




Application Program Development

Segment : Interfaces and Implementations




Agenda for Week 2

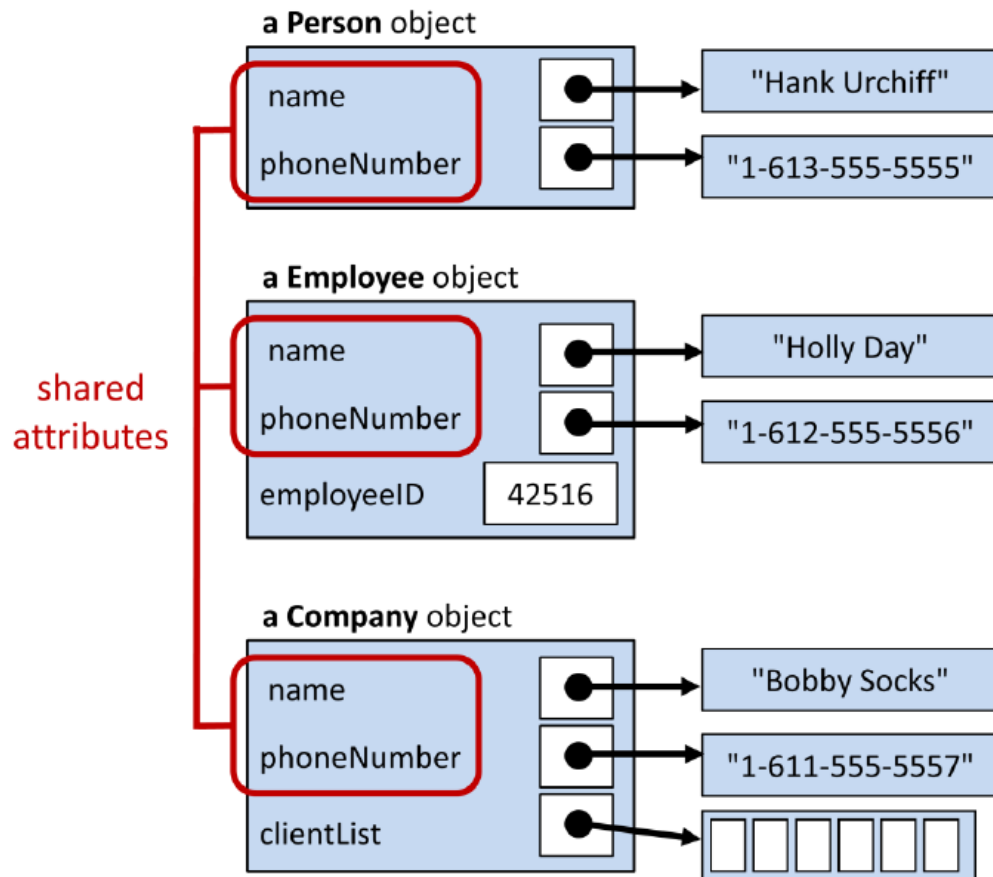
- Lecture
 - Inheritance
 - Abstract classes
 - Multiple Inheritance & Interfaces
 - Organizing Classes
 - Java Interfaces
 - Lab
 - Test 1
- 



Outcomes

- Organizing Classes
 - Understanding of Inheritance.
 - Understanding of Abstract classes.
- 

