Input & Output (I/O)

General pattern for interactive I/O:
1. Prompt for input:
   ```java
   System.out.print("Enter your favorite whole number: ");
   ```
2. Read input from keyboard:
   ```java
   int fave = kb.nextInt(); // assumes existence of Scanner-type object kb
   ```
3. Echo back to user:
   ```java
   System.out.println("You entered "+ fave);
   ```

Console output methods:
- `print`: prints content of argument to screen, leaving cursor on same line
- `println`: prints content of argument to screen, moving cursor down one line when finished
- Example:
  ```java
  System.out.println("Testing"); // cursor moves to next line
  System.out.print(45); // cursor stays on same line
  System.out.print(56); // same line again
  System.out.println(73); // moves to new line after output
  System.out.print(49 + "\n"); // \n is the newline character
  System.out.println(); // prints blank line
  System.out.println("All done");
  ```

Output from example:
```
Testing
455673
49
All done
```

Formatted output to console:
- Limited amount of formatting available with `print()` and `println()`; can use horizontal tabs ("\t") and newline characters to create white space
- The `printf()` method is designed to do custom formatting:
  - Format is specified in the first argument to the method, the control string
  - Control string includes conversion (aka formatting) specifiers
  - Additional arguments provide data to be printed using the specified format; arguments are expressions that match the data type(s) of the format specifier(s), in the same quantity and order.
  ```java
  System.out.printf("%d", 49 + "\n");
  System.out.printf("%s", "All done");
  ```

- Syntax:
  ```java
  System.out.printf(control string, arg(s));
  ```
- Format specifiers:

<table>
<thead>
<tr>
<th>Data type: int</th>
<th>Data type: String</th>
<th>Data type: double</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>%s</td>
<td>%f (fixed-point notation)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%e (scientific notation)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%g (system chooses most compact notation)</td>
</tr>
</tbody>
</table>
Field width specification: field width is the amount of horizontal space provided for output; we can specify a field width by placing a whole number value between the % and the letter in specifier:

```
System.out.printf("%15s\n%15d\n", "Output", 999);  // example; output shown below:
```

```
Output
999
```

- Field width can be positive or negative; positive values right justify the output, negative values left justify
- If a field width is the same as or less than the number of output characters, the specification is ignored

Formatting floating-point numbers:
- In addition to field-width specifier, can incorporate a precision specifier in %f, %e and %g notation
- The precision is specified using a decimal point and a whole number between the % and the letter; if there is a field width already specified, the precision notation comes after the field width
- Examples:

```
System.out.printf("Pi = %5.2f\n", Math.PI);
System.out.printf("Pi = %20.5f\n", Math.PI);
System.out.printf("Pi = %.32f\n", Math.PI);
```

```
Output:
Pi =  3.14
Pi =              3.14159
Pi = 3.14159265358979300000000000000000
```

Console input methods:
- Methods from the Scanner class can be used to read input from the console. The method specifications are as follows:
  - `int nextInt()`
  - `double nextDouble()`
  - `String next()`
  - `String nextLine()`
- The first three of these methods all behave more or less the same way; each reads input up to the first delimiting (white space) character it encounters, ignoring any leading or trailing white space
- The `nextLine()` method is unique in that it reads most white space characters as data, stopping only when a newline character is encountered. This behavior has a downside: if previous calls to the other three methods left behind a newline character, a call to `nextLine` will read that character as end of data
- The clumsy workaround: make a dummy call to `nextLine` (call the method but discard any input) after every call to `next`, `nextInt` or `nextDouble`, as in the example below:

```
System.out.println(“Enter a number, a word and a sentence; hit enter after each”);
int x = kb.nextInt();
kb.nextLine();
String y = kb.next();
kb.nextLine();
String z = kb.nextLine();
```

Note: It's a bad idea to prompt for more than one data item at a time – it is much more reliable to prompt for, read, and echo one item, then move on to the next!
Because of this clumsiness, and because the numeric input methods are extremely sensitive to bad input (nextInt(), for example, will crash your program if given anything other than int data), a case can be made for using nextLine() as the only input method.

- Advantage: no need for dummy nextLine() calls
- Disadvantage: all data is read as Strings, and you can't do math on Strings

Converting String data to numeric types using wrapper class methods
- The Integer and Double classes serve as object surrogates (“wrappers”) for the primitive types int and double
- Among other useful methods, these classes include parsing methods that convert String arguments to int or double equivalents – example:

```java
System.out.print("Enter a whole number: ");
String input = kb.nextLine();
int x = Integer.parseInt(input);
System.out.print("Enter a floating-point number: ");
input = kb.nextLine();
double y = Double.parseDouble(input);
System.out.printf("The sum of %d and %f is %f\n", x, y, (x+y));
```

Input and Output in GUI environment
- The JOptionPane class from the javax.swing library contains static methods for reading and writing data in a windowing environment
- The method JOptionPane.showMessageDialog is roughly equivalent to System.out.print; the method JOptionPane.showInputDialog is somewhat like a combination of calls to print and nextline().

Example:
```java
String input = JOptionPane.showInputDialog(null, 
"Enter the size of the circle: ");
size = Integer.parseInt(input);
JOptionPane.showMessageDialog(null, "You entered " + size);
```

Output from example: