CS 106A, Lecture 20
Critters

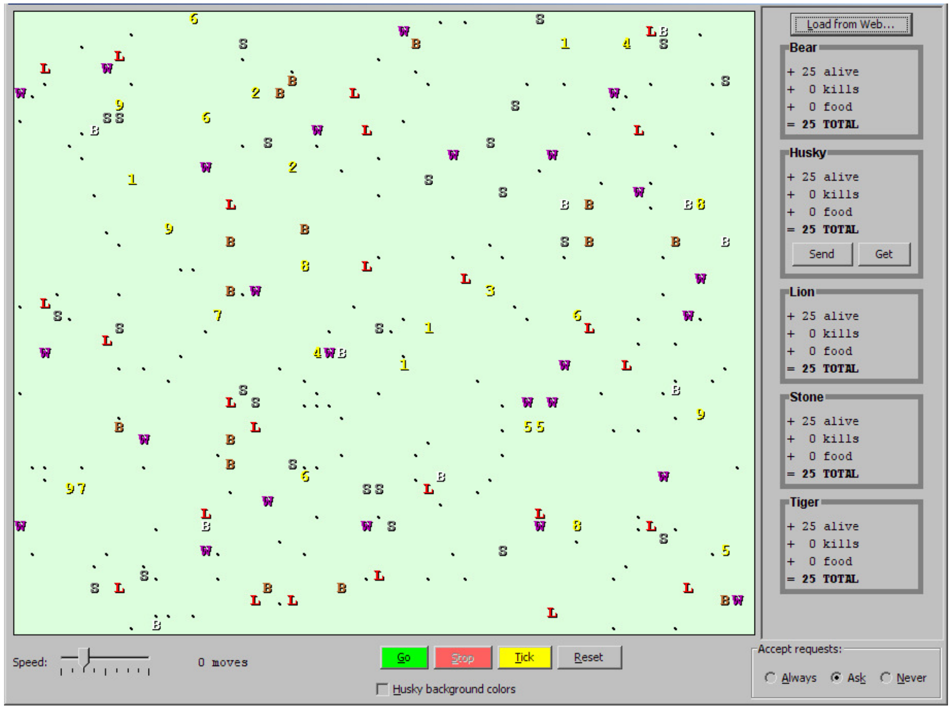
reading:

none

Based on slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, and others.

CS 106A Critters

- **Critters**: A 2D world with "animals" represented as objects.
  - `Ant`
  - `Bird`
  - `Crab`
  - `Hippo`
  - `Vulture`
  - `Wolf` (creative)



```
Bear
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Husky
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Lion
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Stone
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Tiger
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL
```

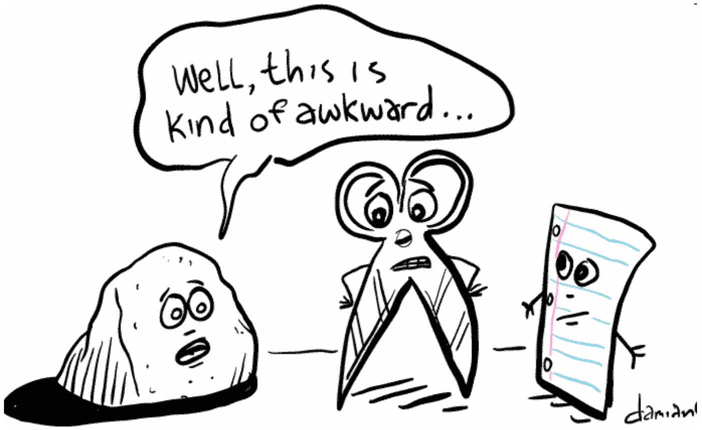
Accept requests: ○ Always ● Ask ○ Never

- each animal's behavior:

– `eat`	eating food
– `fight`	animal fighting
– `getColor`	color to display
– `getMove`	movement
– `toString`	letter to display

