Logistics

Reading from Eck
Ch 5 on Objects/Classes

Goals
- Finish arrays of objects
- Static fields
- Non-static methods

Lab08: Simple object definitions
- Stock object
- Methods in same java file

Project
- Spec up
- Due a week from Wed
Static/Non-static Stuff so far

- The keyword static in Java roughly translates to "belongs to the whole class and all objects"
- So far we have written the following

**static methods**

```java
public class MyClass{
    public static int doSomething(...){
        ...
    }
}
```

- Nothing special about them, invoked with `MyClass.doSomething(..)`
- Must pass in all parameters to the methods

**Non-static fields**

```java
public class Thing{
    int part1;
    double part2;
    String part3;
}
```

- Each Thing has its own `part1, part2, part3`
- 4 Things means 12 pieces of data, 4 ints, 4 doubles, 4 String references
Static Class Fields

- A static field indicates there is only 1 memory location for the entire class, NOT one per object
- Closest thing Java has to a *global variable*
- Seen examples of static fields from some classes
  
  ```java
double pie = Math.PI;
double natbase = Math.E;
PrintStream ps = System.out;
```

- Syntax static establish a static field is simple
  
  ```java
  public class Mixed{
      public static int e;  // static field
      public String f;      // non-static field
  }
  ```

- Every instance of a Mixed has its own f
- **There is only one integer e**, accessible via Mixed.e
Demo of Static vs Non-Static Fields

```java
1 class OnlyStatic{
2   public static int a; // both static
3   public static String b;
4 }
5 class OnlyNon{
6   public int c; // both non-static
7   public String d;
8 }
9 class Mixed{
10  public static int e; // one static
11  public String f; // one non-static
12 }

14 public class StaticFields{
15   public static void main(String args[]){
16     OnlyStatic.a = 5; OnlyStatic.b = "bb";
17     // OnlyNon.c = 4; // ERROR: non-static field
18     // OnlyNon.d = "ddd"; // ERROR: non-static field
19     OnlyNon x = new OnlyNon();
20     x.c = 10; x.d = "dd";
21     OnlyNon y = new OnlyNon();
22     y.c = 15; y.d = "dddd";
23     Mixed.e = 20;
24     // Mixed.f = "ff"; // ERROR: non-static field
25     Mixed z = new Mixed();
26     z.f = "ff";
27     Mixed w = new Mixed();
28     w.f = "ffff";
29   }
30 }
```
Exercise: Recap what we learned about static fields

1. What’s the difference between a static and a non-static field?

2. How many of each kind of field are gotten when calling new?

3. Draw a quick diagram of the following.

```java
public class Thing{
    public int red;
    public double blue;
    public static int green;

    public static void main(String args[]){
        Thing x = new Thing();
        Thing y = new Thing();

        x.red = 5;
        y.blue = 7.0;

        // DRAW HERE

        // which works / doesn’t?
        Thing.green = 9;
        Thing.red   = 10;
    }
}
```
Non-static Methods

- static roughly means *class-level*, as in belonging to the entire class
- Non static roughly means *instance-level*, as in associated with a specific instance/object
- Non-static methods are *ALWAYS* invoked with a specific object/instance
  
  ```java
  String s = "hello";
  String t = "goodbye";

  int len1 = s.length(); // 5
  int len2 = t.length(); // 7
  ```

- During a the execution of a non-static method, the keyword *this* refers to the object on which the method is running
Compare: Static vs Non-static Method Defs/Calls

Static

```java
1 public class Omelet{
2     int eggs;
3     int cheese;
4     double cookedFor;
5     String extras;
6
7     static void cookFor(Omelet om,
8     double time){
9         om.cookedFor += time;
10    }
11    static void addEgg(Omelet om){
12        om.eggs++;
13    }
14 }
15 main(){
16     Omelet standard = new Omelet();
17     int x = 5;
18     Omelet.addEgg(standard);
19     Omelet.cookFor(standard, 2.5);
20 }
```

Non-static

```java
1 public class OOOmelet{
2     int eggs;
3     int cheese;
4     double cookedFor;
5     String extras;
6
7     void cookFor(double time){
8         this.cookedFor += time;
9     }
10     void addEgg(){
11         this.eggs++;
12     }
13 }
14 main(){
15     OOOmelet standard = new OOOmelet();
16     int x = 5;
17     standard.addEgg();
18     standard.cookFor(2.5);
19 }
```

Examine OOOmelet.java to see full implementation
this variable: reference to current object

