Lecture 26:
Writing Classes, Part IV

Sierra College
CSCI-12
Spring 2015
Mon 05/04/15

Announcements

- **General**
- **Schedule**
  - 4 more lectures, then finals week (final exam Weds 5/20)
    - Review session Monday, study guide, in-class exam (like last time)
  - This program, then one LAST one (due after final)
- **Past due assignments**
  - PRGM22: Menu For Demo (accepted thru Weds 5/6 @ 11pm)
- **Current assignments**
  - PRGM25: Dam (due Thurs 5/14 @ 11pm)
    - Create a new class which models a water storage dam
    - Use the systematic procedure we have gone thru in lectures, refer back to prior lecture notes
    - We will go over this program in more detail today

Lecture Topics

- **Last time**:
  - Continued creating ***Person*** class step-by-step
    - Adding main() test code to the class
    - Accessors and mutators
    - Various kinds of equality, and *equals*()
    - Augmenting our ***Person*** class
- **Today**
  - Finishing up our ***Person*** class
    - Data safety
    - Safe transfer of object data
    - Utility methods
    - Testing considerations

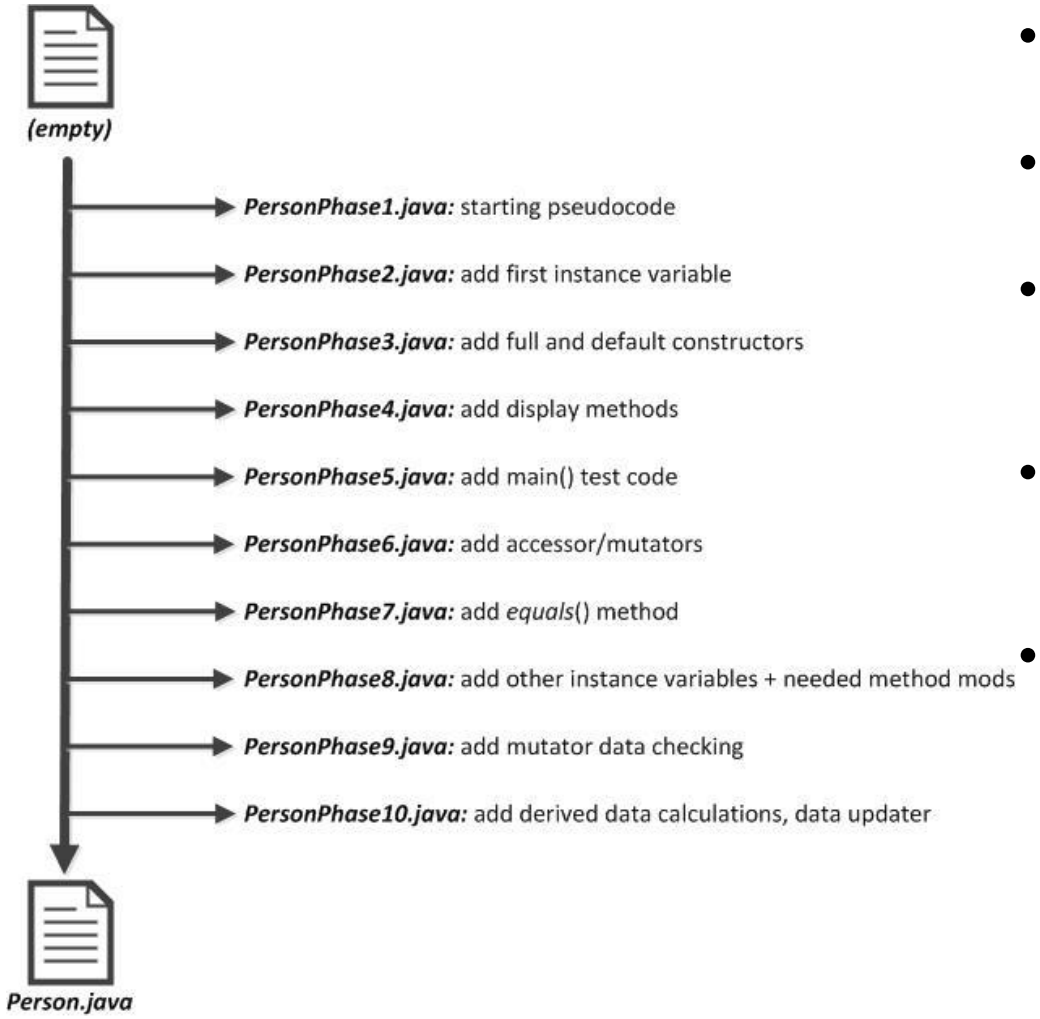
For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Program**
  - Start building up your ***Dam*** class
    - Review the Writing Classes lecture notes
    - Review the assignment and the ***Dam*** API carefully (questions??)
    - Start with an empty Java class, and create the class pseudocode
    - Build a little, test a little (add test code to *main*() as you go)
    - Implement the entire starter class for ONE instance variable