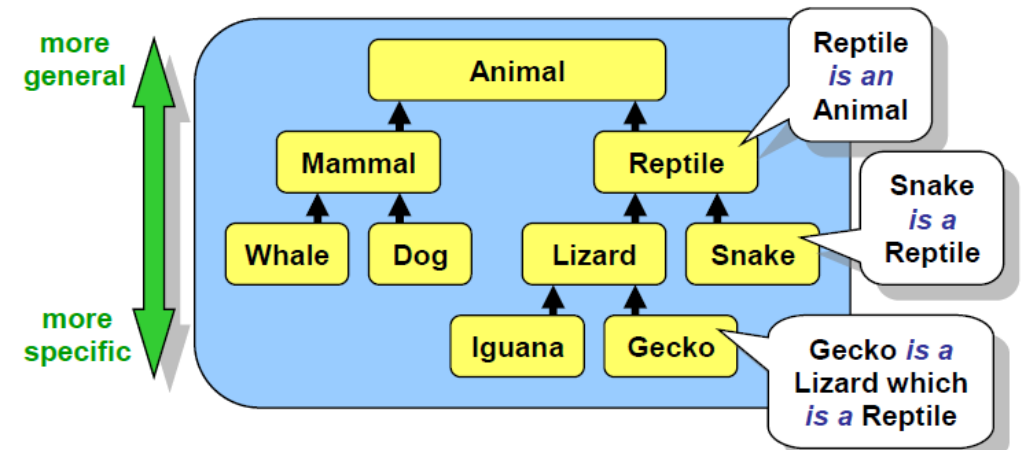
Organizing class: The need



- Defining objects as a new kind of data structure simply involves creating new **classes**, each in their own file (e.g., Car, Person, Address, Bank, etc..).
- However, there are some objects that “share” attributes in common. For example, **Person** objects may have **name** and **phoneNumber** attributes, but so can **Employee**, **Manager**, **Customer** and **Company** objects.
 - For example, an **Employee** object may maintain **employeeID** information or a **Company** object may have a **clientList** attribute, whereas **Person** objects in general do not keep such information
- All object-oriented languages (e.g., JAVA) allow you to organize your classes in a way that allows you to take advantage of the commonality between classes.

Class hierarchy

- A class hierarchy is often represented as an upside-down tree.
- A **child** object defined in the tree is a *more specific kind* of object than its **parent** or *ancestors* in the tree. Hence, there is an "**is a**" (i.e., "is-a-kind-of") relationship between classes.
- Each class is a **subclass** (i.e., a specialization) of some other class which is called its **superclass** (i.e., a generalization). The **direct superclass** is the class right "above" it.




Inheritance (Quick overview)

In JAVA, in order to create a subclass of another class, use the **extends** keyword in our class definition.

