

# Lecture 23:

## Writing Classes, Part I

Sierra College  
CSCI-12  
Spring 2015  
Weds 04/22/15

# Announcements

- **General**

- Get started on new assignment: the next one after this one will be out next Weds (one day overlap in lab)

- **Schedule**

- **Past due assignments**

- PRGM19: Age Utils, accepted thru Sun 4/26 @ 11pm (-3 pts/day)

- **New assignments**

- PRGM22: Menu For Demo (due Weds 4/29 @ 11pm) [\[lab today\]](#)
  - An exercise in nested logic (*while*, *switch*, *for*, and *if-elseif*)
  - Update `readInt()` for type-safe data handling, and also add a `readChar()`
  - Create a user-controlled, menu-based “for-loop demo”
  - You will reuse this structure on the LAST assignment, so understand how it works!

# Lecture Topics

- **Last time:**
  - Using *while* looping to perform type-safe inputs
  - Count-controlled looping: the *for* loop
  - Examples of *for* looping
- **Today:**
  - More examples of *for* looping
  - Review of classes and objects
  - Begin the steps of creating a new user class “***Person***”

# For Next Time

- **Lecture Prep**

- Text readings and lecture notes

- **Program**

- Get started on the new assignment

- Suggestions:

- First, review the new assignment carefully
- Start by updating your `readInt()` and creating `readChar()`
- Then, implement a simple endless *while* loop
- Then, control it using an update read of a *char*
- Then, handle the keyboard input *char* in a *switch* statement
- Then, add the *switch* cases (*int* reads and a *for* loop)
- Then, add the needed logic around the *for* loop (*if-else* logic)

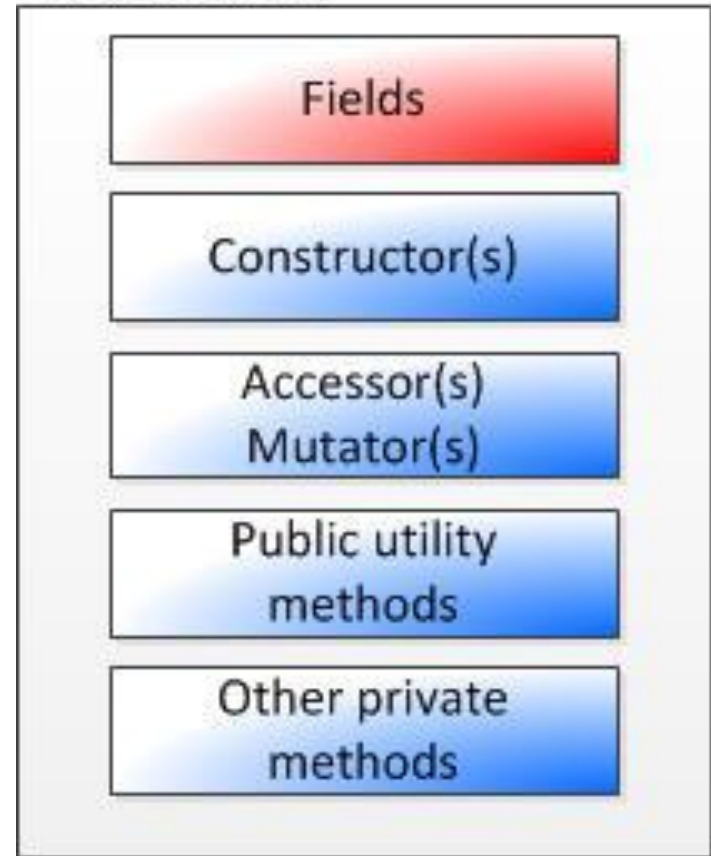
# Class Refresher

- Summary of prior lectures (*please review as needed*):
  - A class is the blueprint, or template, for a software “thing”
    - Class → cookie cutter    Object → one particular cookie
  - Classes allow us model real-world entities
  - Everything in Java is a class
  - Every piece of code you’ve written this semester has been a class
- Advantages of a class:
  - They bundle together (encapsulate) **data**, plus all possible operations upon that data (**methods**)
  - Its implementation details can be hidden
  - A well-written class can be reused over and over again
    - Software development efficiency (*“don’t reinvent the wheel”*)

