Lecture 15: Inheritance II

Building Java Programs: A Back to Basics Approach
by Stuart Reges and Marty Stepp

Interacting with the superclass

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 (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;
    }
}
```

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) { //this replaces default
        years = initialYears;           //constructor
    }

    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 few slides)

The Default constructor

- Any class without a constructor receives the default constructor(no parameters) that initializes integers to 0, doubles to 0.0, booleans to false and objects to null. The programmer should write an explicit default constructor if he wishes to initialize these variables to other values.

- If a class has a constructor with parameters, the default constructor given by the compiler no longer exists. (Unless, there is an explicit default constructor.)

- A subclass that doesn't have any constructors will also receive a default constructor that simply calls its parent's default constructor.

The detailed explanation

- Constructors are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.
  - A subclass without any constructors will call its parent's default constructor.
- But our `Employee(int)` replaces the default `Employee()`.
  - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

Subclass Constructor

– Example:

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        this.years = years;// error! Why?
    }
    …
}
```

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();
       //why not super.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);
    }
}
```

Subclasses Rules

- A subclass inherits all public data members and methods of its parent.
- A subclass do not inherit the private instance variables or private methods. However, objects of subclasses contain memory for those private instance variables, even though they can't directly access them.
- A subclass can override inherited methods.
- Private methods cannot be overridden.

Subclasses Rules

- A subclass can add new private instance variables.
- A subclass DO NOT inherit its parent's constructors. A subclass should define its own constructors.
- A subclass can add new public, private or static methods.
- A subclass may not override static methods of the superclass.
- A subclass may not redefine a public method as private.

Lab 1

Modify the previous lab Student and subclass GradStudent and create a new class Undergrad.

•The Student class has a private string `name`, public integer id, private integer array `tests`, and private string `grade`. It also has a public final static integer `NUM_TESTS=3`. It has two constructors: 1) A default constructor that initializes `name` and `grade` to the empty string , id to 0, and create a new array for `tests` of length `NUM_TESTS=3`. 2) A constructor with four parameters for name, id, array of integers and grade.

•Student has `getName(), setName(String n), printWelcome(), getGrade(), setGrade(String newGrade), getTestAverage()` and `computeGrade()` PUBLIC methods. The `getTestAverage()` returns the average of the tests. The `computeGrade()` method set `grade` to "" if name="" and to "Pass" if test average is >=65 and "fail" otherwise.

…continued next page

Lab 1(continued)

- The UnderGrad class has no new variables. It has a default constructor and a constructor with three parameters: name, array of integers and grade.
- It overrides the `computeGrade()` method. If the tests average is >=70, grade is "Pass" and "Fail" otherwise.
- The GradStudent extends Student has an additional private String variable `dissertationTopic`. It has a default constructor that called `super()` and set `dissertationTopic` to empty string. It also has a constructor with five parameters: name, id, array of integers test, grade and topic an called constructor with parameters of Student.
- GradStudent overrides `computeGrade()`. It computes the grade in the same way as the Student class but set `grade` to "Pass with distinction" if the test average is >=90.