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**`);`

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.

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`

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";
        }
    }
}
```

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";
        }
    }
}
```

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);
    }

    ...
```

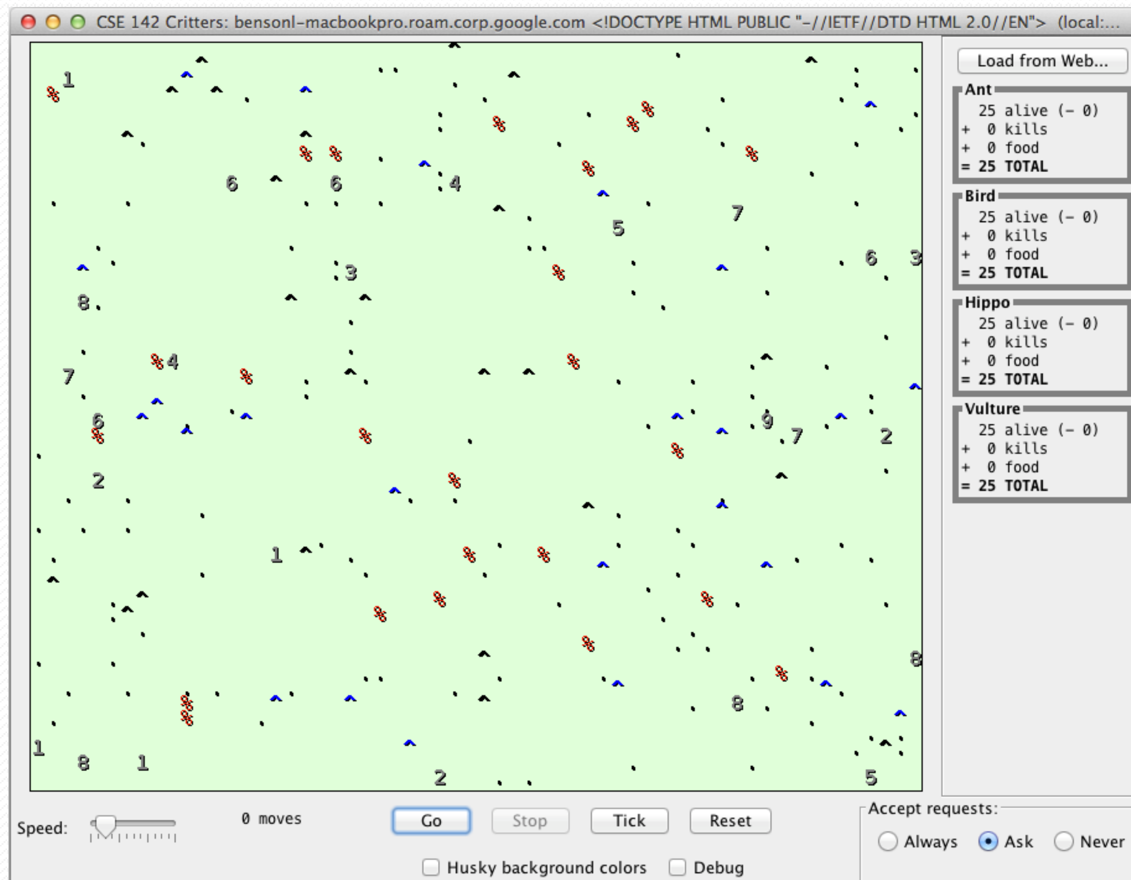
```
    public void jump() {
        setChannel(previousChannel);
    }

    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

Critters

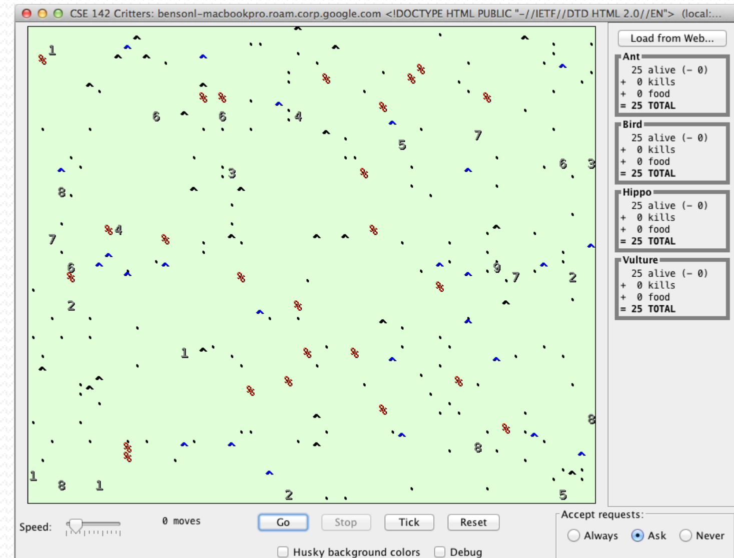
- A simulation world with animal objects.
- Animals move around, eat, and can fight one another.



