Java Console Input/Output
The Basics

Lecture 3
CGS 3416 Spring 2016

January 18, 2016
Console Output

- **System.out** – *out* is a *PrintStream* object, a static data member of class *System*. This represents standard output
  - Use this object to call functions *print*, *println*, and even *printf*
    - **print()** – converts parameter to a string (if not already one) and prints it out
    - **println()** – prints parameter, and also prints a newline after
    - **printf** – works like in C programming. Formatted string, followed by parameters to ”fill in the blanks”
Sample Calls

System.out.print("Hello, World"); // no newline
System.out.println("Hello\n\nWorld");
    // adds newline at end

int feet = 6, inches = 3;
System.out.printf("I am %d feet and %d inches tall\n", feet, inches);
    // just like printf in C
If the + operator is used with at least one string operand, then the operation is string concatenation.

Other types will be auto-converted to type string if needed

```java
System.out.println("The number of states in the U.S. is " + 50);
int sides = 8;
System.out.println("Number of sides on a stop sign = " + sides);
```
Formatting with printf

- When printing values with decimal precision it is often useful to be able to specify how many decimal places should be printed.
- The C-style `printf` function provides an easy way to format decimal precision.
- Format of `printf` calls:
  ```java
  System.out.printf(format string, list of parameters);
  ```
- The format string is a string in quotes, with special format symbols inserted:
  - `%d` specifies an integer
  - `%c` specifies a character
  - `%s` specifies a String
  - `%f` specifies a floating point type
- Consider the format symbols to be “fill-in-the-blanks” spots in the format string. These are filled in with the list of parameters.
printf Example

```java
int numStudents = 25;
char letterGrade = 'A';
double gpa = 3.95;

System.out.printf("There are %d students\n", numStudents);
System.out.printf("Bobby’s course grade was %c, and his GPA is %f\n", letterGrade, gpa);

// The output from this example is:
// There are 25 students
// Bobby’s course grade was A, and his GPA is 3.950000
```
To specify how many decimal places for the output of a floating point value, modify the ‘%f’ symbol in this format:

%.Nf // where N is the number of decimal places

Example:

double gpa = 3.275;
double PI = 3.1415;

System.out.printf("gpa = %.2f", gpa);
System.out.printf("PI = %.3f", PI);

// Output is:
// gpa = 3.28
// PI = 3.142
Before Java version 1.5.0, console input was harder. Since 1.5.0, we have the Scanner class.

Class Scanner is a text parser. Contains easy methods for grabbing different types of input.

System.in is an InputStream object that represents standard input.

To use Scanner to read from standard input:

1. Put the appropriate import statement at the top of the file:
   ```java
   import java.util.Scanner;
   ```
2. Create a Scanner object
3. Pass in System.in into the Scanner constructor, when creating the object.
import java.util.Scanner;
// yadda yadda

Scanner input = new Scanner(System.in);

// now we can use the object to read data from
  // the keyboard (stdin).
// Some sample calls:

int x = input.nextInt();
double y = input.nextDouble();
String s = input.next();
Different data types require the use of different methods available in the `Scanner` class. These are available only for primitive types and Strings.
For the following table, assume the `Scanner` object is called "in".

<table>
<thead>
<tr>
<th>Type</th>
<th>Reading in one variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td><code>byte b = in.nextByte();</code></td>
</tr>
<tr>
<td>short</td>
<td><code>short s = in.nextShort();</code></td>
</tr>
<tr>
<td>int</td>
<td><code>int i = in.nextInt();</code></td>
</tr>
<tr>
<td>long</td>
<td><code>long l = in.nextLong();</code></td>
</tr>
<tr>
<td>float</td>
<td><code>float f = in.nextFloat();</code></td>
</tr>
<tr>
<td>double</td>
<td><code>double d = in.nextDouble();</code></td>
</tr>
<tr>
<td>boolean</td>
<td><code>boolean bool = in.nextBoolean();</code></td>
</tr>
<tr>
<td>String</td>
<td><code>String st = in.next();</code></td>
</tr>
</tbody>
</table>
The Scanner class does not have a method for reading in characters.

However, there are several ways to read in a single character using the Scanner class.

One such way is to read in a String and then grab the first character.

```java
String st = in.next();
char c = st.charAt(0);
```
We usually want to take a peek into the Scanner buffer to see if an input token exists before we scan it in.

This is to avoid exceptions/errors that occur when we try to read in data that doesn’t exist or data of incompatible types.

Just like the reading methods, the Scanner class provides methods that check the Scanner buffer for data of a particular type.

All of these methods return a boolean, where ‘true’ indicates data exists.

These methods are very useful when we don’t know the length of the input and have to read indefinitely from the Scanner.
## Scanner methods to check for tokens

<table>
<thead>
<tr>
<th>Type</th>
<th>Checking for data of that type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>in.hasNextByte();</td>
</tr>
<tr>
<td>short</td>
<td>in.hasNextShort();</td>
</tr>
<tr>
<td>int</td>
<td>in.hasNextInt();</td>
</tr>
<tr>
<td>long</td>
<td>in.hasNextLong();</td>
</tr>
<tr>
<td>float</td>
<td>in.hasNextFloat();</td>
</tr>
<tr>
<td>double</td>
<td>in.hasNextDouble();</td>
</tr>
<tr>
<td>boolean</td>
<td>in.hasNextBoolean();</td>
</tr>
<tr>
<td>String</td>
<td>in.Next();</td>
</tr>
</tbody>
</table>