# Anatomy of a Class

- This is a general layout of a .java source code file
- Order is not set in stone, however:
  - Fields appear first
  - Constructors are usually the first methods

*MyClass.java*



# Intended *Person* Class API

Person Class API	
Person Class Instance Variables	
String <b>firstName</b>	- first name
String <b>lastName</b>	- family name or surname
SimpleDate <b>birthdate</b>	- date of birth
char <b>gender</b>	- M or F
double <b>height</b>	- height in [m]
double <b>weight</b>	- weight in [kg]
Person Class Constructors	
<b>Person()</b>	
Creates a Person object with default initial values	
<b>Person(String firstName, String lastName, SimpleDate birthdate, char gender, double height, double weight)</b>	
Creates a Person object with all values specified	
<b>Person(String firstName, String lastName)</b>	
Creates a Person object with specified first/last names, and all other instance variable with default values	
Person Class Methods	
String <b>toString()</b>	Returns all instance variables in a comma-separated String format
void <b>print()</b>	Displays all instance variables in a labeled output format
String <b>getFirstName()</b>	Returns the value of firstName
void <b>setFirstName(String firstName)</b>	Sets the value of firstName
void <b>setFirstName(boolean inputMode)</b>	Sets the value of firstName, using user input prompts
String <b>getLastName()</b>	Returns the value of lastName
void <b>setLastName(String lastName)</b>	Sets the value of lastName
void <b>setLastName(boolean inputMode)</b>	Sets the value of lastName, using user input prompts
SimpleDate <b>getBirthdate()</b>	Returns the value of birthdate
void <b>setBirthdate(SimpleDate birthdate)</b>	Sets the value of birthdate
void <b>setBirthdate(boolean inputMode)</b>	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 API Legend

- We will build up our class in roughly the order shown
  - This is where we're "going"
- Notice that the list of API methods has some "color coding" specified
- The color legend below shows related methods grouped together



A vertical legend with five colored rectangular boxes, each containing a text label. From top to bottom: a yellow box with 'display methods', a light green box with 'accessors/mutators', a pink box with 'equivalence method', a light blue box with 'derived data accessors', and a light orange box with 'utility methods'.

display methods
accessors/mutators
equivalence method
derived data accessors
utility methods

- Display methods
  - Methods which allow us to "see" the current object content
- Accessors/mutators
  - The **get...()/set...()** methods, 2 for each public instance variable
- Equivalence method
  - **equals()**, checks whether 2 objects are field-for-field equal
- Derived data accessors
  - Accessors only for various data derived from instance variables
- Utility methods
  - Various "actions" our class might want to do



# *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*”

# CRUD

- In programming or database development, you may sometimes encounter the acronym “CRUD”
- **CRUD = create, read, update, delete**
- Here, this helps us keep in mind the basic types of operations we typically want to provide for as we build up new classes
- **Create**
  - Constructors
- **Read**
  - Display methods
  - Accessor (get) methods
  - Derived data accessors
- **Update**
  - Mutator (set) methods
  - Utility methods
- **Delete**
  - N/A for our purposes at this time

# Starting Skeleton of a Class

- The code at right is the starting pseudocode “outline” for the **Person** class API
  - It is compilable, but not executable
  - Note use of visual separators
- Conventions:
  - Class names begin with CAPS
  - Class names are nouns (“things”)
  - Use CamelCase naming
  - Class names are always **public**
    - We want classes to be seen/used by the outside world

```
public class Person {  
  
    // instance variables -----  
  
    // constants -----  
  
    // other class data -----  
  
    // data above here  
    //=====   
    // methods below here  
  
    // constructors -----  
  
    // display methods -----  
  
    // accessors, mutators -----  
  
    // equivalence -----  
  
    // derived data accessors -----  
  
    // utility methods -----  
  
} // end class
```

This snapshot of the *Person* class is saved as ***PersonPhase1.java*** in **Example Source Code**

