Hi Folks, welcome to the CS 170, Java Programming 1 lecture on Loops, Jumps and Iterators.

In previous lessons, you learned how to use the for loop and while loop to repeat different kinds of actions. With the for loop, you relied on a counter, but with the while loop, you could control your loop with a counter or by examining some condition, like a sentinel value found in input, or the threshold condition used in a limit loop.

But, even though for loops and while loops are often used for different purposes, they do have one thing in common: both of them are top-tested, look-before-you-leap kind of loops. In this lecture, you’ll start by learning about loops designed for the impulsive, risk-taking programmer: the “hasty” do-while loop, for those of you who like to jump first, and check the effects afterward.

And, speaking of jumping, you’ll also take a look at Java’s unconditional flow-of-control statements, the break and continue.

Finally, we’ll finish up this lesson by looking at a practical loop application: using loops as iterators to read and process all of the data in a text file.

Ready? Let's get started.

In addition to the for and while loops, Java also has a loop that performs it's Boolean test after the loop body, instead of before. This loop, the do-while loop, illustrated here, will always execute the statements inside its body at least once. The body of the do-while loop appears between the keywords do (which precedes the loop body) and while. Like all loops, the body of the do-while loop can be a single statement, ending with a semicolon, or it can be a compound statement enclosed in braces. As with the other loops, you'll have a more fulfilling life if you train yourself to always use a compound (brace-delimited) loop body.

The do-while loop's Boolean condition follows the while keyword. As with all selection and iteration statements in Java, the Boolean condition must be enclosed in parentheses, which are part of the statement syntax. In the do-while loop, the Boolean condition is followed by a semicolon, unlike the while loop, where following the condition with a semicolon can lead to subtle, hard to find bugs.
When To Use Do-While?

• If you want to execute actions **before** making the test
  - Most of the time, you **don’t** want to do that!

When To Use Do-While?
Duration: 00:01:27
Advance mode: Auto

Notes:
The do-while loop is often employed by beginning
programmers, because, for some reason, it seems more
natural. If you find yourself in this situation, however, you
should think twice. 99% of the time, a while loop or a for loop
can be employed to better effect than using a do-while. In fact,
except for salting your food which should always be done
before tasting, I can think of very few situations where a test-
at-the-bottom strategy is superior to "looking before you leap."
Here’s an illustration:

At the dining table, Dogbert is quietly salting his food, when
Dilbert interrupts him: "You shouldn’t salt your food before
tasting it!"

Using impeccable logic Dogbert replies:

"It's a calculated risk. The average mouthful of food is five
percent of the total serving. So timid salters eat five percent of
almost every meal with to little salt. Because, only one time in
a thousand is food too salty to begin with.

Dogbert then finishes Dilbert off with the logical coupe de
grace, "Therefore, over a lifetime, you experience almost five
percent less salt-related happiness than I do."

Dilbert is obviously overmatched, but just to put up some kind
of token opposition, he replies, "No necessarily. I usually salt
my tongue after the first swallow."

Notes:
Let’s see how much work it is to convert a do-while loop into a
while loop. Open the file DoToWhile.java (in ic14) and run the
program. Notice that it uses a limit loop. Print me a screen-
shot of the result. When you’re done, I want you to rewrite the
program using a while loop instead. As you do, remember that
n, s, and x must have the same ending values. Show me a
screen-shot of your while version running correctly (showing
both your code and the console window.)
Jump Statements

- Jumps are "low-level" flow of control statements
  - In contrast to "high-level" statements like if and loops
- A jump statement is sometimes called a goto
  - Although Java, its not used
- Java does have goto statements
  - break: used to exit a switch or a loop (but not an if)
  - continue: used to prematurely "restart" a loop
- Java also has
  - Used to exit a particular nested loop

Notes:

Java's selection and iteration statements are called flow-of-control statements because they provide a highly structured method of conditionally and repeatedly executing a particular section of code.

In years past, however, before the control structures we use today, programmers still needed the equivalent of if statements and loops. Rather than relying on built-in language features, though, those iron-age programmers constructed their own flow of control statements, using the jump.

