Lecture 13:
Useful Classes

Sierra College
CSCI-12
Spring 2015
Weds 03/11/15

Announcements

- **General**
  - Still catching up on grading, but making good progress…
- **Schedule**
  - Due dates for remaining assignments before the midterm have been "pushed out" (1 wk for each one: lab time and/or a weekend)
  - Midterm exam in 2 wks (Weds 3/25), before spring break
    - More details and a study guide next week
- **Past due assignments**
  - HW10: Methods, accepted thru Monday 3/16
    - Deductions lightened somewhat for this one
- **Current assignments**
  - HW11: SimpleDate, due Thursday 3/12 (*lab time today*)
  - HW12: Strings, due Tuesday 3/17 (*lab time today*)
- **New assignments**
  - HW13: Useful Classes, due Thursday 3/19 (*lab time <u>next</u> week*)

Lecture Topics

- **Last time**:
  - Finished up the *SimpleDate* class
  - Object references
  - Began the *String* class and its API
- **Today**:
  - The Java API

The Java API

- Every class in Java has an **API (Application Programming Interface)**, which tells the user:
  - How to create an object of that class (all **constructor** forms)
  - How to use that object (all available **methods**)

- Similarly, Java itself has the **Java API**, or **Java Class Library**
  - A listing of ALL available Java **packages** (to be defined…)
  - A listing of ALL available, pre-built classes
  - The API for each individual class

Accessing the Java API

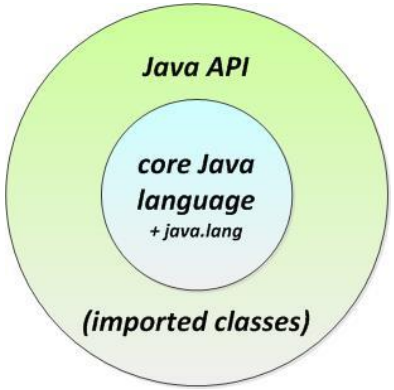
- You have several ways of accessing the Java API online
  1) Browser
     - http://docs.oracle.com/javase/7/docs/api/index.html
  2) Search engine
     - Google: **Java 7 API**
     - Probably the first search result
  3) Canvas
     - See **Modules: Resource Links: Java SE 7 API Documentation**
     - A link to the same place as in (1) or (2)
  4) jGRASP ← probably easiest while writing code
     - From jGRASP, with some file open in the editor: **Help: Java**
     - A link to the same place as in (1) or (2)

Contents of a Java Class API

- In the Java API, click on **All Classes** (top left) to get a scrollable alphabetical listing of all classes
- Click on any individual class to bring up its API
- In a class API, you will typically find the following:
  - The package "address" (hierarchy) of the class
  - The interface of the class
  - Some commentary, or perhaps examples, on using the class
  - All constructor forms
  - A summary of all available (public) class methods
  - Details of all available (public) class methods
- The Java API entries are all examples of automatically generated documentation using **javadoc comments /**…*/**

Java Packages

- Java **packages** are groupings, by functionality, of all Java classes in the **Java Class Library**
  - There are over 2000 predefined classes in Java!
- The classes within packages represent user-selectable extensions to the basic Java language
- Some classes are available to your program automatically
  - Anything within the ***java.lang*** package is already available, with no further action needed (*String*, *Math*, etc.)
- Other classes must first be **imported** in order to be used (many of these we'll encounter in coming lectures…)
  - *Random*, datatype wrapper classes
  - Various I/O classes
  - Numerical formatting classes
  - And many others…



Some Common Java Packages

Classes are grouped in **packages** according to functionality

Package	Categories of Classes
java.lang	Basic functionality common to many programs, such as the *String* class and the *Math* class This package is automatically available without import
java.awt	Graphics classes for drawing and using colors
javax.swing	User-interface components, popup I/O dialogs (*JOptionPane* class resides here)
java.text	Classes for formatting numeric output (*DecimalFormat* and *NumberFormat* classes reside here)
java.util	The *Scanner* class, the *Random* class, and other miscellaneous classes
java.io	Classes for reading from and writing to files

