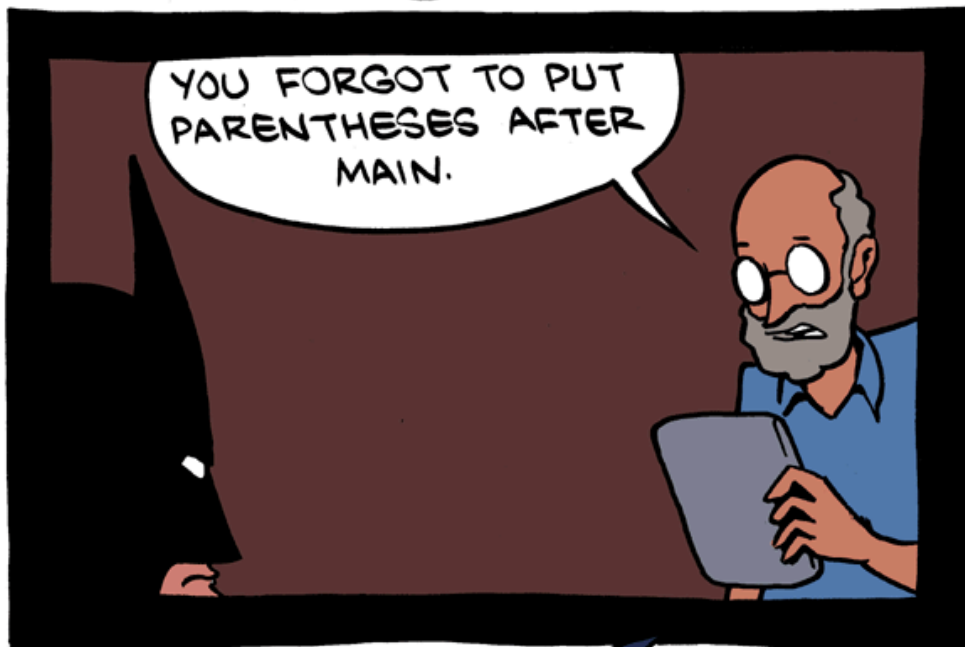


Homework 8: Critters

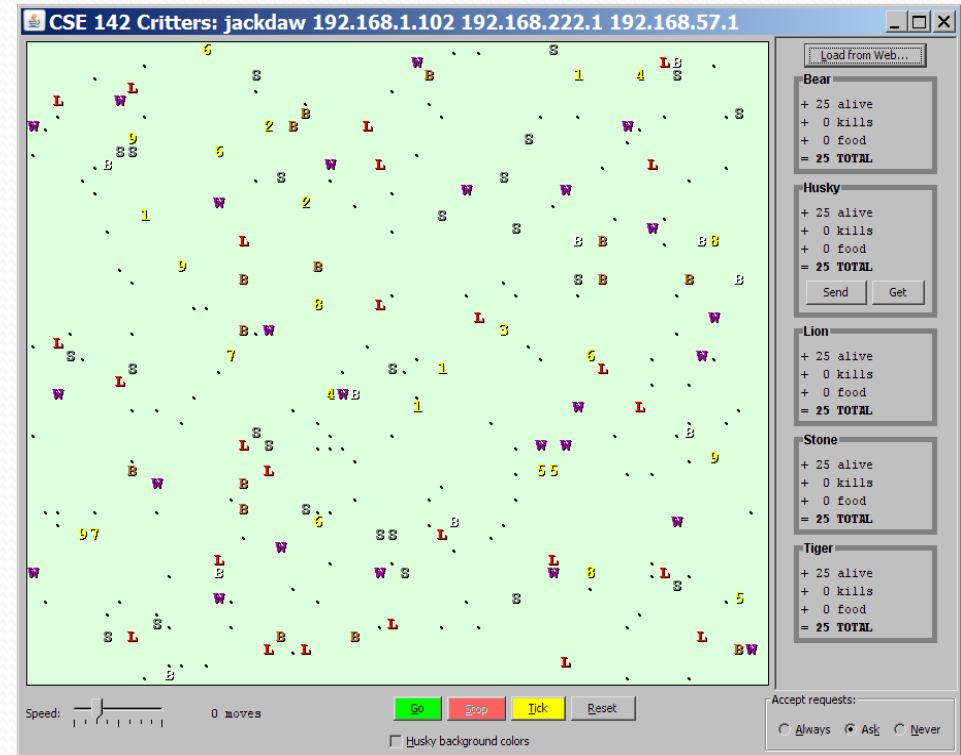
reading: HW8 spec

BATMAN vs. PROGRAMMING



CSE 142 Critters

- Ant
 - Bird
 - Hippo
 - Vulture
 - Husky
- (creative)
- **behavior:**
 - eat eating food
 - fight animal fighting
 - getColor color to display
 - getMove movement
 - toString letter to display