# Types of Classes

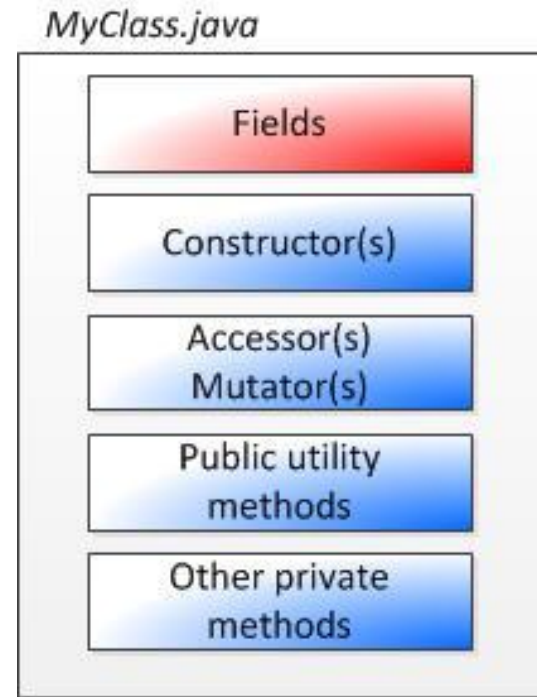
- Predefined classes
  - Those already available as part of Java
  - Java API give the full list of what's available (~2000 or more)
  - Some are automatically available for use:
    - *String, Math, etc.*
  - Some must be imported before using:
    - *Scanner, JOptionPane, Random, etc.*
- User-defined classes
  - Allow us to extend the core Java language with our own capabilities
    - Anything you've written so far in class this semester
    - Every one of my lecture examples
    - *SimpleDate*
    - All textbook examples
    - Any 3<sup>rd</sup> party classes or packages

# Some Class Terminology

- **Class members**
  - **Fields/instance variables**      data
  - **Methods**      operations on the data
- **Object**
  - One specific instance of a class
  - A class is the “general”, an object is the “specific”
- **Client**
  - Some outside code which uses a class as an object
  - For us, this will also represent **test driver code**
- **Access modifier**
  - Specifies the visibility, or availability, of a class or its members to the outside world
  - **public**, **private**, or package (we will only deal with the first two)
- **Scope**
  - The range of visibility of variables or methods within their class

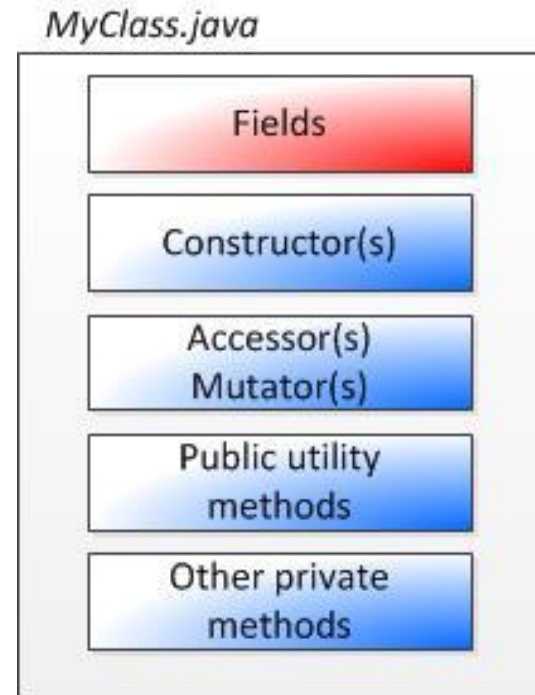
# Data Scope

- The **scope** (visibility) of instance data of a class is “global” within the class
- Every method within a class has access to every instance variable within that class (public or private)
  - No extra declarations needed
  - No passing via method interfaces needed
- All methods share one common version of the SAME instance data



# Method Scope

- The **scope** (visibility ) of methods of a class is “global” within the class
- Any method within a class can call any other method within that class (public or private)
  - The public methods subset can also be called by outside clients
- Generally speaking, a class is:
  - A collection of some data
  - Plus lots of small, individual operations upon that data (methods)
- When designing a class:
  - First, outline, or “rough out” (pseudocode) all the desired methods
  - Then, write them one-by-one