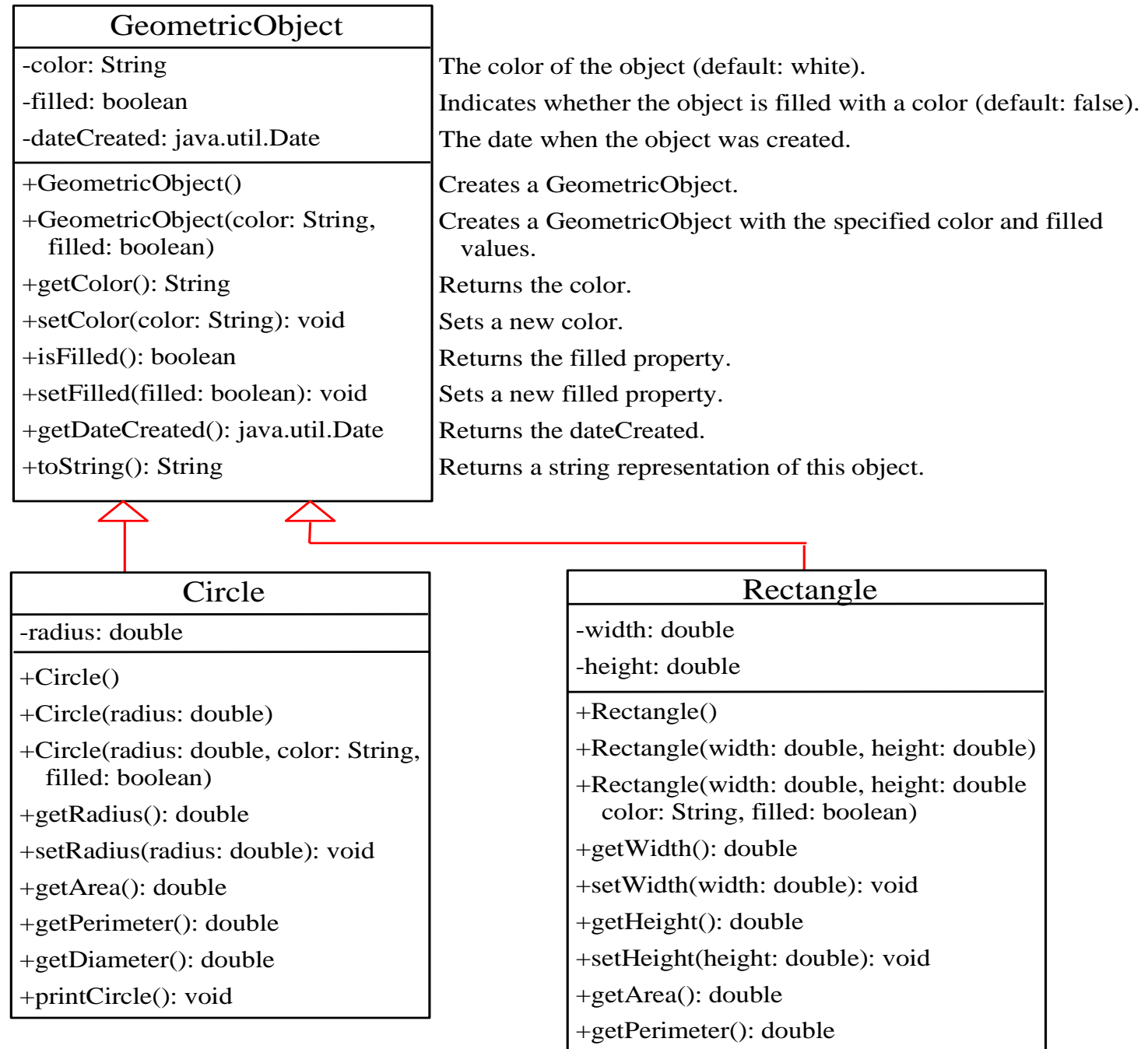
- The **superclass** refers to a direct ancestor.
- A **subclass** inherits state and behavior from all its ancestor.
- Subclass inherits all, but private superclass's members

```
public class SuperClass
{ ... }

public class SubClass extends SuperClass {
    ...
}
```

An upward-pointing arrow is positioned between the 'extends' keyword in the SubClass definition and the SuperClass name, indicating that SubClass inherits from SuperClass.

Superclass and Subclasses



Are Superclass's Constructor Inherited?

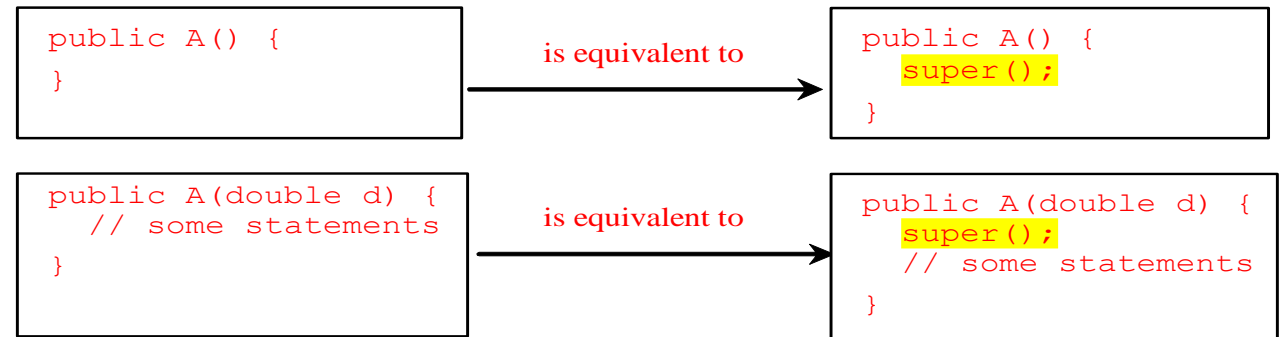
- No. They are not inherited.
- They are invoked explicitly or implicitly.
- Explicitly using the ***super*** keyword.
- Unlike properties and methods, a superclass's constructors are not inherited in the subclass.
- Superclass constructor can only be invoked from the subclasses' constructors, using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

```
super(<expression>, <expression>, ..., <expression>);
```


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
- To call a superclass method



Note:

- You must use keyword `super` to the superclass's constructor.
- Invoking the name of the superclass's constructor in a subclass causes an error.
- `super` keyword must appear first in the constructor.

