CS 211: Defining Classes

Chris Kauffman

Week 3-2

Logistics

P2: Instant Runoff Voting

- Can anyone explain?
- Class decomposition

Topics Today

- Strategies for `VotingMachine` methods
- Creating classes/objects (project)

Reading

- Building Java Programs Ch 8
- Lab Manual Ch 4 and 5

PracticeIt! BJP 3rd Ed Exercises

- Ch 8 Exercise 18
- Ch 8 Exercise 20
- Ch 8 Exercise 21
- Ch 8 Exercise 22

Aggregate Data

Define	Now there's a type `bleh`, it looks like blah
Declare	Here is a variable, its type is `bleh`
Assign	Element foo of variable bar gets value blip
Access	Retrieve element foo of variable bar

Arrays

Create Homogeneous Aggregate Data

- Each constituent element is the same type
- Access via number index: `a[5] = something;`

Classes

Define Heterogeneous Aggregate Data

- Constituent elements can be of different types
- Access via symbolic field name

```
a.field1 = 1;
a.Xfiled = "init!";
```

Basic Objects are Just Data

Omelets in `SOmelet.java`, no `static` fields

```
public class SOmelet{
  public int eggs;
  public int ozCheese;
  public String extraIngredients;
  public double totalCookMinutes;
}

main(){
  SOmelet o = new SOmelet();
  o.eggs = 3;
  o.ozCheese = 4;
  o.extraIngredients = "";
  System.out.println("Cooked "+o.totalCookMinutes);
}
```

Exercise: One Class, Many Objects

Draw a Memory Diagram for the `main()` method below at the location indicated

```
main(){
  SOmelet small = new SOmelet();
  small.eggs = 2;
  small.ozCheese = 3;

  SOmelet big = new SOmelet();
  big.eggs = 4;
  big.ozCheese = 6;

  SOmelet shallow = small;

  SOmelet [] oa = new SOmelet[5];
  for(int i=0; i<oa.length; i++){
    oa[i] = new SOmelet();
    oa[i].eggs = i; oa[i].ozCheese = 2*i;
  }
  // Draw memory diagram HERE
}
```

```
public class SOmelet{
  public int eggs;
  public int ozCheese;
  public String extraIngredients;
  public double totalCookMinutes;
}
```

Typically Want to *do* stuff with data

- `static` Methods defined in `SOmeletMethods.java`
- Used in `UseSOmelet.java` (excerpt below)

```
// Create an omelet
SOmelet standard = SOmeletMethods.constructSOmelet();
// Calculate calories
calories= SOmeletMethods.getBaseCalories(standard);
// Cook an omelet
SOmeletMethods.cookFor(standard, 4.0);
// Cooked long enough?
safe = !SOmeletMethods.foodPoisoningIminent(standard);
```

Notice always invoking static method through `SOmeletMethods` class (irritation)

Defining Static Methods on Objects

Take a reference to the object and do something with it; from `SOmeletMethods.java`

```
// Cook an omelet for the given amount of time
public static void cookFor(SOmelet thisOmelet,
                           double cookMinutes){
  thisOmelet.totalCookMinutes += cookMinutes;
}

// Determine if consumption of the given omelet is risky
public static
boolean foodPoisoningIminent(SOmelet thisOmelet){
  return
    thisOmelet.totalCookMinutes < 1.0 * thisOmelet.eggs;
}
```

Notice reference `thisOmelet` is always required (irritation)

Remember: `SOmelet` is unconventional

`SOmelet.java` and `SOmeletMethods.java` are weird

- Don't follow java convention
- Requires explicit reference `thisOmelet` in all methods
- Precludes *dynamic dispatch* (next week)

However

Static method approach clearly separates

- Data versus Functions acting on data

Easier to build understanding from there because..

