Building Java Programs

Chapter 9
Critters; Subtype Polymorphism

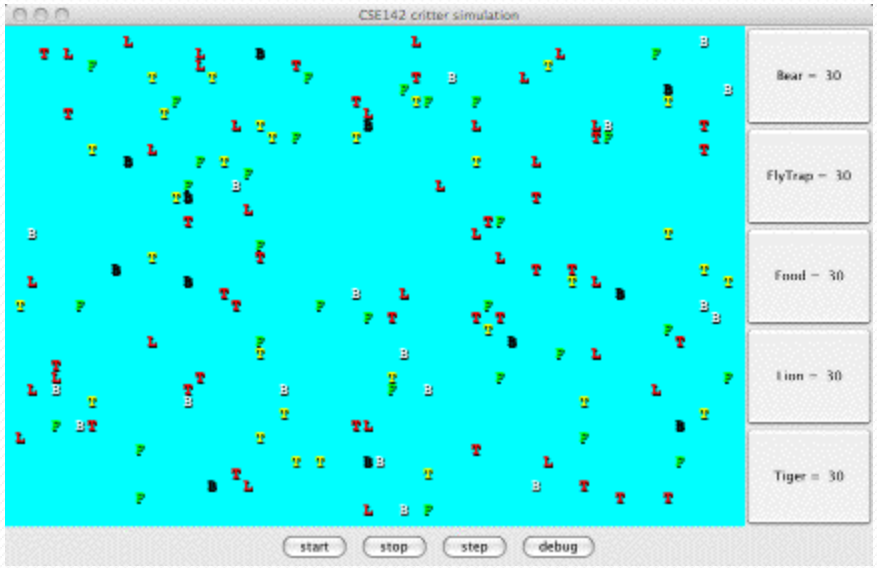
Reading: HW9 Handout, Chapter 9.2

Critters

- A 2-D simulation world with animal objects with behavior:
  - `getMove` what to do "on each turn"
  - `toString` letter to display for this animal
  - `getColor` color to display for this animal

- You implement 4 classes (kinds of critters):
  - `Bear`
  - `Lion`
  - `Tiger`
  - `Husky` (creative)

- All other classes written for you



A `Critter` subclass

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

- `extends Critter` tells the simulator your class is a critter
  - an example of *inheritance*
- Write a constructor to initialize each critter's state
- Implement the 3 methods that define the critter's behavior

How the simulator works

- `CritterMain.java` (written for you) makes a bunch of critters and puts them randomly in the world.
  - All you do is (un)comment-out relevant lines

- When you press "start", the simulator enters a loop:
  - moves each animal once (`getMove`), in random order
  - uses `getColor` and `toString` to display your critter

- 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.

- Your animal must keep *state* (as fields) so that it can make a *single move*, and know what moves to make later.

Actions

Each critter is in some *position* facing some *direction*

Every `getMove` method returns an `Action`,
which is 1 of 4 constants:

- `Action.HOP`: Forward 1 space (no effect if occupied or wall)
- `Action.LEFT`: Turn 90-degrees counter-clockwise
- `Action.RIGHT`: Turn 90-degrees clockwise
- `Action.INFECT`: Infect critter in front of you (no effect if no critter in front or your own species)
  - Turns other critter into one of your species (!)

CritterInfo

- The argument to `getMove` is an object with methods that provide lots of useful information:
  - Neighbors: what is in front, behind, to left, and to right
    - wall, nothing, same species, another species
  - Direction: what way are you facing
    - North, South, East, West
  - Infection count: number of critters you have infected
    - Only useful if trying for world domination (see the handout)

- But your critters will also need state (fields) to remember enough about what they have done in the past
  - Example need:

    "Hop forward unless that is what I did on my last move"
  - Example in section tomorrow

Tournament

- Your `Husky` class can do whatever you want
  - Some style points dedicated to creativity
- To win the tournament, must best "survive" in a world filled with other species (your opponents)
  - Details posted later
- "Playoffs" in class on last day

Example Critters

The code provided to you also includes two simple critters

- Yours will be more interesting

- Food: Stay in one place, easy to be infected
  - Does try to infect others (rather unlike "food")

- FlyTrap: Stay in one place, but spin around and always try to infect
  - A surprisingly good strategy

Critter exercise

- Write a critter class `Cougar` (the dumbest of all animals):

Method	Behavior
`getMove`	Hop unless at wall then turn left
`getColor`	red
`toString`	`"C"`

Ideas for state

- You must not only have the right state, but update that state properly when relevant actions occur.