Constructor Chaining

- Constructing an instance of a class invokes all the super classes' constructors along the inheritance chain. This is called *constructor chaining*.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

Find out the errors in the program:

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

Example

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

    /** Construct a default geometric object */
    public GeometricObject() {
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color
     *  and filled value */
    public GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor() {
        return color;
    }

    /** Set a new color */
    public void setColor(String color) {
        this.color = color;
    }
```



```
/** Return filled. Since filled is boolean,
    its get method is named isFilled */
    public boolean isFilled() {
        return filled;
    }

    /** Set a new filled */
    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    /** Get dateCreated */
    public java.util.Date getDateCreated() {
        return dateCreated;
    }

    /** Return a string representation of this object */
    public String toString() {
        return "created on " + dateCreated + "\n" + "color: " + color
+
        " and filled: " + filled;
    }
}
```

Example

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


    public CircleFromGeometricObject() {
    }

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

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

    /** Return radius */
    public double getRadius() {
        return radius;
    }

    /** Set a new radius */
    public void setRadius(double radius) {
        this.radius = radius;
    }
}
```



```
/** Return area */
public double getArea() {
    return radius * radius * Math.PI;
}

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

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

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

Example

```
public class RectangleFromGeometricObject extends
    GeometricObject {

    private double width;
    private double height;

    public RectangleFromGeometricObject() {
    }

    public RectangleFromGeometricObject(
        double width, double height) {
        this.width = width;
        this.height = height;
    }

    public RectangleFromGeometricObject(
        double width, double height, String color, boolean
filled) {
        this.width = width;
        this.height = height;
        setColor(color);
        setFilled(filled);
    }

    /** Return width */
    public double getWidth() {
        return width;
    }
```

```
    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }

    /** Return height */
    public double getHeight() {
        return height;
    }

    /** Set a new height */
    public void setHeight(double height) {
        this.height = height;
    }

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

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

Example

```
public class TestCircleRectangle {  
    public static void main(String[] args) {  
        CircleFromGeometricObject circle = new CircleFromGeometricObject(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());  
  
        RectangleFromGeometricObject rectangle = new RectangleGeometricObject(2, 4);  
        System.out.println("\nA rectangle " + rectangle.toString());  
        System.out.println("The area is " + rectangle.getArea());  
        System.out.println("The perimeter is " + rectangle.getPerimeter());  
    }  
}
```

Output

The circle is created Sun Nov 06 15:47:12 EST 2022 and the radius is 1.0

A Circle color is white

The radius of the circle is 1.0

Th area of the circle is 3.141592653589793

The diameter of the circle is 2.0

A rectangle created Sun Nov 06 14:47:12 EST 2022

Color: white and filled: false

The area is 8.0

The perimeter is 12.0

Overriding

- Definition:

Replacing the superclass's implementation with a new method in a subclass is called *overriding*.

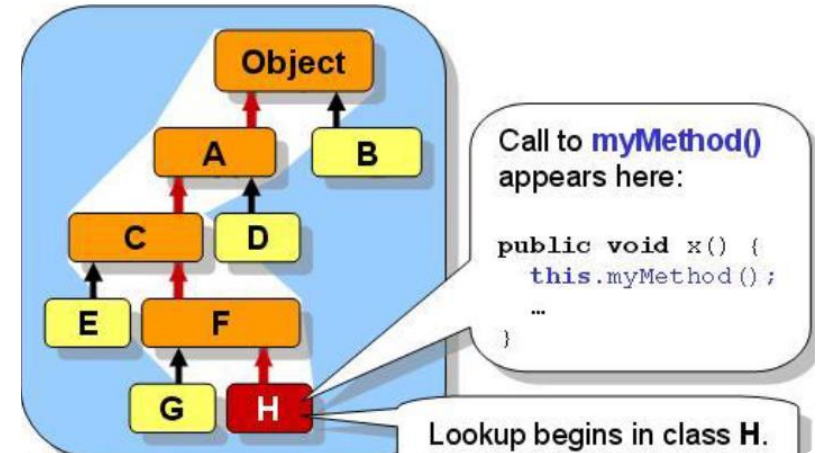
- The signature should be identical.
- Only accessible non-static method can be overridden.
- Access modifier could be different in overridden method as long as the subclass modifier is less restrictive than the superclass.

