CS 112 Introduction to Programming

(Spring 2012)

Lecture #32: Inheritance and Class Hierarchy

Zhong Shao

Department of Computer Science
Yale University
Office: 314 Watson

http://flint.cs.yale.edu/cs112

Acknowledgements: some slides used in this class are taken directly or adapted from those accompanying the two textbooks: Introduction to Programming in Java: An Interdisciplinary Approach by Robert Sedgewick and Kevin Wayne and Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp
super and Constructor

Java constructor automatically inserts super() as the first statement in the constructor of a child class:

```java
public class Lawyer extends Employee {
    public Lawyer() {
        // super() is automatically inserted
        System.out.println("In Lawyer()");
    }

    public String vacationForm() {
        return "pink";
    }

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

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```
public class Lawyer extends Employee {
    public Lawyer() {
        // super() is automatically inserted
        System.out.println("In Lawyer()");
    }
    ...
}

public class Employee {
    public Employee() {
        System.out.println("In Employee()");
    }
    ...
}

public static void main(String[] args) {
    Lawyer larry = new Lawyer();
}

Output:
In Employee()
In Lawyer()
Exercise: Add name to Employee

```java
public class Employee {
    private String name;
    public Employee(String name) {
        this.name = name;
    }
    ...
}
```
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 explanation

- **Constructors aren't inherited.**
  - The Employee subclasses don't inherit the public Employee(String name) constructor.

- **After defining** public Employee(String), Java sees that we have a constructor, and will no longer provide the default Employee constructor.
  - i.e., public Employee() is not defined unless we define it explicitly

- **But**
  ```java
  public Lawyer() {
    // super() is automatically inserted but not defined
    // in Employee
    System.out.println("In Lawyer() ");
  }
  ```
super and Constructor

- To pass parameter to parent’s constructor, add `super(...)` as first statement in child’s constructor

```java
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
        System.out.println("In Lawyer()");
    }
    ...
}

public class Employee {
    private String name;
    public Employee(String name) {
        System.out.println("In Employee()");
        this.name = name;
    }
    ...
}
```
Summary: Method/data Inheritance

- Use `extends` to allow a derived class to access data/methods defined in parent class

- A child can override a method defined in the parent class
  - if you do not want a method to be over-ridden, declare it `final`

- Access a method defined in a parent class using `super`
Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money (10% more) and can file legal briefs.

