Building Java Programs

Chapter 9

Lecture 9-2: Static Data; More Inheritance

reading: 9.3 - 9.4

DrunkenFratGuy critter

- All the frat guys are trying to get to the same party
- The party is at a randomly-generated board location
  (On the 60-by-50 world)
- They stumble north then east until they reach the party

A flawed solution

```
import java.util.*;

public class DrunkenFratGuy extends Critter {
    private int partyX;
    private int partyY;

    public DrunkenFratGuy() {
        Random r = new Random();
        partyX = r.nextInt(60);
        partyY = r.nextInt(50);
    }

    public Direction getMove() {
        if (getY() != partyY) {
            return Direction.NORTH;
        } else if (getX() != partyX) {
            return Direction.EAST;
        } else {
            return Direction.CENTER;
        }
    }
}
```

- Problem: Each guy goes to his own party, not a shared one.

Static fields

- **static**: Part of a class, rather than part of an object.
  - Classes can have *static fields*.
  - Static fields are not replicated into each object; a single field is shared by all objects of that class.

```
private static type name;
```

or,

```
private static type name = value;
```

  - Example:

```
private static int count = 0;
```

Static field example

```
public class Husky implements Critter {
    // count of Huskies created so far
    private static int objectCount = 0;

    private int number;    // each Husky has a number

    public Husky() {
        objectCount++;
        number = objectCount;
    }

    ...

    public String toString() {
        return "I am Husky #" + number +
               "out of " + objectCount;
    }
}
```

Static methods

- **static method**: Part of a class, not part of an object.
  - shared by all objects of that class
  - good for code related to a class but not to each object's state
  - does not understand the *implicit parameter*, `this`; therefore, cannot access an object's fields directly
  - if `public`, can be called from inside or outside the class

- Declaration syntax: *(same as we have seen before)*

```
public static type name(parameters) {
    statements;
}
```

Static method example 1

- Java's built-in `Math` class has code that looks like this:

```
public class Math {
    ...
    public static int abs(int a) {
        if (a >= 0) {
            return a;
        } else {
            return -a;
        }
    }

    public static int max(int a, int b) {
        if (a >= b) {
            return a;
        } else {
            return b;
        }
    }
}
```

Static method example 2

```
public class Point {
    ...
    // Converts a String such as "(5, -2)" to a Point.
    // Pre: s must be in valid format.
    public static Point parse(String s) {
        s = s.substring(1, s.length() - 1);  // "5, -2"
        s = s.replaceAll(",", "");           // "5 -2"

        // break apart the tokens, convert to ints
        Scanner scan = new Scanner(s);
        int x = scan.nextInt();              // 5
        int y = scan.nextInt();              // 2

        Point p = new Point(x, y);
        return p;
    }
}
```

Calling static methods

class.method(parameters);

- This is the syntax client code uses to call a static method.

- Examples:

```
int absVal = Math.max(5, 7);
Point p3 = Point.parse("(-17, 52)");
```

- From inside the same class, the **class.** is not required.

method(parameters);

- This is the syntax you used to call methods in your programs.

Fixed DrunkenFratGuy

```
import java.util.*;

public class DrunkenFratGuy extends Critter {
    private static int partyX = -1;
    private static int partyY = -1;

    public DrunkenFratGuy() {
        if (partyX < 0 || partyY < 0) {
            Random r = new Random();   // the 1st frat guy created
            partyX = r.nextInt(60);    // chooses the party location
            partyY = r.nextInt(50);    // for all frat guys to go to
        }
    }

    public Direction getMove() {
        if (getY() != partyY) {
            return Direction.NORTH;
        } else if (getX() != partyX) {
            return Direction.EAST;
        } else {
            return Direction.CENTER;
        }
    }
}
```

Inheritance with constructors and fields

reading: 9.3

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.
  - They 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*.

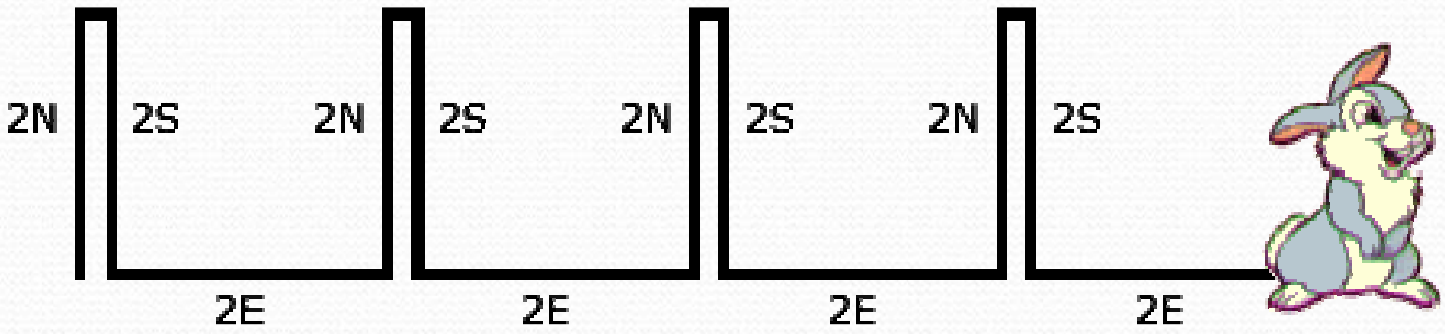
```
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }

    // 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);
    }
}
```

Recall: `Rabbit` critter

Method	Behavior
constructor	`public Rabbit()`
color	dark gray (`Color.DARK_GRAY`)
eating	alternates (`true`, `false`, `true`, ...)
fighting	if opponent is a Lion, then scratch; otherwise, roar
movement	2 N, 2 S, 2 E, repeat
`toString`	`"V"`



Exercise: WhiteRabbit

- In section 9, 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 8

    (N*8, S*8, E, E,
    N*8, S*8, E, E, ...)