- Variable `this` is automatically created in non-static methods
- Gets filled in with the value of the object being operated on

```java
standard.addEgg();
coronary.addEgg();
```

during `addEgg()`, this will refer to `Omelet` standard

during `addEgg()`, this will refer to `OOOmelet` coronary

---

**static Omelet.addEgg(standard)**

<table>
<thead>
<tr>
<th>STACK</th>
<th>main()</th>
<th>#1024 standard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1028 x</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>static</td>
<td>addEgg() #1032 om</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**HEAP**

- `Omelet` #2048 eggs
- `cheese` #2052 4
- `cookedFor` #2056 4.5
- `extras` #2064 #4060

---

**NON-static standard.addEgg()**

<table>
<thead>
<tr>
<th>STACK</th>
<th>main()</th>
<th>#1024 standard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1028 x</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>addEgg()</td>
<td>#1032 this #2048</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**HEAP**

- `OOOmelet` #2048 eggs
- `cheese` #2052 4
- `cookedFor` #2056 4.5
- `extras` #2064 #4060

...
Constructors

- Objects usually have necessary fields initialized at creation
- Special method called a **constructor**
- Method name is always identical to class name, return type is omitted
- CK commonly uses `this.field = param;` to initialize fields

```java
public class OOOmelet{
    // Constructor to initialize fields to given values. cookedFor is always initialized to 0.0.
    public OOOmelet(int eggs, int cheese, String extras){
        this.eggs = eggs; // set field eggs to parameter eggs
        this.cheese = cheese; // set field cheese to parameter cheese
        this.extras = extras; // set field extras to parameter extras
        this.cookedFor = 0.0; // always set cookedFor to 0.0
    }
    ...
}

public class OOOmeletMain{
    public static void main(String args[]){
        OOOmelet small = new OOOmelet(2,5,"ham"); // smallish OOOmelet
        OOOmelet large = new OOOmelet(5,8,"bacon"); // largeish OOOmelet
        ...
    }
}
```
Exercise: Draw a Memory Diagram

- Show the OOOMeletMain.java and OOOMelet.java.exercise
- Running the main() method, trace execution
- Draw memory diagrams of what things look like at the numbered locations
- **Note:** May hit some locations more than once
- **Important:** Don’t forget the automatic this variable in non-static methods
Easy Printing: `toString()` method

- Most complex objects provide a `toString()` method to produce nice output
- Compare
  
  ```java
  OOOmelet small = new OOOmelet(2,5,"ham");
  System.out.println(small);
  ```
- NO `toString()` method:
  ```java
  OOOmelet@2a139a55
  ```
- WITH `toString()` method:
  ```java
  3 eggs, 5 oz cheese, cooked for 1.5 mins, extras: ham
  ```

```java
public class OOOmelet{
    private int eggs; private int cheese;
    private double cookedFor; private String extras;

    // Create a pretty string version of the OOOmelet.
    public String toString(){
        return String.format("%d eggs, %d oz cheese, cooked for %.1f mins, extras: %s",
             this.eggs, this.cheese, this.cookedFor, this.extras);
    }
}
```
String.format() for toString()

- Extremely useful method static method of String class
- Works like printf() but instead of printing to the screen, creates a string and returns it
- Example:

  ```java
  String s =
      String.format("apples: %d weight: %.1f kind: %s", 5, 1.27, "Honeycrisp");
  System.out.println(s);
  // apples: 5 weight: 1.3 kind: Honeycrisp
  ```

- Often used in toString() methods to format info on object for display
- Also used in testing files to produce error messages containing data for debugging
Exercise: Dog Constructor and toString()

- Define constructor for Dog class to the right
- Infer arguments/defaults from use in main()
- Define toString() method
- Infer format from use in main()
- Make use of String.format()

```java
public class Dog{
    public String name;
    public int age;
    public boolean hasBone;

    // CONSTRUCTOR

    // toString()

    public static void main(String args[]){
        Dog s = new Dog("Stout",3);
        Dog r = new Dog("Rufus",1);
        r.hasBone = true;
        System.out.println(s.toString());
        System.out.println(r.toString());
    }
}
```

> javac Dog.java
> java Dog
Name: Stout  Age: 3  Bone? false
Name: Rufus Age: 1  Bone? true
Access Modifiers

Access Levels for Fields/Methods by other stuff

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- Mostly concerned with public and private, read about others on your own
- Most projects will specify required public methods, maybe public fields
- Most of the time you are free to create additional private methods and fields to accomplish your task

Official docs on access modifiers

http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
Common Java convention is to make all fields private. 
Private fields are only visible within on .java file accessor and mutator methods provided to work with object data.