Critters

- Critter objects have the following behavior:
  - `fight` what type of attack
  - `eat` whether to eat
  - `getColor` color to display
  - `getMove` direction to move
  - `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* (see later)
- Write some/all 5 methods to give your animals behavior.

Implementing a Critter

- Critters redefine the following methods (defaults shown):

```
public boolean eat() {
    return false;
}

public Attack fight(String opponent) {
    return Attack.FORFEIT;
}

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

public Direction getMove() {
    return Direction.CENTER;
}

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

A completely valid critter

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

- The critters of this species are black question marks that don’t move, don’t fight, and never eat.

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.

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 `Direction` class:

Constant	Description
`Direction.CENTER`	Stay in place
`Direction.NORTH`	Move one space to the top of the screen
`Direction.SOUTH`	Move one space to the bottom of the screen
`Direction.EAST`	Move one space to the right of the screen
`Direction.WEST`	Move one space to the left of the screen

The `fight` method

- The `fight` method must return one of the following constants from the `Attack` class:

Constant	Description
`Attack.ROAR`	"Roar" beats "scratch"
`Attack.POUNCE`	"Pounce" beats "roar"
`Attack.SCRATCH`	"Scratch" beats "pounce"

- Rock-paper-scissors
  - http://en.wikipedia.org/wiki/Roshambo

Example Critter

```
import java.awt.*;

public class Martian extends Critter {
    public boolean eat() {
        return true;
    }

    public Attack fight(String opponent) {
        return Attack.SCRATCH;
    }

    public Direction getMove() {
        return Direction.NORTH;
    }

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

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

Critter exercise: `Blinker`

Method	Behavior
constructor	`public Blinker()`
`eat`	never eats
`fight`	default behavior (forfeit)
`getColor`	alternates between red and green
`getMove`	stays in place
`toString`	`"X"`

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

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?

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

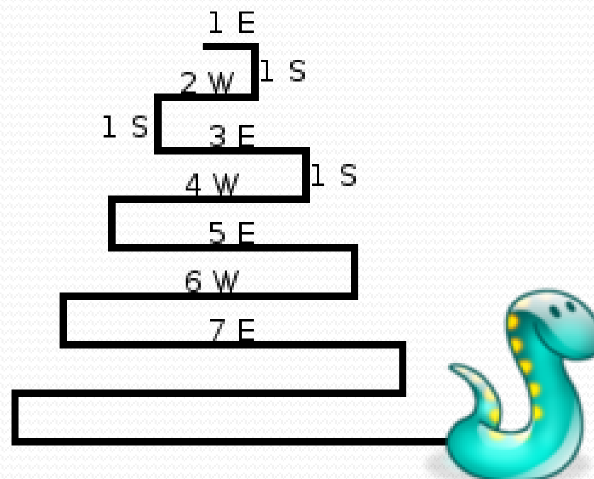
```
import java.awt.*;

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

    public Direction getMove() {
        moves++;
        return Direction.CENTER;
    }

    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
  - Or use `MiniMain`
- Look at the behavior one step at a time
  - Use "Tick" rather than "Go"

Critter exercise: `Snake`

Method	Behavior
constructor	`public Snake()`
`eat`	Never eats
`fight`	always forfeits
`getColor`	black
`getMove`	1 E, 1 S; **2** W, 1 S; **3** E, 1 S; **4** W, 1 S; **5** E, ...
`toString`	`"S"`



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

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";
    }
}
```

Building Java Programs

Chapter 8

Lecture 8-4: Static Methods and Fields

(OPTIONAL)