Note:

- A static method can be inherited.
- A static method cannot be overridden.
- If you redefine the static method from the superclass into its subclass then the method of the superclass will be hidden.

Method calling VS Overriding

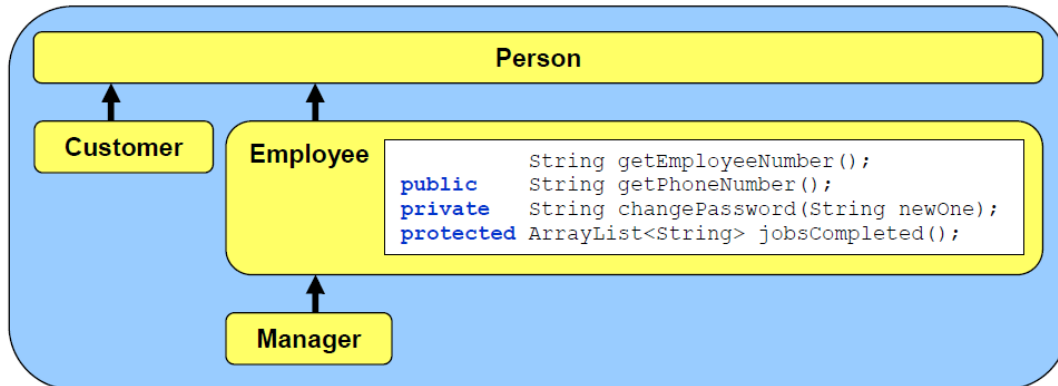
- How JAVA "finds" a method in the class hierarchy when you try to call it.
 - It can be confusing if there are many "overridden" methods (i.e., all with the same name and parameter lists).
 - Whenever you call a method from a class directly (e.g., **this.myMethod()**), JAVA looks first to see whether or not you have such a method in the class that you are calling it from.
 - If it finds it there, it evaluates the code in that method.
 - Otherwise, JAVA tries to look for the method up the hierarchy (never down the hierarchy) by checking the **superclass**.
 - If not found there, JAVA continues looking up the hierarchy until it either finds the method that you are trying to call, or until it reaches the **Object** class at the top of the tree.
 - Use of **this** simply tells the compiler to start looking from this class (Current class)



How are access Modifiers affected by Inheritance?

- It would be good to consider the effects that access modifiers have on attributes and methods within the class hierarchy.
- When an inherited attribute is declared as **private**, the subclasses still inherit it, but they cannot access it directly from within their own "local" code.

How do **private** and **protected** modifiers affect methods?



Now consider some code within the **Manager** class that attempts to access these methods

```
public class Manager extends Employee {
    public void tryThingsOut() {
        System.out.println(this.getEmployeeNumber()); // access allowed
        System.out.println(this.getPhoneNumber()); // access allowed
        System.out.println(this.changePassword("12345678")); // compile error
        System.out.println(this.jobsCompleted()); // access allowed
    }
}
```

- The only method not allowed to be accessed is the **private** method, since the **tryThingsOut()** method is written in the **Manager** class, not in **Employee**.
- Consider now the **Customer** class restrictions

```
public class Customer extends Person {  
    public void buyFrom(Employee emp) {  
        System.out.println(emp.getEmployeeNumber()); // access allowed  
        System.out.println(emp.getPhoneNumber()); // access allowed  
        System.out.println(emp.changePassword("12345678")); // compile error  
        System.out.println(emp.jobsCompleted()); // compile error  
    }  
}
```

