Building Java Programs

Chapter 8
Lecture 8-3: Object state;
Homework 8 (Critters)

reading: 8.3 - 8.4

The keyword `this`

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>`***`);`

3

Variable shadowing

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

```
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.

4

Fixing shadowing

```
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`

5

Object state

The `Parent` class

```
public class Parent {
    private int count;

    public Parent() {
        count = 0;
    }

    public String areWeThereYet() {
        count++;
        if (count >= 7) {
            return "NO!!!!  Now sit down and shut up, you ungrateful little brat!";
        } else if (count % 2 == 0) {
            return "We'll be there soon";
        } else {
            return "We're almost there";
        }
    }
}
```

7

The `Parent` class: Version 2

```
public class Parent {
    private int count;
    private int threshold;

    public Parent(int threshold) {
        count = 0;
        this.threshold = threshold;
    }

    public String areWeThereYet() {
        count++;
        if (count >= threshold) {
            return "NO!!!! Now sit down and shut up, you ungrateful little brat!";
        } else if (count % 2 == 0) {
            return "We'll be there soon";
        } else {
            return "We're almost there";
        }
    }
}
```

8

Exercise

- Write a class `Remote` that implements a TV remote control with a "jump" button. The remote keeps track of the TV channel. When the user presses "jump", the channel is set to the previous channel.

The remote should have the following methods:

- `up()`: sets the channel to be the next one up
- `down()`: sets the channel to be the next one down
- `setChannel(int)`: sets the channel to an arbitrary channel
- `jump()`: sets the channel to the previous channel

Solution

```
public class Remote {
    private int channel;
    private int previousChannel;

    public Remote() {
        channel = 2;
        previousChannel = 2;
    }

    public void up() {
        setChannel(channel + 1);
    }

    public void down() {
        setChannel(channel - 1);
    }

    ...
```

```
    public void jump() {
        setChannel(previousChannel);
    }

    public void setChannel(int num) {
        previousChannel = channel;
        channel = num;
        printChannel();
    }

    public void printChannel() {
        System.out.println("The channel is "
                           + channel);
    }
}
```

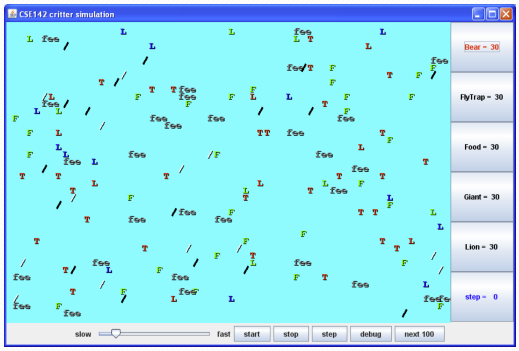
Homework 8: Critters

reading: HW8 assignment spec

11

Critters

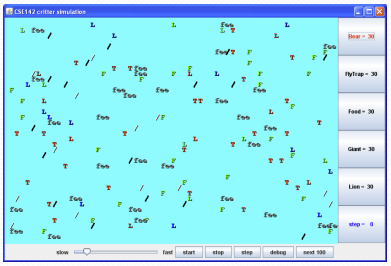
- A simulation world with animal objects.
- Animals move around and can infect one another.
  - If an animal A infects an animal B, then B becomes the same species as A.



12

Critters

- Critter objects have the following behavior:
  - `getColor` color to display
  - `getMove` action
  - `toString` letter to display
- You must implement:
  - `Bear`
  - `Giant`
  - `Lion`
  - `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 3 methods to give your animals behavior.

A completely valid critter

```
public class Default extends Critter {
}
```

- The critters of this species are black question marks that always turn left.

15

How the simulator works

- When you press "Go", the simulator enters a loop:
  - Asks each animal (`getMove`) once what move it wants to make
    - The order that the animals are asked changes over the course of the simulation
- 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.