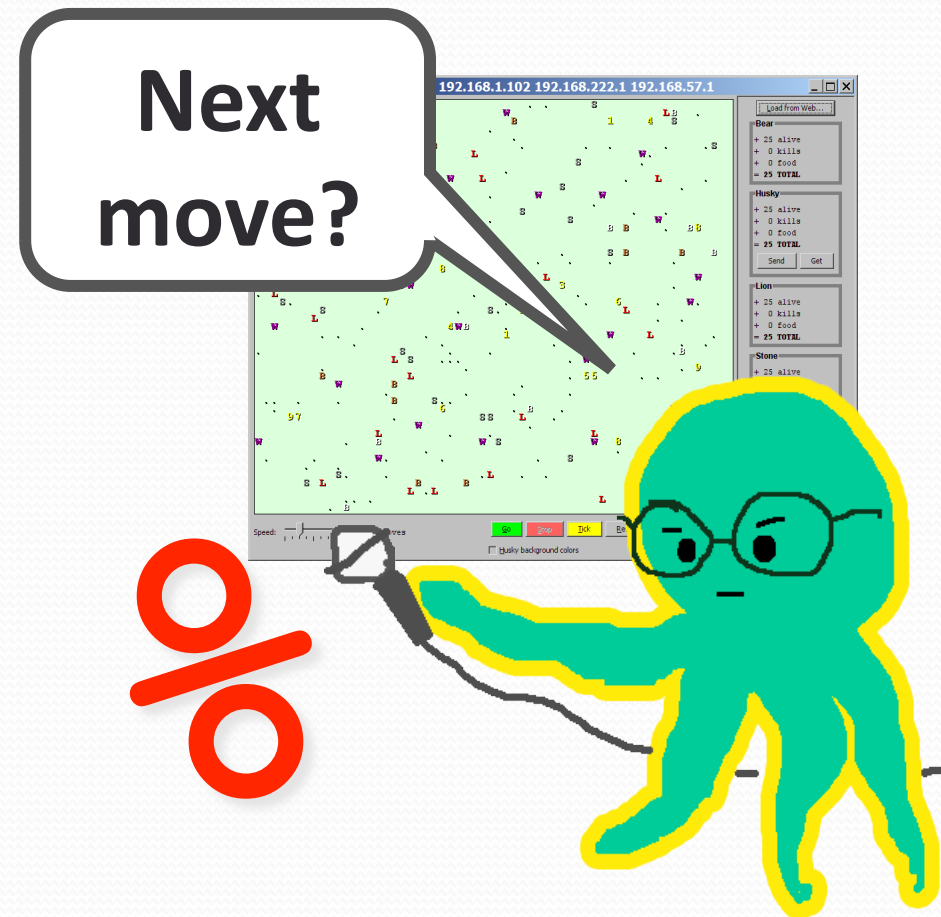
A Critter subclass

```
public class name extends Critter { ... }
```

```
public abstract class Critter {  
    public boolean eat()  
    public Attack fight(String opponent)  
        // ROAR, POUNCE, SCRATCH  
    public Color getColor()  
    public Direction getMove()  
        // NORTH, SOUTH, EAST, WEST, CENTER  
    public String toString()  
}
```

How the simulator works

- "Go" → loop:
 - move each animal (`getMove`)
 - if they collide, fight
 - if they find food, eat
- Simulator is in control!
 - `getMove` is one move at a time
 - (no loops)
 - Keep state (fields)
 - to remember future moves



Development Strategy

- Do one species at a time
 - in ABC order from easier to harder (Ant → Bird → ...)
 - debug `println`
- Simulator helps you debug
 - smaller width/height
 - fewer animals
 - **"Tick"** instead of "Go"
 - **"Debug"** checkbox
 - drag/drop to move animals

Critter exercise: Cougar

- Write a critter class `Cougar`:

Method	Behavior
<code>constructor</code>	<code>public Cougar()</code>
<code>eat</code>	Always eats.
<code>fight</code>	Always pounces.
<code>getColor</code>	Blue if the <code>Cougar</code> has never fought; red if he has.
<code>getMove</code>	Walks west until he finds food; then walks east until he finds food; then goes west and repeats.
<code>toString</code>	"C"

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?

