Lecture 14: Input/Output

Sierra College
CSCI-12
Spring 2015
Mon 03/15/15

Announcements

- **General**
  - **Please make sure you've read the Thursday announcement on assignment due dates!**
    - Due dates have been extended to give you more time to complete and better UNDERSTAND the underlying material
    - Due dates extended, but now assignments are due as posted or zero credit (no late deductions)
    - Do not WAIT until lab time to read or begin assignments, start now
- **Schedule**
  - Midterm exam next week (Weds 3/25), before spring break
    - More details and a study guide later this week
- **Current assignments**
  - HW11: SimpleDate, due Friday 3/20 (*lab time Weds*)
  - HW12: Strings, due Friday 3/20 (*lab time Weds*)
  - HW13: Useful Classes, due Wednesday 3/25 (*lab time Weds*)
    - Based on today's material, see also posted example source code
- **New assignments**
  - HW14: Input/Output, due Wednesday 3/25
    - Based on Weds material, see also posted example source code

Lecture Topics

- **Last time**:
  - Finished up the *String* class and its API
  - Began talking about the Java API, packages, and importing classes
- **Today**:
  - Importing classes
  - Static classes
  - The ***Random*** class
  - The ***Math*** class
  - Wrapper classes

Motivations for I/O

- Earlier in this course, we considered and created programs which had hardwired (fixed) inputs
- Such programs are:
  - Useful for instruction
  - But, not terribly useful (or realistic) in real life
- We need some ways of getting user-specified input at run-time, and displaying back any results
  - This is the **input/output** (or **I/O**) problem

Types of I/O

- In general terms, there are multiple possible pathways for I/O in programs
- The actual means used by a program or app depends on the situation or program constraints
- Some I/O options (by no means an exhaustive list) include:
  - **Command line I/O** (stdin, stdout)
  - **Popup dialogs**
  - GUI windows
  - Web page front-ends
  - Touch screens
  - **File I/O**
  - Database I/O
  - Network I/O
- In this course, we will consider the means shown in **bold**
  - Other CS courses do/may here get into the other alternatives

I/O New Considerations

- I/O capability is great in a program, because it makes the program more general and more useful
- However, I/O now adds some new considerations:
  - Defensive programming
    - Checking that user inputs are proper, correct and complete
    - Making sure bad inputs won't crash a program
    - "Bulletproofing", "idiot-proofing", etc.
    - Use of logic, exception handling, etc.
  - Testing considerations
    - How do we adequately test anything that might reasonably (or unreasonably) get thrown at a program?
    - How do we know if we've tested our I/O capability "enough"?
    - Can our program safely handle bad data? Incomplete data? Out of bounds data?

Workhorse I/O Classes

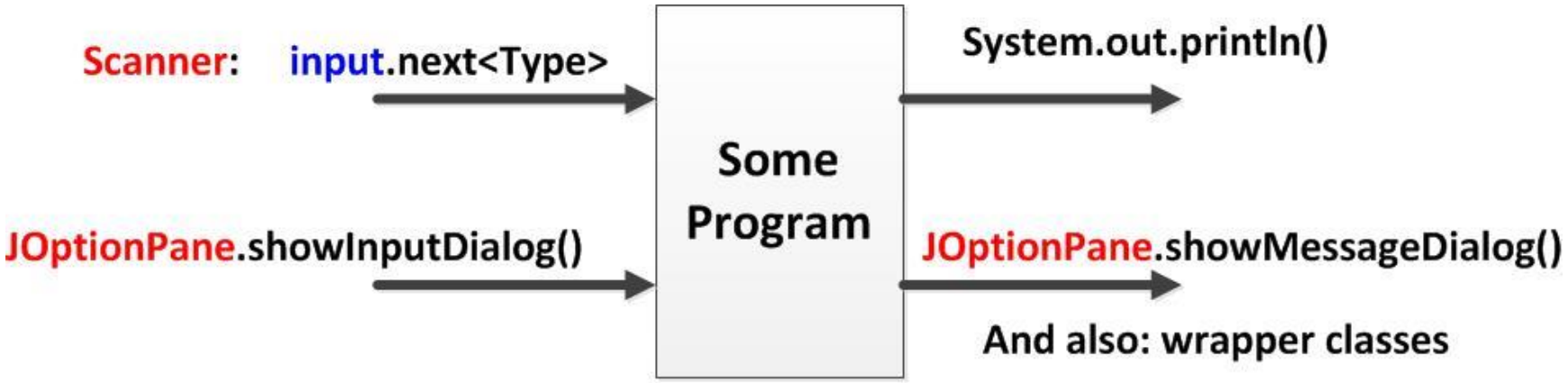
Scanner class

- **Object-based**: declare a new *Scanner* object, then use its methods
- In ***java.util*** package, must be imported
- Must provide prompt via System.out.print()/println()
- Reads user inputs via command line
- Inputs are already of primitive datatypes, or Strings