16

The `getMove` method

- The simulator will ask your critter for a move via the `getMove` method
- The `getMove` method must return one of the following constants from the `Action` class:

Constant	Description
`Action.HOP`	Move forward one square in its current direction
`Action.LEFT`	Turn left (rotate 90 degrees counter-clockwise)
`Action.RIGHT`	Turn right (rotate 90 degrees clockwise)
`Action.INFECT`	Infect the critter in front of you

17

Implementing a Critter

- Critters redefine the following methods

```
public Action getMove(CritterInfo info) {
     ...
}

public Color getColor() {
   ...
}

public String toString() {
   ...
}
```

18

Example Critter

```
import java.awt.*;

public class Food extends Critter {
    public Action getMove(CritterInfo info) {
        return Action.INFECT;
    }

    public Color getColor() {
        return Color.GREEN;
    }

    public String toString() {
        return "F";
    }
}
```

19

`getMove` and `CritterInfo`

- The `getMove` method takes a `CritterInfo` object as parameter:

```
public Action getMove(CritterInfo info) {
    ...
}
```

- `CritterInfo` methods:

Method	Description
`public Neighbor getFront()`	returns neighbor in front
`public Neighbor getBack()`	returns neighbor in behind
`public Neighbor getLeft()`	returns neighbor to the left
`public Neighbor getRight()`	returns neighbor to the right
`public Direction getDirection()`	returns direction critter is facing
`public int getInfectCount()`	returns # of critters infected by critter

20

Hello `Neighbor`!

Constant	Description
`Neighbor.WALL`	The neighbor in that direction is a wall
`Neighbor.EMPTY`	The neighbor in that direction is an empty square
`Neighbor.SAME`	The neighbor in that direction is a critter of your species
`Neighbor.OTHER`	The neighbor in that direction is a critter of another species

```
public Action getMove(CritterInfo info) {
    if (info.getFront() == Neighbor.EMPTY) {
        return Action.HOP;
    } else {
        return Action.LEFT;
    }
}
```

21

`Direction`

Constant	Description
`Direction.NORTH`	facing north
`Direction.SOUTH`	facing south
`Direction.EAST`	facing east
`Direction.WEST`	facing west

```
public Action getMove(CritterInfo info) {
    if (info.getDirection() == Direction.NORTH) {
        return Action.INFECT;
    } else {
        return Action.LEFT;
    }
}
```

22

Critter exercise: `FlyTrap`

- Write a critter class `FlyTrap`:

Method	Behavior
constructor	`public FlyTrap()`
`getColor`	red
`getMove`	always infect if an enemy is in front otherwise turn left
`toString`	`"T"`

`FlyTrap`

```
public class FlyTrap extends Critter {
    public Action getMove(CritterInfo info) {
        if (info.getFront() == Neighbor.OTHER) {
            return Action.INFECT;
        } else {
            return Action.LEFT;
        }
    }

    public Color getColor() {
        return Color.RED;
    }

    public String toString() {
        return "T";
    }
}
```

Critter exercise: `Blinker`

Method	Behavior
constructor	`public Blinker()`
`getColor`	alternates between red and green
`getMove`	always infects
`toString`	`"X"`

- NOTE: The simulator calls the `getMove` method once per turn. All other methods may be called more than once per turn.

25

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?

26

Keeping state

- How can a critter alternate colors?

```
public Color getColor () {
    boolean isRed = false;
    while (true) {
        isRed = !isRed;
        if (isRed) {
            return Color.RED
        } else {
            return Color.GREEN;
        }
    }
}
```


Blinker

```
import java.awt.*;

public class Blinker extends Critter {
    private int moves; // total moves made by this Critter

    public Action getMove(CritterInfo info) {
        moves++;
        return Action.INFECT;
    }

    public Color getColor() {
        if (moves % 2 == 0) {
            return Color.GREEN;
        } else {
            return Color.RED;
        }
    }

    public String toString() {
        return "X";
    }
}
```

Testing critters

- Focus on one specific critter
  - Only spawn 1 animal of the species being debugged
- Make sure your fields update properly
  - Use `println` statements to see field values
  - Or use the debugger
- Look at the behavior one step at a time
  - Use "step" rather than "start"
- Use "debug" to see what direction critters are facing

Critter exercise: `Snake`

Method	Behavior
constructor	`public Snake()`
`getColor`	black
`getMove`	hop if possible otherwise turn around
`toString`	`"S"`

Snake solution

```
import java.awt.*;

public class Snake extends Critter {
    boolean turning;

    public Snake() {
        turning = false;
    }

    public Action getMove(CritterInfo info) {
        if (turning) {
            turning = false;
            return Action.LEFT;
        } else if (info.getFront() == Neighbor.EMPTY) {
            return Action.HOP;
        } else {
            turning = true;
            return Action.LEFT;
        }
    }

    public Color getColor() {
        return Color.BLACK;
    }

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

31