A Critter subclass

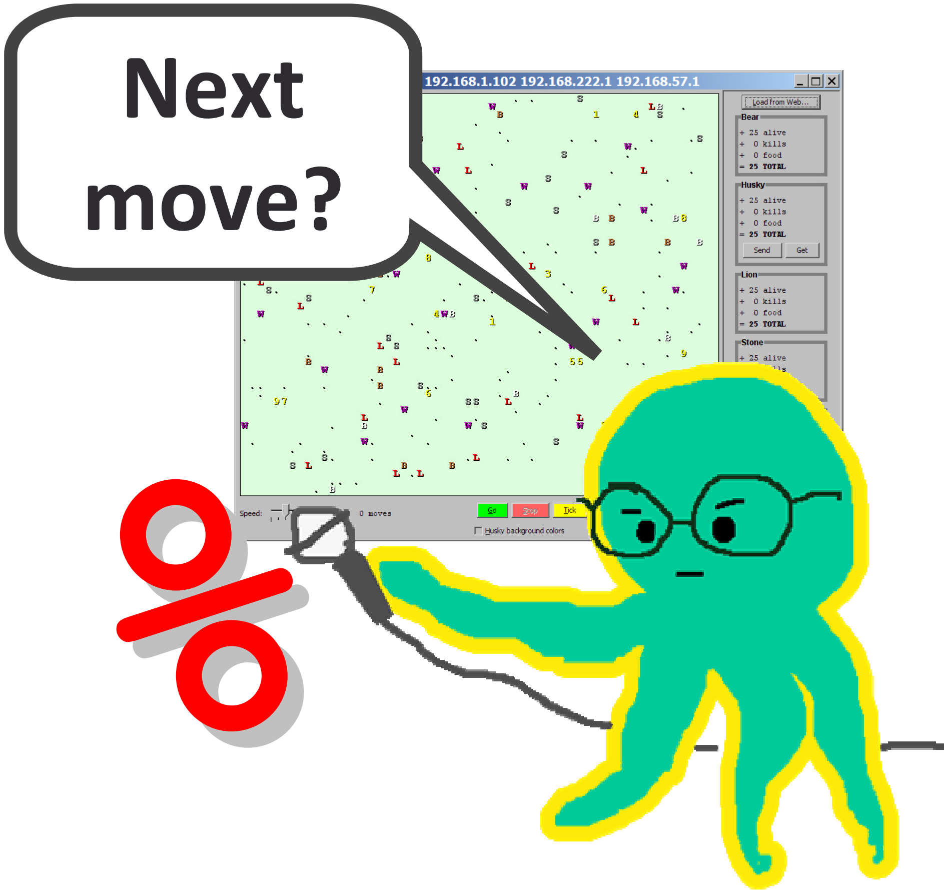
```
public class Name extends Critter { ... }

public 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:
  - for each animal:
    - move the animal (`getMove`)
    - if 2 animals collide: `fight`
    - if animal finds food: `eat`
  - update/redraw the screen

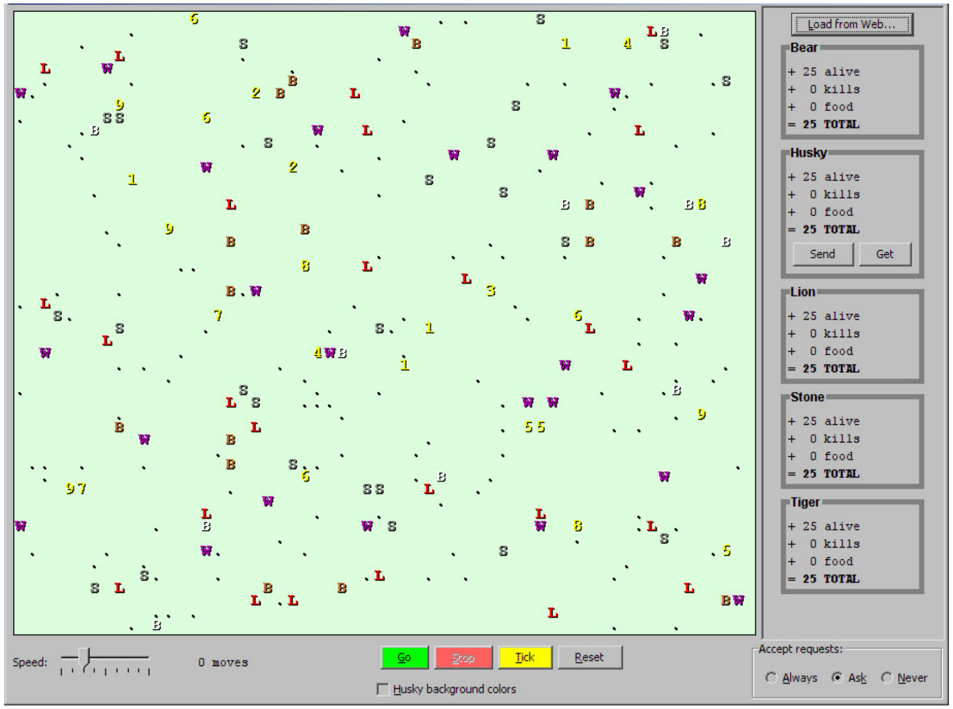
- Simulator is in control!
  - `getMove` is <u>one move</u> at a time
    - (*no loops*)
  - Keep <u>state</u> (fields)
    - to remember future moves and behavior



Development strategy

- Do one species at a time
  - roughly in ABC order from easier to harder (Ant → Bird → ...)
  - use `println` statements for debugging

- Simulator helps you debug
  - smaller width/height
  - fewer animals
  - **"Tick"** instead of "Go"
  - **"Debug"** checkbox
  - drag/drop to move animals



