Building Java Programs

Chapter 8
Lecture 8-3: Encapsulation;
Homework 8 (Critters)

reading: 8.3 - 8.4

Encapsulation

reading: 8.4
Encapsulation

- **encapsulation**: Hiding implementation details from clients.

  - Encapsulation forces abstraction.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object’s data

Private fields

*A field that cannot be accessed from outside the class*

- **private** type name;

- Examples:
  ```java
  private int id;
  private String name;
  ```

- Client code won't compile if it accesses private fields:
  ```java
  PointMain.java:11: x has private access in Point
  System.out.println(p1.x);
  ^
  ```
Accessing private state

// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}

Client code will look more like this:
System.out.println(p1.getX());
p1.setX(14);

Point class, version 4

// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }
    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
}
Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an Account's balance.

- Can change the class implementation later
  - Example: Point could be rewritten in polar coordinates \((r, \theta)\) with the same methods.

- Can constrain objects' state (invariants)
  - Example: Only allow Accounts with non-negative balance.
  - Example: Only allow Dates with a month from 1-12.

The keyword \textit{this}

\textbf{reading: 8.3}
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);`
  - One constructor can call another: `this(parameters);`

Variable 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 void setLocation(int x, int y) {
        ...
    }
}
```

- In most of the class, `x` and `y` refer to the fields.
- In `setLocation`, `x` and `y` refer to the method’s parameters.
Fixing shadowing

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

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

- **Inside** setLocation,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

Calling another constructor

```java
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
Homework 8: Critters

reading: HW8 assignment spec

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
  - Husky (creative)
A Critter subclass

public class name extends Critter {
    ...
}

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

- Write some/all 5 methods to give your animals behavior.

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>

Ideas for state

- You must not only have the right state, but update that state properly when relevant actions occur.

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

- Remembering recent actions in fields is 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?
Keeping state

- How can a critter move west until it finds food?

```java
public Direction getMove() {
    while (animal has not eaten) {
        return Direction.EAST;
    }
    while (animal has not eaten a second time) {
        return Direction.EAST;
    }
}
```

```java
private int moves;    // total moves made by this Critter

public Direction getMove() {
    moves++;
    if (moves % 4 == 1 || moves % 4 == 2) {
        return Direction.WEST;
    } else {
        return Direction.EAST;
    }
}
```

Cougar solution

```java
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;
    }

    ...
```
Cougar solution

... 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";
    }

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"
Critter exercise: *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>always forfeits</td>
</tr>
<tr>
<td>getColor</td>
<td>black</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 in
  - Length of current cycle
  - Number of moves made in current cycle

- Remembering things you've done in the past:
  - an int counter?
  - a boolean flag?
import java.awt.*; // for Color

public class Snake extends Critter {
    private int length; // # steps in current horizontal cycle
    private int step; // # of cycle's steps already taken

    public Snake() {
        length = 1;
        step = 0;
    }

    public Direction getMove() {
        step++;
        if (step > length) { // cycle was just completed
            length++;
            step = 0;
            return Direction.SOUTH;
        } else if (length % 2 == 1) {
            return Direction.EAST;
        } else {
            return Direction.WEST;
        }
    }

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