# Instance Variables

- The **instance variables** (or **fields**) of a class are simply its data (what the class “is”)
  - They are nothing more than a collection of related variables!
  - They follow all the usual naming conventions of variables (camelCase, descriptive, etc.)
- They can be:
  - Any of the 8 core datatypes
  - Objects of any Java-provided class (*String*, etc.)
  - Objects of any user-defined class (*SimpleDate*, etc.)
- **Access specifiers** for instance variables:
  - In a well-designed Java class, instance variables are **private**
  - The only outside access is via **public accessors** and **mutators**
- **Scope** of instance variables:
  - All instance variables are visible to all methods of the class (“**class scope**”)
  - It is not necessary to pass them into methods as arguments, or declare them within methods: we get free class-wide access to them
  - They are akin to “global” variables, within the confines of the class



# Access Modifiers

Access Modifier	Class or member can be referenced by...
<i>public</i>	methods of the same class and methods of other classes <b>visible/usable by outside code</b>
<i>private</i>	methods of the same class only <b>NOT visible/usable by outside code</b>
<i>protected</i>	methods of the same class, methods of subclasses, and methods of classes in the same package
No access modifier ( <i>package</i> access)	methods in the same package only

In this course, we will use *public* and *private* exclusively

# Instance Variable Specification

- Here we show the creation of ONE instance variable
  - All instance variables have ***private*** access specification
- Once we have the first one in place, adding additional ones later is straightforward
  - We will add more instance variables later on
- We can use inline comments to document instance variables' purpose, units, etc.

```
13 public class Person {  
14  
15     // instance variables -----  
16     private String firstName;    // person's first name  
17 }
```

This snapshot of the *Person* class is saved as ***PersonPhase2.java*** in **Example Source Code**

# Constructors

- A class **constructor** is nothing but a special-purpose method
  - It is the method called when a new object instance is created using the *new* keyword:

```
String greeting = new String("Hello World!");  
SimpleDate defaultDate = new SimpleDate();
```
  - Its purpose is to set up and initialize a new instance (object) of the class
- Some special things about a constructor method
  - It has the **same name** as the class (with leading capital)
  - Its access is **public**, so the outside world can use it
  - It has NO return value, not even a **void**
- Example: SimpleDate default constructor (internally)

```
public void SimpleDate( ) {...}
```

# Constructor Purpose

- The purpose of a constructor is to set up and initialize a new instance of the class (an object)
  - All instance variables should be initialized to specified values, or else set to “reasonable” defaults
  - Any other startup actions should be performed
    - Constructors can call other methods within their class
    - Constructors can do anything else that any Java method can do
  - A constructor is actually optional
    - If not provided, the compiler provides a **default constructor**
      - We made use of this on our “Methods” assignment
    - Instance variables get set to default values thru **autoinitialization**
    - **CS-12 coding standard:** always provide one or more constructors, and explicitly initialize each instance variable (do not rely upon autoinitialization)

# Autoinitialization Initial Values

Data Type	Default Value
<i>byte, short, int, long</i>	0
<i>float, double</i>	0.0
<i>char</i>	The null character
<i>boolean</i>	<i>false</i>
Any object reference (for example, a <i>String</i> )	<i>null</i>

# Constructor Examples (ver.1)

- **Default constructor**
  - Sets “reasonable” default values
  - Should be “recognizable” values (designer’s choice)
  - “Specify nothing” (no control)
- **Full constructor**
  - Passes in the parameters which are to be used
  - “Specify everything” (full control)
- All constructors are nothing more than **methods** carrying the same name as the class itself
- For this course, always provide at least these two forms
- Providing multiple like-named constructors illustrates **overloading**

```
--  
26 // constructors -----  
27  
28 // default constructor  
29 public Person() {  
30     firstName = "unknown";  
31 }  
32  
33 // full constructor  
34 public Person(String personFirstName) {  
35     firstName = personFirstName;  
36 }  
--
```

# Overloading

- In O-O programming, methods can (and often do) have multiple versions
  - This is called **overloading**
  - The method variant interfaces must differ in one or more of: {arg #, arg types, arg order}
  - Argument names are NOT taken into account as differences
- For constructors:
  - A **default constructor** takes no arguments
  - A **full constructor** specifies ALL instance variables
  - Other constructor forms are the designer's choice
    - May want to provide useful alternate or shortcut forms
  - **CS-12 coding standard: always provide a default and a full constructor, others are your choice**