- Now we can no longer call the **jobsCompleted()** method, since it has been declared **protected** and **Customer** is not a subclass of **Employee**

Final Classes / Methods

- A class can be declared as final with the declaration:

```
public final class X { ...}
```

- A class that is declared final cannot be subclassed

Example: `java.lang.String`

- A method can be declared as final with the declaration:

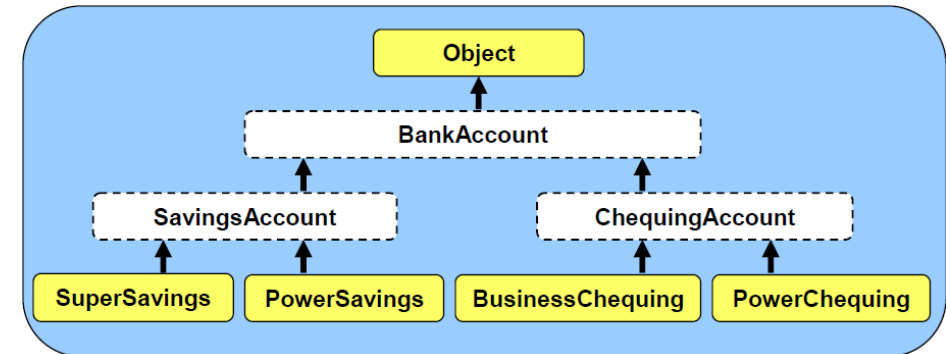
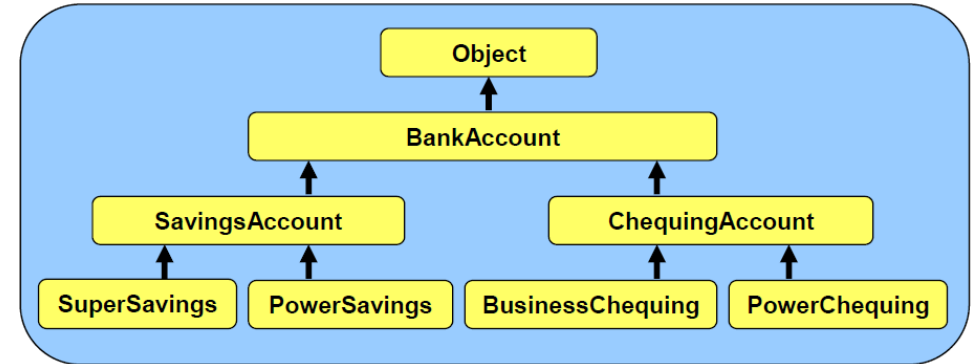
```
public class Y {  
    public final void m() {...}  
}
```

- A method that is declared final cannot be overridden or hidden by subclasses

Abstraction

- As abstraction says hide the implementation from the user by showing only functionality.
- How to achieve Abstraction?
 - Two ways to achieve abstraction in java:
 1. Abstract classes.
 2. Interfaces.

- Assume that the real bank actually has exactly 4 types of accounts so that when someone goes to the bank teller to open a new account, they specify whether or not they want to open a **SuperSavings**, **PowerSavings**, **BusinessChequing** or **PowerChequing** account.
- The four classes representing the accounts that we can actually open are called **concrete** classes.
- A **concrete class in JAVA is a class that we can make instances of directly by using the new keyword.**
- In JAVA, we actually use the term **abstract class** to define a class that we do not want to make instances of.
- An **abstract class is a class for which we cannot create instances.**



Abstract classes and Abstract Methods

- Class declared with a keyword **abstract**, known as an abstract class.

```
abstract class Bookmark{  
    ...  
}
```

- Needs to be extended.
- Cannot be instantiated.
- An abstract class can contain abstract methods that are to be implemented in concrete subclasses.
- Abstract classes can also contain non-abstract methods.
- If a subclass doesn't want to override the abstract methods of the superclass, then subclass needs to be an abstract as well

An Abstract method

- Declared with the keyword **abstract** and does not contain any implementation

```
abstract void someMethod();
```

- An abstract method cannot be contained in a *non-abstract* class.
- A subclass can be abstract even if its superclass is concrete.