JOptionPane class

- **Class-based**: call *JOptionPane* class **static** methods directly
- In ***javax.swing*** package, must be imported
- Must provide prompt as argument to the method call
- Prompts user for inputs via popup dialog
- All outputs are *Strings*, must be type-converted first using **wrapper classes**

Our General I/O Capabilities



And also for any displayed Strings:

DecimalFormat: template.format(data)

The *Scanner* Class

- The ***Scanner*** class provides a means to read command line (console) user inputs
  - Located in *java.util* package, so must be imported
  - Tied to a data source *System.in*, which by default is the keyboard
- Using *Scanner*:
  - 4 steps:
    - **import java.util.Scanner;**
    - Declare and instantiate a *Scanner* object
    - Provide "meaningful" user prompts for each desired input
    - Use *Scanner* object's various ***next<Type>()*** methods to read data of an anticipated datatype
  - See app note "***User Input Using Scanner***" in Canvas

Scanner Constructor

Scanner Constructor
Scanner(InputStream dataSource) creates a *Scanner* object for reading from *dataSource.* The typical *dataSource* is ***System.in***, which instantiates a *Scanner* object for reading from the Java console (keyboard input)

Example:

```
Scanner scan = new Scanner( System.in );
```

scan is now our "input object", and we can reuse it for any types of command line inputs we want to prompt for and read

Scanner next...() Methods

Return type	Method name and argument list
dataType	**nextByte**( ), **nextShort**( ), **nextInt**( ), **nextLong**( ), **nextFloat**( ), **nextDouble**( ), **nextBoolean**( ) returns the next token in the input stream as a *dataType.* (no *char*: see upcoming example…)
String	**nextLine**( ) returns the remainder of the line as a *String*
String	**next**( ) returns the next token in the input stream as a *String*

Scanner Prompts

- Neither Scanner nor any of its next…() methods provides any user input prompts
  - So, it is the developer's responsibility to prompt for inputs of the anticipated type
  - If the input data does not match its next…() method, an ***InputMismatchException*** will occur

- Good user input prompts should be:
  - **On or near the same line** as their user entry
    - ***System.out.println()*** prints a newline, but ***System.out.print()*** does NOT
  - **Clear and specific** as to what is expected
    - Units? Allowable data range? Expected data type? (text? number?)
  - **Obvious** that user data entry is expected
    - Use a '>' or':' prompt symbol
    - Leave a trailing space for readability
  - Written **in user-centric language**
    - Most people are NOT programmers: What's a "String"??? (use "text" instead?)

Scanner Example

- Some things to note:
  - *import* statement before class (line 13)
  - *Scanner* setup is done one time (line 19), and *Scanner* object reused for all inputs
  - Same-line prompts using *print*() as opposed to *println*()
  - Input buffer clearing read using *nextLine*() (line 38)
  - Read of a single character using compound dot notation (line 48)
    - *scan.nextLine*() returns a *String*
    - then *charAt*() is applied to that *String*
    - **Reading the first char of a *String*:**

