Lecture 24: Writing Classes, Part II

Sierra College
CSCI-12
Spring 2015
Mon 04/27/15

Announcements

- **General**
  - Age Utils programs all graded: great job overall, 2 general comments:
    - **Program spec**: read it carefully, and provide ALL that is req'd (in assignment, in rubric, in test code, in sample output)
    - **Good coding conventions**: nest ALL code in braces, comment ALL code (we are 75% thru the course, so these things are expected by now)
- **Schedule**
  - Review next 4 wks roadmap…
  - You will have lab time for current program on Weds, but we will also introduce NEXT assignment (next 2 labs after that)
  - 2 more programs after current one, last one will be due AFTER final exam (with extra lab session(s) AFTER final exam)
- **Current assignments**
  - PRGM22: Menu For Demo (due Weds 4/29 @ 11pm) [lab Weds]
- **New assignments**
  - Next assignment will be posted Weds (creating a new class, using procedure we are going thru now)

Lecture Topics

- **Last time**:
  - More examples of *for* looping
  - Review of classes and objects
  - Begin the steps of creating a new user class "***Person***"
- **Today**:
  - Continue creating ***Person*** class step-by-step

For Next Time

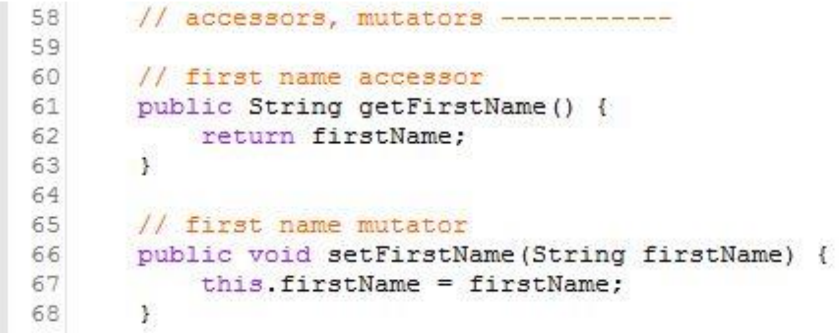
- **Lecture Prep**
  - Text readings and lecture notes
- **Program**
  - Continue working on next assignment
  - Suggestions:
    - First, review the new assignment carefully
    - Start by updating your readInt() and creating readChar()
    - Then, implement a simple endless *while* loop
    - Then, 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)

Accessors and Mutators

- **Accessors** and **mutators** are specific public methods in a Java class, which **get** and **set** private instance data of a class
- Recall that, in a well-designed class:
  - All instance variable data is **private**
  - The only access to it is via **public** accessors and mutators
- Conventions:
  - Both accessors and mutators are **public** methods
  - We generally provide both an accessor AND mutator for each instance variable
    - Unless the instance variable is a derived quantity: in that case, usually only an accessor
- Both are very simple, one-line (or so) methods, with standard interfaces:
  - Accessors: **<datatype> getVarName()**
  - Mutators: **void setVarName(<datatype> varName)**

