Homework 8: Critters (cont.)

Lecture 21: Critters, static fields

reading: HW8 spec

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)
YOU FORGOT TO PUT PARENTHESES AFTER MAIN.

I HAVE NO PARENS.
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?
Snake solution

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";
}
}
Critter exercise: Hipster

- All hipsters want to get to the bar with the cheapest PBR
- That bar is at a randomly-generated board location (On the 60-by-50 world)
- They go north then east until they reach the bar
A flawed solution

```java
import java.util.*; // for Random

public class Hipster extends Critter {
    private int cheapBarX;
    private int cheapBarY;

    public Hipster() {
        Random r = new Random();
        cheapBarX = r.nextInt(60);
        cheapBarY = r.nextInt(50);
    }

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

- **Problem:** Each hipster goes to a different bar. We want all hipsters to share the same bar location.
Static members

- **static**: Part of a class, rather than part of an object.
  - Object classes can have static methods *and fields*.
  - Not copied into each object; shared by all objects of that class.

```java
class
state:
private static int staticFieldA
private static String staticFieldB
behavior:
public static void someStaticMethodC()
public static void someStaticMethodD()

object #1
state:
int field2
double field2
behavior:
public void method3()
public int method4()
public void method5()

object #2
state:
int field1
double field2
behavior:
public void method3()
public int method4()
public void method5()

object #3
state:
int field1
double field2
behavior:
public void method3()
public int method4()
public void method5()
```
Static fields

private static type name;
or,
private static type name = value;

- Example:
  private static int theAnswer = 42;

**static field**: Stored in the class instead of each object.
- A "shared" global field that all objects can access and modify.
- Like a class constant, except that its value can be changed.
Accessing static fields

- From inside the class where the field was declared:

  ```java
  fieldName // get the value
  fieldName = value; // set the value
  ```

- From another class (if the field is `public`):

  ```java
  ClassName.fieldName // get the value
  ClassName.fieldName = value; // set the value
  ```

- generally static fields are not `public` unless they are `final`

- Exercise: Write the working version of Hipster.
import java.util.*;  // for Random

public class Hipster extends Critter {
    // static fields (shared by all hipsters)
    private static int cheapBarX = -1;
    private static int cheapBarY = -1;

    // object constructor/methods (replicated into each hipster)
    public Hipster() {
        if (cheapBarX < 0 || cheapBarY < 0) {
            Random r = new Random(); // the 1st hipster created
            cheapBarX = r.nextInt(60); // chooses the bar location
            cheapBarY = r.nextInt(50); // for all hipsters to go to
        }
    }

    public Direction getMove() {
        if (getY() != cheapBarY) {
            return Direction.NORTH;
        } else if (getX() != cheapBarX) {
            return Direction.EAST;
        } else {
            return Direction.CENTER;
        }
    }
}
Static methods

// the same syntax you've already used for methods
public static type name(parameters) {
    statements;
}

- **static method**: Stored in a class, not in an object.
  - Shared by all objects of the class, not replicated.
  - Does not have any *implicit parameter*, **this**; therefore, cannot access any particular object's fields.
Critter exercise: Chameleon

• All Chameleons should be the same random color, and it should switch every turn.

• Some chameleons are displayed as <:=)==(=- and some are displayed as -=)==(=:>. This is determined by a boolean parameter passed to the Chameleon’s constructor.
  • true → -=)==(=:>
  • false → <:=)==(=-
import java.awt.*; // For Color
import java.util.*; // For Random

public class Chameleon extends Critter {
    private static int sharedMoveCount;
    private static Color sharedColor;
    private int moveCount;
    private static Random rand;
    private String display;

    public Chameleon(boolean backwards) {
        if (rand == null) {
            rand = new Random();
        }
        moveCount = sharedMoveCount;
        if (backwards) {
            display = "-=)==(>=";
        } else {
            display = "<=)==(<=";
        }
    }

    public String toString() {
        return display;
    }
}

...
... public Direction getMove() {
    moveCount++;
    if (moveCount > sharedMoveCount) {
        sharedMoveCount++;
        sharedColor = new Color(rand.nextInt(256),
                                rand.nextInt(256), rand.nextInt(256));
    }
    return Direction.CENTER;
}

public Color getColor() {
    return sharedColor;
}
Multi-class systems

- Most large software systems consist of many classes.
  - One main class runs and calls methods of the others.

- Advantages:
  - code reuse
  - splits up the program logic into manageable chunks
Summary of Java classes

- A class is used for any of the following in a large program:
  - a `program`: Has a main and perhaps other static methods.
    - example: Bagels, Birthday, BabyNames, CritterMain
    - does not usually declare any static fields (except `final`)
  - an `object class`: Defines a new type of objects.
    - example: Point, BankAccount, Date, Critter, Hipster
    - declares object fields, constructor(s), and methods
    - might declare static fields or methods, but these are less of a focus
    - should be encapsulated (all fields and static fields `private`)
  - a `module`: Utility code implemented as static methods.
    - example: Math