Intended ***Person*** Class API

	Person Class API
	Person Class Instance Variables
String	**firstName** - *first name*
String	**lastName** - *family name or surname*
SimpleDate	**birthdate** - *date of birth*
char	**gender** - *M or F*
double	**height** - *height in [m]*
double	**weight** - *weight in [kg]*
	Person Class Constructors
	Person()
	Creates a Person object with default initial values
	Person(String **firstName**, String **lastName**, SimpleDate **birthdate**, char **gender**, double **height**, double **weight**)
	Creates a Person object with all values specified
	Person(String **firstName**, String **lastName**)
	Creates a Person object with specified first/last names, and all other instance variable with default values
	Person Class Methods
String	**toString**()
	Returns all instance variables in a comma-separated String format
void	**print**()
	Displays all instance variables in a labeled output format
String	**getFirstName**()
	Returns the value of firstName
void	**setFirstName**(String **firstName**)
	Sets the value of firstName
void	**setFirstName**(boolean **inputMode**)
	Sets the value of firstName, using user input prompts
String	**getLastName**()
	Returns the value of lastName
void	**setLastName**(String **lastName**)
	Sets the value of lastName
void	**setLastName**(boolean **inputMode**)
	Sets the value of lastName, using user input prompts
SimpleDate	**getBirthdate**()
	Returns the value of birthdate
void	**setBirthdate**(SimpleDate **birthdate**)
	Sets the value of birthdate
void	**setBirthdate**(boolean **inputMode**)
	Sets the value of birthdate, using user input prompts
char	**getGender**()
	Returns the value of gender
void	**setGender**(char **gender**)
	Sets the value of gender
void	**setGender**(boolean **inputMode**)
	Sets the value of gender, using user input prompts
double	**getHeight**()
	Returns the value of height
void	**setHeight**(char **height**)
	Sets the value of height
void	**setHeight**(boolean **inputMode**)
	Sets the value of height, using user input prompts
double	**getWeight**()
	Returns the value of weight
void	**setWeight**(char **weight**)
	Sets the value of weight
void	**setWeight**(boolean **inputMode**)
	Sets the value of weight, using user input prompts
void	**update**(boolean **inputMode**)
	Sets all instance variables, using user input prompts
boolean	**equals**(Object **obj**)
	Compares this Person object to another Person object
int	**getAge**()
	Returns the age of the Person
int	**getIQ**()
	Returns the IQ of the Person.
	IQ is assumed to be some TBD function of the person's age.
double	**getBMI**()
	Returns the BMI (Body Mass Index) of the Person.
	BMI is a function of the person's height and weight.
void	**eat**(double **kcal**)
	Allows person to ingest food calories.
	Assume that weight increases according to 0.1 kg per 1000 kcal.
void	**exercise**(double **hrs**)
	Allows to person to exercise for a specified period of time.
	Assume that weight reduces according to 0.1 kg per hr of exercise.

Person Class Development Progression



- This is the development progression we will follow
- We will build up the ***Person*** class from scratch in stages
- Each code snapshot "stepping stone" represents added capability
- You will then use this process to create your OWN class on the next program
- **Incremental development like this may feel slower, but over the long haul, it saves time and prevents frustration**
  - *"Get something simple to work first, then keep adding to it"*