Accessor and Mutator Examples

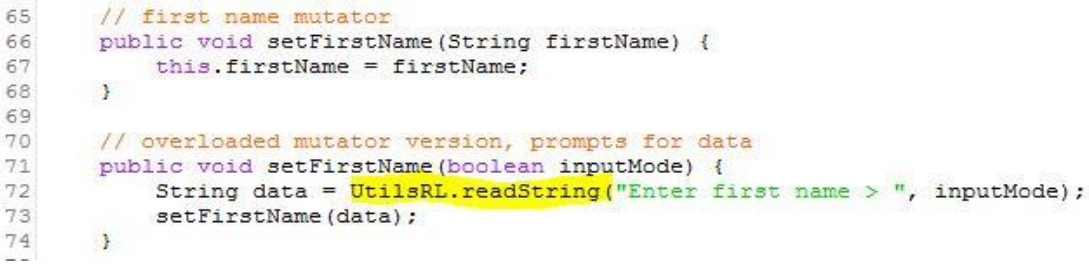
- Note that accessors/mutators typically occur in get/set pairs
  - Write one pair, same pattern for all other instance variables
- Note that accessors/mutators are ***public***
  - We want outside client code to be able to use them
- Note the use of the ***this*** keyword for mutators
  - This lets us use the same variable name for instance variable and argument, without ambiguity
- Note the difference in return type between the method types
- **Simple accessors and mutators are <u>literally</u> "one-liners"!**



```
// accessors, mutators -----------

// first name accessor
public String getFirstName() {
    return firstName;
}

// first name mutator
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

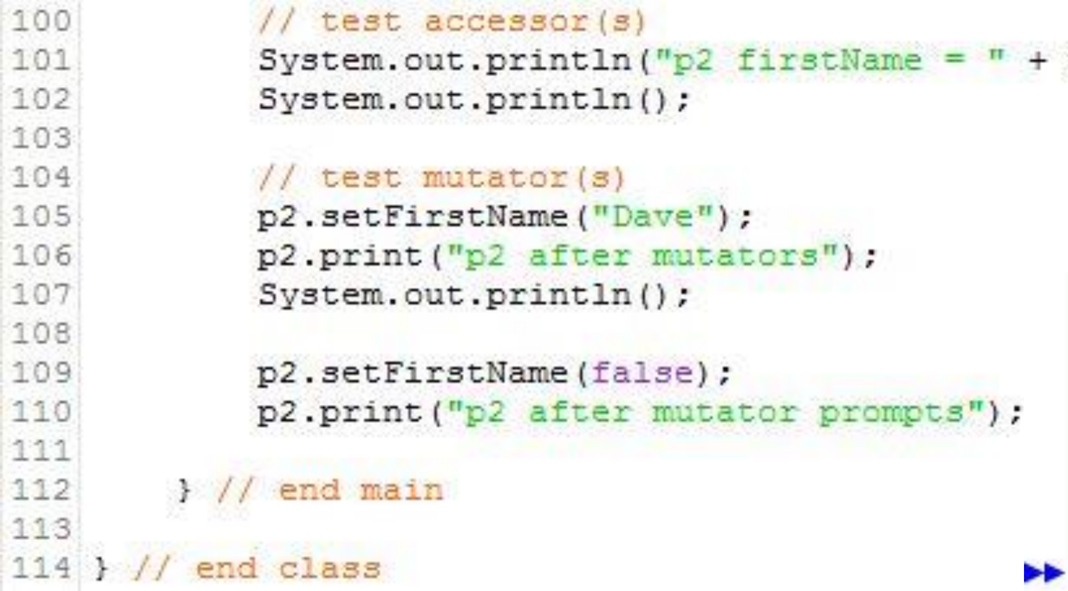
Mutator Nesting

- Avoid the temptation to do “everything”, or “too much”, inside a mutator
  - Keep it short, focused, and on task
- Your mutator should ONLY update the instance variable with a provided value, nothing more
  - **Don’t** add in code to prompt user for values
  - **No “kitchen sink” solutions**
- If you want to also prompt the user for the data:
  - Create an <u>overloaded</u> or <u>alternate</u> version
  - Call (nest) the mutator within it
  - Once again, note **code reuse!**
  - **Utils example shown is instructor specific; you’ll soon write your own version**

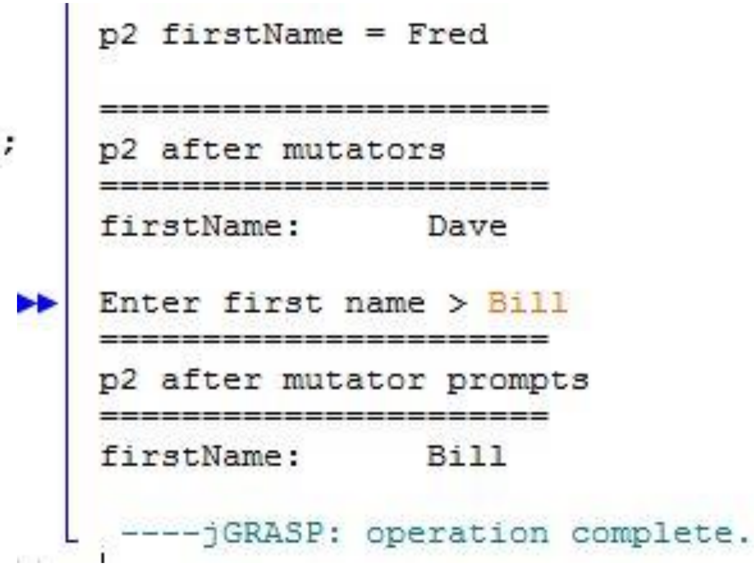


```
// first name mutator
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

// overloaded mutator version, prompts for data
public void setFirstName(boolean inputMode) {
    String data = UtilsRL.readString("Enter first name > ", inputMode);
    setFirstName(data);
}
```

Accessor/Mutator Testing



```
        // test accessor(s)
        System.out.println("p2 firstName = " + p2.getFirstName());
        System.out.println();

        // test mutator(s)
        p2.setFirstName("Dave");
        p2.print("p2 after mutators");
        System.out.println();

        p2.setFirstName(false);
        p2.print("p2 after mutator prompts");

    } // end main

} // end class
```



```
p2 firstName = Fred

======================
p2 after mutators
======================
firstName:      Dave

Enter first name > Bill
======================
p2 after mutator prompts
======================
firstName:      Bill

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

- Embed blank lines and comments in your test code to make the SOURCE more readable
- Embed *println*() newlines and commentary in your test code to make the OUTPUT more readable

This snapshot of the *Person* class is saved as
PersonPhase6.java in **Example Source Code**

Types of Data Comparisons

- We can compare variables of the fundamental datatypes for equality and relative value:
  - Equality operators: == , !=
  - Relational operators: > , >= , < , <=
- There are other types of “things” that we would like to perform **equality comparisons** on:
  - **Objects**
  - **Strings**
  - **Floating point numbers**

Comparing Objects

- When we compare two objects (*SimpleDate*, *String*, etc.), be aware that there are two differing options:
  - The **equality operator (==)**
    - Compares the values of the **object references** only
    - Checks only whether the object references "point to" the same memory location
    - Does NOT check whether the objects' data are identical
  - The ***equals*() method**
    - Compares whether all the **fields** of the objects are identical
    - ALL Java classes inherit this from the *Object* (ultimate ancestor) class
    - Many classes (but not all) implement their own custom versions of it
      - In O-O terms, this is called **overriding a method**
    - Standard interface is: ***public boolean equals(Object obj)***

The *equals()* Method

Return type	**Method name and argument list**
boolean	**equals**(*Object* obj) returns *true* if the data of the object *obj* is equal to the data in the object used to call the method*;* *false* otherwise

Example:

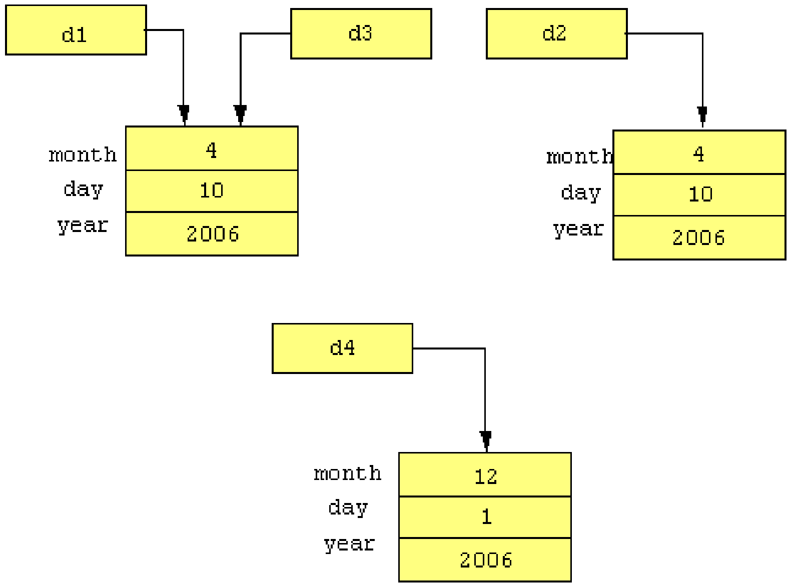
If *d1* and *d2* are *SimpleDate* object references

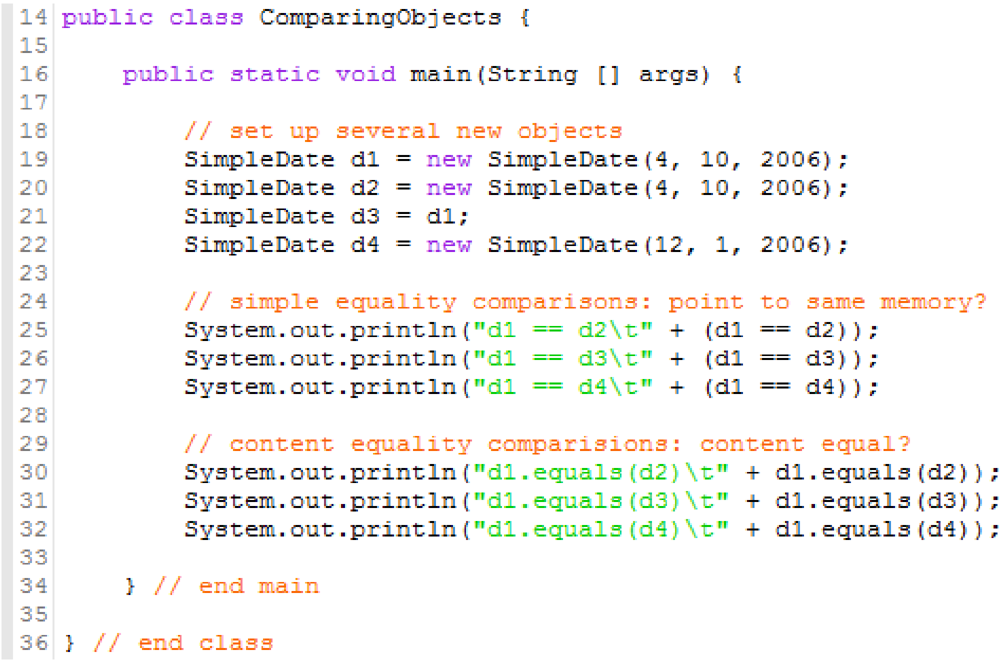
d1.equals(d2)

returns *true* if:

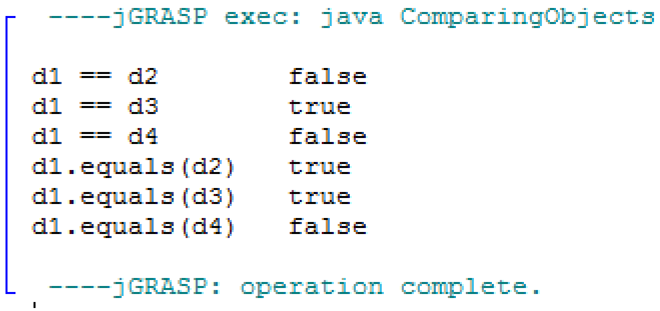
the *month, day*, and *year* of ***d1*** are identical to the *month, day,* and *year* of ***d2***

