Building Java Programs

Chapter 9

Lecture 9-2: Interacting with the Superclass (`super`)

reading: 9.3 - 9.4

Changes to common behavior

- Let's return to our previous company/employee example.
- Imagine a company-wide change affecting all employees.

  Example: Everyone is given a $10,000 raise due to inflation.
  - The base employee salary is now $50,000.
  - Legal secretaries now make $55,000.
  - Marketers now make $60,000.
- We must modify our code to reflect this policy change.

Modifying the superclass

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;              // works 40 hours / week
    }

    public double getSalary() {
        return 50000.0;         // $50,000.00 / year
    }

    ...
}
```

- Are we finished?

- The `Employee` subclasses are still incorrect.
  - They have overridden `getSalary` to return other values.

An unsatisfactory solution

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 55000.0;
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() {
        return 60000.0;
    }
    ...
}
```

- Problem: The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this.

Calling overridden methods

- Subclasses can call overridden methods with `super`

  `super.`**`method`**`(`**`parameters`**`)`

  - Example:

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

  - Exercise: Modify `Lawyer` and `Marketer` to use `super`.

Improved subclasses

```
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}

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

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

Calling overridden methods

`super.`**`method`**`(`**`parameters`**`)`

- Example:

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

- Recall: Subclasses can call overridden methods with `super`.

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.

Modified `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";
    }
}
```

Problem with constructors

- 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:

```
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

```
super(parameters);
```

- Example:

```
public 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.

Modified `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:

```
public class Lawyer extends Employee {
    ...
    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
    ...
}
```

- Does not work; the error is the following:

```
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?

Improved `Employee` code

Add an accessor for any field needed by the subclass.

```
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.

```
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*.

```
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);
    }
}
```

Critter exercise: `Toad`

- Write a critter class `Toad`:

Method	Behavior
constructor	`public Toad()`
`eat`	Never eats (the default eating behavior)
`fight`	Always forfeits (the default fighting behavior)
`getColor`	brown (red=192, green=128, blue=0)
`getMove`	Walks west once every 5 moves: C, C, C, C, **W**, C, C, C, C, **W**, ...
`toString`	`"F"`

  - Does some of this behavior sound familiar?

Modified Frog

```
import java.awt.*;   // for Color

public class Frog extends Critter {
    private int age;
    private int count;

    public Frog(int age) {
        this.age = age;
        this.count = 0;
    }

    public Direction getHopDirection() {  // added so that it can
        return Direction.EAST;            // be overridden by Toad
    }

    public Direction getMove() {
        count++;
        if (count >= age) {    // go EAST once every 'age' moves
            count = 0;
            return getHopDirection();
        } else {
            return Direction.CENTER;
        }
    }

    ...
}
```

Toad solution

```
import java.awt.*;   // for Color

public class Toad extends Frog {
    private static final Color BROWN = new Color(192, 128, 0);

    public Toad() {
        super(5);
    }

    public Color getColor() {
        return BROWN;
    }

    public Direction getHopDirection() {   // overrides the version
        return Direction.WEST;             // from Frog
    }
}
```

Critter: `WhiteRabbit`

- In section, you wrote a `Rabbit` critter
  - Hops: N N, S S, E E, N N, S S, E E, ...

- Let's write `WhiteRabbit`
  - White, not brown
  - Hops in cycles of 5
    (N*5, S*5, E, E,
    N*5, S*5, E, E, ...)