# The *this* Keyword

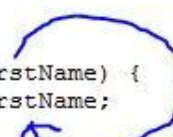
- The prior example is kind of... “clumsy”
  - We are using two variables to refer to the same quantity, in the full constructor
  - Can we improve this??
- All objects of the same class share one copy of its methods
  - ***this*** is an implicit parameter sent to all methods
  - It is an object reference to the specific object for which the method was called
  - In reality, the full name of any instance variable *varName* is really: ***this.varName***
  - So, we can use “***this***” to distinguish between an instance variable and its method argument



# Constructor Examples (ver.2)

- **Default constructor**
  - No ambiguity, no change
- **Full constructor**
  - Input parameter carries same name as its instance variable destination
- Now, we are using one common variable name to refer to the same data entity
- **this** indicates the instance variable for the current (“this”) object

```
13 public class Person {  
14  
15     // instance variables -----  
16     private String firstName;    // person's first name  
17  
18     // constants -----  
19  
20     // other class data -----  
21  
22     // data above here  
23     //=====   
24     // methods below here  
25  
26     // constructors -----  
27  
28     // default constructor  
29     public Person() {  
30         firstName = "unknown";  
31     }  
32  
33     // full constructor  
34     public Person(String firstName) {  
35         this.firstName = firstName;  
36     }  
}
```



This snapshot of the *Person* class is saved as ***PersonPhase3.java*** in **Example Source Code**

# Display Methods

- Once we have created objects with our class constructors, we next need some way of “seeing” inside those objects
- One way to do this would be by using a debugger
  - OK if developing code
  - But not all users would be using an IDE
- A better way would be to be able see inside our objects as part of our code
- Two methods: ***toString()*** and ***print()***
  - ***toString()***
    - This is a standard method most all classes provide
    - Creates a *String* representation of the object (designer’s choice of format)
    - Returns that *String* to any calling client program
  - ***print()***
    - This is an alternate method we will use for this course (I have found this very helpful)
    - Prints multi-line, labeled text with all desired data
    - Nothing is returned to calling client (printout only)

# The *toString()* Method

- The ***toString()*** method returns a *String* version of an object's data
- Interface is standard for ANY class (line 41)
- Format is designer's choice
  - Comma-separated?
  - Space-separated?
  - Other?
- Here, simply concatenate all instance variables somehow into one return *String*
- Good design practice is to always provide this method
  - CS-12 coding standard: always provide this method

```
40  
41  
42  
43
```

```
// string version of object data  
public String toString() {  
    return firstName;  
}
```

# The *print()* Method

- Another way of “seeing inside” an object is to provide a formatted `print()` method
- Not a standard; however, I have found this to be quite helpful in development
  - CS-12 coding standard: always provide this method
- Simply perform a *println()* of each instance variable, along with some identifying label
- Here we also provide an alternate **overloaded** version, which also prints an identifying label

```
45 // formatted version of object data
46 public void print() {
47     System.out.println("firstName:\t" + firstName);
48 }
49
50 // overloaded version of print, accepts a label string
51 public void print(String message) {
52     System.out.println("=====");
53     System.out.println(message);
54     System.out.println("=====");
55     print();
56 }
```

This snapshot of the *Person* class is saved as ***PersonPhase4.java*** in **Example Source Code**

# Overriding a Method

- **Overriding a method** means to replace its inherited default version with your own version of the method
- Example: ***toString()***
  - *Object* is the ultimate common ancestor of EVERY Java class
  - Every new class inherits *Object*'s default *toString()* method
  - It is good software practice to always **override** this method by providing your own version
  - Default *toString()* from *Object*, versus our overridden one:

```
----jGRASP exec: java Person  
toString is: Person@291aff  
----jGRASP: operation complete.
```

```
----jGRASP exec: java Person  
toString is: Fred  
----jGRASP: operation complete.
```