Importing a Java Class

- Java classes are **imported** using the ***import*** statement
  - In application code, the *import* statement appears **outside and before any class declaration**
  - You can import individual classes, or an entire package
- Import considerations:
  - No runtime impact from entire package, perhaps minor compile time hit
  - Entire package does not bloat up bytecode, compiler is selective
  - Also, less chance of naming collisions
  - Class-by-class often preferred, easier to track specific classes used
  - Specific imports better documents the dependencies in your code
- **RECOMMEND: import individual classes, unless most of an entire package is warranted**
  - Example: import javax.swing.*; // for GUI and graphics development

- Example:

```
import  java.util.Scanner;                    // import one specific class (preferred)
//    -- OR --
import java.util.*;                           //import entire package (not preferred)

public class SomeNewClass {
     Scanner input = new Scanner(System.in);

     // body of your class appears here…
}
```

Object Methods

- Up to this point:
  - We have seen that objects are instances of classes
  - Methods are called in the context of existing objects
    - Use **dot notation**: **objectName.methodName()**
- Examples:

```
SimpleDate halloween = new SimpleDate();
String label = “Today is a Wednesday”;
halloween.setDate(10, 31, 2014);
System.out.println( label.toUpperCase() + “   length = “ + label.length() );
```

- Now we will consider the exceptions to this
  - “We didn’t tell you the full story…”
  - Several such example classes will be presented today

Static Methods

- **Static methods** (also called **class methods**) are the exception to the usual object-based methods
- For static methods:
  - They are called without instantiating any object
  - Method is created using the ***static*** keyword before the return type:
    - General: **static** returnType methodName(argList…)
    - Example: **static** double abs(double num)
  - Method is called by using the class name, rather than an object reference:
    - General: **ClassName**.staticMethod(argList…);
    - Example: absValue = **Math**.abs(-52.5);

Why Static Methods?

- Static means “belonging to the class”, rather than to any one specific object of that class
- Static methods:
  - Can provide quick, one time functionality for common operations
    - ***“I just wanna do this one thing…”***
  - Don’t require objects to be initiated and maintained
  - Don’t waste memory and CPU when an object isn’t warranted
- Some examples:
  - The main() method: **public static void main(String [] args) { … }**
    - “Chicken-and-egg” problem: at startup, no objects exist yet
    - Run it once and you are done with it (until next time)
  - Math methods
  - Datatype wrapper methods
  - Pop-up dialogs (next lecture)
- We will see several usage examples in this lecture

Methods Example: Using An Object

3 utility separate utility methods, and also grouped together into one method

```java
public class MethodExamples {

    // note that this class has no instance variables (data),
    // for now, it's just a collection of methods

    // add three integers together
    public int addThreeNumbers(int num1, int num2, int num3) {

        int sum;

        sum = num1 + num2 + num3;
        return sum;
    }

    // calculate the percentage represented by two ints
    public double calculatePercent(int num1, int num2) {

        double result;

        result = (double) num1/num2 * 100.0;
        return result;
    }

    // format a string and print it: a header, in this example
    public void printMessage(String str) {

        System.out.println("=======================================");
        System.out.println(str);
        System.out.println("=======================================");

    }

    // do all 3 above steps internally inside one method
    // ADDED since initial posting on Canvas
    public void performMultipleSteps() {

        int sum;
        double percent;

        sum = addThreeNumbers(10, 20, 30);
        printMessage("Same sum from inside one method: sum = " + sum);

        percent = calculatePercent(57, 89);
        printMessage("Same sum from inside one method: percent = " + percent + "%");

    }

} // end class
```

To use the methods, must call them via some intervening "bridge" object of the class type

```java
public class MethodExamplesClient {

    public static void main(String [] args) {

        // declarations
        int sum;
        double percent;
        MethodExamples obj = new MethodExamples();

        // reuse methods appearing in another file, MethodExamples.java
        sum = obj.addThreeNumbers(10, 20, 30);
        obj.printMessage("   Reuse some useful methods to get:");
        System.out.println("sum = " + sum);

        percent = obj.calculatePercent(57, 89);
        obj.printMessage("percentage is:\t" + percent + '%');

        // same steps, except wrapped inside one method call
        System.out.println();
        obj.performMultipleSteps();

    } // end main

} // end class
```

