

# Section 1 - Introduction and Resources

## 3A.1.1 Overview

Last week we learned the basic rules of Java programs. We learned how to work with Integers (**ints**) and put some character **Strings** on the screen.

### User Input

It's now time to have a conversation with our user, so we will learn to ask the user questions and read answers into our program.

### Selection

We will also teach our programs how to "think." By that I mean that our programs will do one thing under certain circumstances, but something entirely different under a different set of circumstances: If it's raining, leave 15 minutes early and take an umbrella. If it's cloudy but dry, leave on time. If it's sunny, call in sick and go to the beach.

This is called *selection* and we will see it in the form of the *if* and *if/else* statements. If you haven't done so already, this week you will begin to feel as though you are really becoming a programmer.

It will be important that your computer and *Eclipse IDE* be able to handle *Java 6.0 or later*. This is standard in new versions of Eclipse, but if you are using an older version, you'll have to manually set it. See the handout Using Java Version 6 or Later in this course for details.

## 3A.1.2 Assigned Reading

As usual, read the week's modules and paste every code sample into your IDE so you can test and expand upon it. Do this *before looking at the assignment*. If you want more source material after reading the modules, look up any new terms in your textbook index, and see what it has to say about these things.

### Additional Module Readings

Review the Class Schedule , Syllabus and Style Rules briefly every week, so you are reminded of important rules and dates

## 3A.1.3 Resources for the Entire Course

Once again, I will remind you about optional resources:

- The *Elements of Java Style* listed in the syllabus.
- <http://docs.oracle.com/javase/7/docs/index.html>

The web page referenced above has many links and you can easily get lost. Once you are on that page, here are some of the most helpful places to look for answers:

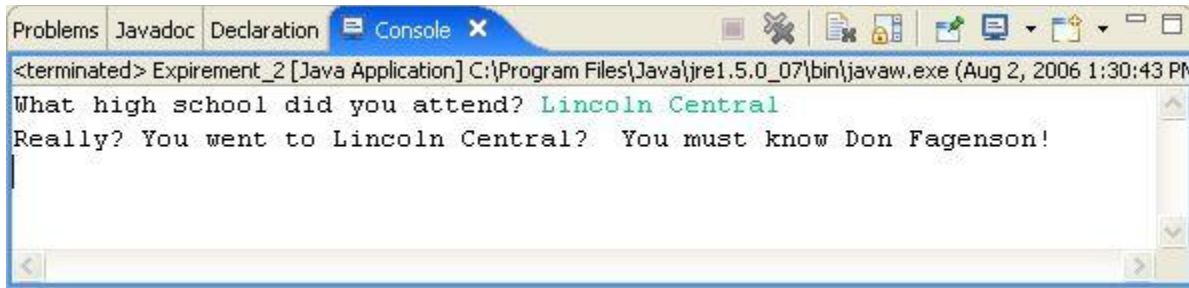
- **For the Java Language Technical Specification:** <http://docs.oracle.com/javase/specs/jls/se7/html/index.html>
- **For Information On The Built-In Java Classes:** [search for **String**, **Math**, **Integer**, etc. on this page then follow link]: <http://docs.oracle.com/javase/7/docs/api/index.html>
- **For Information on Swing:**  
<http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>
- **For Non-Swing GUI Classes:**  
<http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html>
- **For Common Utility Classes:** [search for Scanner, Data, Formatter, etc on this page then follow link]: <http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html>

## Section 2 - User Input from the Console

### 3A.2.1 Using `nextLine()` to Get User Input

Okay, are you ready to have a conversation with your user? The thing about that is you have to be ready. I was learning Spanish many years ago during which period I traveled to Madrid. I finally got the hang of expressing myself. The trouble was, when I asked questions, I got answers: quickly spoken, long, elaborate answers that I could not understand. We are in a similar situation. We can ask the user questions, but we have to make sure we know what to do with answers that the user gives us.