Mutator Data Checking

- Mutators can be used to perform **data checking** on input values
  - Enforces **data integrity** (makes sure our data is always valid)
  - If value is improper or out of bounds, notify the user and leave the value unchanged (or whatever else is reasonable)
- Examples
  - Gender: restrict M/F, upper case
  - Height: restrict to [0.0-3.0]
  - Weight: restrict to [0.0-400.0]

```
25      // constants ----------------------
26      private final double TOL = 0.0001;   // equals FP tolerance
27      private final double MIN_HT = 0.0;   // min allowable height
28      private final double MAX_HT = 3.0;   // min allowable height
29      private final double MIN_WT = 0.0;   // min allowable weight
30      private final double MAX_WT = 400.0; // max allowable weight
```

```
186     // gender mutator with data checking
187     public void setGender(char gender) {
188         // convert inputs to all CAPS exclusively
189         char genderUpper = Character.toUpperCase(gender);
190
191         if ((genderUpper == 'M') || (genderUpper == 'F')) {
192             this.gender = genderUpper;
193         }
194         else {
195             System.out.println("ERROR: unrecognized value for gender, unchanged");
196         }
197     }
```

```
210     // height mutator with data checking
211     public void setHeight(double height) {
212         if ((height >= MIN_HT) && (height <= MAX_HT)) {
213             this.height = height;
214         }
215         else {
216             System.out.println("ERROR: invalid height value, unchanged");
217         }
218     }
```

```
231     // weight mutator with data checking
232     public void setWeight(double weight) {
233         if ((weight >= MIN_WT) && (weight <= MAX_WT)) {
234             this.weight = weight;
235         }
236         else {
237             System.out.println("ERROR: invalid weight value, unchanged");
238         }
239     }
```

This snapshot of the *Person* class is saved as ***PersonPhase9.java*** in **Example Source Code**

Mutator Data Checking in Constructors

- To take full advantage of mutator data checking, use mutators exclusively within full and alternate constructors to initialize instance variables
- Also, make sure these constructors have defaults in place, by using *this*()
- Otherwise, bad data can be used to create objects, that otherwise should not be allowed

```
39    // constructors ------------------
40
41    // default constructor
42    public Person() {
43        firstName = "unknown";
44        lastName = "unknown";
45        birthdate = new SimpleDate();
46        gender = 'M';
47        height = 0.0;
48        weight = 0.0;
49    }
50
51    // full constructor
52    public Person(String firstName, String lastName,
53                  SimpleDate birthdate, char gender,
54                  double height, double weight) {
55
56        // pulls in ALL defaults, must be first statement
57        this();
58
59        // now set all specified values
60        setFirstName(firstName);
61        setLastName(lastName);
62        setBirthdate(birthdate);
63        setGender(gender);
64        setHeight(height);
65        setWeight(weight);
66    }
67
68    // alternate constructor: names only
69    public Person(String firstName, String lastName) {
70
71        // pulls in ALL defaults, must be first statement
72        this();
73
74        // now overwrite only the ones with changes
75        setFirstName(firstName);
76        setLastName(lastName);
77    }
```

This snapshot of the *Person* class is saved as ***PersonPhase9.java*** in **Example Source Code**

Testing Mutator Data Checking

- To test the added mutator data checks, try setting instance variables to some invalid values
- Also, try creating new objects having invalid values, using the constructor(s)

```
337         // test mutator data checks
338         p2.setGender('X');
339         p2.setHeight(5.0);
340         p2.setWeight(1000.0);
341         Person p4 = new Person("John", "Jones",
342                                new SimpleDate(1, 1, 2001), 'X',
343                                5.0, 1000.0);
344         p4.print("Checking results of bad constructor data");
345         System.out.println();
```

```
ERROR: unrecognized value for gender, unchanged
ERROR: invalid height value, unchanged
ERROR: invalid weight value, unchanged
ERROR: unrecognized value for gender, unchanged
ERROR: invalid height value, unchanged
ERROR: invalid weight value, unchanged
======================
Checking results of bad constructor data
======================
firstName:      John
lastName:       Jones
birthdate:      1/1/2001
gender:         M
height:         0.00
weight:         0.00
```

This snapshot of the *Person* class is saved as ***PersonPhase9.java*** in **Example Source Code**

Problem: Passing Objects Into Methods

- Consider this example:
  - We pass an object (a *SimpleDate*) to our *Person* mutator
  - Later changes to the outside client object corrupt the internal *Person* object's data
  - This violates encapsulation!
  - How do we avoid this?
  - And why does the *String* not have the same issue?

