CS 112 Introduction to Programming

Critters/Event-Driven Programming

Yang (Richard) Yang
Computer Science Department
Yale University
308A Watson, Phone: 432-6400
Email: yry@cs.yale.edu

Admin

- Class project teaming is due today: please use the Google doc to record the teaming
- We will provide special sessions to cover more background on
  - Android App
  - Google App Engine (Web app backend)
- Javascript will be covered in class starting from next week
  - Please read the tutorials listed on class page
Example projects

Games

- Sudoku:
  - Target to solve non-solvable by PS6 using recursion
- Madlib:
  - [http://mrt-cs112-o.appspot.com/](http://mrt-cs112-o.appspot.com/)
  - Used Google App Engine as backend
- 2048

Research, validation

- Evaluate or train the counting method of blackjack

Tools

- Health care cost calculator for Yale students
- Image processing tools (face recognition) based on OpenCV
  - [https://www.openshift.com/blogs/day-12-opencv-face-detection-for-java-developers](https://www.openshift.com/blogs/day-12-opencv-face-detection-for-java-developers)
- Transloc api ([http://api.transloc.com/doc/](http://api.transloc.com/doc/)) to build a shuttle application for Yale
Recap

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

Polymorphic Reference through Inheritance

- A polymorphic reference variable of type $T$ can hold an object of class $T$ or descendent of $T$, e.g.,

  Employee emp = new Employee(“Ed”);
  emp = new Lawyer(“Larry”);
  emp = new LegalSecretary(“Lisa”);

- When you invoke a method through a polymorphic reference variable, it is the type of the object being referenced, not the reference type, that determines which method is invoked.

Recap: Polymorphism and Arrays

- A common usage of polymorphism is to define an array of a base type, but different entries refer to different types of objects

  - To handle a heterogeneous population of objects with uniformity
Polymorphism and Arrays: Example

```java
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[4];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
    }
}
```

Works on any mix of Employee objects

```java
public void payday() {
    for (int count = 0; count < staffList.length; count++) {
        System.out.printf("%-10s:", staffList[count].name());
        System.out.printf("$%.2f\n", staffList[count].pay());
        System.out.println("-----------------------------------");
    }
}
```

Extending the Program: Hourly

- Include a new type of secretary who works variable number of hours and is paid by the hours.
Add a new Type of Employee: Hourly

```java
public class Hourly extends Secretary {
    private double payRate;
    private int hours;

    public Hourly(String name, double payRate)
        super(name);
        this.payRate = payRate;
        hours = 0;
    }

    public void addHours(int hours) {
        this.hours += hours;
    }

    public int hours() { return hours; }

    public double pay() { return hours() * payRate; }
}
```

Polymorphism and Arrays: Example

```java
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[5];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
        Hourly holly = new Hourly("Holly"); holly.addHours(10);
        staffList[4] = holly;
    }

    public void payday() {
        for (int count = 0; count < staffList.length; count++) {
            System.out.printf("%-10s:", staffList[count].name());
            System.out.printf("$%.2f\n", staffList[count].pay());
            System.out.println("-----------------------------------");
        }
    }
}
```

No need to change the payday method at all.
Through a given type of reference variable, we can invoke only the methods defined in that type.

```java
class Employee{
    public double pay()
    {...
}
}
class Lawyer extends Employee {
    public void sue()
    {...
}
}
Employee ed = new Lawyer("Larry");
```

Can we do the following statements:

```java
ed.pay();
ed.sue();
```
Comment: Variable Type and Method

We can “promote” an object back to its original type through an explicit narrowing cast:

```java
staffList = new Employee[5];
staffList[0] = new Lawyer("Lisa");
staffList[1] = new Secretary("Sally");
staffList[2] = new Marketer("Mike");
staffList[3] = new LegalSecretary("Lynne");
staffList[4] = new Hourly("Holly");

Hourly holly = (Hourly)staffList[4];
holly.addHours (5);
```

Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - overriding and using overridden methods
  - inheritance and field access
  - inheritance hierarchy
  - inheritance and polymorphism
  - example: Critters
Critters