The jump statement often uses another name; even programmers who have never heard of the jump have heard of its nom de plum, the goto statement. The computer scientist Edsger Dijkstra, (shown here), is best known for his landmark paper published in the journal, Communications of the ACM, titled GOTO Considered Harmful. The paper, which pointed out the dangers of using the goto statement, and urged its retirement, ignited a storm of controversy and is often credited with starting the structured programming revolution.

The goto is one of Java's fifty reserved words, but it isn't actually implemented as part of the language, (like it is in C++ and C#). If your dream is to arbitrarily jump willy-nilly hither-and-yon through your code, you'll have to find another programming language to accommodate you. When it comes to loop control, however, Java hasn't abandoned the jump altogether. Java has two statements—break and continue—that let you do your jumping in a controlled fashion.

Java also has some specialized labeled versions of break and continue that can be used to exit the outer loop from a nested inner loop.

The break Statement

- break jumps out of a switch or loop (but not an if)
- In a loop, break jumps to the first statement following the loop body as shown here.
- Can make your code cleaner when used to construct a loop-and-a-half

Notes:

The break statement is used in two different circumstances, but both circumstances are quite similar. The break statement allows you to jump out of:

- a switch statement (as you saw in Chapter 5)
- a loop

When a break statement is encountered in the body of a loop, your program jumps to the statement immediately following the loop body, as you can see in the illustration shown here. Any remaining statements within the loop body are skipped. The test expression is not re-evaluated; the termination occurs regardless of the value of the test expression.

Although it can be misused, allowing you to write code that is as convoluted as any produced by the goto statement, the break can sometimes be used to make your loops simpler and clearer, especially when used to create a loop-and-a-half.

Your textbook discourages the use of the break statement as part of a loop, but I disagree. Research has shown that
students who use the break statement following the loop-and-a-half pattern shown below, understand the termination condition better and consistently write correct loops more often. To help you make an informed decision, I’ve placed an article by Eric Roberts of Stanford University on the class Web site. He makes a good case for using this style of loop. You don't have to read it, but you might find the information useful.

A "Loop-and-a-half"

- A loop-and-a-half is available in languages like Ada
- Very useful for "primed" loops with complex setup
  - In Java, use if with break to do the same thing
  - Put the necessary bounds in your loop condition
  - Put the intentional bounds in the if statement
- Exercise 2: Open BreakFor.java, locate comments
  - Write a loop that visits every character in the String
  - Extract the current character
  - If the current character is a period, then break
  - Otherwise, append it to the answer

Notes:

As you know, the built-in loops test either before the loop body, or after the loop body. Sometimes, however, its much more convenient to test inside the loop body. This is especially true when a primed loop requires many lines of code just to set up the loop condition. Some languages, like Ada, actually have this capability built in.

In Java, to create a loop and a half, you can use an if statement along with a Boolean flag (like you saw in the last lecture). Often, though, it's easier to use a break statement instead. To do this you:

- put the necessary bounds inside your loop condition. These are the conditions you need to check if your loop doesn't reach it's goal, like going beyond the end of string or coming to the end of a file.
- put the intentional bounds inside an if statement. If you meet the bounds condition in the if statement, then just use a break to exit the loop.

OK. Now let's see if you can put this into practice. Inside the ic14 project, open the file BreakFor.java. This is a Java applet that contains a loop designed to extract the first sentence (that is, all the characters up to a period) from a set of sentences. Locate the portion of the code that contains comments, and then add your code to:

- first, write a loop that visits every character in the String. This will be the intentional bounds, since you don't want to go beyond the end of the String and have your program crash. You can use a while loop, but a for loop is probably easier.
- Next, inside the body of the loop, use the charAt() method to extract the current character.
- If the current character is a period, then break. This is the intentional bounds of your loop.
- As the last statement in the loop body, simply append the current character on to the output variable.

When you've finished, and your code compiles, run it as an applet. Type a couple of sentences into the TextArea in the applet and then hit the Enter key. If you've written the code correctly, only the first sentence should be retained.