Example: Comparing Objects





```
public class ComparingObjects {

    public static void main(String [] args) {

        // set up several new objects
        SimpleDate d1 = new SimpleDate(4, 10, 2006);
        SimpleDate d2 = new SimpleDate(4, 10, 2006);
        SimpleDate d3 = d1;
        SimpleDate d4 = new SimpleDate(12, 1, 2006);

        // simple equality comparisons: point to same memory?
        System.out.println("d1 == d2\t" + (d1 == d2));
        System.out.println("d1 == d3\t" + (d1 == d3));
        System.out.println("d1 == d4\t" + (d1 == d4));

        // content equality comparisions: content equal?
        System.out.println("d1.equals(d2)\t" + d1.equals(d2));
        System.out.println("d1.equals(d3)\t" + d1.equals(d3));
        System.out.println("d1.equals(d4)\t" + d1.equals(d4));

    } // end main

} // end class
```



```
 ----jGRASP exec: java ComparingObjects

d1 == d2        false
d1 == d3        true
d1 == d4        false
d1.equals(d2)   true
d1.equals(d3)   true
d1.equals(d4)   false

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

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

Comparing Strings

- A *String* is a specific type of an *Object*
- The prior comparisons for objects hold here, as well as some new considerations
  - String reference comparison:
    - The **equality operator**: **==**
  - String data comparison:
    - ***equals***() method case-sensitive comparison
    - ***equalsIgnoreCase***() method case-insensitive comparison
  - Lexicographical comparison:
    - ***compareTo***() method case-sensitive comparison
    - ***compareToIgnoreCase***() method case-insensitive comparison

    - **Lexicographical** → "coming first in a dictionary"
      - Used for alphabetical sorting
      - Uses ASCII/Unicode values, character by character

The *equalsIgnoreCase()* Method

Return type	Method name and argument list
boolean	**equalsIgnoreCase**(*String* str) compares the value of two *Strings,* treating upper and lower case characters as equal. Returns *true* if the *Strings* are equal; *false* otherwise.

Example:

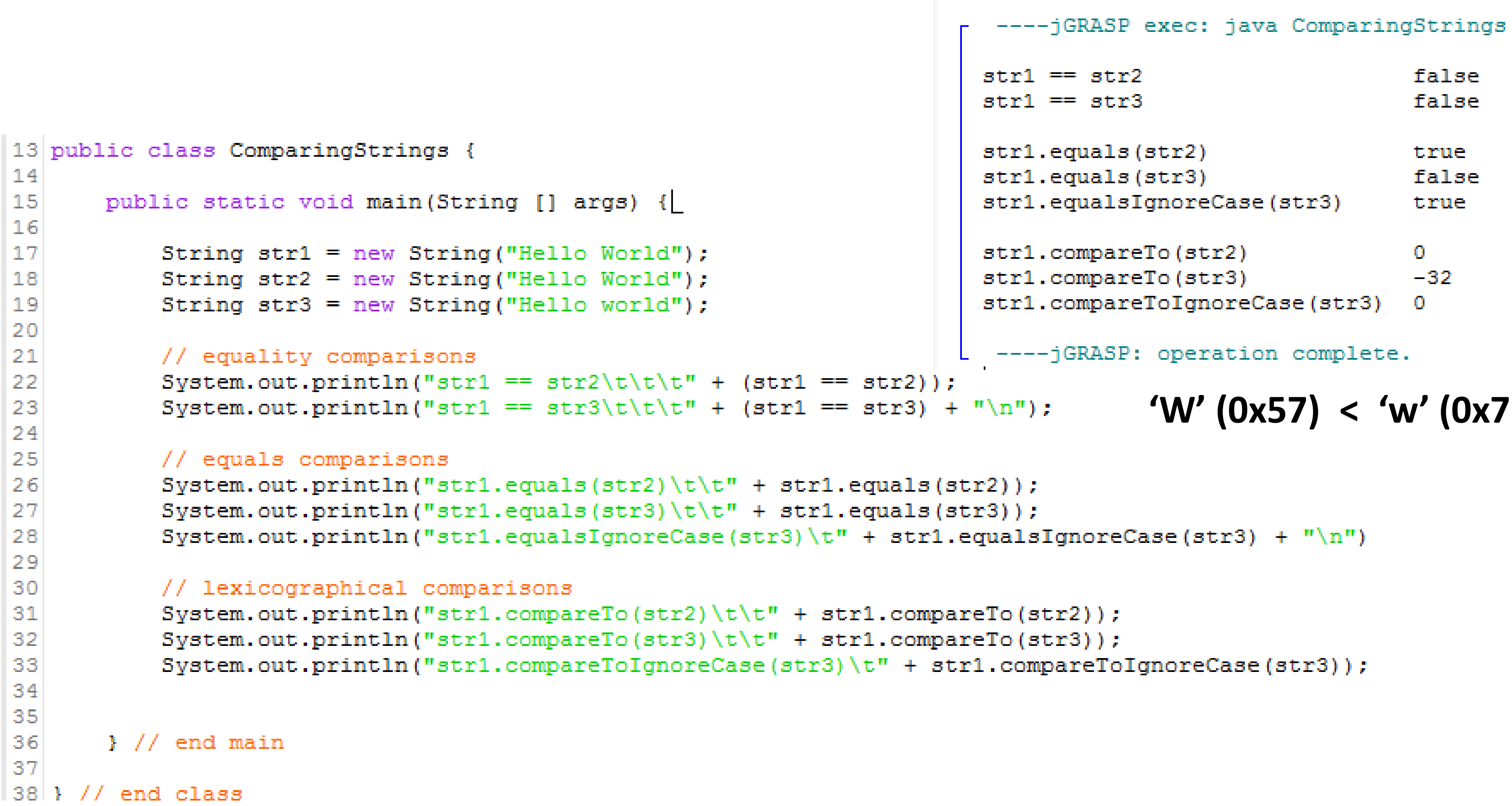
```
String s1 = "Exit", s2 = "exit";
if ( s1.equalsIgnoreCase( s2 ) )
        System.exit( 0 );
```

The *compareTo()* Method

Return type	Method name and argument list
int	compareTo(*String* str) compares the value of the two *Strings*. If the *String* object is less than the *String* argument, *str*, a **negative integer** is returned. If the *String* object is greater than the *String* argument, a **positive integer** is returned If the two *Strings* are equal, **0** is returned.

- A character with a lower Unicode numeric value is considered less than a character with a higher Unicode numeric value
  - *a* is less than *b* *a* = 0x61, *b* = 0x62
  - *A* is less than *a* *A* = 0x41, *a* = 0x61
  - See ASCII or Unicode tables

Example: Comparing Strings