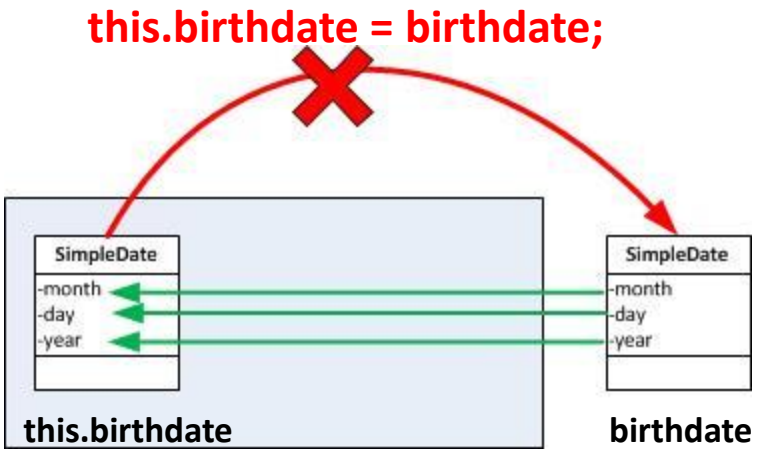
```
14 public class PersonDataCorruption {
15
16     public static void main(String [] args) {
17
18         Person p = new Person();
19         SimpleDate date = new SimpleDate(11, 17, 2014);
20         String name = new String("Fred");
21
22         // update the Person with the given date and name
23         p.setBirthdate(date);
24         p.setFirstName(name);
25
26         // Person has been updated
27         p.print("before");
28
29         // Now the outside client changes the SimpleDate and String
30         date.setDate(12, 25, 2014);
31         name = new String("Anne");
32
33         // Uh-oh! The object's private data has changed w/o permission
34         // BUT, the String is OK: strings are "immutable" in Java
35         p.print("after");
36
37     } // end main
38
39 } // end class
```

```
======================
before
======================
firstName:      Fred
lastName:       unknown
birthdate:      11/17/2014
gender:         M
height:         0.00
weight:         0.00
======================
after
======================
firstName:      Fred
lastName:       unknown
birthdate:      12/25/2014
gender:         M
height:         0.00
weight:         0.00
```

See ***PersonDataCorruption.java*** in **Example Source Code**

The Problem and the Solution

- The problem arises because when we pass an object to a mutator, we are passing an object reference to the outside world
- If the outside object changes, so too does the internal instance variable object
- We need to "transfer" the data from the passed object reference into a safe, local instance variable




```
(this.birthdate).setMonth( birthdate.getMonth() );
(this.birthdate).setDay( birthdate.getDay() );
(this.birthdate).setYear( birthdate.getYear() );
```

Safely Passing Objects to Class Methods

- To safely pass outside objects via mutator into another object:
  - Create a new instance variable object
  - Transfer data over, field by field, from the outside object into the instance variable one
  - Use accesors and mutators
  - Strings are not an issue because they are **immutable**; from the *String* Java API:

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared.

```
162     // birthdate mutator
163     public void setBirthdate(SimpleDate birthdate) {
164         // don't simply do this: outside changes affect internal data
165         //this.birthdate = birthdate;
166
167         // instead, transfer data over field by field
168         // now internal birthdate is truly isolated from outside
169         (this.birthdate).setMonth(birthdate.getMonth());
170         (this.birthdate).setDay(birthdate.getDay());
171         (this.birthdate).setYear(birthdate.getYear());
```

```
======================
before
======================
firstName:      Fred
lastName:       unknown
birthdate:      11/17/2014
gender:         M
height:         0.00
weight:         0.00
======================
after
======================
firstName:      Fred
lastName:       unknown
birthdate:      11/17/2014
gender:         M
height:         0.00
weight:         0.00
```

This snapshot of the *Person* class is saved as ***PersonPhase9.java*** in **Example Source Code**

Safely Returning Objects from Class Methods

- The opposite operation (returning an instance variable object reference via an accessor) is also problematic
- A similar solution applies:
  - Create an internal copy
  - Transfer data into it, using accessors and mutators
  - Return the object reference of the copy

```
149     // birthdate accessor
150     public SimpleDate getBirthdate() {
151         // don't simply do this: don't return a reference to the internal data
152         //return birthdate;
153
154         // instead, make a local "scratch" copy and return the reference to that
155         SimpleDate temp = new SimpleDate();
156         temp.setMonth(this.birthdate.getMonth());
157         temp.setDay(this.birthdate.getDay());
158         temp.setYear(this.birthdate.getYear());
159         return temp;
160     }
```