Shoot me a screen-shot of the applet window after you've hit
Another Loop-and-a-Half

Here’s a sentinel loop replaced by loop-and-a-half

```java
while (true) {
    System.out.print("Dollar value (Q to quit): ");
    String input = Std.in.next();
    if (input.equalsIgnoreCase("Q")) break;
    double dollar = Double.parseDouble(input);
    double euro = dollar * rate;
    System.out.printf("%.2f Dollar = %.2f Euros", dollar, euro);
}
```

Enter, and then paste a copy of your loop code immediately below that.

Notes:
Here’s another solution for the sentinel loop used in the CurrencyConverter program. (This is the third or fourth different version that we’ve looked at.) Since this loop doesn’t have a necessary bounds, the loop uses while (true) as the bounds condition, which is normally an infinite loop. The intentional bounds, the user typing in “Q” to quit, appears in the middle of the loop and allow the user to exit.

The continue Statement

- The `continue` statement only works inside loops
- Instead of leaving the loop, it starts the next iteration
- In `while` and `do-while` `continue` jumps to the loop test expression
- In `for`, it jumps to the update expression

Notes:
The `continue` statement is a jump statement like `break`, but unlike `break`, the `continue` jumps back to the loop condition, rather than jumping out of the loop. Exactly what that means depends upon which loop you’re using:

- For the `while` loop and the `do-while` loop, the `continue` statement jumps to the boolean test, skipping backward (while) or forward (do-while) as necessary.
- With the `for` loop, control jumps to the update expression.

Notice that this means that counter-controlled for loops and counter-controlled while loops behave differently when `continue` is involved. Because a counter-controlled while loop puts its update expression at the bottom of the loop body, when a `continue` is encountered, the counter would not be updated. In a `for` loop, it would be. That’s another reason to prefer the `for` loop whenever you use a counter to control the repetition.
Completing ContinueFor

Suppose you want to convert 1,234,456 to an int
- Can't use Integer.parseInt() because of commas
- You can write a loop to convert the number manually
  - Use the String's length as the loop bounds
  - Extract each character from the String
  - Use Character.isDigit() to see if it is a number
  - If not, then restart the loop using continue
  - If it is, then convert the digit into a binary number
  - Add it to the running total

Exercise 3: Complete and snap 2 pics (running/code)

Notes:
Suppose you want to convert a String variable containing a number like 2,345,567 into an int variable so that you can use it in a calculation. If you try to use the Integer.parseInt() method, like you've done in previous lessons, you'll find that it crashes because it doesn't know what to do with the commas.

All is not lost, though. You can simply write a loop that processes the String manually, throwing away the commas and then converting the remaining digits to a number. All you need to know is the "recipe" or algorithm for converting a String to an integer. Here it is:

1. Using a loop, step through each character in the String. You'll use the length of the String as the loop bounds.
2. Inside the loop body, extract each character from the String using the charAt() method. Store the character in a char variable.
3. Use an if statement to see if the character is a digit. You can do that by passing the character variable to the Character.isDigit() method. It will return true if it's a digit and false otherwise. When your character contains the comma, it would return false.
4. If the character is not a digit, then use the continue statement to jump to the next repetition in the loop.
5. Convert the digit character to a binary number by subtracting the character '0'. (Remember that '0' has a numeric (Unicode) value of 48 decimal, while 0 has a numeric value of 0. If you subtract '0' from '0', the result is a binary 0; if you subtract '0' from '1', the result is a binary 1, and so on.) Store this value in an int variable. This is the binary value for the current digit.
6. As each digit is encountered, you need to add it to a running total. Before you add it to the running total, though, you need to multiply the current value of the total variable by 10. Then, add the new digit.

For Exercise 3, open the file ContinueFor.java, and locate the comments in the messyStringToInt() method. I've already created the running total variable (named result). You'll just need to write the rest of the Java code to complete the program.