```
Bear
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Husky
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Lion
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Stone
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL

Tiger
+ 25 alive
+  0 kills
+  0 food
= 25 TOTAL
```

0 moves

□ Husky background colors

Accept requests: ○ Always ● Ask ○ Never

FrenchBulldog exercise

- Write a critter class **`FrenchBulldog`**:

Method	Behavior
constructor	`public FrenchBulldog()`
`eat`	Always eats.
`fight`	Always pounces.
`getColor`	White if she has never fought; red if she has.
`getMove`	Walks west until she finds food; then walks east until she finds food; then goes west and repeats.
`toString`	`"A"`

Ideas for state

- You need to have the right state, and you must update that state properly when relevant actions occur.

- Integer fields for **counting** are often 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?

FrenchBulldog solution

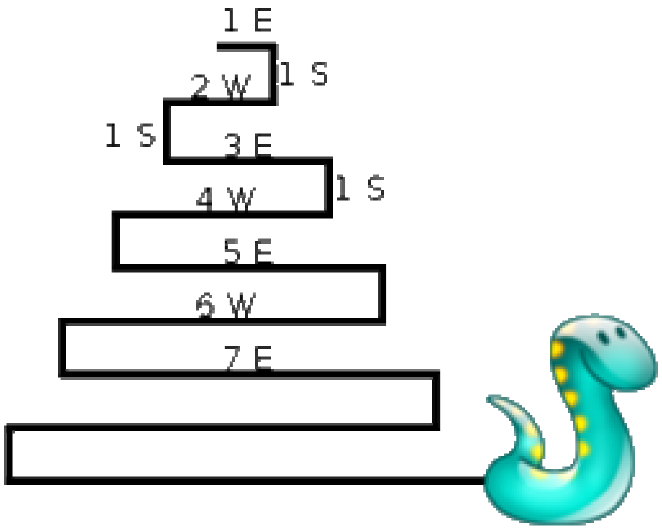
```
import java.awt.*;  // for Color

public class FrenchBulldog extends Critter {
    private boolean west;
    private boolean fought;

    public FrenchBulldog() {
        west = true;
        fought = false;
    }

    public boolean eat() {
        west = !west;
        return true;
    }

    public Attack fight(String opponent) {
        fought = true;
        return Attack.POUNCE;
    }

    ...
```

FrenchBulldog 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 "A";
    }
}
```

Critter: Snake

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



Determining fields

- Information required to decide what move to make?
  - Direction to go in
  - Length of current cycle
  - Number of moves made in current cycle

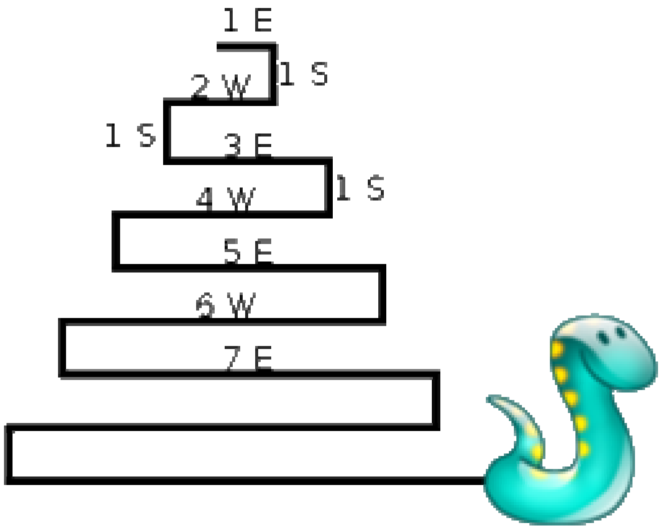
- Remembering things you've done in the past:
  - an `int` counter?
  - a `boolean` flag?