See ***MethodExamples.java*** and ***MethodExamplesClient.java*** In **Example Source Code**

Methods Example: Static

Same code, except that each method is now *static*

```
14 public class MethodStaticExamples {
15
16     // note that this class has no instance variables (data),
17     // for now, it's just a collection of methods
18
19     // add three integers together
20     public static int addThreeNumbers(int num1, int num2, int num3) {
21
22         int sum;
23
24         sum = num1 + num2 + num3;
25         return sum;
26     }
27
28     // calculate the percentage represented by two ints
29     public static double calculatePercent(int num1, int num2) {
30
31         double result;
32
33         result = (double) num1/num2 * 100.0;
34         return result;
35     }
36
37     // format a string and print it: a header, in this example
38     public static void printMessage(String str) {
39
40         System.out.println("=====================================");
41         System.out.println(str);
42         System.out.println("=====================================");
43
44     }
45
46     // do all 3 above steps internally inside one method
47     // ADDED since initial posting on Canvas
48     public static void performMultipleSteps() {
49
50         int sum;
51         double percent;
52
53         sum = addThreeNumbers(10, 20, 30);
54         printMessage("Same sum from inside one method: sum = " + sum);
55
56         percent = calculatePercent(57, 89);
57         printMessage("Same sum from inside one method: percent = " + percent + "%");
58
59     }
60
61 } // end class
```

Now, there is no intervening object, and all the methods are called using the class name

```
15 public class MethodStaticExamplesClient {
16
17     public static void main(String [] args) {
18
19         // declarations
20         int sum;
21         double percent;
22
23         // reuse methods appearing in another file, MethodStaticExamples.java
24         sum = MethodStaticExamples.addThreeNumbers(10, 20, 30);
25         MethodStaticExamples.printMessage("   Reuse some useful methods to get:");
26         System.out.println("sum = " + sum);
27
28         percent = MethodStaticExamples.calculatePercent(57, 89);
29         MethodStaticExamples.printMessage("percentage is:\t" + percent + '%');
30
31         // same steps, except wrapped inside one method call
32         System.out.println();
33         MethodStaticExamples.performMultipleSteps();
34
35     } // end main
36
37 } // end class
```

See ***MethodStaticExamples.java*** and ***MethodStaticExamplesClient.java*** In **Example Source Code**

Static Data

- Just as we can have **static** methods, we can also have **static** data
- **Static data**:
  - Is declared using the *static* keyword
  - Belongs to the class, not to any one object instance
- Example usage:
  - One typical usage is to keep track of the number of class instances (objects) that have been created
  - Every object has access to the one single count of objects

Static Constants

- One common usage of static data: named **static constants** (or **class constants**)
- Examples:
  - *Math* constants
  - *Color* constants
- As with methods, these are accessed using the class name rather than any object name:
  - General: ClassName.STATIC_CONSTANT
  - Examples: Math.PI, Color.BLUE

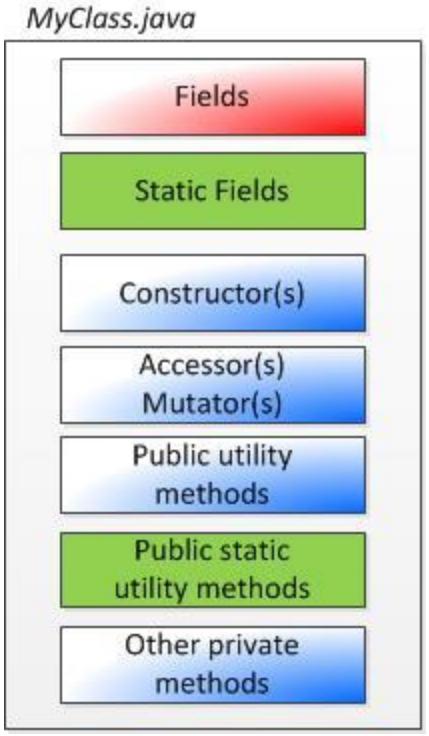
- But, doesn't this violate the usual principle of **data encapsulation**??
  - Usually: data is private, accessed by public accessors/mutators only
    - Intention is to prevent outside corruption of data
  - For static constants: the data is typically also ***final***
    - Means it cannot be changed by clients (so no need for a mutator/"set" method)
    - In this case, direct access to the data easier than calling an accessor/"get" method

