Inheritance

Introduction to Programming and Computational Problem Solving: Accelerated Pace
CSE 11
Lecture 12
Announcements

• Assignment 5 is due May 15, 11:59 PM
  – Upgrade beginning May 18, 12:01 AM
• Assignment 6 will be released May 15
  – Due May 22, 11:59 PM
Inheritance

• Suppose you define classes to model circles, rectangles, and triangles
• These classes have many common features
• What is the best way to design these classes so to avoid redundancy?
• Object-oriented programming allows you to define new classes from existing classes
• This is called inheritance
Superclasses and subclasses

• Inheritance enables you to define a general class (i.e., a *superclass*) and later extend it to more specialized classes (i.e., *subclasses*)

• A subclass inherits from a superclass
  – For example, both a circle and a rectangle are geometric objects
    • GeometricObject is a superclass
    • Circle is a subclass of GeometricObject
    • Rectangle is a subclass of GeometricObject

• Models *is-a* relationships
  – For example
    • Circle *is-a* GeometricObject
    • Rectangle *is-a* GeometricObject
Superclasses and subclasses

### GeometricObject

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-color: String</code></td>
<td><code>-width: double</code></td>
</tr>
<tr>
<td><code>-filled: boolean</code></td>
<td><code>-height: double</code></td>
</tr>
<tr>
<td><code>-dateCreated: java.util.Date</code></td>
<td>+Rectangle()</td>
</tr>
<tr>
<td>GeometricObject()</td>
<td>+Rectangle(width: double, height: double)</td>
</tr>
<tr>
<td>+GeometricObject(color: String, filled: boolean)</td>
<td>+Rectangle(width: double, height: double, color: String, filled: boolean)</td>
</tr>
<tr>
<td>+getColor(): String</td>
<td>+getWidth(): double</td>
</tr>
<tr>
<td>+setColor(color: String): void</td>
<td>+setWidth(width: double): void</td>
</tr>
<tr>
<td>+isFilled(): boolean</td>
<td>+getHeight(): double</td>
</tr>
<tr>
<td>+setFilled(filled: boolean): void</td>
<td>+getArea(): double</td>
</tr>
<tr>
<td>+getDateCreated(): java.util.Date</td>
<td>+getHeight(height: double): void</td>
</tr>
<tr>
<td>+toString(): String</td>
<td>+getPerimeter(): double</td>
</tr>
<tr>
<td>+printCircle(): void</td>
<td>+getPerimeter(): double</td>
</tr>
</tbody>
</table>

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

### Circle

- `radius: double` 
- Circle() 
- Circle(radius: double) 
- Circle(radius: double, color: String, filled: boolean) 
- getRadius(): double 
- setRadius(radius: double): void 
- getArea(): double 
- getPerimeter(): double 
- getDiameter(): double 
- printCircle(): void 

### Rectangle

- `width: double` 
- `height: double` 
- +Rectangle() 
- +Rectangle(width: double, height: double) 
- +Rectangle(width: double, height: double, color: String, filled: boolean) 
- getWidth(): double 
- setWidth(width: double): void 
- getHeight(): double 
- setHeight(height: double): void 
- getArea(): double 
- getPerimeter(): double 
- getPerimeter(): double
Superclasses and subclasses

- A subclass *inherits* accessible data fields and methods from its superclass and may also add *new* data fields and methods
  - A subclass is not a subset of its superclass
    - A subclass usually contains *more* information and methods than its superclass
  - For example
    - A rectangle has a width and height
    - A circle has a radius
    - Both have a color
Superclasses and subclasses

- A **superclass** is also called a *parent class* or *base class*
- A **subclass** is also called a *child class*, *extended class*, or *derived class*
  - A child class inherits from a parent class
  - A subclass extends a superclass
  - A derived class derives from a base class
Superclasses and subclasses

• Remember, 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
  – For example
    • Circle is a subtype of GeometricObject, and GeometricObject is a supertype for Circle
Inheritance