      oneChar = someStr.charAt(0);
  - Grouping of related statements, and commenting each group
- See ***ScannerUsage.java*** in **Example Source Code**

```
13 import java.util.Scanner;
14
15 public class ScannerUsage {
16
17     public static void main(String [] args) {
18
19         Scanner scan = new Scanner(System.in);
20         String firstName, fullName, tempStr;
21         int age;
22         char inputKey;
23
24         // prompt for and read first name
25         System.out.print("Enter your first name > ");
26         firstName = scan.next();     // reads next String token
27         System.out.println("Hello, " + firstName);
28
29         // prompt for and read age
30         System.out.print("\nEnter your age [in yrs], " + firstName + " > ");
31         age = scan.nextInt();   // reads next integer
32         System.out.println("I see you are " + age + " years old");
33
34         // sometimes needed to "consume" any residual
35         // input buffer characters: NN + <cr>
36         // if you get exception or missing text errors, this is often why
37         // try this example with and without the following line
38         scan.nextLine();  // reads input buffer, dumps it on the ground
39
40         // prompt for enemy annihilation go-ahead
41         System.out.print("\nTerminate all enemy intruders? [y/n] > ");
42
43         // reading one character in two individual steps
44         tempStr = scan.nextLine();
45         inputKey = tempStr.charAt(0);
46
47         // reading one character in one combined step
48         //inputKey = scan.nextLine().charAt(0);
49
50         System.out.println(firstName + ", your order is: " + inputKey);
51
52     } // end main
53
54 } // end class
```

How *Scanner* Works

- Scanner works by **tokenizing** an input stream
  - Default separators are the Java whitespace characters (space, tab, newline, etc.)
  - An input line can contain more than one token
- Scanner attempts to read a next token of the specified type
  - If the user's input does not match the datatype of the next *next…()* method call, an *InputMismatchException* occurs and execution stops
  - Sometimes, if there are residual characters remaining in an input buffer, an extra *nextLine*() may be needed to clear the input buffer (as in the prior example)

System.in and *System.out*

- ***System*** is an existing Java class in the *java.lang* package
  - Already available without needing to import it
  - Has a few familiar ***static*** constants and useful methods
    - One of these is ***in (System.in)***, another one is ***out (System.out)***
- ***System.in***
  - A *static* constant, represents the **standard input** device
  - By default, this is the **keyboard**
  - *in* by itself is a static object of the *InputStream* class
- ***System.out***
  - A *static* constant, represents the **standard output** device
  - By default, this is the **Java console (command line)**
  - *out* by itself is a static object of the *PrintStream* class

System.out Print Methods

Return type	Method name and argument list
void	**print**(anyDataType argument) prints *argument* to the standard output device ***WITHOUT*** a newline character
void	**println**(anyDataType argument) prints *argument* to the standard output device ***WITH*** a newline character

Example:

```
System.out.print( "The answer is " );
System.out.println( 42 );
```

Output:

```
The answer is 42
```

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

The *toString* Method

Return type	Method name and argument list
String	**toString**( ) converts the object's data to a *String* equivalent

- The purpose of the *toString* method is to return a *String* representation of its object's data
- All classes have an associated *toString* method
  - Usually, each class explicitly provides one
  - Otherwise, ultimately, its parent class or the Object class will provide one
- The *toString* method of a class is **called automatically (implicitly)** any time one of its objects is used with *print* or *println*
- The following outputs are identical:

```
SimpleDate today = new SimpleDate(3, 18, 2015);

System.out.println(today);            // prints      3/18/2015
System.out.println(today.toString()); // also prints 3/18/2015
```

Code Clarity For Extended Strings

- Remember that Java allows statements and strings to span multiple lines
  - New lines of text for multiple method arguments
  - String concatenations for lengthy strings
  - Embedded newlines (\n) or tabs (\t) in long strings
- Typical situations:
  - Long string concatenations
  - Methods calls involving multiple arguments
  - Lengthy user prompts
- Use the above fact to achieve better code clarity
  - Allow your code to “breathe” on the page
  - Allow your output to be more readable

Extended String Examples

```
// a multi-line user input prompt
Scanner input = new Scanner(System.in);
char userOption;
final String PROMPT_TEXT = "ACCOUNT OPTIONS:\n" +
                           "Add new account  [A]\n" +
                           "Update account    [U]\n" +
                           "Print account info [P]\n" +
                           "Delete account     [D]\n" +
                           "Quit program       [Q]\n" +
                           "Enter option >> ";
System.out.print( PROMPT_TEXT );
userOption = input.nextLine().toUpperCase().charAt(0);
```

```
// a lengthy output string within a method call
Person john = new Person();
System.out.println("Name = "  +  john.getName() +
                   "Age = "  +  john.getAge() +
                   "Birthdate = "  +  john.getBirthdate() );
```