```
public class ComparingStrings {

   public static void main(String [] args) {

      String str1 = new String("Hello World");
      String str2 = new String("Hello World");
      String str3 = new String("Hello world");

      // equality comparisons
      System.out.println("str1 == str2\t\t\t" + (str1 == str2));
      System.out.println("str1 == str3\t\t\t" + (str1 == str3) + "\n");

      // equals comparisons
      System.out.println("str1.equals(str2)\t\t" + str1.equals(str2));
      System.out.println("str1.equals(str3)\t\t" + str1.equals(str3));
      System.out.println("str1.equalsIgnoreCase(str3)\t" + str1.equalsIgnoreCase(str3) + "\n")

      // lexicographical comparisons
      System.out.println("str1.compareTo(str2)\t\t" + str1.compareTo(str2));
      System.out.println("str1.compareTo(str3)\t\t" + str1.compareTo(str3));
      System.out.println("str1.compareToIgnoreCase(str3)\t" + str1.compareToIgnoreCase(str3));


   } // end main

} // end class
```

```
 ----jGRASP exec: java ComparingStrings

str1 == str2                    false
str1 == str3                    false

str1.equals(str2)               true
str1.equals(str3)               false
str1.equalsIgnoreCase(str3)     true

str1.compareTo(str2)            0
str1.compareTo(str3)            -32
str1.compareToIgnoreCase(str3)  0

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

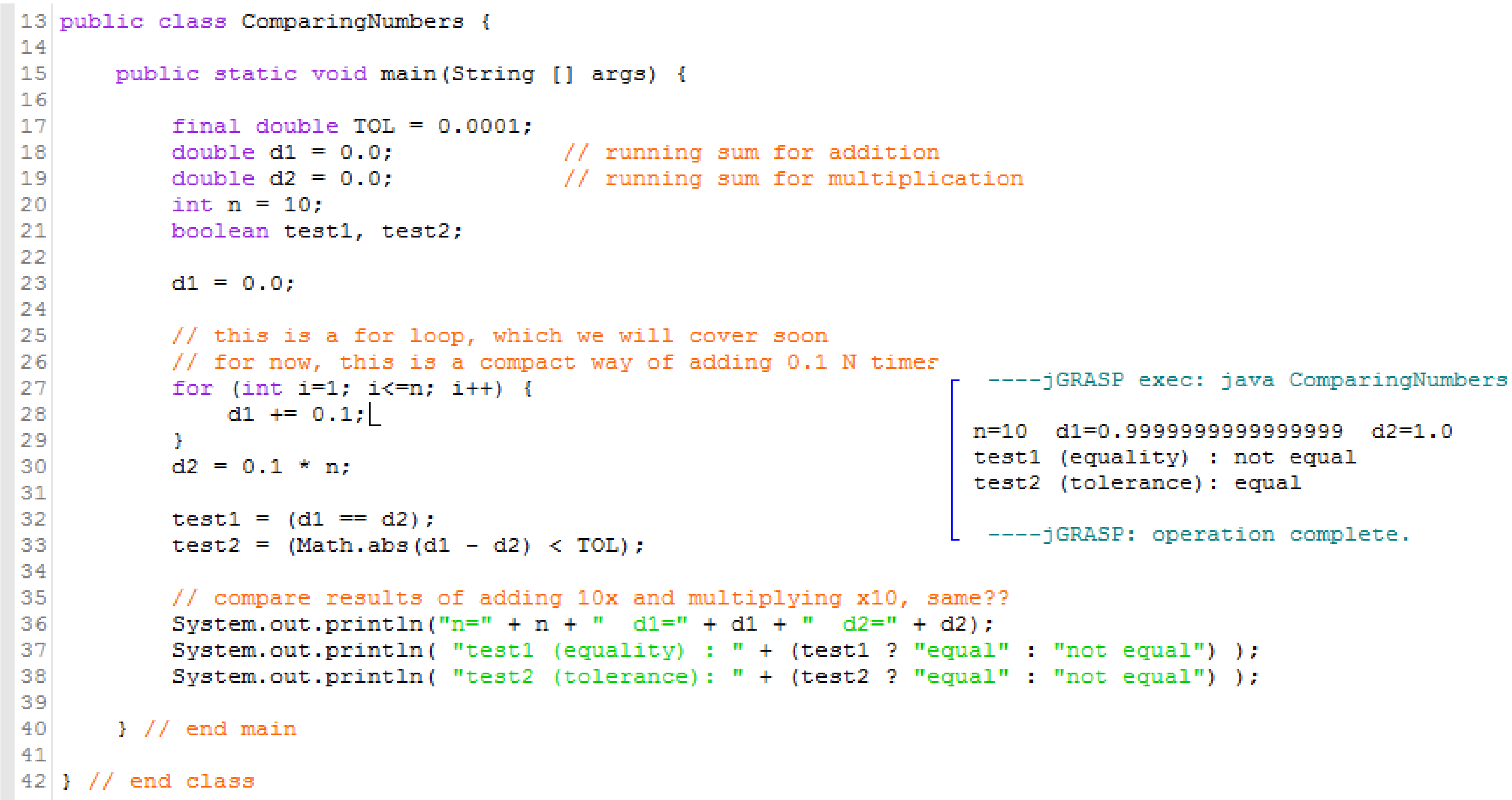
‘W’ (0x57) < ‘w’ (0x77)