• The keyword extends tells the compiler that the (sub)class extends another (super)class

• A Java class may inherit directly from only one superclass
  – This restriction is known as single inheritance
  – Some other programming languages allow classes to inherit from one or more classes
    • This is known as multiple inheritance
extends keyword

• The keyword extends tells the compiler that the (sub)class extends another (super)class

• For example
  – The Circle class extends the GeometricObject class using the syntax
    ```
    public class Circle extends GeometricObject
    ```
  – The Circle class inherits the accessible data fields and methods of GeometricObject
Circle extends GeometricObject

<table>
<thead>
<tr>
<th>GeometricObject</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-color: String</td>
<td>The color of the object (default: white).</td>
</tr>
<tr>
<td>-filled: boolean</td>
<td>Indicates whether the object is filled with a color (default: false).</td>
</tr>
<tr>
<td>-dateCreated: java.util.Date</td>
<td>The date when the object was created.</td>
</tr>
<tr>
<td>+GeometricObject()</td>
<td>Creates a GeometricObject.</td>
</tr>
<tr>
<td>+GeometricObject(color: String, filled: boolean)</td>
<td>Creates a GeometricObject with the specified color and filled values.</td>
</tr>
<tr>
<td>+getColor(): String</td>
<td>Returns the color.</td>
</tr>
<tr>
<td>+setColor(color: String): void</td>
<td>Sets a new color.</td>
</tr>
<tr>
<td>+isFilled(): boolean</td>
<td>Returns the filled property.</td>
</tr>
<tr>
<td>+setFilled(filled: boolean): void</td>
<td>Sets a new filled property.</td>
</tr>
<tr>
<td>+getDateCreated(): java.util.Date</td>
<td>Returns the dateCreated.</td>
</tr>
<tr>
<td>+toString(): String</td>
<td>Returns a string representation of this object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius: double</td>
<td></td>
</tr>
<tr>
<td>+Circle()</td>
<td></td>
</tr>
<tr>
<td>+Circle(radius: double)</td>
<td></td>
</tr>
<tr>
<td>+Circle(radius: double, color: String, filled: boolean)</td>
<td></td>
</tr>
<tr>
<td>+getRadius(): double</td>
<td></td>
</tr>
<tr>
<td>+setRadius(radius: double): void</td>
<td></td>
</tr>
<tr>
<td>+getArea(): double</td>
<td></td>
</tr>
<tr>
<td>+getPerimeter(): double</td>
<td></td>
</tr>
<tr>
<td>+getDiameter(): double</td>
<td></td>
</tr>
<tr>
<td>+printCircle(): void</td>
<td></td>
</tr>
</tbody>
</table>

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

    public Circle() {
    }

    public Circle(double radius) {
        this.radius = radius;
    }
}
```
Unified Modeling Language (UML)

+ public
# protected
- private
• Static variables and methods are underlined
• Abstract class names and methods are *italicized*
• Interface names and methods are *italicized*
• Open or no arrow is association
• Hollow diamond is aggregation
• Filled diamond is composition
• Hollow triangle is inheritance
• Dashed line with hollow triangle is implementation of interface

Covered later in the quarter
# Methods and data fields visibility

<table>
<thead>
<tr>
<th>Modifiers on Members in a Class</th>
<th>Accessed from the Same Class</th>
<th>Accessed from the Same Package</th>
<th>Accessed from a Subclass in a Different Package</th>
<th>Accessed from a Different Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Default (no modifier)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Covered later in the quarter
Methods and data fields visibility

- Private members cannot be accessed outside of a class, including one of its subclasses
  - Use accessor (getter) and mutator (setter) methods

```java
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);
    }
```
Superclass constructors and the super keyword

• Remember, 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 constructors and the super keyword

• For example, replace this

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

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

with this

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

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

Invoking the superclass constructor using `super` must be the first statement in the subclass’s constructor.
Superclass constructors and the super keyword

• If the keyword `super` is not explicitly used, the superclass's no-arg constructor is automatically invoked (as the **first** statement in the constructor)

