CS 112 Introduction to Programming
(Spring 2012)

Lecture #31: Software Reuse through Inheritance

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
Recap: Object Oriented Programming

- Progressively adding new perspectives
  - Complex data type → encapsulation → inheritance → polymorphism

- A class definition may include
  - fields (instance variables per object)
  - static variables (shared by all objects)
  - static methods (method usable with or without object)
  - instance methods (implicit parameter)
    - Constructors
    - Accessors (do not change object state)
    - Mutators (modify object state)
public class Point {
    private int x;
    private int y;

    // Constructs a Point at the given x/y location.
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }

    ...
}
Common Constructor Issues

Re-declaring fields as local variables ("shadowing"):

```java
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        int x = initialX;
        int y = initialY;
    }
    ...
}
```

- This declares local variables with the same name as the fields, rather than storing values into the fields. The fields remain 0.
Shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, *except when one variable is a field*

```java
public class Point {
    private int x;
    private int y;
    ...

    // this is legal
    public Point(int x, int y) {
        System.out.println("x = " + x);// para x
    }
}
```

- In most of the class, `x` and `y` refer to the fields.
- In `Point(int x, int y)`, `x` and `y` refer to the method's parameters.
The **this** keyword

- **this** : Refers to the implicit parameter inside your class.  
  *(a variable that stores the object on which a method is called)*

  - Refer to a field:  
    `this.field`

  - Call a method:  
    `this.method(parameters);`
Fixing Shadowing

- To refer to the data field x, say this.x
- To refer to the parameter x, say x

```java
public class Point {
    private int x;
    private int y;

    ...  
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```
Calling another constructor

public class Point {
  private int x;
  private int y;

  public Point() {
    this(0, 0);    // calls (x, y) constructor
  }

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  ...
}

• Avoids redundancy between constructors
• Only a constructor (not a method) can call another constructor
public class Point {

    // fields (instance variables, attributes)
    private int x;
    private int y;

    // constructors
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }

    public Point() {
        this(0, 0);
    }

    // static methods
    public static Point readPoint(Scanner scan) {
        Point p = new Point();
        p.x = scan.nextInt();
        p.y = scan.nextInt();
        return p;
    }
}
Point class

// mutators
public void translate(int dx, int dy) {
    x = x + dx;
    y = y + dy;
}
public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
}

// accessors
public double distanceFromOrigin() {
    return Math.sqrt(x * x + y * y);
}

public String toString() {
    return "(" + x + ", " + y + ")";
}
Point class as blueprint

Point class
state:
int x, y
behavior:
translate(int dx, int dy)
...
...

Point object #1
state:
x = 5, y = -2
behavior:
translate(int dx, int dy)
...

Point object #2
state:
x = -245, y = 1897
behavior:
translate(int dx, int dy)
...

Point object #3
state:
x = 18, y = 42
behavior:
translate(int dx, int dy)
...
Recap: The Encapsulation Principle

- An object should be an encapsulated entity (a black box) with well defined external state and behaviors

Client

Client can see only the external state and behaviors

Client should not see the internal state or behaviors

External state and behaviors

Internal state and behaviors
The **public and private** Access Modifiers

- **access modifiers** enforce encapsulation
  - **public** members (data and methods: can be accessed from anywhere)
  - **private** members: can be accessed from only inside the class
  - Members without an access modifier: default **private** accessibility, i.e., accessible in the same package; otherwise, not accessible.
Using Access Modifiers to Implement Encapsulation: Data Fields