- A simulation (game) world of animal objects (e.g., Ants, Birds, Cougars) with behaviors such as
  - eat: eating food
  - fight: animal fighting
  - getColor: color to display
  - getMove: movement
  - toString: letter to display

The Critter Class

// abstract class means not implement every method
public abstract class Critter {
    public boolean eat()
    public Attack fight(String opponent)
        // ROAR, POUNCE, SCRATCH, FORFEIT
    public Color getColor()  
    public Direction getMove(String[][] grid)
        // NORTH, SOUTH, EAST, WEST, CENTER  
    public String toString()
        ...
    // read the class for other methods available
}
Defining a Critter subclass

public class name extends Critter {
    ...
}

- extends Critter tells the simulator your class
  is a critter
  - an example of inheritance

- Override some methods to give your new type of
  animal behaviors.

Example Critter World Class Hierarchy
Critter Example: Stone

```java
import java.awt.*;

public 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 "St"; // the game displays a stone
    }
}
```

The Simulator (Controller)

- The simulator is in CritterMain.java
- It searches local dir. for all critters types
- The simulator creates an array of critters
- "Go" → loop:
  - move each animal (getMove)
  - if they collide, fight
  - if they find food, eat
**Simulator Pseudo-code**

```java
Critter[] critters = new Critter[N];
critters[0] = new Ant();
critters[1] = new Bird();
...

loop
  foreach critter i in critters
    call getMove of critter i if it can move
    foreach critter i in critters
      if new pos of critter i results in fight
        ask how critter i will fight
      else if new pos finds food
        ask critter i whether it will eat
      else if new pos results in mate possibility
        ask if critter i will mate
    compute new state of critters
```

**Event-Driven Programming**

- 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.
  - We say that you focus on writing the `callback` functions of objects
Critter exercise: Cougar

- Write a critter class Cougar (among 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 roars.</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>

Implement Cougar’s eat, fight, toString.

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(String[][] grid) {
    initial currentDirect = WEST
    loop
        if (eat) {
            reverse currentDirect;
            print currentDirection;
        }
    }
```
**getMove for Cougar**

- **State machine**
  - eat()
  - Move West
  - eat()
  - Move East

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

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

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

---

**getColor for Cougar**

- **State machine**
  - ! Has fought
  - Has fought

- **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

Blue if the Cougar has never fought; red if he has.
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(String[][] grid) {
        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>public Snake()</td>
</tr>
<tr>
<td>eat</td>
<td>Never eats</td>
</tr>
<tr>
<td>fight</td>
<td>random pounce or roar</td>
</tr>
<tr>
<td>getColor</td>
<td>Color(20, 50, 128)</td>
</tr>
<tr>
<td>getMove</td>
<td>1 E, 1 S; 2 W, 1 S; 3 E, 1 S; 4 W, 1 S; 5 E, ...</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
Non-EDP Version

A non-event driven version

cycleLength = 1;
while (true) {
    // go one cycle
    go South
    cycleLength ++
}

Non-EDP Version

A non-event driven version

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

A non-event driven version

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

Non-EDP-> EDP: Guarding Condition

A non-event driven version

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

```java
import java.awt.*; // for Color
public class Snake extends Critter {
    private int cycleLength; // # steps in curr. Horiz.
    private int steps; // # of cycle's steps all
    public Snake() {
        cycleLength = 1;
        steps = 0;
    }
    public Direction getMove() {
        if (steps < cycleLength) {
            steps++; // steps < cycleLength
            if (cycleLength % 2 == 1) {
                return Direction.EAST;
            } else {
                return Direction.WEST;
            }
        } else {
            steps = 0;
            cycleLength ++;
            return Direction.SOUTH;
        }
    }
    public String toString() {
        return "S";
    }
}
```

- If `steps` is less than `cycleLength`, increment `steps` and determine the direction based on the current `cycleLength`:
  - If `cycleLength` is odd, go East.
  - If `cycleLength` is even, go West.
- If `steps` equals `cycleLength`, reset `steps` to 0, increment `cycleLength`, and go South.

---

18