We'll start out working with Strings -- no numbers yet. Let's see how one gets input Strings from the user in the *console* environment. Here's what we are trying to accomplish:



To ask questions and place answers on the screen we will use the same output statement we already learned, **System.out.println()**, along with a newer version that prints a string *without* following it with a line feed. The new version is called **print()** (rather than **println()**).

To get answers from the user, we introduce the **Scanner** class which provides us with **next()** and **nextLine()** *methods*. Here we see the program acquiring the answer from the user and placing it into the string `strUserAnswer`:

```
// Declare a String reference to hold the user's answer
String strUserAnswer;

// ask the question and get the answer
strUserAnswer = inputStream.nextLine();
```

Notice that we do not need to *instantiate* a **String** object for `strUserAnswer`. In other words we do not need to do this:

```
strUserAnswer = new String(" ... ");
```

That's because, as we saw earlier, if the **String** reference is used on the *LHS* (left hand side) of the assignment operator, as it is in the `nextLine()` statement, a prior instantiation is not needed.

## Closing the Input Stream

When you are done with the input, or at some point before the program ends, you might want to add a line to close your input stream:

```
inputStream.close();
```

This will make a compiler warning go away, but is otherwise not essential.

## 3A.2.2 The Functional Return

You'll notice that in the above code fragment we were using a *method call* on the *RHS* (right hand side) of the assignment operator:

```
strUserAnswer = inputStream.nextLine();
```

We are *invoking* the **nextLine()** *function* and getting a *return value* from it. That return value is assigned to whatever **String** variable appears on the *LHS* of the = sign.

Using a *method invocation* to obtain a value like this is called *getting a functional return* from the method. That's because when the *method invocation* is complete, it will have *returned* some information back to our program. We say "the **nextLine()** method *returns* a **String** to the program."

Here is the technique in action.

```
import java.util.Scanner;

public class Experiment_2
{
    public static void main(String[] args)
    {
        // declare some string variables
        String strUserAnswer;
        String strQuestion;
        String strFinalStatement;

        // declare an object that can be used for console input
        Scanner inputStream = new Scanner(System.in);

        // Prepare a question for the user
        strQuestion = new String("What high school did you attend? ");

        // Show the user the question ...
        System.out.print(strQuestion);

        // ... and get a response
        strUserAnswer = inputStream.nextLine();

        // build up a final statement using the user's input
        strFinalStatement = "Really? You went to " + strUserAnswer + "? ";
        strFinalStatement = strFinalStatement + " You must know Don Fagenson!";

        // show the final output
        System.out.println(strFinalStatement);
    }
}
```

When the program starts, you may have to manually place the cursor in the console window before you start typing the answer to the question. If you start typing without doing that, you will be modifying your source with text, and you'll have to undo that text or your next build will have compiler errors.

## The Scanner Class

We need a new package, the **java.util** *package* to use the **Scanner** class. That's what the **import** statement, at the top of the program, is for.

In the program body, we declared and instantiated a **Scanner** object, **inputStream** (but we could have called it **scanner** or whatever we wanted). We use that object to get input from the console at run time using the **nextLine()** method.

### 3A.2.3 next()

If we want to only return individual words from the user input, rather than the entire line, we would have used **inputStream.next()** instead. The **next()** version returns the next **String** up to the first whitespace it comes across, which may be less than the entire line. So what would the program output look like if we used **next()** instead of **nextLine()**? (Assume that the user enters "Lincoln Central" as before and try to answer that question. Then modify that one line and run the program to see if you were right.)