Cougar solution

```
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;  
    }  
}
```

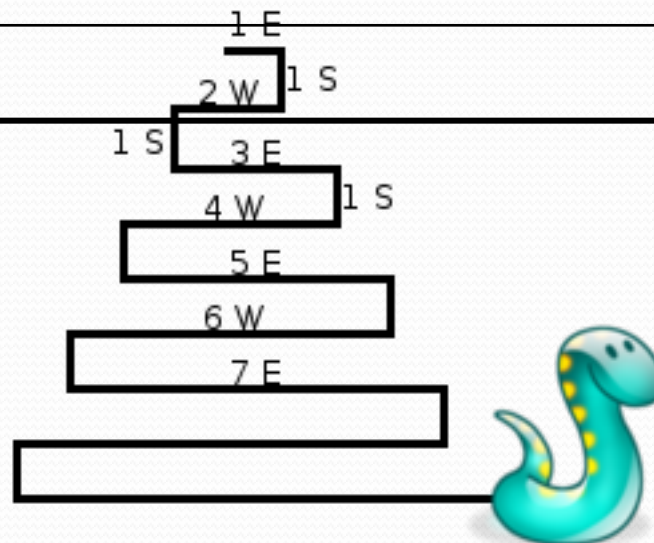
```
public Direction getMove() {  
    if (west) {  
        return Direction.WEST;  
    } else {  
        return Direction.EAST;  
    }  
}
```

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

```
}
```

Critter exercise: Snake

Method	Behavior
constructor	<code>public Snake()</code>
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
- 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

```
import java.util.*;    // for Random
public class Hipster extends Critter {
    private int cheapBarX;
    private int cheapBarY;

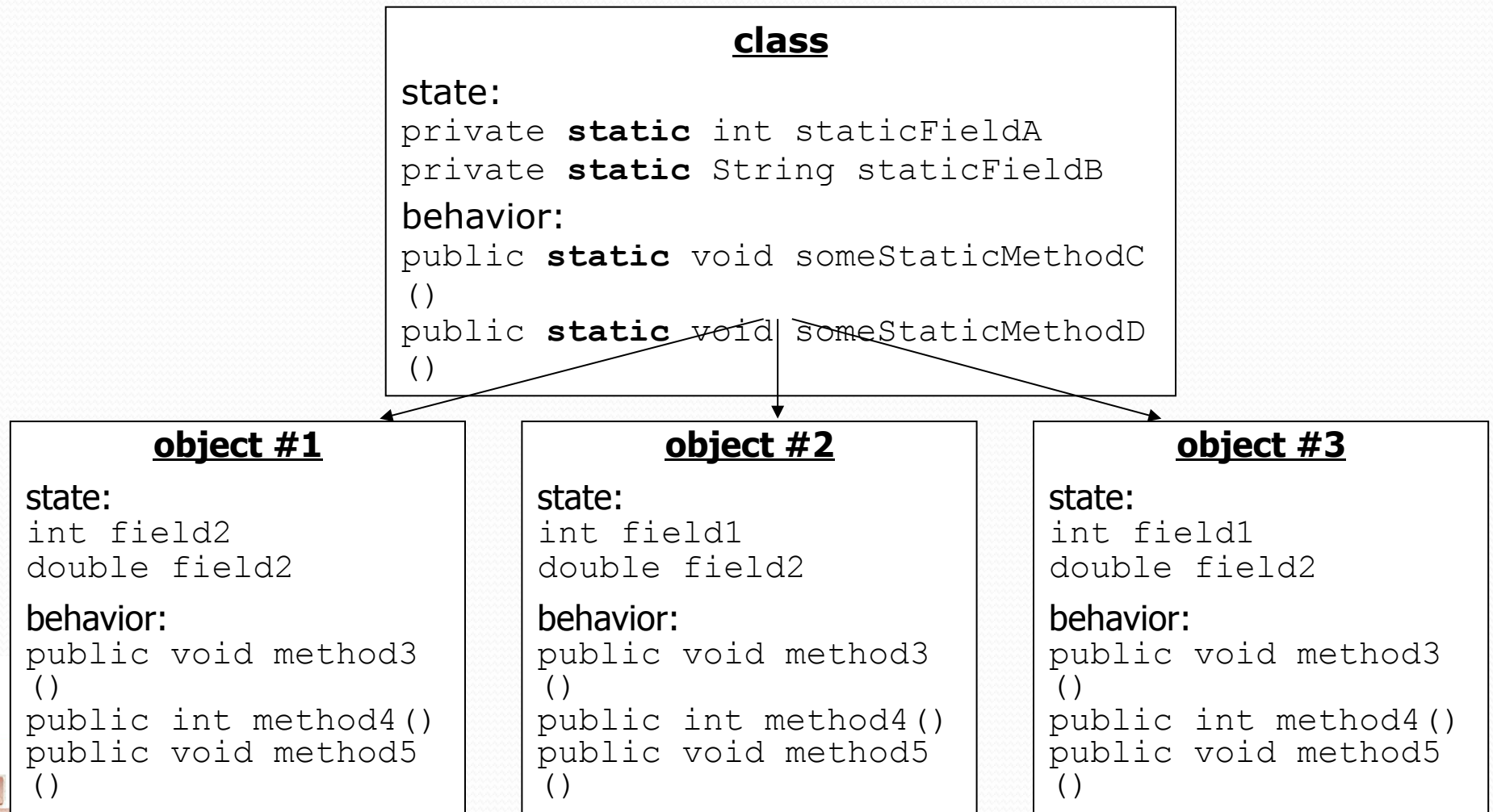
    public Hipster() {
        Random r = new Random();
        cheapBarX = r.nextInt(60);
        cheapBarY = r.nextInt(50);
    }

    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`.

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 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;  
}
```