When you're done, type in a number with commas into the TextField and hit the Enter key. It should convert your number with commas into the actual integer value. Shoot me a screenshot of your program running. Below that, paste the code for the messyStringToInt() method that you wrote.
Nested Loops

- Loops can be **nested**, just like if statements
  - We call the nested loop the **inner loop**
  - We call the enclosing loop the **outer loop**

```java
for (int outer = 1; outer <= 10; outer++) {
    for (int inner = 1; inner <= 10; inner++) {
        System.out.printf("%4d", outer * inner);
    }
    System.out.println();
}
```

- **Exercise 4:** Open Dr. Java interaction pane. Type code above. Snap a screenshot.

Notes:

When you put one if statement inside the body of another if statement, you have a nested if statement. You can do the same thing with loops. As you'd expect, these are called nested loops. When you stick one loop inside another loop, the first loop encountered is called the outer loop, and the next is called the inner loop.

In the example shown here, the outer loop goes from 1 to 10. Each time that the outer loop repeats with a new value, the inner loop also goes from 1 to 10. That means that the entire loop goes around 100 times.

To see how this works, I want you to open the Dr. Java interactions pane and type the code shown above. When you're done, shoot me a screenshot for Exercise 4.

If you don't see the Dr. Java interactions pane, then Choose View from the Window menu. You'll probably need to select Other and then search for the Dr. Java pane.

Iterators

- Objects or methods that allow you to sequentially access all of the elements in a collection, regardless of how those elements are physically stored in memory
  - We'll look at some more sophisticated iterators later
  - For now, let's look at the **Scanner** as an iterator
- In addition to reading the keyboard, a **Scanner** can:
  - Read and process each line of text in a text file
  - Process each individual word or token in a line of text
- **Exercise 5:** Open **WordCounter.java**

Notes:

Now let's look at a practical application of loops. An iterator is an object that allows you to apply some operation to all of the members of a group of related variables, using a loop. Iterators provide a consistent and simple mechanism for systematically processing a group of items.

Such groups of related items or variables are known as collections, and Java has several different types. We'll take a closer look at the array and ArrayList collection types in the next Chapter. In this lesson, though, let's take a look at an iterator object you've already met: the Scanner class.

Up until now, we've used the Scanner class to read console input. The class is actually much more versatile than that, however. Using a Scanner as an iterator allows us to:

- *read and process each line of text in a file*
- *process each individual word in a line of text*

Before we process files, though, we'll try this with the text typed into an applet's TextArea. For Exercise 5, go ahead and open WordCounter.java and locate the countWords() method.
Counting Words

Step 1: Locate the `countWords()` method, and then construct a `Scanner` object using the `text` variable.

```java
Scanner textScanner = new Scanner(text);
```

- Note that the `Scanner` object will "step through" the `text` String, just like it steps through keyboard input.
- The `Scanner` sees both as a stream of characters.

Step 2: All iterators have a `hasNext()` method.
- Use that as the bounds of your loop.

Notes:

In the `countWords()` method you can see that I've already extracted the text from the `TextArea` and stored it in a `String` variable named `text`. I've also created an `int` variable named `numWords` to store the total number of words in the `TextArea`.

Start your code by constructing a `Scanner` object as shown here. Now up until now, you've only passed `System.in` to the `Scanner` constructor. Passing a `String` means that we'll treat its text as our input source.

Note that now, the `Scanner` object will step through the `String` named `text`, just like it stepped through the input you typed at the keyboard. As far as the `Scanner` object is concerned, both are simply streams of characters that it needs to iterate over.

Now, you're ready to write a loop. All iterators, and the `Scanner` is no exception, have a method named `hasNext()`. This method will return true if there is more input to read. Write a while loop using the `Scanner` object's `hasNext()` method to control it. In other words, you want to continue processing while your `Scanner` has more data to process.

Slide 13 🎨

Counting Words
Duration: 00:01:20
Advance mode: Auto

Counting Words

Step 3: Inside the loop, use the `next()` method to extract the item (word) that you want to process.
- As you read each word, increment the counter.

Exercise 5: Copy text of Moby Dick. Shoot a screenshot of the running program. Paste your code underneath that.

Notes:

Inside the body of your loop, you can extract the next word by using the iterator's `next()` method. In our case, of course, we don't do anything with the word we get back; we just count it.