### 3A.2.4 nextInt(), nextDouble, nextFloat()

I'm going to show you something, but you have to promise not to overuse it. You can get numbers directly into numeric variables, bypassing the **String** phase, using any of the three **Scanner** methods in the title of this section. For example, to get a **double** directly into a double variable, we do this:

```
double someNumber;  
  
// then later in the program ...  
someNumber = inputStream.nextDouble();
```

The same sort of move works for **floats** and **ints**. Here is the technique used in a full program.

```
import java.util.Scanner;

public class Foothill
{
    public static void main(String[] args)
    {
        // declare some variables
        double someNumber;
        String strQuestion;
        String strFinalStatement;

        // declare an object that can be used for console input
        Scanner inputStream = new Scanner(System.in);

        // Prepare a question for the user
        strQuestion = new String("Pick a number: ");

        // Show the user the question ...
        System.out.print(strQuestion);

        // ... and get a response
        someNumber = inputStream.nextDouble();

        // build up a final statement using the user's input
        strFinalStatement = "Your choice, " + someNumber
            + " was an excellent one.";
        strFinalStatement = strFinalStatement + "\nMy compliments";

        // show the final output
        System.out.println(strFinalStatement);
    }
}

/* ----- run -----
Pick a number: 3.1415927
Your choice, 3.1415927 was an excellent one.
My compliments
----- */
```

The reason this is not as useful as you might think is that, for multiple input values, it is easy for your program to get out-of-sync with the user, resulting in runaway or nonsense behavior. Therefore, when your program is any more complicated than the one above, I recommend always using `nextLine()` and converting the acquired strings to numbers using the conversion methods I have given you in module 2B.5.

We'll do some more user input examples next.

# Section 3 - Computing from User Input

## 3A.3.1 Doing Calculations with User Input

In earlier sections, we learned how to convert **String** data to numeric data with statements like:

```
numVar = Integer.parseInt(strVar); // convert String to int
```

We would do this, for instance, if we read a user response into the **String** object **str** and needed to convert it to a number in order to compute with it.

For instance, let's ask the user for two numbers and add them:

```
import java.util.Scanner;

public class Foothill
{
    public static void main(String[] args)
    {
        // declare some string and int variables
        String strUserInput;
        int num1, num2, answer;

        // declare an object that can be used for console input
        Scanner inputStream = new Scanner(System.in);

        // ask the user for the first number:
        System.out.print("Enter your first number: ");

        // get the answer in the form of a string:
        strUserInput = inputStream.nextLine();

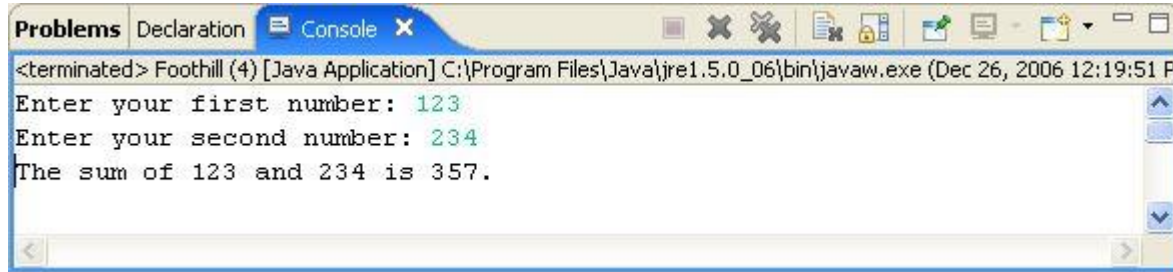
        // now convert it to a number so we can compute:
        num1 = Integer.parseInt(strUserInput);

        // now do it again (reusing the strUserInput variable):
        System.out.print("Enter your second number: ");
        strUserInput = inputStream.nextLine();
        num2 = Integer.parseInt(strUserInput);

        // add the two numbers (or try other operations if you wish):
        answer = num1 + num2;

        // show the final output
        System.out.println("The sum of " + num1 + " and " + num2 + " is " +
            answer + ".");
    }
}
```

Here is the output of the above program:



```
<terminated> Foothill (4) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 26, 2006 12:19:51 P
Enter your first number: 123
Enter your second number: 234
The sum of 123 and 234 is 357.
```