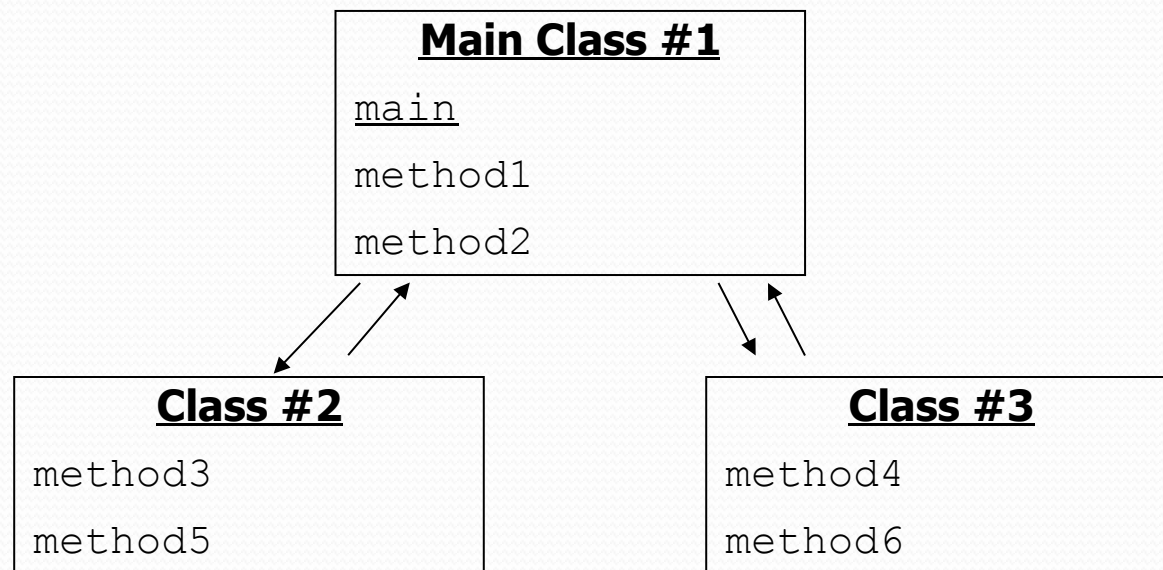
- **static method:** Stored in a class, not in an object.
 - Shared by all objects of the class, not replicated.
 - Does not have any *implicit parameter*, `this`; therefore, cannot access any particular object's fields.
- **Exercise:** Make it so that clients can find out how many total `BankAccount` 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;  
    }  
}
```

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`