TH Assessment 8: Critters

reading: A8 spec
KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.

LEMMIE SEE— I'M SURE IT'S FINE.

...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.

IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.

IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.

OKAY, I'LL READ A STYLE GUIDE.
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
How the simulator works

- "Go" → loop:
  - move each animal \( \text{getMove} \)
  - if they collide, fight
  - if they find food, eat

- Simulator is in control!
  - \text{getMove} \text{ is one move at a time}
    - \text{(no loops)}
  - Keep state (fields)
    - to remember future moves
A Critter subclass

```java
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()
}
```
Sidebar: Color

• Specified as predefined `Color` class constants:

```java
Color.CONSTANT_NAME
```

where `CONSTANT_NAME` is one of:

- BLACK
- BLUE
- CYAN
- DARK_GRAY
- GRAY
- GREEN
- LIGHT_GRAY
- MAGENTA
- ORANGE
- PINK
- RED
- WHITE
- YELLOW

• Example:

```java
Color.MAGENTA
```
Making your own colors

- Create colors using **Red-Green-Blue (RGB)** values of 0-255

  ```java
  Color name = new Color(red, green, blue);
  ```

- Example:
  ```java
  Color brown = new Color(192, 128, 64);
  ```

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**: 

<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?
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;
    }
}
```

...
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

<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";
    }
}

Snake solution
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()
```

```java
object #1
state:
int field2
double field2

behavior:
public void method3()
public int method4()
public void method5()
```

```java
object #2
state:
int field1
double field2

behavior:
public void method3()
public int method4()
public void method5()
```

```java
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: Modify the `BankAccount` class shown previously so that each account is automatically given a unique ID.
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.
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
    }

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

```java
Main Class #1
main
method1
method2

Class #2
method3
method5

Class #3
method4
method6
```
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