Restrictions on Static

- ***static* methods**
  - Can only access other (class-level) *static* data
  - Cannot access any instance variables of the class (since no objects might yet exist)
  - Can only directly call (without intervening objects) other ***static*** methods
- The ***main()*** method
  - At program startup, no other objects exist yet
  - So to be truly standalone, *main()* must itself be ***static***
  - If you want to create any utility methods alongside your *main()* method, those methods must also be declared ***static***

Revised Class Structure

- There are effectively two “types” of data and methods
- Static methods can ONLY access other static data
  - No access to instance variable data, since none might yet exist
  - They can also call other static methods
- However, the converse is NOT true
  - Normal methods CAN access static data
  - It belongs to the CLASS and not to any one instance (object) of the class



The *Random* Class

- The ***Random*** class generates pseudorandom numbers
  - Appear random, but actually mathematically calculated
  - Not truly random, but "random enough" for our purposes
  - Located in ***java.util*** **package**, so it must be imported
- Instantiate a *Random* object, then call its **next…() methods** to get a "random" number
  - Multiple datatype forms exist: see the *Random* API

Random Usage

Random Class Constructor
`Random( )` Creates a random number generator.

Return type	Method name and argument list
`int`	`nextInt( int number )` returns a random integer ranging from 0 up to, but not including, *number*

- There are similar methods nextFloat(), next Double(), etc.
- Example: generate a random dice roll from 1-6

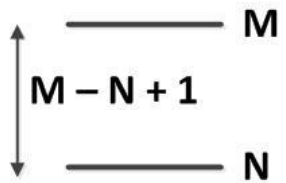
```
Random dice = new Random();
int  nextRoll  =  dice.nextInt( 6 ) + 1;     // 0-5 is offset to 1-6
```

Random Numbers Over a Specific Range

- Ref: textbook Sect. 6.9, pp.283-284
- To generate a random number over the specific range [N→M] (with N < M), the general algorithm is:

randObj.nextInt(M – N + 1) + N

(M-N) is the default range
add +1 to include the upper endpoint in a new range
offset the entire new range by +N



- Example:

```
int  minNum, maxNum, randNum;
Random rand = new Random();
minNum = 100;
maxNum = 200;
randNum = rand.nextInt(maxNum – minNum + 1) + minNum;
```

- See ***RandomExamples.java*** in **Example Source Code**

The *Math* Class

- The ***Math* class** provides:
  - Two *static* math **constants** (**E** and **PI**)
  - Multiple *static* **methods** for common math calculations
    - Book API, as usual, is the abridged version
    - ***Math* class API** gives full list of capabilities
- The *Math* class is in the ***java.lang* package**
  - Does not need to be imported, automatically available
- All the *Math* class methods are *static*
  - They are called as: **Math.methodName(argList)**
- Many of the methods are overloaded, and accept multiple input datatypes (including *double*)
  - This means they will accept <u>any</u> type of numeric argument
  - <u>All</u> input types can implicitly promote to *double*

The *Math* Class Constants

- Two *static* constants

  PI - the value of pi

  E - the base of the natural logarithm

- Example:

```
System.out.println( Math.PI );
System.out.println( Math.E );
```

- The output is:

```
3.141592653589793
2.718281828459045
```

- See ***MathExamples.java*** in **Example Source Code**

Some Methods of the *Math* Class

Return type	Method name and argument list
dataTypeOfArg	**abs(dataType arg)** returns the absolute value of the argument *arg*, which can be a *double, float, int,* or *long*.
double	**log(double a)** returns the natural logarithm (in base e) of its argument.
double	**sqrt(double a)** returns the positive square root of *a*
double	**pow(double base, double exp)** returns the value of *base* raised to the power *exp*