- Two approaches:
  - How many moves of some sort has this animal made?
  - What has this animal done recently?

(The first approach is often shorter.)

- Food, FlyTrap, and Cougar are too simple to need state.

Testing critters

- Focus on one specific Critter of one specific type
  - Only spawn 1 of each Critter type
  - (Be sure to test with more later)
- Make sure your fields update properly
  - Use `println` statements to see field values
- Look at the behavior one step at a time
  - Use "step" rather than "start"
- Debug: Shows direction faced rather than normal String
- Example: Cougar without most other species

Building Java Programs

Chapter 9
Lecture 9-3: Polymorphism

reading: 9.1-9.2
self-check: #5-9

Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects.
  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
  - `CritterMain` can interact with any type of critter.
    - Each one moves, infects, etc. in its own way.
- Java supports polymorphism in a few ways
  - We will learn about *subtyping* via *inheritance*

Coding with polymorphism

- A variable of type *T* can hold an object of any subclass of *T*.

```
Employee ed = new Lawyer();
```

  - You can call any methods from `Employee` on `ed`.
  - You *cannot* call any methods specific to `Lawyer` (e.g. `sue`).

- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary());          // 40000.0
System.out.println(ed.getVacationForm());    // pink
```

Polymorphism and parameters

- You can pass any subtype of a parameter's type.

```
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer leslie = new Lawyer();
        TechnicalWriter toby = new TechnicalWriter();
        printInfo(leslie);
        printInfo(toby);
    }

    public static void printInfo(Employee empl) {
        System.out.println("salary = " + empl.getSalary());
        System.out.println("days = " + empl.getVacationDays());
        System.out.println("form = " + empl.getVacationForm());
        System.out.println();
    }
}
```

OUTPUT:

```
salary = 40000.0
vacation days = 15
vacation form = pink

salary = 40000.0
vacation days = 10
vacation form = yellow
```

Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] e = { new Lawyer(),   new TechnicalWriter(),
                         new Marketer(), new Lawyer() };

        for (int i = 0; i < e.length; i++) {
            System.out.println("salary: " + e[i].getSalary());
            System.out.println("v.days: " + e[i].getVacationDays());
            System.out.println();
        }
    }
}
```

Output:

```
salary: 40000.0
v.days: 15

salary: 40000.0
v.days: 10

salary: 50000.0
v.days: 10

salary: 40000.0
v.days: 15
```

Polymorphism problems

- A few classes with inheritance relationships are shown.
  - Can have multiple levels of subclasses
- A client program calls methods on objects of each class.
- You must read the code and determine the client's output.
- (On the final exam, at least a “simple” version)

A polymorphism problem

- Assume that the following four classes have been declared:

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

A polymorphism problem

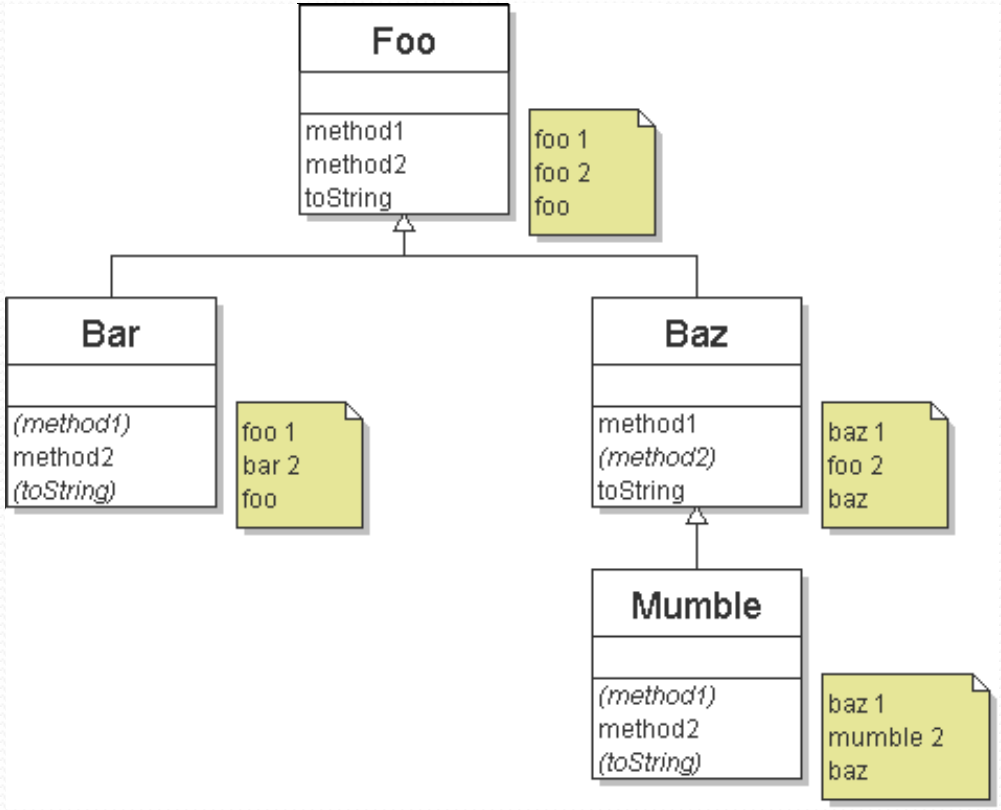
```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

- What would be the output of the following client code?

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Diagramming the classes

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.



Finding output with tables

method	Foo	Bar	Baz	Mumble
`method1`	`foo 1`	*`foo 1`*	`baz 1`	*`baz 1`*
`method2`	`foo 2`	`bar 2`	*`foo 2`*	`mumble 2`
`toString`	`foo`	*`foo`*	`baz`	*`baz`*

Polymorphism answer

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

- Output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

A harder problem

- The order of the classes is jumbled up (easy).
- The methods sometimes call other methods (tricky!!)

```
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}
```

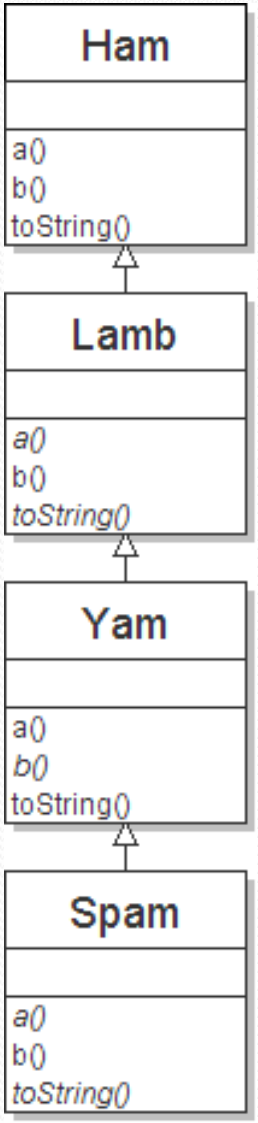
Another problem 2

```
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b   ");
    }
}
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a   ");
    }
    public String toString() {
        return "Yam";
    }
}
```

- What would be the output of the following client code?

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();      // to end the line of output
    food[i].b();
    System.out.println();      // to end the line of output
    System.out.println();
}
```

Class diagram



Polymorphism at work

- `Lamb` inherits `Ham`'s `a`. `a` calls `b`. But `Lamb` overrides `b`...

```
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
```

- `Lamb`'s output from `a`:

```
Ham a    Lamb b
```

The table

method	Ham	Lamb	Yam	Spam
a	Ham a **b()**	Ham a **b()**	Yam a	Yam a
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	Ham	Yam	Yam

The answer

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

- Output:

```
Ham
Ham a   Lamb b
Lamb b

Ham
Ham a   Ham b
Ham b

Yam
Yam a
Spam b

Yam
Yam a
Lamb b
```