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: Cobra

Method	Behavior
constructor	`public Cobra()`
`eat`	Never eats
`fight`	always chooses POUNCE
`getColor`	red (`Color.RED`)
`getMove`	1 E, **2 S**; 2 W, **2 S**; 3 E, **2 S**; 4 W, **2 S**; 5 E, ...
`toString`	`"S"`



Cobra solution idea

- When critter types are very similar, one can **extend** the other.
  - The more specific type should extend the more general/broad type.

```
public class Cobra extends Snake {
    ...
}
```

Static data and behavior

Bee exercise

- Write a `Critter` class named **Bee**.
  - The bees buzz together in a swarm.
  - They all want to fly to the same destination.

  - When the simulator loads up, the bees collectively choose a random board location to which they will all fly.

    (On the 60-by-50 world)

- They go north then east until they reach this location.

Flawed solution

- **Q:** What's wrong with this solution?

```
public class Bee extends Critter {
    private int flyX, flyY;

    public Bee() {
        RandomGenerator randy = RandomGenerator.getInstance();
        flyX = randy.nextInt(60);
        flyY = randy.nextInt(50);
    }

    public Direction getMove() {
        if (getY() != flyY) {
            return Direction.NORTH;
        } else if (getX() != flyX) {
            return Direction.EAST;
        } else {
            return Direction.CENTER;
        }
    }
}
```

A. The code does not compile.

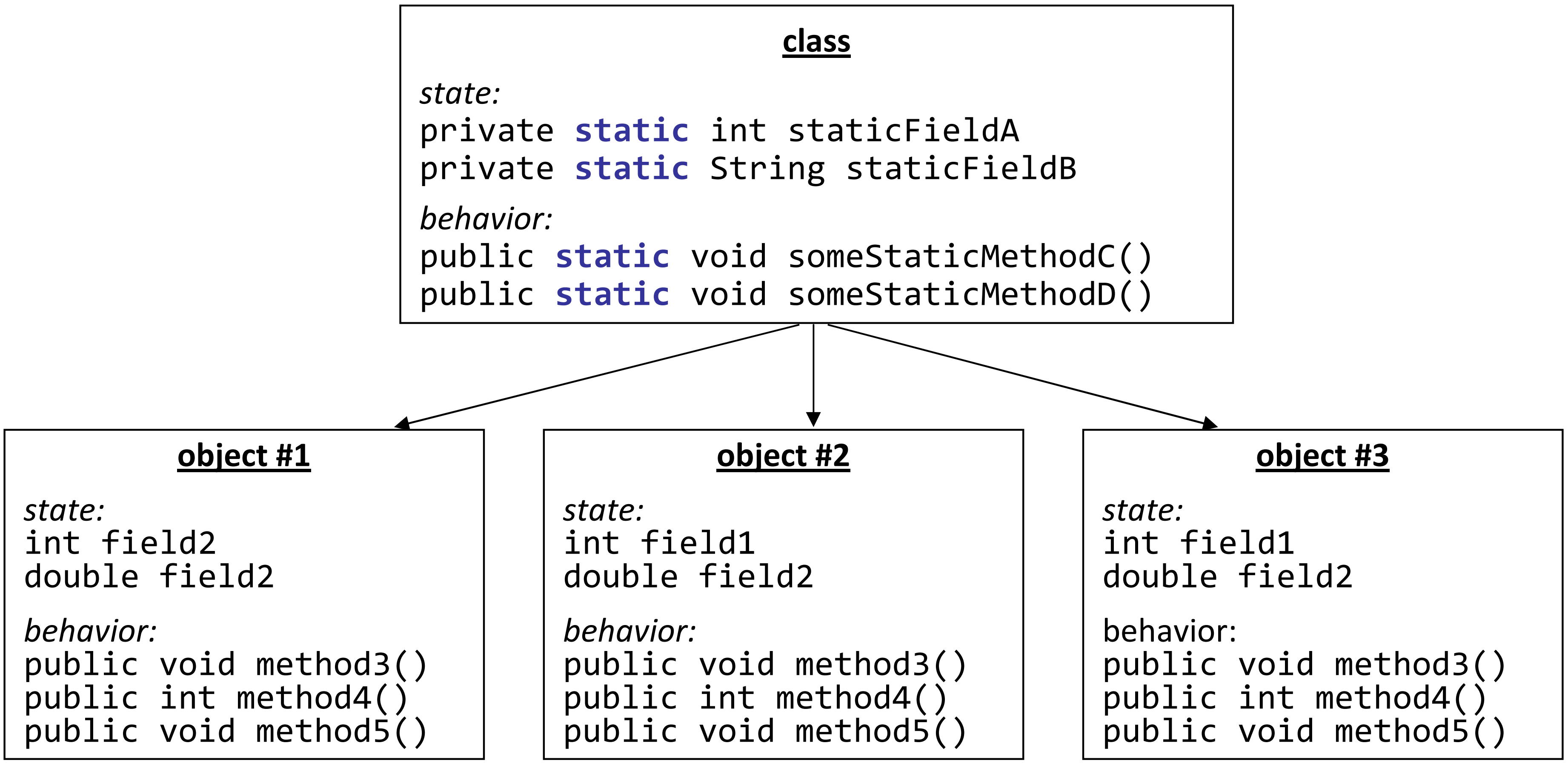
B. Goes to same location every time it is run.