- As a general rule, no object's data (state) should be declared with public accessibility.

- Access or change to the object's state be accomplished only by that object's methods.
Benefit of Encapsulating Data

- **Consistency**: so that we make it impossible for others to "reach in" and directly alter another object's state
  - Protect object from unwanted access
    - Example: Can't fraudulently increase an BankAccount balance.
  - Maintain state *invariants*
    - Example: Only allow BankAccounts with non-negative balance.
    - Example: Only allow Dates with a month from 1-12.

- **Flexibility**: so that you can change the state representation later without worrying about breaking others’ code
  - Example: Point could be rewritten in polar coordinates \((r, \theta)\) so long you provide translation in the interface, clients will not see difference.
Using Access Modifiers to Implement Encapsulation: Methods

- **Only service methods** should be made **public**

- **Support or helper methods** created simply to assist service methods should be declared **private**
The Effects of Public and Private Accessibility

<table>
<thead>
<tr>
<th>Public</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>violate Encapsulation Use Caution</td>
<td>enforce encapsulation</td>
</tr>
<tr>
<td>provide services to clients</td>
<td>support other methods in the class</td>
</tr>
</tbody>
</table>
Above is a class diagram representing the Coin class.
“-” indicates private data or method
“+” indicates public data or method
Question: What programming language feature(s) have we covered to allow software reuse?
Consider the following law firm:

- Employees work 40 hours / week.
- Pay policy: Employees make a base salary of $50,000 per year, except
  - legal secretaries who make 10% extra per year ($55,000 total),
  - marketers who make 20% extra per year ($60,000 total).
  - lawyers who reach partner level get bonus.
- Vacation policy: Employees have 2 weeks of paid vacation leave per year, except
  - lawyers who get an extra week (a total of 3).
  - employees should use a yellow form to apply for leave, except for lawyers who use a pink form.
A Law Firm Problem

- Each type of employee has some job functions:
  - Lawyers know how to sue.
  - Marketers know how to advertise.
  - Secretaries know how to prepare ordinary document.
  - Legal secretaries know how to prepare ordinary documents and also legal documents.
An Employee class

// A class to represent employees in general.
public class Employee {

    public int hours() {
        return 40;  // works 40 hours / week
    }

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

    public int vacationDays() {
        return 10;  // 2 weeks' paid vacation
    }

    public String vacationForm() {
        return "yellow";  // use the yellow form
    }

    public String toString() {
        String result = "Hours: " + hours() + "\n";
        result += "Pay: " + pay() + "\n";
        result += "Vacation days: " + vacationDays() + "\n";
        result += "Vacation Form: " + vacationForm() + "\n";
        return result;
    }

} // end of Employee
Question: Writing class Secretary

- Secretaries can prepare documents.
Secretary class: Attempt 1

// A class to represent employees in general (20-page manual).
public class Secretary {

    public int hours() {
        return 40; // works 40 hours / week
    }
    public double pay() {
        return 50000.0; // $50,000.00 / year
    }
    public int vacationDays() {
        return 10; // 2 weeks' paid vacation
    }
    public String vacationForm() {
        return "yellow"; // use the yellow form
    }
    public String toString() {
        String result += "Hours: " + hours() + "\n";
        result += "Pay: " + pay() + "\n";
        result += "Vacation days: " + vacationDays() + "\n";
        result += "Vacation Form: " + vacationForm() + "\n";
        return result;
    }
    public void prepareDoc(String text) {
        System.out.println("Working on Document: " + text);
    }
}
Desire for code-sharing

prepareDoc is the only unique behavior in Secretary.

We'd like to be able to say:

```java
// A class to represent secretaries.
public class Secretary {
    <copy all the contents from the Employee class>

    public void prepareDoc(String text) {
        System.out.println(“Work on Document: “ + text);
    }
}
```
Inheritance

- **Inheritance**: A way to allow a software developer to reuse classes by deriving a new class from an existing one
  - The existing class is called the *parent class*, or *superclass*, or *base class*
  - The derived class is called the *child class* or *subclass*.

- As the name implies, the child inherits characteristics of the parent
  - The child class *inherits* every method and every data field defined for the parent class
Inheritance

- Inheritance relationships are often shown graphically in a class diagram, with the arrow pointing to the parent class.

```
Animal
- weight : int
+ getWeight() : int

Bird
+ fly() : void
```

Inheritance should create an *is-a* relationship, meaning the child is a more specific version of the parent.
The child class inherits the methods and data defined for the parent class.

**Animal**
- weight : int
+ getWeight() : int

**Bird**
- flySpeed : int
+ fly() : void

an animal object

weight = 120
getWeight()

a bird object

weight = 100
flySpeed = 30
getWeight()
fly()
Deriving Subclasses: Syntax

public class <name> extends <superclass> {
}

For example:

class Animal {
    // class contents
    private int weight;
    public void int getWeight() {...}
}

class Bird extends Animal {
    // class contents
    public void fly() {...};
}
Improved Secretary code

// A class to represent secretaries.
public class Secretary extends Employee {

    public void prepareDoc(String text) {
        System.out.println("Working on document: " + text);
    }

}

• By extending Employee, each Secretary object now:
  • receives methods hours, pay, vacationDays, vacationForm, toString from Employee’s definition automatically

  • can be treated as an Employee by client code (seen later)

  • Now we only write the parts unique to each type.
Implementing the Lawyer class: Attempt 1

// A class to represent lawyers.
public class Lawyer extends Employee {

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


Problem

- We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior:
  - Lawyers who get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
Defining Methods in the Child Class: Overriding Methods

- A child class can (have the option to) *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent’s method, but can have different code in the body

- The method invoked is always the one defined in the child class, if the child class refines a method
// A class to represent lawyers.
public class Lawyer extends Employee {

    // overrides getVacationDays from Employee class
    public int vacationDays() {
        return 15; // 3 weeks vacation
    }

    // overrides getVacationForm from Employee class
    public String vacationForm() {
        return "pink";
    }

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

Lawyer class
Marketer class

- Exercise: Complete the Marketer class. Marketers make 20% more than the base ($60,000 total) and know how to advertise.
// A class to represent marketers.
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act while supplies last!");
    }

    // override
    public double pay() {
        return 60000.0; // $60,000.00 / year
    }
}
Overloading vs. Overriding

- **Overloading** deals with multiple methods in the same class with the same name but different signatures.

- **Overloading** lets you define a similar operation in different ways for different data.

- **Overriding** deals with two methods, one in a parent class and one in a child class, that have the same signature.

- **Overriding** lets you define a similar operation in different ways for different object types.
An Issue

```java
public class Marketer extends Employee {
    public double pay() {
        return 60000.0;
    }

    ...
}

• Problem: The Marketer’s salaries are based on the Employee’s base salary (20% more than base), but the pay code does not reflect this.
```
Changes to Common Behavior

- 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 $60,000.
  - Marketers now make $72,000.

- 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 hours() {
        return 40; // works 40 hours / week
    }
    public double pay() {
        return 60000.0; // $60,000.00 / year
    }
    ...
}

- **Issue**: the Employee subclasses are still incorrect.
  - They have overridden pay to return other values.
Calling overridden methods

Subclasses can call overridden methods with `super`.
`super. <method>(<parameters>)`

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

```java
class Lawyer extends Employee {
    public String vacationForm() {
        return "pink";
    }
    public int vacationDays() {
        return super.vacationDays() + 5;
    }
    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    public double pay() {
        return super.pay() * 1.2;
    }
}
```
Critters

- A simulation world with animal objects with behavior:
  - eat eating food
  - fight animal fighting
  - getColor color to display
  - getMove movement
  - toString letter to display

- You must implement:
  - Ant
  - Bird
  - Hippo
  - Vulture
  - Bulldog (creative)
A Critter subclass

public class \texttt{name} extends Critter { 
  ... 
}

- extends \texttt{Critter} \textit{tells the simulator your class is a critter}
  - an example of \textit{inheritance}

- \textit{Override some/all 5 methods to give your animals behavior.}
import java.awt.*;

class Stone extends Critter {
    public Attack fight(String opponent) {
        return Attack.ROAR; // good ol' ROAR... nothing beats that!
    }

    public Color getColor() {
        return Color.GRAY; // stones are gray in color
    }

    public String toString() {
        return "Stone"; // the game displays a stone
    }
}

Critter Example: Stone
How the Simulator Works

- When you press "Go", the simulator enters a loop:
  - move each animal once (getMove), in random order
  - if the animal has moved onto an occupied square, fight!
  - if the animal has moved onto food, ask it if it wants to eat

- Key concept: The simulator is in control, NOT your animal.
  - Example: getMove can return only one move at a time.
    - getMove can't use loops to return a sequence of moves.
      - It wouldn't be fair to let one animal make many moves in one turn!

- Your animal must keep state (as fields) so that it can make a single move, and know what moves to make later.
Critter exercise: Cougar

- Write a critter class **Cougar** (the dumbest of all animals):

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</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>
Key Issue

- Each object must have the right state, e.g.,
  - Did the animal eat the last time it was asked?
  - Did the animal fight the last time it was asked?
  - ...

- Each object must update that state properly when relevant actions occur.
getColor for Cougar

- State machine

- How to remember the state?
  - A boolean instance variable:
    
    ```
    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

How can a critter move west until it finds food and then moves to east until find food and repeat?

```java
public Direction getMove() {
    while (animal has not eaten) {
        return currentDirect;
    }
    reverse currentDirect;
}
```
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:
    ```java
    west = true;
    ```

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