# Overloading a Method

- **Overloading a method** means to provide some alternate version(s) with differing method interface signatures
  - Must differ in one or more of: {# args, args order, arg datatypes}
- Examples:
  - Providing multiple constructors is an example of overloading
  - We could provide another *print()* method, this time accepting a text label, then calling itself internally (nested calls):

```
public void print(String label) {  
    System.out.println(label);    // print a leading label  
    print();                      // followed by nested default print call  
}  
  
// client invocation example:  
myObj.print("This are the current values of myObj:");
```

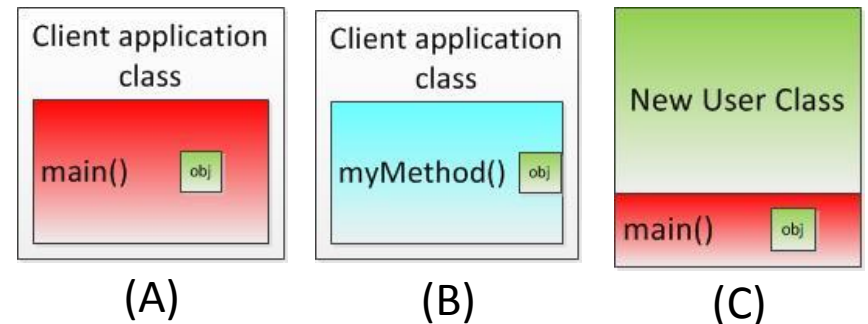
# Other Types of “Data”

- Notice that we have provided for other types of “data” in a class
- Not every declared variable or object has to be an **instance variable**
- Some data might be for internal usage only
  - Would also be declared as **private**
  - No need for any outside access, so no accessors/mutators needed
  - Usable throughout the entire class
- Examples:
  - *Scanner*, *JOptionPane* for inputs
  - Computation constants
  - Conversion factors
  - *DecimalFormat* objects for output formatting
  - We’ll use some of these as we gradually build up our new class

```
13 public class Person {  
14  
15     // instance variables -----  
16     private String firstName;    // person's first name  
17  
18     // constants -----  
19  
20     // other class data -----  
21  
22     // data above here  
23     //=====   
24     // methods below here
```

# Using a New Class

- A new user-written class is just another reusable software component
  - Use it to create new objects of the new user-defined type
- It can be used as any other Java class would be used:
  - A) Within a *main()* of some client application class
  - B) Within any other method of some client application class
  - **C) Within a *main()* method inside the same Java class (as unit test code)**
- **For this course:**
  - **Always add a *main()* method to the end of any new class**
  - **Create unit test code as you develop your new class**
- Advantages:
  - User class and its test code are self-contained
  - The class may be run and tested standalone (no extra test code file is needed)





# Testing the New Class

- Add a **main()** method to the end of our new class
  - But we could just as easily have relocated this static *main()* method into an outside class file
- Do an initial test of the class by exercising all created methods:
  - Creating objects with each constructor
  - Displaying the objects as Strings using **toString()**
  - Displaying the objects as a formatted print using **print()**
- Continue adding to this unit test code as each new feature is gradually added (***“build and test”***)

This snapshot of the *Person* class is saved as ***PersonPhase5.java*** in **Example Source Code**

```
66 // unit test code -----
67
68 // test driver for this class
69 public static void main(String [] args) {
70
71     Person p1 = new Person();
72     Person p2 = new Person("Fred");
73
74     System.out.println(p1); // implicit toString()
75     p1.print();
76     p1.print("Default constructor Person is:");
77     System.out.println();
78
79     System.out.println(p2); // implicit toString()
80     p2.print();
81     p2.print("Full constructor Person is:");
82
83 } // end main
84
85 } // end class
```

```
----jGRASP exec: java Person

unknown
firstName:      unknown
=====
Default constructor Person is:
=====
firstName:      unknown

Fred
firstName:      Fred
=====
Full constructor Person is:
=====
firstName:      Fred

----jGRASP: operation complete.
```

# Restrictions on *main()* in the New Class

- When including a *main()* method at the end of a new class as unit test code, keep in mind the following:
  - The *main()* method cannot directly access any of the instance variables
  - The *main()* method cannot directly access any of the methods, even accessors/mutators
  - The ONLY way to use the class code “above”, is via some created object(s) of the class type
- There is effectively an unbreachable “wall” between the two sides of the class code file