No
Body

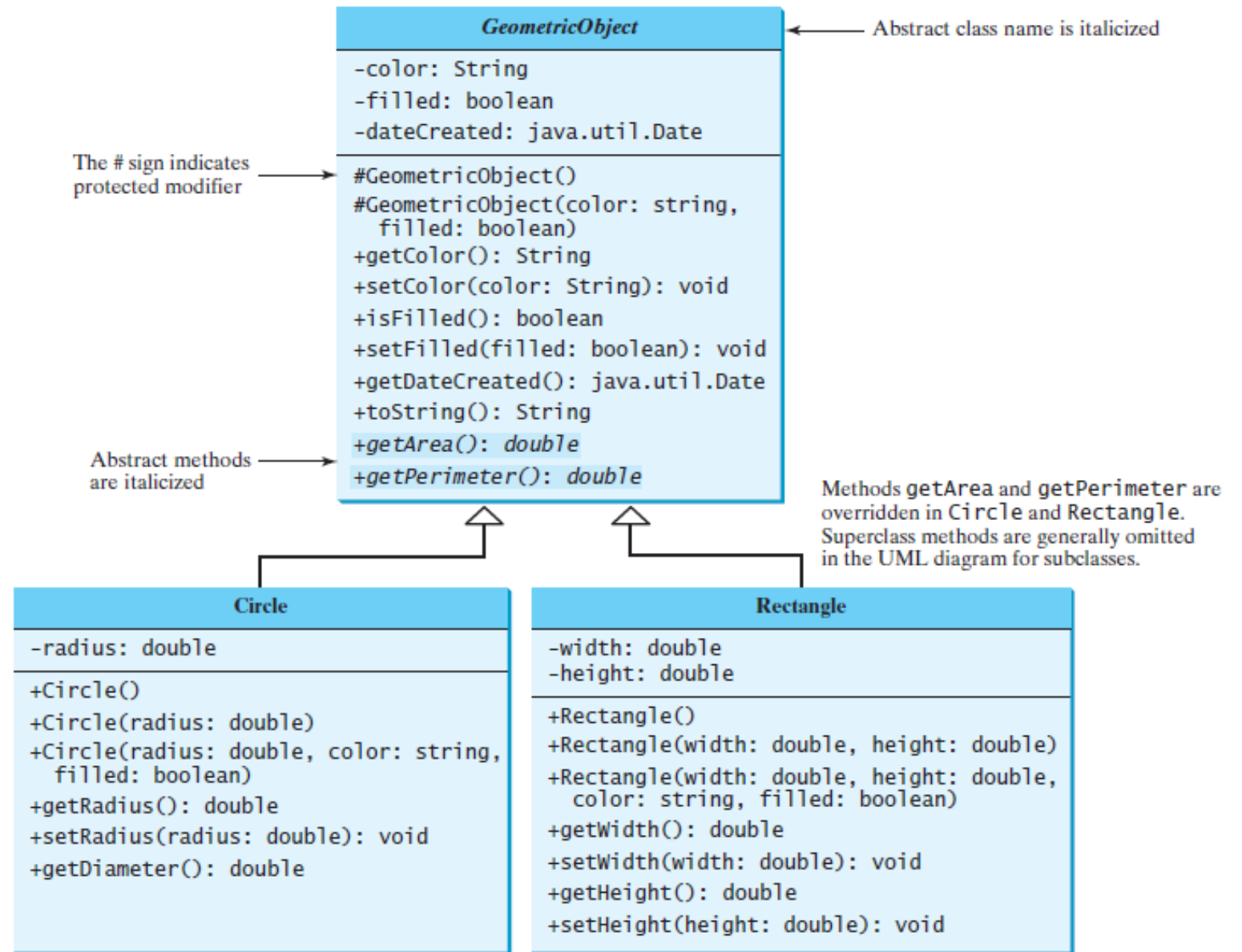
Can't be
static

Must be
overridden

Abstract Method: Why?