```java
public A() {
    // some statements
}
```

is equivalent to

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

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

is equivalent to

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

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

is equivalent to

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

```java
public A(double d) {
}
```

is equivalent to

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

• Constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain

• This is known as constructor chaining
Constructor chaining

```java
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() {
        this("(2) Invoke Employee's overloaded constructor");
        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");
    }
}
```
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() {
        this("(2) Invoke Employee’s overloaded constructor");
        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");
    }
}
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() {
        this("(2) Invoke Employee's overloaded constructor");
        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");
    }
}
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() {
        this("(2) Invoke Employee's overloaded constructor");
        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");
    }
}
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() {
        this("(2) Invoke Employee's overloaded constructor");
        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");
    }
}
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() {
        this("(2) Invoke Employee’s overloaded constructor");
        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");
    }
}
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() {
        this("(2) Invoke Employee’s overloaded constructor");
        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");
    }
}
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() {
        this("(2) Invoke Employee’s overloaded constructor");
        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");
    }
}

7. Execute println
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() {
        this("(2) Invoke Employee’s overloaded constructor");
        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");
    }
}

8. Execute println
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() {
        this("(2) Invoke Employee’s overloaded constructor");
        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");
    }
}
Default constructor

- Remember, a class may be defined without constructors.
- In this case, a no-arg constructor with an empty body is implicitly defined in the class.
- This constructor, called a *default constructor*, is provided automatically only if no constructors are explicitly defined in the class.
- Best practice is to provide (if possible) a no-arg constructor for every class to make the class easy to extend and avoid compile-time errors during constructor chaining.
Defining a subclass

• A subclass inherits from a superclass
• You can also
  – Add new properties
  – Add new methods
  – Override the methods of the superclass
Add new methods

• For example
  – Add printCircle() method in the Circle class
    
    ```java
    public void printCircle() {
        System.out.println("The circle is created " +
            getDateCreated() + " and the radius is " + radius);
    }
    ```

  Call superclass method
Override the methods of 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
Override the methods of the superclass

• To override a method, the method must be defined in the subclass using **the same signature** as in its superclass, and **same or subtype of the overridden method’s return type**

• A best practice to avoid mistakes is to use a special Java syntax, called **override annotation**
  – Annotated method is required to override a method in its superclass
    • If it does not, then there will be a compile-time error

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

    @Override
    public String toString() {
        return super.toString() + "\nradius is " + radius;
    }
}
```
Overriding vs overloading

• Overridden methods
  – Have the same signature
  – Are in different classes related by inheritance

• Overloaded methods
  – Have the same name, but different parameter lists
  – Can be either
    • In the same class
    • In different classes related by inheritance
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
    // This method overloads the method in B
    public void p(double i) {
        System.out.println(i);
    }
}

Remember to use @Override annotation
(not shown so lines align)
Private methods of the superclass

• An instance method can be overridden only if it is accessible

• As such, 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, then the two methods are completely unrelated
Static methods of the superclass

• 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
this and super keywords

• Similar to using this to reference the calling object, the keyword super refers to the superclass of the class which super appears

• The keyword this is the name of a reference that refers to an object itself
  – One common use of the this keyword is to reference a hidden class member

• The keyword super refers to the superclass of the class in which super appears
  – One common use of the super keyword is to reference a hidden superclass member
this keyword

• The keyword this refers to an object itself
• The keyword this can be used to
  – Call another constructor of the same class
    • Syntax
      this(arguments);
  – Reference a hidden class variable
    • Syntax
      this.variableName
super keyword

• The keyword `super` refers to the superclass of the class in which `super` appears

• The keyword `super` can be used to
  – Call a superclass constructor
    • Syntax
      ```java
      super(arguments);
      ```
  – Call a superclass method
    • Syntax
      ```java
      super.methodName(arguments);
      ```
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`

```java
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.
- For example
  ```java
  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.
Next Lecture

- Polymorphism