Standard Java

Lets mix data and functions together and season with `this`

The "Normal" Way

- See `OOOmelet.java`
- No `static` methods or fields (except constants)
- Equivalent in most ways to `SOmelete.java` + `SOmeletMethods.java`

```
public class OOOmelet{
  // No static fields
  public int eggs;
  public int ozCheese;
  public String extraIngredients;
  public double totalCookMinutes;

  // Constructors
  public OOOmelet(int eggs, int ozCheese){ ... }
  public OOOmelet(){...}

  // No static methods
  public void addIngredient(String ingredient){...}
  public void cookFor(double cookMinutes){...}
  ...
}
```

Methods

Discuss `this`: hidden parameter to method invocation

Standard: `OOOmelet`

```
public void
cookFor(double cookMinutes){

  this.totalCookMinutes +=
       cookMinutes;
}

public boolean
isBurned(){
  return
    this.totalCookMinutes
       > 2.0 * this.eggs;
}

main(){
  OOOmelet oo =
    new OOOmelet();
  oo.cookFor(4.0);
}
```

Static: `SOmelet`

```
public static void
cookFor(SOmelet thisOmelet,
        double cookMinutes){
  thisOmelet.totalCookMinutes +=
      cookMinutes;
}

public static boolean
isBurned(SOmelet thisOmelet){
  return
    thisOmelet.totalCookMinutes
        > 2.0 * thisOmelet.eggs;
}

main(){
  SOmelet so =
    SOmeletMethods.constructSOmelet();
  SOmeletMethods.cookFor(so, 4.0);
}
```

Constructors

Weird methods that build an object but don't return it. Compare:

Standard

```
public class OOOmelet{

  public OOOmelet(int eggs,
                  int ozCheese){
    // No allocation
    this.eggs = eggs;
    this.ozCheese = ozCheese;
    this.extraIngredients =
      new String("");
    this.totalCookMinutes = 0.0;
    // No return
  }
```

Static

```
public class SOmeletMethods {
  public static SOmelet
  constructSOmelet(int eggs,
                   int ozCheese){
    SOmelet thisOmelet = new SOmelet();
    thisOmelet.eggs = eggs;
    thisOmelet.ozCheese = ozCheese;
    thisOmelet.extraIngredients =
      new String("");
    thisOmelet.totalCookMinutes = 0.0;
    return thisOmelet;
  }
```

Error Checking

Modify the constructor for `OOOmelet` to throw a `RuntimeException` if eggs or ozCheese is a negative number.

Exercise: To String, or Not To String.

That is not a question. 'Tis almost always better to endure writing a `toString()` method that prints a pretty version of the object.

Write `toString()` for `OOOmelet`

```
Welcome to DrJava.
> OOOmelet standard = new OOOmelet();
> System.out.println(standard.toString());
3 egg 4 oz cheese omelet, cooked for 0.0 minutes

> standard.cookFor(2.3)
> System.out.println(standard)
3 egg 4 oz cheese omelet, cooked for 2.3 minutes

> OOOmelet coronary = new OOOmelet(5,12);
> coronary.addIngredient("bacon");
> coronary.cookFor(4.6785)
> System.out.println(coronary)
5 egg 12 oz cheese omelet, cooked for 4.7 minutes
```

Don't touch that

Java enables *Access Control* for insides of classes

- Visibility of fields and methods to other stuff
- `public`, `protected`, none, `private`
- Put them in front of methods and fields
- Play with these in `OOOmelet`

Access Modifiers

Access Levels for Fields/Methods by other stuff

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- 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

Getter, Setter, Class Invariant

Common Java convention is to make all fields `private` and provide getter and setter methods to change them

Getter/Setter for Eggs

```
public class OOOmelet{
  public int eggs;
  public int ozCheese;
  ...
  public double getEggs(){
    return this.eggs;
  }
  public void setEggs(int e){
    if(this.totalCookMinutes > 0){
      throw new
       RuntimeException("yuck");
    }
    this.eggs = e;
  }
  ...
}
```

Questions

- Does it make sense to change the number of eggs after an omelet is cooked?
- Does it make sense to add `setCookMinutes(double)` to arbitrarily change `totalCookMinutes`?
- Why use getters/setters?

Typically Fields are `private`

POmelet: Private fields

Provide getters to report fields like eggs and cook time

```
public class POmelet{
  private int eggs;
  private double
    totalCookMinutes;
  ...
  public double getEggs(){
    return this.eggs;
  }
  public double
  getTotalCookMinutes(){
    return this.eggs;
  }
  ...
}
```

Use of Getters v. Private Fields

```
POmelet x=new POmelet(3,4);
// Correct
int eggs = x.getEggs();
// Error
x.eggs = 5; // No such symbol

x.cookFor(2.5);
// Correct
if(x.getTotalCookMinutes() > 0.0)
  ...
}
// Error
if(x.totalCookMinutes > 0.0){
  ...
}
```

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.totalCookMinutes = 0.0;`
  - Add eggs after being cooked
  - `POmelet` with private fields prevents this
- Complex objects like `Scanner` 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 `next()` and `setX(v)` 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.