So, as you read each word, add another statement to increment the counter named `numWords`. When the method is finished, we return the word count.

Once you have the program finished and it compiles, let's see how well it works. Down at the bottom of the ic14 project, I've placed a text file named "moby.txt" that contains the entire text of Herman Melville's classic book. Let's use our program to count the number of words that it contains.

Double-click the file to open it in Eclipse as a text document. Select and copy all of the text it contains (use `Control` + `A` in Windows), and then paste it into the `TextArea` in your running `WordCounter` program. Click the Count Words button, and, if you've done everything correctly, your program will count the words in Moby Dick as shown here. Show me a screenshot and then paste the code for your `countWords()` method right below it.

Slide 14 🎨

Counting Words
Duration: 00:01:27
Advance mode: Auto
Iterators and Text Files

• You can use a Scanner to read the lines in a text file
• Exercise 6: open the file LineCounter.java
  – Construct a Scanner like this:

  ```java
  Scanner reader = new Scanner(new File("moby.txt"));
  ```

• Use hasNext() and nextLine() to read text
  – Using next() would read next word instead
• As you read each line, increment your line counter
• Run, shoot a screen-shot. Paste your code.

Notes:

You can also use the Scanner class to iterate over the text in a file as well as over the characters in a String or the input from the keyboard. Let's see how that works. To do this, you're going to complete the program named LineCounter.

• Inside your main() method, construct a Scanner object like this. Instead of scanning a String in memory, this Scanner will open the file moby.txt and process it.

• After you've created your Scanner, notice the variable that holds the number of lines in the file. It is initialized to 0. Once the file is open, loop through each line of text by adding a while loop and testing the hasNext() method. Inside the body of your loop, call the nextLine() method, instead of the next() method. If you used the next() method, the program would read the next word from the file, just like WordCounter, rather than the next line.

• As you read each line, increment your line counter.

At the end of the file, the printf() statement prints out the number of lines in the file.

Once the program compiles, run it and make sure it counts the number of lines inside Moby Dick. Shoot me a screen-shot of the console window, and below that screen-shot, paste the code from your program.

Nested Loops: Words & Lines

• Exercise 7: create a program to count both lines and words named FileStats.java

Notes:

Let's do one last exercise that puts all of this together. I want you to create a program named FileStats that counts both the words and lines in Moby Dick. Of course you could do this just by running the two programs you've already written. How do you do it in a single program, though? You can use nested loops. The outer loop will run through the file, counting the number of lines. Inside the loop, you'll create a second Scanner to scan through the String returned by nextLine(), counting and accumulating the words that it contains.

Instead of modifying an existing program that I've started, you can make a copy of the LineCounter program, and then modify your copy. To make a copy:

2. On the context menu, select Copy.
3. Now, right click the ic14 project and
4. Click the Paste option on the context menu.
5. Because your project already has a LineCounter program, you'll need to supply a new name. Type in FileStats (not
FileStats.java) and

6. Click the OK button. Eclipse will make a copy of your file under the new name. It will also change all of the code references (such as the class name) that used to say LineCounter to FileStats. It won't change comments, though, so you'll have to modify those references to LineCounter yourself.

---

**Notes:**

Here are the steps you'll need to follow to complete your program.

- At the beginning of the file, change the comment so that it no longer refers to the LineCounter program.
- Find the variable that holds the number of lines and right next to it, create a second int variable to hold the number of words. Initialize it to 0 as well.
- Inside the body of the loop, right after the statements that read and count each line of text, construct a second Scanner object. You'll pass the String variable containing the line of text you just read from the file to this second Scanner constructor. This scanner will iterate over a String instead of over a file.
- Add a nested loop that uses hasNext() with the second Scanner object. In the body of this nested loop, use the next() method to read each word from the String. Update your word counter as each word is read.
- Finally, at the end of the program, modify the printf statement so that it displays the number of words as well as the number of lines. That means that you'll need to add another placeholder to the formatting string and pass the word-counter variable as an additional parameter.

When you're finished, run the program and shoot me a screen-shot of the output. Finish up this lecture by pasting your code right below the screen-shot.