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

Comparing Floating Point Numbers

- **Floating point numbers** (*float* and *double* variables) are stored using **IEEE-754 format** (see Appendix G)
- Calculations involving such numbers can (will) introduce low decimal place, **residual rounding errors**
  - (10 * 0.1) is NOT equal to (0.1 + 0.1 + 0.1 + … + 0.1) [10 times]
- Numeric equality is generally handled not with ==, but instead using **tolerances**, or **threshold values**
  - If the difference between two numbers is less than a certain numeric threshold, the numbers are considered “equal”
- The Java ***BigDecimal*** class
  - An alternative for exact precision in calculations with large decimal numbers
  - Provides methods such that arithmetic operations (+, -, *, /) are exact, without floating point rounding errors
  - Part of Java Class Library, in *java.math* package
  - See textbook Ch. 4.1 for details if interested, but we won’t cover in this course

Example: Comparing Floating Point Numbers



```java
public class ComparingNumbers {

    public static void main(String [] args) {

        final double TOL = 0.0001;
        double d1 = 0.0;            // running sum for addition
        double d2 = 0.0;            // running sum for multiplication
        int n = 10;
        boolean test1, test2;

        d1 = 0.0;

        // this is a for loop, which we will cover soon
        // for now, this is a compact way of adding 0.1 N times
        for (int i=1; i<=n; i++) {
            d1 += 0.1;
        }
        d2 = 0.1 * n;

        test1 = (d1 == d2);
        test2 = (Math.abs(d1 - d2) < TOL);

        // compare results of adding 10x and multiplying x10, same??
        System.out.println("n=" + n + "  d1=" + d1 + "  d2=" + d2);
        System.out.println( "test1 (equality) : " + (test1 ? "equal" : "not equal") );
        System.out.println( "test2 (tolerance): " + (test2 ? "equal" : "not equal") );

    } // end main

} // end class
```

```
 ----jGRASP exec: java ComparingNumbers

n=10  d1=0.9999999999999999  d2=1.0
test1 (equality) : not equal
test2 (tolerance): equal

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

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

The ***equals()*** Method

- Do not use the equality operator == to compare two objects for equality
- Instead, it is good practice to always provide an ***equals()*** method
  - Has a standard API form
  - Determines **data equality** of two objects
  - Checks data field-by-field
  - First uses the ***instanceof*** operator, to establish whether we are comparing like objects (“apples to apples”)

equals() Method Interface

Determines if the data in another object is equal to the data in this object:

Return value	Method name and argument list
boolean	**equals**(Object obj) returns *true* if the data in the *Object obj* is the same as in this object; *false* otherwise.

Example client code using *SimpleDate* references *date1* and *date2*:

```
if ( date1.equals( date2 ) ) {
        System.out.println( “date1 equals date2" );
}
```

The *instanceof* Operator

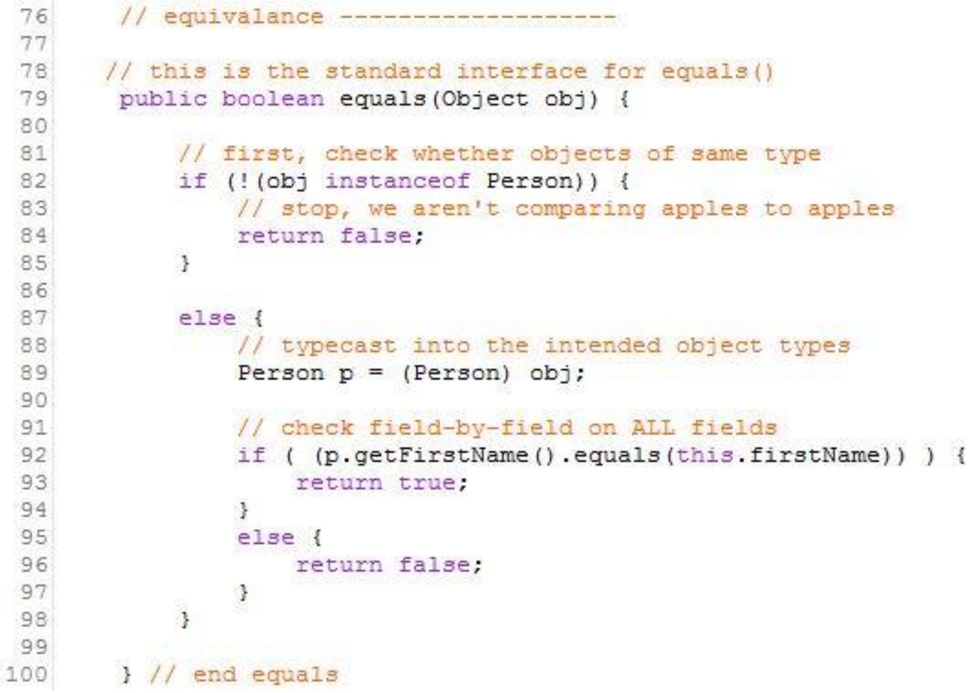
- Because ***equals()*** uses a generic ***Object*** parameter, the two objects being compared don't even have to be of the same object datatype!
- The boolean, binary operator ***instanceof*** checks this:

  objName *instanceof* ClassType

- The first thing we need to establish is whether the two objects are of the same type ("apples to apples")
  - If they are NOT of the same type, don't bother checking further
  - If they ARE of the same type:
    - Datatype cast the (input) generic object to the intended type
    - Continue checking field-by-field
  - See examples on next slides

Adding the *equals*() Method

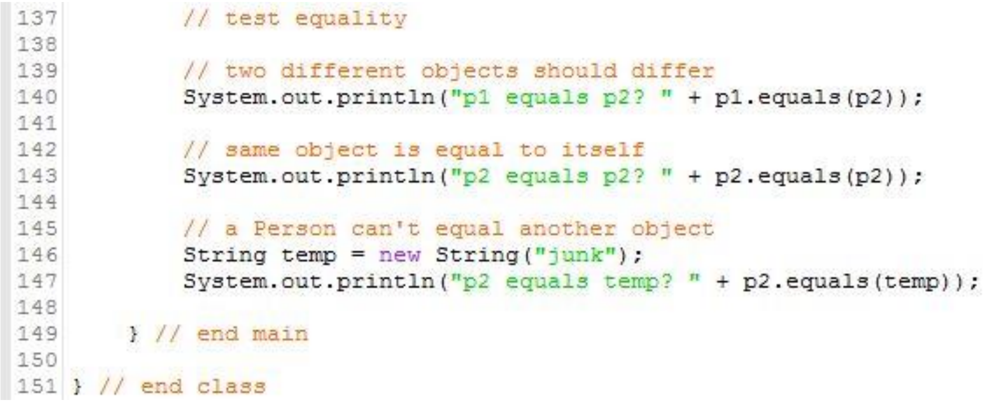
- Standard method interface: ALWAYS the same
- Outer *if-else* tests whether objects are of same class type
- Inner *if-else* checks instance variables versus input object, field-by-field
  - p. → input *Person*
  - this. → current object *Person*
- Comparison (line 92) is <u>very easy</u> to extend for added instance variables, as we will see



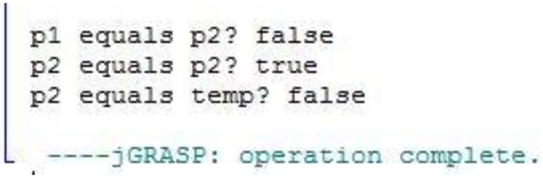
```
 76     // equivalance --------------------
 77
 78    // this is the standard interface for equals()
 79     public boolean equals(Object obj) {
 80
 81         // first, check whether objects of same type
 82         if (!(obj instanceof Person)) {
 83             // stop, we aren't comparing apples to apples
 84             return false;
 85         }
 86
 87         else {
 88             // typecast into the intended object types
 89             Person p = (Person) obj;
 90
 91             // check field-by-field on ALL fields
 92             if ( (p.getFirstName().equals(this.firstName)) ) {
 93                 return true;
 94             }
 95             else {
 96                 return false;
 97             }
 98         }
 99
100     } // end equals
```

Testing the *equals*() Method

- To test the *equals*() method:
  - Compare two different *Person* objects
  - Compare the same *Person* object against itself
    - It had better be equal!
  - Compare any *Person* against any other type of object
    - It had better not be equal!