C. The bees do not move properly at all.

D. Each bee goes to its own separate destination.

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.



class

state:

```
private static int staticFieldA
private static String staticFieldB
```

behavior:

```
public static void someStaticMethodC()
public static void someStaticMethodD()
```

object #1

state:

```
int field2
double field2
```

behavior:

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

object #2

state:

```
int field1
double field2
```

behavior:

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

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:

```
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`

Static initializer

```
static {
    statements;
}
```

- **static initializer**: A special block of code to initialize static fields.
  - Runs a single time when the program/class first get loaded.
  - Needed when the static fields can't be set to simple constant values.

- *Exercise:* Write a working version of **Bee** .

Bee solution

```
public class Bee extends Critter {
    // static fields (shared by all bees)
    private static int flyX;
    private static int flyY;

    static {
        RandomGenerator randy = RandomGenerator.getInstance();
        flyX = randy.nextInt(60);
        flyY = randy.nextInt(50);
    }

    // object constructor/methods (replicated into each object)
    public Bee() {
        // empty
    }

    public Direction getMove() {
        if (getY() != flyY) {
            return Direction.NORTH;
        } else if (getX() != flyX) {
            return Direction.EAST;
        } else {
            return Direction.CENTER;
        }
    }
}
```

Static methods

```
public static returnType 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:* Modify the BankAccount class so that each account stores a unique ID number. 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 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;
    }
}
```

Advanced Bee

- Modify **Bee** so that groups of 10 bees will swarm together.
  - Every 10th bee should choose a new location for itself and the next 9 of his bee friends.
    - first ten bees go to location #1
    - next ten bees go to location #2
    - ...

Advanced Bee code

```
public class Bee extends Critter {
    // static fields (shared by all bees)
    private static int ourFlyX;
    private static int ourFlyY;
    private static int objectCount = 0;

    static {
        chooseSpot();
    }

    // chooses the location for future bees to go to
    public static void chooseSpot() {
        RandomGenerator randy = RandomGenerator.getInstance();
        ourFlyX = randy.nextInt(60);
        ourFlyY = randy.nextInt(50);
    }

    // object fields/constructor/methods (replicated in each bee)
    private int myFlyX;
    private int myFlyY;

    ...
```

Advanced Bee code

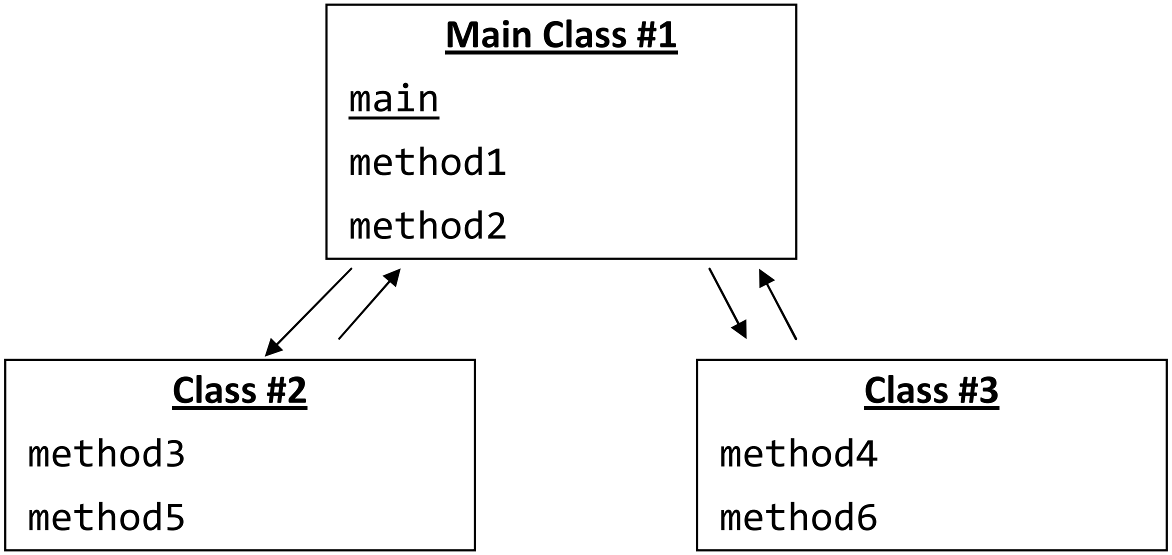
```
    ...