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

The *JOptionPane* Class

- The ***JOptionPane*** class allows us to create popup GUI dialogs for either input or output
  - In the ***javax.swing*** package, so it must be imported to be used
  - This package contains many GUI classes, but *JOptionPane* can be used with GUIs or in standalone programs for simple I/O
  - Only simplest usages are discussed in text or in lecture
  - See Java API for more details on its full capabilities
- *JOptionPane* methods are **static/class methods**, meaning we call them using the class name
  - Provide a *String* user prompt
  - Returned data is of *String* type, so may need to be converted using the static wrapper classes (to numeric data)

JOptionPane static Methods

Return value	Method name and argument list
String	**showInputDialog**(*Component* parent, *Object* prompt) pops up an **input dialog box**, where *prompt* asks the user for input (usually a *String*) For standalone purposes, parent is *null*
void	**showMessageDialog**(*Component* parent, *Object* message) pops up an **output dialog box** with *message* displayed (usually a *String*) For standalone purposes, parent is *null*

Using the *JOptionPane* Class

- Setup:
  - Import **javax.swing.JOptionPane** (before the class definition)
- Usage:
  - Use directly in a program as a static class method
  - For the *Component parent* argument, simply use a null (this would be needed only if in some GUI context)
  - Provide a **prompt string** (for input) or a **display string** (for output)
- **Any return values are of the *String* datatype**
  - Any intended numeric values must be datatype-converted
  - Use the static wrapper classes' ***parse…()* methods**
  - If the return *String* cannot be converted to the expected datatype, some sort of exception will result

Example *JOptionPane* Usage

```
13 import javax.swing.JOptionPane;
14
15 public class DialogExamples {
16
17     public static void main(String [] args) {
18
19         String name, ageStr;
20         int ageNum;
21         final int DAYS_PER_YEAR = 365;
22
23         // prompt for name and age
24         name = JOptionPane.showInputDialog(null, "What is your name?");
25         ageStr = JOptionPane.showInputDialog(null, "What is your age?");
26
27         // show that age is a String, then convert it with a wrapper class
28         System.out.println("Length of input string " + ageStr + " is " + ageStr.length());
29         ageNum = Integer.parseInt(ageStr);
30
31         // show that age is now numeric, and create an output string
32         JOptionPane.showMessageDialog(null,
33                                       name + ", your approximate age is:\n" +
34                                       (ageNum * DAYS_PER_YEAR) + " days");
35
36     }
37
38 }
```

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

Formatting Output Text

- The ***DecimalFormat*** and ***NumberFormat*** classes:
  - Set up **display formats** for numerical values
  - Display formats are like templates, or patterns, or "data masks", for Strings created from that numerical data
  - We can use such formatted Strings in some output, just as we would any other substring
  - These 2 formats must be imported from the ***java.text*** **package**
- **Example situations**
  - Limiting decimal places in an output
    - A calculated price with sales tax of **$5.8371856…** becomes the expected **$5.84**
    - A calculated batting average **0.333333333333…** becomes the expected **0.333**
  - Currency symbols and comma grouping
    - A large currency amount of **10000000** might be standardized as **$10,000,000.00**
  - Percentage displays
  - Padding with leading zeroes
    - An ID number **4567** might become **00004567** for uniform alignment appearance

DecimalFormat vs. *NumberFormat*

- ***DecimalFormat***:
  - More direct control over formats, but a bit trickier to use
    - Nonetheless, we will probably prefer this one for this course
  - Declare a *DecimalFormat* object using *new* keyword
  - Define output format using available format symbols
  - Create formatted *String* using *format*() method
- ***NumberFormat***:
  - Less control over formats, can't do everything, but easier to use
  - Declare *NumberFormat* object directly WITHOUT using *new* keyword (uses *static* **factory methods**)
  - Output format is predefined by their **factory methods**
  - Create formatted *String* using *format*() method

More Advanced Formatting

- The ***DecimalFormat*** and ***NumberFormat*** classes are some "basic" ones for formatting the appearance of output
- For more industrial-strength control of output formats, Java also provides these:
  - **java.util.Formatter** class
    - Its API contains a detailed treatment of **format string syntax**
  - **System.out.printf()**
    - Similar to C/C++ printf() method
    - Again, it references the same **format string syntax**
  - The **String class format()** method
    - Converts a String object into that described by the attached format
    - Again, it references the same **format string syntax**
- *We will not cover these formats in this iteration of the course*
  - *(I need to work up some good examples for these…)*

DecimalFormat Class

DecimalFormat Constructor
DecimalFormat(String *pattern*) instantiates a *DecimalFormat* object with the format specified by *pattern*

Pattern characters:

0	required digit, include even if 0 (don't suppress)
#	optional digit, suppress if 0
.	decimal point
,	comma separator
$	dollar sign
%	multiply by 100 and display a percent sign

It is also possible to incorporate other text, such as currency symbols, Unicode characters, etc.