- Abstract methods are actually never called, so JAVA never attempts to evaluate their code.
- Just as an abstract class is used to force the user of that class to have subclasses, an abstract method *forces* the subclasses to ***implement*** (i.e., to write code for) that method.
- So, by defining an abstract method, you are really just informing everyone that the concrete subclasses must write code for that method.
- All concrete subclasses of an **abstract** class MUST implement the **abstract** methods defined in their superclasses, there is no way around it.
- When JAVA compiles an **abstract** method for a class (e.g., class **A**), it checks to see whether or not all the subclasses of **A** have implemented the method (i.e., that they have written a method with the same return type, name and parameters).
- Only **abstract** classes are allowed to have abstract methods.
- However, an **abstract** class may have regular methods as well.

Abstract Class



Example

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

    /** Construct a default geometric object */
    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color
     *  and filled value */
    protected GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor() {
        return color;
    }

    /** Set a new color */
    public void setColor(String color) {
        this.color = color;
    }
}
```



```
/** Return filled. Since filled is boolean,
    its get method is named isFilled */
    public boolean isFilled() {
        return filled;
    }

    /** Set a new filled */
    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    /** Get dateCreated */
    public java.util.Date getDateCreated() {
        return dateCreated;
    }

    /** Return a string representation of this object */
    public String toString() {
        return "created on " + dateCreated + "\n" + "color: " + color
            + " and filled: " + filled;
    }

    /** Abstract method getArea */
    public abstract double getArea();

    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}
}
```

Example


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

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

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

```
  
    /** Return radius */  
    public double getRadius() {  
        return radius;  
    }
```

```
  
    /** Set a new radius */  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }
```



```
    /** Return area */  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }
```

```
  
    @Override /** Return diameter */  
    public double getDiameter() {  
        return 2 * radius;  
    }
```

```
  
    @Override /** Return perimeter */  
    public double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }
```

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

Example

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;

    public Rectangle () {
    }

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

    public Rectangle (double width, double height, String color,
                     boolean filled) {

        this.width = width;
        this.height = height;
        setColor(color);
        setFilled(filled);
    }

    /** Return width */
    public double getWidth() {
        return width;
    }
```



```
/** Set a new width */
public void setWidth(double width) {
    this.width = width;
}

/** Return height */
public double getHeight() {
    return height;
}

/** Set a new height */
public void setHeight(double height) {
    this.height = height;
}

@Override /** Return area */
public double getArea() {
    return width * height;
}

@Override /** Return perimeter */
public double getPerimeter() {
    return 2 * (width + height);
}
}
```

Example

```
public class TestGeometricObject {
    /** Main method */
    public static void main(String[] args) {
        // Declare and initialize two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);

        System.out.println("The two objects have the same area? " +
            equalArea(geoObject1, geoObject2));

        // Display circle
        displayGeometricObject(geoObject1);

        // Display rectangle
        displayGeometricObject(geoObject2);
    }

    /** A method for comparing the areas of two geometric objects */
    public static boolean equalArea(GeometricObject object1, GeometricObject object2)
    {
        return object1.getArea() == object2.getArea();
    }

    /** A method for displaying a geometric object */
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println();
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }
}
```

Output

The two objects have the same area? false

The area is 78.53981633974483

The perimeter is 31.41592653589793

The area is 13.0

The perimeter is 16.0

Concrete class or Abstract class?

- How do we know which classes to make **abstract** and which ones to leave as **concrete** ?
 - If we are not sure, it is better to leave them as concrete.
 - However, if we discern that a particular class has subclasses that cover all of the possible concrete classes that we would ever need to create in our application, then it would be reasonable to make the superclass abstract.
- Is there any advantage of making a class **abstract** rather than simply leaving it **concrete** ?
- Yes.
 - By making a class **abstract**, you are informing the users of that class that they should not be creating instances of that class.
 - In a way, you are telling them “**If you want to use this class, you should make your own concrete subclass of it.**”.
 - You are actually *forcing* them to create a subclass if they want to use your abstract class.
 - It forces the user of your class to be more specific in their object creation, thereby **reducing ambiguity** in their code.