This snapshot of the *Person* class is saved as ***PersonPhase9.java*** in **Example Source Code**

Utility Methods

- Classes can (usually do) have methods beyond the ones so far discussed:
  - Constructors, accessors/mutators, print(), toString(), equals()
- Other **utility methods** provide the "business logic" or other services for the class
  - Public methods provide services available to outside client code
    - ***They make a class "do whatever that class is supposed to do"***
  - Private methods are not intended for outside use, only for the internal use of the class
- Class API documentation
  - Public methods are part of an API
  - Private methods are typically NOT part of an API (internal use only)
  - But, private methods would be specified on a UML Class Diagram for designer
- We will consider two types of utility methods for *Person:*
  - **Derived data accessors**
  - **Other utility methods (actions) specific to a *Person***

Base Versus Derived Data

- **"Base" data** are the true "instance variables"
  - Can be directly obtained or measured, are directly set as instance variables
  - Examples:
    - *Person*: height, weight, birthdate
    - *Auto:* miles (odometer), gallons (pump reading)
- **Derived data** is calculated from other base data
  - Examples:
    - *Person*: **age** ← birthdate; **IQ** ← age ← birthdate; **BMI** ← height, weight
    - *Auto*: MPG ← miles, gallons
- Typical usage:
  - Provide both accessors and mutators for (private) base data
  - **Provide (public) accessors only for derived data**
    - **Recalculate the derived data from other base data, any time it is needed**
  - Derived data does not need to become additional instance variables
    - If we wanted to do that, it would need to be recalculated any time any of its base data changed

Derived Data for Person

- **Age**
  - Simple: just use our existing age calculation method in Utils class
- **IQ**
  - Assume: you begin life with IQ=100, add 1 pt per year of age
- **BMI**
  - Calculated according to:

$$BMI = \frac{\text{weight (kg)}}{\text{height}^2\ (\text{m}^2)}$$

- Add these to our *print*() output also
  - Format BMI (and height and weight also) to 2 decimal places
- Notice that no method arguments need to be passed
  - We have all the data needed, available internally as class instance variables

```
32      // other class data --------------
33      private DecimalFormat bodyFmt = new DecimalFormat("0.00");
```

```
81      // string version of object data
82      public String toString() {
83          return firstName + ", " +
84                 lastName + ", " +
85                 birthdate + ", " +
86                 gender + ", " +
87                 bodyFmt.format(height) + ", " +
88                 bodyFmt.format(weight);
89      }
90
91      // formatted version of object data
92      public void print() {
93          System.out.println("firstName:\t" + firstName);
94          System.out.println("lastName:\t" + lastName);
95          System.out.println("birthdate:\t" + birthdate);
96          System.out.println("gender:\t\t" + gender);
97          System.out.println("height:\t\t" + bodyFmt.format(height));
98          System.out.println("weight:\t\t" + bodyFmt.format(weight));
99          System.out.println("age:\t\t" + getAge());
100         System.out.println("IQ:\t\t" + getIQ());
101         System.out.println("BMI:\t\t" + bodyFmt.format(getBMI()));
102     }
```

```
275     // derived data accessors --------
276
277     // compute age
278     public int getAge() {
279         return UtilsRL.getAge(birthdate);
280     }
281
282     // compute IQ
283     public int getIQ() {
284         final int BASE_IQ = 100;
285         return BASE_IQ + getAge();
286     }
287
288     // compute Body Mass Index (BMI)
289     public double getBMI() {
290         if (height > 0.0) {
291             return weight / (height * height);
292         }
293         else {
294             return 0.0;
295         }
296     }
```

This snapshot of the *Person* class is saved as ***PersonPhase10.java*** in **Example Source Code**