Accessor often referred to as "getter" as in getEggs().

Mutator sometimes called a "setters" but often have other names, intended to change object data.

Important: changing object data preserves any invariants of the class: related fields

```java
public class OOOmelet{
    public int eggs;
    public int cheese;

    // Retrieve number of eggs
    public int getEggs(){
        return this.eggs;
    }

    // Add an egg to the omelet
    // if cooking hasn’t begun
    public void addEgg(){
        if(this.cookedFor > 0){
            System.out.println("Yuck");
        } else{
            this.eggs++;
        }
    }
}
```
Invariants in Classes

OOOmelets (In-class)
▶ Once cooking starts, cannot add eggs
▶ Can only add time to cooking, not subtract
▶ Extra ingredients must be specified up front

Linear Equations (Lab09)
▶ \( y = m \cdot x + b \)
▶ Left and right sides of equation are always equal
▶ Changing \( x \) updates \( y \), vice versa

Portfolio (Proj4)
▶ Adding a stock increases the stockCount
▶ Buying stocks deducts from cash
▶ Selling stocks adds to cash
▶ Cannot withdraw() more cash than is available
▶ Cannot sell more shares than available
Why Getters vs. Public Fields

- Simple objects can probably have public fields, direct access
  - Don’t do this as you’ll be penalized on manual inspection
- Slightly more complex objects like OOOmelet might get away with public fields but would allow ..
  - "Uncooking" of omelets: o.cookedFor = 0.0;
  - Add eggs after being cooked
  - Using private fields prevents this
- Complex objects like Printstream from System.out must preserve invariants: different parts must agree with each other.
  - Changing one field might screw up another one
  - Deny direct access via private fields
  - Mutation methods like println() keep all fields synchronized

Abstraction Up and Down

Break a problem into smaller parts. Define public methods between those parts. Think about internal details for one part at a time. Recurse for subparts as needed.
private Fields / public methods

OOOmelet.java

```java
public class OOOmelet{
    private int eggs;
    private int cheese;
    private double cookedFor;
    private String extras;

    public double getEggs(){
        return this.eggs;
    }
    public double getCookTime(){
        return this.cookedFor;
    }
    public void addEgg(){
        ...
    }
    ...
}
```

UseOOOmelet.java

```java
public class UseOOOmelet{
    public static void main(String args[]){
        OOOmelet om =
            new OOOmelet(2,4,"ham");

        // CORRECT: public methods
        int eggs = om.getEggs();
        om.addEgg();

        // INCORRECT: No such symbol
        om.eggs = 5;  // compile error

        // CORRECT: public method
        om.cookFor(1.0);

        // INCORRECT: No such symbol
        om.cookedFor=0.0; // compile error
    }
}
```

Must access fields through public methods
private Fields Visible only in One Java File

- private means visible in current Java File only
- Within `OOOmelet.java`, the name `eggs` is visible for all `OOOmelet`s
- Even if that name is associated with "some other" `OOOmelet`
- See `moreEggs()` method: accessing `that.eggs` despite it being a private variable

```java
// OOOmelet.java
public class OOOmelet{
    private int eggs;
    // Return true if this omelet has more eggs than the parameter omelet
    public boolean moreEggsThan(OOOmelet that){
        if(this.eggs > that.eggs){ // OK!!!
            return true;
        }
        else{
            return false;
        }
    }
}

// OOOmeletMain.java
public class OOOmeletMain{
    public static void main(String args[]){
        OOOmelet small = new OOOmelet(2,5,"ham");
        OOOmelet large = new OOOmelet(5,8,"bacon");
        boolean moreEggs = small.moreEggsThan(large);
    }
}
```
Name Binding Resolution Mechanics

- Java follows rules to determine where names are defined: name binding
- Resolution matters for bare names: no class/object association

```java
om.eggs = 5; // specific object’s field
this.cookedFor = 5; // specific object’s field
int c = om.getCalories(); // specific object’s method
this.addEgg(); // specific object’s method
Omelet.egg_cals = 123; // specific class (static)
cookedFor = 1.23; // BARE NAME for field
addEgg(); // BARE NAME for method
```

- To determine where name var binds look at
  1. Local variables
  2. Parameters to method
  3. Fields of class
  4. Potentially outside class (won’t do this in CS 1103)
Exercise: Binding Resolution

- NUMBERS declare a name
- LETTERS are bare name references
- Match LETTERS to NUMBERS to match bare name to where it is defined