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

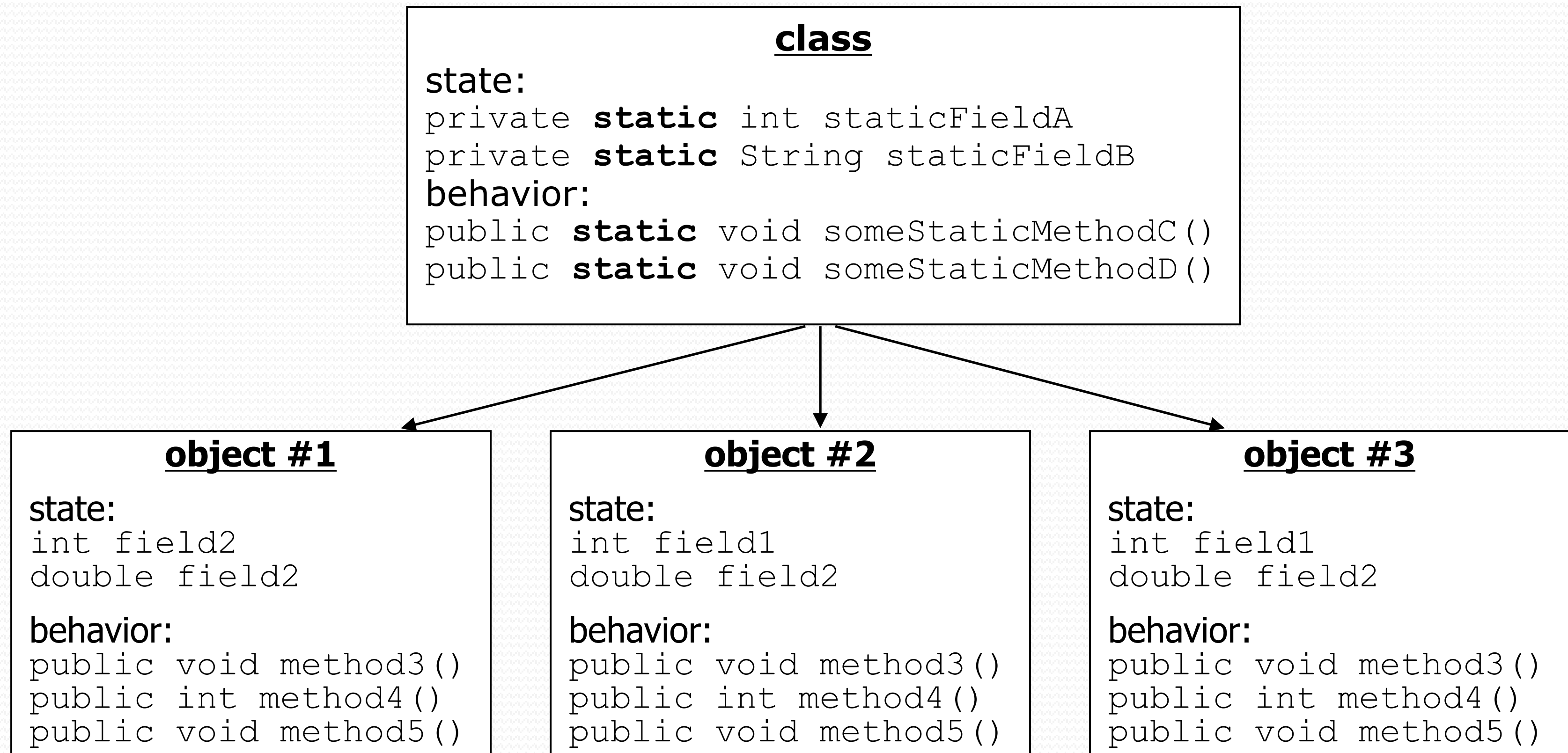
A flawed solution

```
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.



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:

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

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

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

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

- Exercise: Modify the `BankAccount` class shown previously so that each account is automatically given a unique ID.
- Exercise: Write the working version of `Hipster`.

BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;       // advance the id, and
        id = objectCount;    // give number to account
    }

    ...

    public int getID() {     // return this account's id
        return id;
    }
}
```

Hipster solution

```
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 hipter)
    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.

- Exercise: Make it so that clients can find out how many total `BankAccount` objects have ever been created.

BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // clients can call this to find out # accounts created
    public static int getNumAccounts() {
        return objectCount;
    }

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;       // advance the id, and
        id = objectCount;    // give number to account
    }

    ...
    public int getID() {    // return this account's id
        return id;
    }
}
```