Utility Methods for Person

- **eat**()
  - Assume a person's weight increases by 0.1 kg per 1000 food calories
- **exercise**()
  - Assume a person's weight decreases by 0.1 kg per hr of exercise
- These are completely made-up relations, but actual ones could replace these
- We can also do input validity checks for these

```
298     // utility methods ---------------
299
300     // eating food increases the weight (assume 0.1 lb/1000 food cals)
301     public void eat(double kcal) {
302         if (kcal < 0.0) {
303             System.out.println("ERROR: food kcal must be >= 0.0, no change");
304         }
305         else {
306             weight += ((kcal/1000.0) * 0.1);
307         }
308     }
309
310     // exercising decreases the weight (assume 0.1 lb/hr)
311     public void exercise(double hrs) {
312         if (hrs < 0.0) {
313             System.out.println("ERROR: exercise hrs must be >= 0.0, no change");
314         }
315         else {
316             weight -= (hrs * 0.1);
317         }
318     }
```

```
407         // test utility methods
408         p2.print("p2 before eating");
409         p2.eat(1000);
410         p2.print("p2 after eating, needs exercise")
411         p2.exercise(2.0);
412         p2.print("p2 after exercising");
```

```
======================
p2 before eating
======================
firstName:      Joe
lastName:       Cool
birthdate:      7/4/2000
gender:         M
height:         2.00
weight:         80.00
age:            14
IQ:             114
BMI:            20.00
======================
p2 after eating, needs exercise
======================
firstName:      Joe
lastName:       Cool
birthdate:      7/4/2000
gender:         M
height:         2.00
weight:         80.10
age:            14
IQ:             114
BMI:            20.02
======================
p2 after exercising
======================
firstName:      Joe
lastName:       Cool
birthdate:      7/4/2000
gender:         M
height:         2.00
weight:         79.90
age:            14
IQ:             114
BMI:            19.97
```

This snapshot of the *Person* class is saved as ***PersonPhase10.java*** in **Example Source Code**

Object Update For *Person*

- Another useful utility is to group individual overloaded "user prompt" mutators into <u>one</u> update method
- Create a default object, then immediately call this method to update it using user input prompts
- This one method also tests all the individual overloaded mutator update methods

```
320     // update the entire object from user prompts
321     public void update(boolean inputMode) {
322         setFirstName(inputMode);
323         setLastName(inputMode);
324         setBirthdate(inputMode);
325         setGender(inputMode);
326         setHeight(inputMode);
327         setWeight(inputMode);
328     }
```

```
414         // new object creation and update
415         Person p5 = new Person();
416         p5.update(false);
417         p5.print("new p5 object");
```

```
Enter first name > Sheldon
Enter last name > Cooper
Enter birthdate month > 5
Enter birthdate day > 4
Enter birthdate year > 1980
Enter gender [M/F] > M
Enter height [m] > 1.9
Enter weight [kg] > 70
======================
new p5 object
======================
firstName:      Sheldon
lastName:       Cooper
birthdate:      5/4/1980
gender:         M
height:         1.90
weight:         70.00
age:            34
IQ:             134
BMI:            19.39
```

This snapshot of the *Person* class is saved as ***PersonPhase10.java*** in **Example Source Code**

Class Testing Strategy

- **If you created a capability, then test that capability (at least once)**
- **Write your unit test code as you develop your code!**
  - Helps you know right away if what you added works
  - Makes it simpler and faster in the long haul to develop your code
- Create one or more objects, then:
  - Display them
  - Make some changes to them
  - Then re-display them ("before and after" testing)
- Carefully inspect the test outputs
  - Make sure the intended changes have taken place
  - Make sure the numerical results are accurate, or reasonably so
- **Comment your unit test code, and annotate your test outputs!**
  - Use good names for your objects
  - Label what all outputs represent
    - The overloaded print() is very helpful for this: I recommend it highly!
- **See examples of unit testing in our final version of *Person.java***

Class Testing Checklist

- When testing a user-written class, make sure to exercise all of the following (at least ONCE):
  - Create objects using all **constructor forms**
  - Display the object content using ***toString*()**, and (preferably) ***print*()**
    - Check that any default values are correct
  - Test the **accessors** by extracting some data and printing it
  - Test the **mutators** by altering the object data somehow
    - Redisplay the object content and confirm the changes
    - If using mutator data checking, **also test some invalid values**
  - Test the ***equals*()** method by testing the object
    - Against itself (it better be true!)
    - Against another object of the same type
    - Against some other arbitrary object
  - Test all **derived data accessors**
    - By adding these to the print(), you are already doing this!
  - Test all **utility methods**, including user-prompted object updates
- **See examples of unit testing in our final version of *Person.java***