```java
public class LegalSecretary extends Secretary {
    ...
}
```

- Exercise: Complete the `LegalSecretary` class.
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }

    public double pay() {
        return super.pay() * 1.1;
    }
}
Partner class

- Partner is a senior lawyer that can get bonus. Thus it supports:

  awardBonus(double bonus)
// A class to represent partner.
public class Partner extends Lawyer {
    private bonus;
    public void awardBonus(double bonus) {
        this.bonus = bonus;
    }

    public double pay() {
        return super.pay() + bonus;
    }
}
Many large-scale software systems define class hierarchies, where the root defines the common behaviors.
Class Hierarchies: Another Example

- Animal
  - Reptile
    - Snake
    - Lizard
  - Bird
    - Parrot
  - Mammal
    - Horse
    - Bat
The Object Class

- A class called Object is defined in the java.lang package of the Java standard class library

- All classes are derived from the Object class
  - even if a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the Object class
  - the Object class is therefore the ultimate root of all class hierarchies

- The Object class contains a few most basic methods, which are inherited by all classes
  - toString()
  - equals()
  - clone()
The Object Class: the `toString` Method

- That’s why the `println` method can call `toString` for any object that is passed to it - all objects are guaranteed to have a `toString` method via inheritance.

- The `toString` method in the `Object` class is defined to return a string that contains the name of the object’s class and a hash value.

- Every time we have defined `toString`, we have actually been overriding it.
Example: GUI Component Class Hierarchy

- A Java *GUI component* is an object that represents a visual entity in a graphical user interface (such as a button, a label, or slider).

- Component classes are organized into an inheritance hierarchy so that they can easily share characteristics.

- A derived component class overrides the method defined in the ancestor class to define its own visual display.
  - e.g., different component classes redefine the `paintComponent` method to display differently.
Class Hierarchy (A Very Small Subset)

Component

Container

JComponent

paintComponent()

......

JPanel  JAbstractButton  JLabel  JTextComponent

......

JTextField

......

JButton  JToggleButton  JCheckBox  JRadioBox

......

paint()
Using Java GUI Components

- Java GUI manager maintains a collection of GUI components and delivers events (e.g., to draw, to respond to a click) to components
  - Each component provides callback functions for the manager to invoke
  - This is called *event driven* programming
Assignment 9 Class Hierarchy

Critter

Ant  Bird  Hippo  Bulldog

Vulture
Critter : Cougar

public class Cougar extends Critter {

... 

}

<table>
<thead>
<tr>
<th>Method</th>
<th>Behaviors to Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>public Cougar()</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>
getColor for Cougar

- **State machine**
  - `fight()`
  - ![State machine diagram](diagram.png)

- **How to remember the state?**
  - a boolean instance variable:
    ```java
    boolean fought
    ```

- **What is initial state and where to set it?**
  - In constructor: `fought = false;`

- **Who/when updates the state?**
  - In `fight()`: `fought = true`
getMove for Cougar

- **State machine**

- **How to remember the state?**
  - a boolean instance variable:
    ```java
    boolean west
    ```

- **What is initial state and where to set it?**
  - In constructor: `west = true;`

- **Who/when updates the state?**
  - In `eat():` reverse state
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;
    }

    ...
}
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";
}
**Critter : Snake**

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td><code>public Snake()</code></td>
</tr>
<tr>
<td>eat</td>
<td>Never eats</td>
</tr>
<tr>
<td>fight</td>
<td>always forfeits</td>
</tr>
<tr>
<td>getColor</td>
<td>black</td>
</tr>
<tr>
<td>getMove</td>
<td><code>1 E, 1 S; 2 W, 1 S; 3 E, 1 S; 4 W, 1 S; 5 E, ...</code></td>
</tr>
<tr>
<td>toString</td>
<td>&quot;S&quot;</td>
</tr>
</tbody>
</table>
Determining necessary fields

- Information required to decide what move to make?
  - Direction to go
  - Length of current cycle
  - Number of moves made in current cycle
Determining necessary fields

A non-event driven version
Determining necessary fields

A non-event driven version

cycle = 1;
while (true) {
    for (steps = 0; steps < cycleLength; step++)
        if cycleLength % 2 == 1
            go E
        else
            go W
            go S
    cycleLength ++
}
Guarding Condition

A non-event driven version

cycle = 1;
while (true) {
    for (steps = 0; steps < cycleLength; steps++)
        if cycleLength % 2 == 1
            go East
        else
            go West
    go South
    cycleLength ++
}

steps < cycleLength

if (cycleLength % 2 == 1)
    go East
else
    go West
steps++;

steps == cycleLength

go South
cycleLength ++
cycleLength ++
steps=0;
import java.awt.*;  // for Color

class Snake extends Critter {
    private int cycleLength;  // # steps in curr. Hori.
    private int steps;  // # of cycle's steps already taken

    public Snake() {
        cycleLength = 1;
        steps = 0;
    }

    public Direction getMove() {
        if (steps < cycleLength) {
            steps++;
            if (cycleLength % 2 == 1) {
                return Direction.EAST;
            } else {
                return Direction.WEST;
            }
        } else {
            steps = 0;
            cycleLength ++;
            return Direction.SOUTH;
        }
    }

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

Comment: States

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

- **Remembering recent actions in fields may be 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?
Testing critters

- Focus on one specific critter of one specific type
  - Only spawn 1 of each animal, for debugging

- Make sure your fields update properly
  - Use `println` statements to see field values

- Look at the behavior one step at a time
  - Use "Tick" rather than "Go"
Designing Bulldog

- Be open minded
- Think about strategies, e.g.,
  - How much state do your bulldogs keep and probe state (e.g., do your bulldogs work together to map the world?)
  - Do your bulldogs play strategic games (e.g., bluffing)?
  - Do your bulldogs coordinate their behaviors to form some kind of patterns?
Coordination
Coordination
Comment: Class/Static Variables

- Sometimes it might be useful if all object instances of a class share the same copy of a variable, e.g.,
  - common constant variables
  - global statistics
  - coordination

- Declare variables using keyword `static` to create only one copy of the variable

- Such variables are called static or class variables

- Class/static variables are accessible by all methods in the class
Example: Keeping Track of Employee Objects

- We want to keep track of the number of Employee objects we ever created.

- In our Employee class, we could add the following data:
  - static int counter, an integer that represents the number of accounts we ever created.
  - since counter is static, it is shared by all account objects.
  - in constructor, we increase counter by 1.
public class Employee {
    private String name;
    private static int counter = 0;

    public Employee(String name) {
        this.name = name;
        counter++;
    }

    public static int nEmployees() {
        return counter;
    }
    ...
}

public class Firm {

    public static void main(String[] args) {
        Lawyer larry = new Lawyer("Larry");
        Marketer mike = new Marketer("Mike");
        Partner peter = new Partner("Peter");
    }
}

Employee with Static
Example: The Employee Objects

0

Employee.counter
Example: The Employee Objects

<table>
<thead>
<tr>
<th>larry: Lawyer</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = “Larry”</td>
</tr>
</tbody>
</table>

After `Lawyer larry = new Lawyer("Larry");`
Example: The Employee Objects

<table>
<thead>
<tr>
<th>larry: Lawyer</th>
<th>Employee.counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = “Larry”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mike: Marketer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name = “Mike”</td>
<td></td>
</tr>
</tbody>
</table>

After Marketer mike = new Marketer("Mike");
### Example: The Employee Objects

<table>
<thead>
<tr>
<th>Employee</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>larry: Lawyer</td>
<td>name = “Larry”</td>
</tr>
<tr>
<td>mike: Marketer</td>
<td>name = “Mike”</td>
</tr>
<tr>
<td>peter: Partner</td>
<td>name = “Peter”, bonus = 0</td>
</tr>
</tbody>
</table>

After Partner peter = new Partner("Peter");

Employee.counter = 3