Building Java Programs

Chapter 9
Lecture 20: Interacting with the Superclass (super);
Discussion of Homework 9: Critters

reading: 9.2

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)
I STARTED A TASK FORCE TO ELIMINATE REDUNDANCIES IN OUR INTERNAL PROCESSES.

REALLY? I'M DOING THE SAME THING.
Calling overridden methods

• Subclasses can call overridden methods with `super`.

\[
\text{super.method(parameters)}
\]

• Example:

```java
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ...
}
```
Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they've been with the company.
  - For each year worked, we'll award 2 additional vacation days.
  
- When an Employee object is constructed, we'll pass in the number of years the person has been with the company.

- This will require us to modify our Employee class and add some new state and behavior.

- Exercise: Make necessary modifications to the Employee class.
public class Employee {
    private int years;
    
    public Employee(int initialYears) {
        years = initialYears;
    }
    
    public int getHours() {
        return 40;
    }
    
    public double getSalary() {
        return 50000.0;
    }
    
    public int getVacationDays() {
        return 10 + 2 * years;
    }
    
    public String getVacationForm() {
        return "yellow";
    }
}
Now that we've added the constructor to the Employee class, our subclasses do not compile. The error:

Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
^

The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

The long explanation: (next slide)
The detailed explanation

- Constructors are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.
  - Subclasses receive a default constructor that contains:
    ```java
    public Lawyer() {
        super(); // calls Employee() constructor
    }
    ```

- But our `Employee(int)` replaces the default `Employee()`.
  - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.
Calling superclass constructor

\[ \text{super(parameters);} \]

- Example:
  
  ```java
class Lawyer extends Employee {
  public Lawyer(int years) {
    super(years); // calls Employee constructor
  }
  ...
}
```

- The `super` call must be the first statement in the constructor.

- Exercise: Make a similar modification to the `Marketer` class.
// A class to represent marketers.
public class Marketer extends Employee {
    public Marketer(int years) {
        super(years);
    }

    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}

- Exercise: Modify the Secretary subclass.
  - Secretaries' years of employment are not tracked.
  - They do not earn extra vacation for years worked.
Modified Secretary class

// A class to represent secretaries.
public class Secretary extends Employee {
    public Secretary() {
        super(0);
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}

- Since Secretary doesn't require any parameters to its constructor, LegalSecretary compiles without a constructor.
  - Its default constructor calls the Secretary() constructor.
Inheritance and fields

- Try to give lawyers $5000 for each year at the company:
  ```java
  public class Lawyer extends Employee {
      ...
      public double getSalary() {
          return super.getSalary() + 5000 * years;
      }
      ...
  }
  ```

- Does not work; the error is the following:
  ```java
  Lawyer.java:7: years has private access in Employee
          return super.getSalary() + 5000 * years;
  ^
  ```

- Private fields cannot be directly accessed from subclasses.
  - One reason: So that subclassing can't break encapsulation.
  - How can we get around this limitation?
Add an accessor for any field needed by the subclass.

```java
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getYears() {
        return years;
    }
    ...
}

public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * getYears();
    }
    ...
}
```
Revisiting Secretary

- The Secretary class currently has a poor solution.
  - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
  - If we call `getYears` on a Secretary object, we'll always get 0.
  - This isn't a good solution; what if we wanted to give some other reward to all employees based on years of service?

- Redesign our Employee class to allow for a better solution.
Improved Employee code

• Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```java
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getVacationDays() {
        return 10 + getSeniorityBonus();
    }

    // vacation days given for each year in the company
    public int getSeniorityBonus() {
        return 2 * years;
    }
    ...
}
```

• How does this help us improve the Secretary?
Improved Secretary code

- Secretary can selectively override `getSeniorityBonus`; when `getVacationDays` runs, it will use the new version.
- Choosing a method at runtime is called *dynamic binding*.

```java
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }

    // Secretaries don't get a bonus for their years of service.
    public int getSeniorityBonus() {
        return 0;
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```
Homework 8: Critters

reading: HW8 spec
CSE 142 Critters

- Ant
- Bird
- Hippo
- Vulture
- Husky (creative)

**behavior:**
- **eat**
  - eating food
- **fight**
  - animal fighting
- **getColor**
  - color to display
- **getMove**
  - movement
- **toString**
  - letter to display
A Critter subclass

public class name extends Critter { ... }

public abstract class Critter {
    public boolean eat()
    public Attack fight(String opponent)
        // ROAR, POUNCE, SCRATCH
    public Color getColor()
    public Direction getMove()
        // NORTH, SOUTH, EAST, WEST, CENTER
    public String toString()
}

How the simulator works

• "Go" → loop:
  • move each animal (getMove)
  • if they collide, fight
  • if they find food, eat

• Simulator is in control!
  • getMove is one move at a time
    • (no loops)
  • Keep state (fields)
    • to remember future moves
Development Strategy

- Do one species at a time
  - in ABC order from easier to harder (Ant → Bird → ...)
  - debug printlns

- Simulator helps you debug
  - smaller width/height
  - fewer animals
  - "Tick" instead of "Go"
  - "Debug" checkbox
  - drag/drop to move animals
Critter exercise: Cougar

- Write a critter class **Cougar**:

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td><code>public Cougar()</code></td>
</tr>
<tr>
<td>eat</td>
<td>Always eats.</td>
</tr>
<tr>
<td>fight</td>
<td>Always pounces.</td>
</tr>
<tr>
<td>getColor</td>
<td>Blue if the Cougar has never fought; red if he has.</td>
</tr>
<tr>
<td>getMove</td>
<td>Walks west until he finds food; then walks east until he finds food; then goes west and repeats.</td>
</tr>
<tr>
<td>toString</td>
<td>&quot;C&quot;</td>
</tr>
</tbody>
</table>
Ideas for state

- You must not only have the right state, but update that state properly when relevant actions occur.

- Counting is helpful:
  - How many total moves has this animal made?
  - How many times has it eaten? Fought?

- Remembering recent actions in fields is helpful:
  - Which direction did the animal move last?
    - How many times has it moved that way?
  - Did the animal eat the last time it was asked?
  - How many steps has the animal taken since last eating?
  - How many fights has the animal been in since last eating?
import java.awt.*;  // for Color

public class Cougar extends Critter {
    private boolean west;
    private boolean fought;

    public Cougar() {
        west = true;
        fought = false;
    }

    public boolean eat() {
        west = !west;
        return true;
    }

    public Attack fight(String opponent) {
        fought = true;
        return Attack.POUNCE;
    }

    ...
Cougar solution

... public Color getColor() {
    if (fought) {
        return Color.RED;
    } else {
        return Color.BLUE;
    }
}

public Direction getMove() {
    if (west) {
        return Direction.WEST;
    } else {
        return Direction.EAST;
    }
}

public String toString() {
    return "C";
}