- See ***MathExamples.java*** in **Example Source Code**

The Math *round* Method

Return type	Method name and argument list
long	**round(double a)** returns the closest integer to its argument *a*

Rounding rules:

- Any factional part < 0.5 is rounded down
- Any fractional part >= 0.5 is rounded up

- See ***MathExamples.java*** in **Example Source Code**

The Math *min/max* Methods

Return type	Method name and argument list
<dataType of args>	min(dataType a, dataType b) returns the smaller of the two arguments. The arguments can be *doubles, floats, ints,* or *longs.*
<dataType of args>	max(dataType a, dataType b) returns the larger of the two arguments. The arguments can be *doubles, floats, ints,* or *longs.*

Find the smallest of three numbers, in stages:

```
int smallest = Math.min( num1, num2 );
smallest = Math.min( smallest, num3 );
```

Find the smallest of three numbers, in one operation:

```
int smallest = Math.min( Math.min(num1, num2) , num3);
```

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

Wrapper Classes

- **Wrapper classes** "wrap" the value of a primitive datatype into a corresponding object
  - For example: varInt (*int* variable) → objInt (*Integer* object)
  - Why?? Sometimes, API methods require an object argument rather than a primitive datatype argument
- Wrapper classes are frequently used to convert GUI input field data (*Strings*) into primitive types
  - Data from many GUI components are *Strings*, not numbers
  - Each wrapper class (other than *Character*) has a ***parse*…() method**, which converts a *String* into that primitive type
    - You can't safely convert a *String* into a *char*
- There are wrapper classes for each of the primitive types
  - The wrapper classes are in *java.lang*, so no need to import

Wrapper Classes

Primitive Data Type	Wrapper Class
double	*Double*
float	*Float*
long	*Long*
int	*Integer*
short	*Short*
byte	*Byte*
char	*Character*
boolean	*Boolean*

Integer and *Double* Methods

static Integer Methods

Return value	Method Name and argument list
int	**parseInt(String s)** returns the *String s* as an *int*
Integer	**valueOf(String s)** returns the *String s* as an *Integer* object

static Double Methods

Return value	Method Name and argument list
double	**parseDouble(String s)** returns the *String s* as a *double*
Double	**valueOf(String s)** returns the *String s* as a *Double* object

parse… Methods

- The **parse<Type> methods**
  - Often used to transform GUI or dialog input
  - GUI/dialog input is usually of *String* datatype
  - **Convert *Strings* to primitive types** for operations/calculations
- Example:

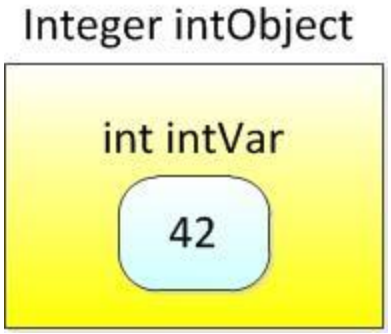
```
String strSalary;    // receives user input from GUI or dialog
double salary;

// let's say this got read in from a GUI or a popup dialog somehow
strSalary= "40000";                          // read from UI as a String
salary= Double.parseDouble(strSalary);  // now a double, ready for calcs
```

- See ***WrapperExamples.java*** in **Example Source Code**

Autoboxing and Unboxing

- **Autoboxing**
  - Automatic conversion: **primitive type → wrapper object**
  - Where a primitive type is used, but an object is expected
  - Example: Integer intObject = 42;



- **Unboxing**
  - Automatic conversion: **wrapper object → primitive type**
  - Where a wrapper object is used, but a primitive type is expected
  - Example: int intVar = intObject;
- Both are similar to automatic, implicit casting
  - Special Java support for converting between a primitive numeric type, and its wrapper class

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Assignments**
  - See slide 2 for new/current/past due assignments