To determine where name var binds look at

1. Local variables
2. Parameters to method
3. Fields of class

```java
public class OOOmelet{
    private int eggs;       // 1
    private int cheese;     // 2
    private double cookedFor; // 3
    private String extras;  // 4

    public int getEggs(){    // 5
        return eggs;        // A
    }

    public void cookFor(double time){ // 6
        double cookedFor =       // 6
            this.cookedFor;     // B
        cookedFor += time;      // C
    }

    public void addCheese(int cheese){ // 7
        cheese += cheese;      // D and E
    }

    public boolean foodPoisoningIminent(){ // 8
        return cookedFor < (1.0 * getEggs()); // F and G
    }
}
```
## Answers: Binding Resolution

<table>
<thead>
<tr>
<th>Let</th>
<th>Num</th>
<th>Note</th>
<th>Field/Param/Local/Field</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>field eggs</td>
<td>private int</td>
<td>public class O00melet{ public int getEggs(){ return eggs; }</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>field cookedFor</td>
<td>private int</td>
<td>public void cookFor(double time){ double cookedFor = this.cookedFor; cookedFor += time; }</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>local cookedFor</td>
<td>private double</td>
<td>public void addCheese(int cheese){ cheese += cheese; }</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>param cheese</td>
<td>private String extras</td>
<td>public boolean foodPoisoningIminent(){ return cookedFor &lt; (1.0 * getEggs()); }</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>param cheese</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>field cookedFor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>this.getEggs()</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Exercise: Gotchya’s with Constructor Name Binding

- Common to initialize fields in constructors
- Determine what’s wrong with these constructors
- Give a correct constructor

```java
public class OOOmelet{
    public int eggs;
    public int cheese;
    public double cookedFor;
    public String extras;

    // BAD CONSTRUCTOR 1
    public OOOmelet(int eggs,
                     int cheese,
                     String extras)
    {
        eggs = eggs;
        cheese = cheese;
        extras = extras;
        cookedFor = 0.0;
    }
}

public class OOOmelet{
    public int eggs;
    public int cheese;
    public double cookedFor;
    public String extras;

    // BAD CONSTRUCTOR 2
    public OOOmelet(int eg,
                     int ch,
                     String ex)
    {
        int eggs = eg;
        int cheese = ch;
        String extras = ex;
        double cookedFor = 0.0;
    }
}
```
Answer: Gotchya’s with Constructor Name Binding

- The names of parameters like eggs or local variable int eggs can *shadow* fields
- Fields never get modified as shadows receive assignments
- Use `this.name = name;` or change names of parameters

```java
public class OOOmelet{
    public int eggs;
    public int cheese;
    public double cookedFor;
    public String extras;

    // CORRECT CONSTRUCTOR 1
    // Use this.field to specify
    // field initialization
    public OOOmelet(int eggs,
                     int cheese,
                     String extras)
    {
        this.eggs = eggs;
        this.cheese = cheese;
        this.extras = extras;
        this.cookedFor = 0.0;
    }
}
```
Multiple Methods: Overloading

- In Java, several methods can share the same name SO LONG as each has a distinct a number and/or type of arguments
- Called overloading a method

```java
class OOMelet {
    public OOMelet(int eggs, int cheese, String extras) {
        this.eggs = eggs;
        this.cheese = cheese;
        this.extras = extras;
        this.cookedFor = 0.0;
    }
    public OOMelet(int eggs, int cheese) {
        this.eggs = eggs;
        this.cheese = cheese;
        this.extras = "";
        this.cookedFor = 0.0;
    }
    public void addEgg() {
        if (this.cookedFor > 0) {
            System.out.println("Yuck");
        } else {
            this.eggs++;
        }
    }
    public void addEgg(int nEggs) {
        for (int i = 0; i < nEggs; i++) {
            addEgg();
        }
    }
    public static void main(String args[]) {
        OOMelet omA = new OOMelet(3, 2, "ham");
        OOMelet omB = new OOMelet(4, 6);
        omA.addEgg(2);
        omB.addEgg();
    }
}
```
1. Describe the difference between a static field and a non-static field. How many of each exit when a class is used?
2. The class Foo has a static method called double bar(int x, String s). Describe how to invoke/call this method.
3. What is a constructor? How are they named? Give an example of how they are called.
4. The class Flurbo has a non-static method named int schmeckle(double z). Describe how to invoke/call it.
5. In what context can the keyword this be used? Where can it not be used?
6. What does the keyword this refer to? Can it ever be null?
7. What order does the Java compiler search for bindings of bare variable names to variable declarations?
8. Why would one choose to make fields of a class private?
9. What are accessor methods? What are mutator methods?