```
        // test equality

        // two different objects should differ
        System.out.println("p1 equals p2? " + p1.equals(p2));

        // same object is equal to itself
        System.out.println("p2 equals p2? " + p2.equals(p2));

        // a Person can't equal another object
        String temp = new String("junk");
        System.out.println("p2 equals temp? " + p2.equals(temp));

    } // end main

} // end class
```



```
p1 equals p2? false
p2 equals p2? true
p2 equals temp? false

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

This snapshot of the *Person* class is saved as ***PersonPhase7.java*** in **Example Source Code**

Overriding a Method

- **Overriding a method** means to replace its default version with your own version of the method
- Another example: ***equals()***
  - *Object* is the ultimate common ancestor of EVERY Java class
  - Every new class inherits Object’s default *equals()* method
  - It is good software practice to always override this method by providing your own version
- Experiment:
  - Disable the created ***equals***() method
  - Are the results as expected?? Interpret the results…

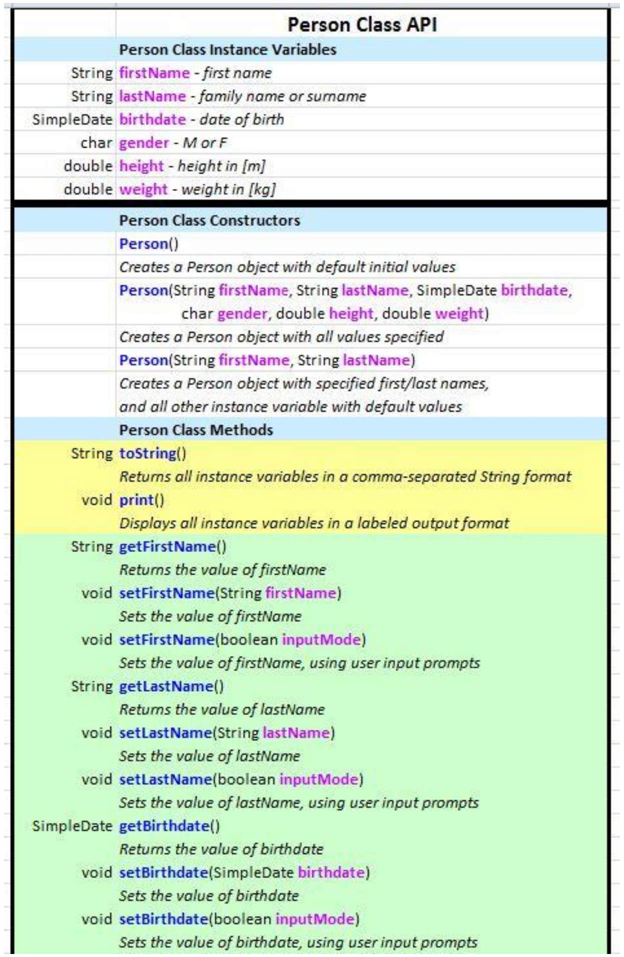
Current State of ***Person*** Class

- Here is a rundown on the current state of our starter ***Person*** class:
  - 1 instance variable
  - 2 constructors (1 default, 1 full)
  - 3 display methods (1 *toString*(), 2 overloaded *print*())
  - 1 accessor, 1 mutator
    - Plus an overloaded mutator which prompt for data (2 forms)
  - 1 *equals*() method
  - Unit test code for all of the above
  - File size: 151 lines (as of ***PersonPhase7.java***)
- Seems like a lot of overhead for one piece of data, right??
  - But now that we have the basic structure in place, extending the class from this point will be EASY!
  - Lots of copy-and-paste cloning of methods