Or, more interestingly, we can ask the user for his age and tell him approximately his year of birth:

```
import java.util.*;

public class Foothill
{
    public static void main(String[] args)
    {
        // declare some string variables
        String strAge, strFinalStatement;
        int numAge, thisYear, numYear;

        // declare an object that can be used for console input
        Scanner inputStream = new Scanner(System.in);

        // ask the user his/her age
        System.out.print("How old are you? ");

        // get the answer in the form of a string:
        strAge = inputStream.nextLine();

        // now convert it to a number so we can compute:
        numAge = Integer.parseInt(strAge);

        // this is a fancy way to get the current year (you can just copy these
two lines):
        Calendar rightNow = Calendar.getInstance();
        thisYear = rightNow.get(Calendar.YEAR);

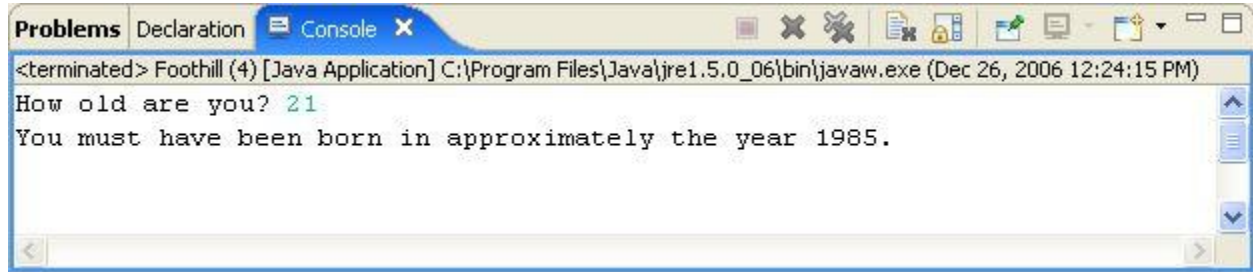
        // determine year of birth from age and current year
        numYear = thisYear - numAge;

        // we won't worry about the exact month detail here - just give an
approximate answer:
        strFinalStatement = "You must have been born in approximately the year
" + numYear + ".";

        // show the final output
        System.out.println(strFinalStatement);
    }
}
```



This time, the output looks like this:



This is a common and effective way to gather information from the user and use it to compute. One could attempt **nextInt()** or **nextDouble()**, instead, , but you will find that reading them as **Strings**, using **nextLine()**, then converting, has advantages.

## Section 4 Mixed Type Concatenation

### 3A.4.1 Revisiting Concatenation

I have been a little sloppy about presenting the **concatenation operator**, **+**. I said that it was a method of combining two **Strings** to make a larger string, as in:

```
String myName;  
myName = "Michael " + "J " + "Loceff";
```

Yet you have seen cases where I combined a String literal with a numeric variable using concatenation:

```
int someNumber;  
  
someNumber = -7;  
someNumber = (someNumber - 2) / 3;  
System.out.println( "the answer is " + someNumber );
```

What's going on? Well the concatenation operator always converts any numeric type such as **int** or **double** to a **String** before it glues it onto the other string. So the value in **someNumber**, namely **-3** is converted to a **String** **"-3"** first, and then the concatenation proceeds.

If one operand is a **String**, the other is converted to **String**.

By the way, **println()** requires a *String* inside the parentheses. If you try to **println()** a purely numeric value, you might get an error. In other words, if **someNumber** is an **int**, you may not be able to do this:

```
System.out.println( someNumber );    // Possible error!!
```

I say "may not" because Java versions are evolving, and while early versions didn't allow it, newer versions might. To be backward compatible, we assume the worst.

In those rare cases in which you need to print out numbers without any words (I will take off points if you try to do that in my class!) there is a trick you can use. *Concatenate* the number with the empty string, "", which will convert it to a *String* during the concatenation process:

```
System.out.println( "" + someNumber );    // Common trick
```

### 3A.4.2 Which + Do You Mean?

When we use the + operator with two numbers, it means add them! When we use the + operator with two strings, or one string and one number, it means concatenate them! Fine. But what about this?

```
long x, y;  
x = 2;  
y = 3;  
System.out.println( "The sum is " + x + y );
```

What gets printed out to the screen? Can you tell us why? Investigate.