each class can present two contracts – one for the users of the class and one for the extenders of the class.
- Make the fields private and accessor methods public if they are intended for the users of the class.
- Make the fields or method protected if they are intended for extenders of the class.
- The contract for the extenders encompasses the contract for the users.
- The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.

Using Visibility Modifiers, cont.

- A class should use the private modifier to hide its data from direct access by clients.
- You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify.
- A class should also hide methods not intended for client use. In this case the method is declared as private and only used by another method inside the class. This is know as helper method.

Using the static Modifier

A property that is shared by all the instances of the class should be declared as a static property.

- Please solve the Exercises.