    public Bee() {
        // every 10th one chooses a new spot for future bees
        if (objectCount % 10 == 0) {
            chooseSpot();
        }

        // must remember its spot so they aren't all the same
        myFlyX = ourFlyX;
        myFlyY = ourFlyY;
    }

    public Direction getMove() {
        if (getY() != myFlyY) {
            return Direction.NORTH;
        } else if (getX() != myFlyX) {
            return Direction.EAST;
        } else {
            return Direction.CENTER;
        }
    }
}
```

Overflow (extra) slides

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



Main Class #1

`main`

`method1`

`method2`

Class #2

`method3`

`method5`

Class #3

`method4`

`method6`

Redundant program 1

```
// This program sees whether some interesting numbers are prime.
public class Primes1 extends ConsoleProgram {
    public void run() {
        int[] nums = {1234517, 859501, 53, 142};
        for (int i = 0; i < nums.length; i++) {
            if (isPrime(nums[i])) {
                println(nums[i] + " is prime");
            }
        }
    }

    // Returns the number of factors of the given integer.
    public int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++;   // i is a factor of the number
            }
        }
        return count;
    }

    // Returns true if the given number is prime.
    public boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

Redundant program 2

```
// This program prints all prime numbers up to a maximum.
public class Primes2 extends ConsoleProgram {
    public void run() {
        int max = readLine("Max number? ");
        for (int i = 2; i <= max; i++) {
            if (isPrime(i)) {
                print(i + " ");
        }   }
        println();
    }

    // Returns the number of factors of the given integer.
    public int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++;   // i is a factor of the number
            }
        }
        return count;
    }

    // Returns true if the given number is prime.
    public boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

Class as module

- **module**: A reusable piece of software, stored as a class.
  - Example module classes: Math, Arrays, System

```
// This module contains methods related to factors and prime numbers.
public class Factors {
    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++;   // i is a factor of the number
            }
        }

        return count;
    }

    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

Details about modules

- A module is a partial program, not a complete program.
  - It does not have a `run` method nor does it extend `Program` .
    - You don't run it directly.
  - Modules are meant to be utilized by other *client* classes.

- Syntax:

```
ClassName.methodName(parameters);
```

- Example:

```
int fact = Factors.countFactors(24);
```

Using a module

```
// This program sees whether some interesting numbers are prime.
public class Primes extends ConsoleProgram {
    public void run() {
        int[] nums = {1234517, 859501, 53, 142};
        for (int i = 0; i < nums.length; i++) {
            if (Factors.isPrime(nums[i])) {
                println(nums[i] + " is prime");
            }
        }
    }
}

// This program prints all prime numbers up to a given maximum.
public class Primes2 extends ConsoleProgram {
    public void run() {
        int max = readLine("Max number? ");
        for (int i = 2; i <= max; i++) {
            if (Factors.isPrime(i)) {
                print(i + " ");
        }   }
        println();
    }
}
```

Modules in Java libraries

```
// Java's built in Math class is a module
public class Math {
    public static final double PI = 3.14159265358979323846;

    ...

    public static int abs(int a) {
        if (a >= 0) {
            return a;
        } else {
            return -a;
        }
    }

    public static double toDegrees(double radians) {
        return radians * 180 / PI;
    }
}
```

Summary of Java classes

- A class is used for any of the following in a large program:
  - a *program* : extends `Program`, has a `run` method. Executed directly.
    - example: `Hangman`, `Breakout`, `CritterMain`
    - does not usually declare any static fields (except `final` constants)
  - an *object class* : Defines a new type of objects.
    - example: `BankAccount`, `Date`, `GOval`, `Critter`, `Bee`
    - declares object fields, constructor(s), and methods
    - might declare static fields/methods to help implement objects' behavior
    - should be encapsulated (all fields and static fields `private`)
  - a *module* : Utility code implemented as static methods.
    - example: `Math`, `Factors`