Scope and this

Name resolution rules don't always require use of keyword this

Using this

```
public class POmelet{
  private int eggs;
  private double
    totalCookMinutes;

  public int getEggs(){
    return this.eggs;
  }
  public void
  cookFor(double cookMinutes){
    this.totalCookMinutes
      += cookMinutes;
  }
}
```

Without this

```
public class POmelet{
  private int eggs;
  private double
    totalCookMinutes;

  public int getEggs(){
    return eggs;
  }
  public void
  cookFor(double cookMinutes){
    totalCookMinutes
      += cookMinutes;
  }
}
```

Exercise static Methods the Best Omelet

- static is stand-alone, independent shared by all objects
- Write code for bestOmelet(arr)

```
public class OOOmelet{
  // Return the "best" omelet in an array; better omelets have higher
  // calorie counts as reported by the o.getBaseCalories() method.  If
  // the array is empty, return null.
  public static OOOmelet bestOmelet(OOOmelet [] arr){...}
}
```

```
Welcome to DrJava.
> OOOmelet arr[] = {new OOOmelet(3,4), new OOOmelet(2,10),
                    new OOOmelet(8,2), new OOOmelet(3,3)};
> OOOmelet best = OOOmelet.bestOmelet(arr);
> best
2 eggs 10 oz cheese omelet, cooked for 0.0 minutes
> best.getBaseCalories()
1328
> OOOmelet empty[] = {};
> OOOmelet other = OOOmelet.bestOmelet(empty);
> other
null
```

Recall: Equality and ==

```
main(){
  int li1=3, li2=3;
  boolean eq1 = (li1 == li2);   // T/F??

  Integer bi1 = new Integer(4);
  Integer bi2 = new Integer(4);
  boolean eq2 = (bi1 == bi2);   // T/F??

  OOOmelet om1 = new OOOmelet(3,4);
  OOOmelet om2 = new OOOmelet(3,4);
  boolean eq3 = (om1 == om2);   // T/F??
}
```

- Draw a memory diagram for the above `main` method
- Determine the values of `eq1,eq2,eq3`

`x.equals(y)` methods

- Provide a deep equality check of `x` to `y`
- What's *deep* vs *shallow*?
- All objects have one... why?
- Most objects define their own
- Technical note: difference between

```
public boolean equals(Object other)
public boolean equals(Omelet other)
```

Exercise: Equality of Omelets

```
public class POmelet{
  private int eggs;                    // How many eggs in the omelet
  private int ozCheese;                // How many ounces of chees
  private String extraIngredients; // Extra ingredients added
  private double totalCookMinutes; // How long the omelet has cooked

  // Define me
  public boolean equals(POmelet other){...}
```

- `POmelet x` and `POmelet y`
- `x.equals(y)` is true when
  1. `x` and `y` have equal `eggs` (`int`)
  2. and equal `ozCheese` (`int`)
  3. and equal `extraIngredients` (`String`)
- 1 and 2 are easy
- 3 is slightly trickier
- Write the equality method
  - Remember that `x` will be `this`, `y` will be `other`

Note: `equals(..)` is a funky method

- All classes automatically have an `equals(Object o)` method due to inheritance
- Will discuss next week in detail, but all proper `equals(..)` methods following the pattern mentioned in the spec

```
public class POmelet{
  private int eggs, ozCheese;
  private String extraIngredients;
  private double totalCookMinutes;
  public boolean(Object other){ // Compare to arbitrary object
    if ( ! ( other instanceof POmelet) ) {
      return false;           // Not anothe omelet, can't b equal
    }
    POmelet that = (POmelet) other; // Caste other to omelet
    return                  // check relevant fields equal
      this.eggs == that.eggs &&
      this.ozCheese == that.ozCheese &&
      this.extraIngredients.equals(that.extraIngredients);
      // && this.totalCookMinutes == that.totalCookMinutes;
  }
}
```