DecimalFormat format() Method

Return type	Method name and argument list
String	**format**(double number) returns a formatted *String* representation of *number*

- Example: we want to print out Euros, with two decimal places, and commas:

```
double bigMoney = 1234567.8;
DecimalFormat euroFormat = new DecimalFormat("#,##0.00" + "  " + '\u20AC');
System.out.println(  euroFormat.format(bigMoney)  );
```

Output becomes: 1,234,567.80 €

- Notice that:
  - The comma format is automatically extrapolated (extended) for larger numbers
  - The original data is UNCHANGED, only its String representation is changed
- See ***Formats.java*** in **Example Source Code**

NumberFormat Class

Return type	Method name and argument list
NumberFormat	**getCurrencyInstance**( ) *static* method that creates an object for printing numbers as money
NumberFormat	**getPercentInstance**( ) *static* method that creates an object for printing percentages
String	**format**(*double* number) returns a formatted *String* representation of *number*

- Presented only for completeness; we won't do much with this class in this course
- See ***Formats.java*** in **Example Source Code**

Formatting Examples

```
13 import java.text.DecimalFormat;
14 import java.text.NumberFormat;
15
16 public class Formats {
17
18     public static void main(String [] args) {
19
20         // batting average formats (percentages)
21         DecimalFormat batAvgFormatDF = new DecimalFormat("0.000");
22         DecimalFormat batAvgFormatDF2 = new DecimalFormat("##%");
23         NumberFormat batAvgFormatNF = NumberFormat.getPercentInstance();
24
25         // currency formats (currency)
26         DecimalFormat usdFormatDF = new DecimalFormat("$###,###.00");
27         NumberFormat usdFormatNF = NumberFormat.getCurrencyInstance();
28         DecimalFormat jpyFormatDF = new DecimalFormat("###,###,###.00" +
29                                                       "\u00A5");
30
31         // spy formats (leading zeroes)
32         DecimalFormat spyFormatDF = new DecimalFormat("000");
33
34         // program data and constants
35         final double PLACER_SALES_TAX = 0.075;
36         final double FOREX_USD_TO_JPY = 98.4045;
37
38         // program data
39         int hits = 100;
40         int atBats = 300;
41         float batAvg;
42         double carPrice = 34587.85;
43         double totalPrice, totalPriceYen;
44         int spyEmployeeId;
45         String spyName;
46
47
```

```
48         // batting average
49         batAvg = (float) hits/atBats;
50         System.out.println("batting average: " + batAvg +
51                            "    " + batAvgFormatDF.format(batAvg) +
52                            "    " + batAvgFormatDF2.format(batAvg) +
53                            "    " + batAvgFormatNF.format(batAvg));
54
55         // total cost in USD
56         totalPrice = (1.0 + PLACER_SALES_TAX) * carPrice;
57         System.out.println("\ncar price: " + carPrice);
58         System.out.println("price with tax: " + totalPrice +
59                            "    " + usdFormatDF.format(totalPrice) +
60                            "    " + usdFormatNF.format(totalPrice));
61         // total cost in JPY
62         totalPriceYen = totalPrice * FOREX_USD_TO_JPY;
63         System.out.println("price in JPY: " + totalPriceYen +
64                            "    " + jpyFormatDF.format(totalPriceYen));
65
66         // spy IDs
67         spyName = new String("James Bond");
68         spyEmployeeId = 7;
69         System.out.println("\nAgent: " + spyName + "\t" +
70                            "ID: " + spyFormatDF.format(spyEmployeeId));
71
72     }
73
74 }
```

```
----jGRASP exec: java Formats

batting average: 0.33333334    0.333    33%    33%

car price: 34587.85
price with tax: 37181.938749999994    $37,181.94    $37,181.94
price in JPY: 3658870.0917243743    3,658,870.09¥

Agent: James Bond        ID: 007

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

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

For Next Time

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