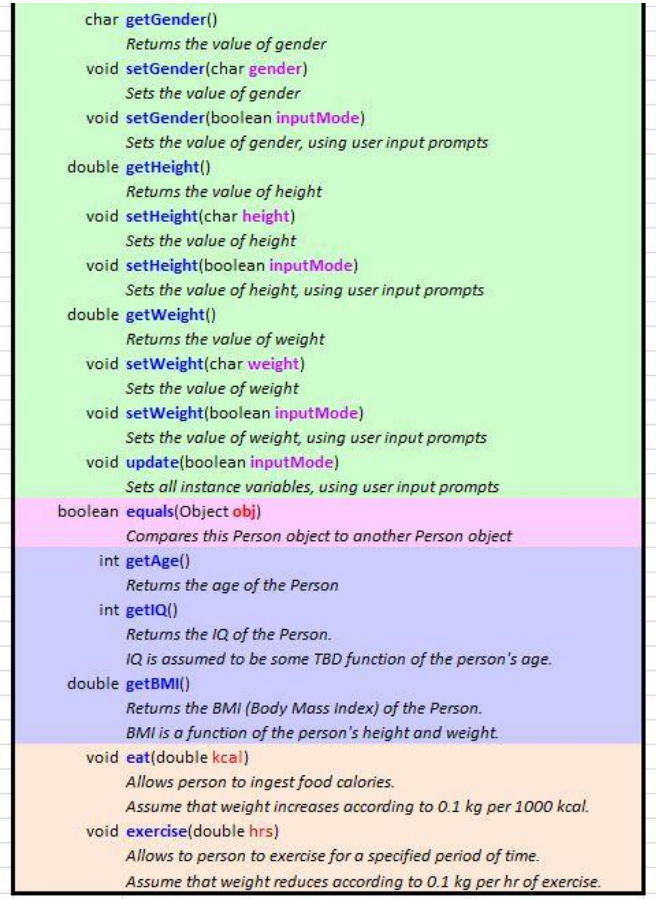
Extending An Existing Class

- Suppose we have a working class, and now we want or need to extend it somehow
- What do we need to do?
  - Declare any added **instance variables**
  - Update **constructor(s)**, and overload new ones as desired
  - Add **accessors/mutators** for the new instance variables
    - Also add overloaded mutators which prompt for data
  - Add new instance variable(s) to the ***toString***() method
  - Add new instance variable(s) to the ***print***() method
  - Add an additional condition(s) to the ***equals***() method
  - Add any other **utility methods** as needed

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.

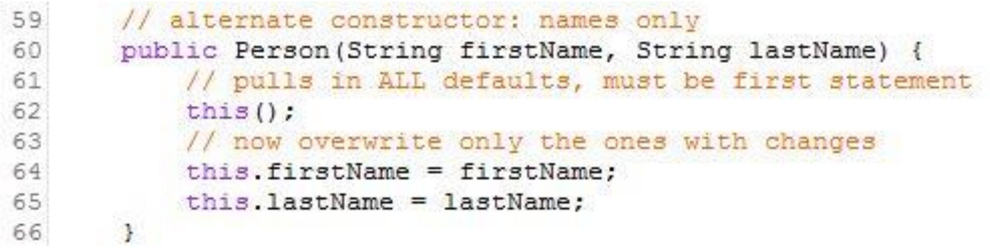
Checklist for Extending ***Person*** Class

- Add remaining instance variables (as outlined in API)
  - So 5 more declarations
- Update constructors
  - Default: 5 more default values
  - Full: 5 more values in argument list, 5 more specific values
- Update display methods
  - *toString*(): add 5 lines
  - *print*(): add 5 lines (tweak output tabbing also)
  - Formatting: perhaps add some *DecimalFormat* format(s)
- Add accessors, mutators
  - 1 accessor, 1 mutator, 1 overloaded mutator for each instance variables
- Equivalence
  - *equals*(): add 5 more comparisons
- Unit test code
  - Add test code for the accessors and mutators ONLY
  - Display methods and *equals*() should remain unchanged
  - Do not need to test methods on EVERY created object, only one of them

Review all these changes in ***PersonPhase8.java*** in **Example Source Code**

Alternate Constructor Example

- **The designer can create as many alternate constructor forms as needed or useful**
- ***this*** can also be used to good advantage for alternate constructor forms
- Rather than repeat all the default values, simply call the default constructor <u>first</u> using: ***this()*** ← MUST be 1st statement
- Then, overwrite <u>only</u> the values which differ from their default values
- Default values are thus only maintained in ONE constructor (the default version)



```
    // alternate constructor: names only
    public Person(String firstName, String lastName) {
        // pulls in ALL defaults, must be first statement
        this();
        // now overwrite only the ones with changes
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

This snapshot of the *Person* class is saved as ***PersonPhase8.java*** in **Example Source Code**