Lecture 21:
Looping, Part I

Sierra College
CSCI-12
Spring 2015
Weds 04/15/15

Announcements

- **General**
  - Keep up with programs from here on out, they will build on each other
  - Next one assigned next week (you will only have ONE at a time)
  - Write/comment these programs well: they will become your exam notes for the final!
- **Schedule**
  - Spring withdraw deadline is Thurs 4/16
    - Final off-ramp: after that point, you will receive a letter grade for this class
    - Please check your grades in Canvas, and assess where you stand ("gut check")
      - Let's talk if any concerns…
- **Current assignments**
  - PRGM19: Age Utils (due Sunday 04/19 @ 11pm) lab time today
    - **Expectations**: follow ALL good software conventions (header, braces/indentations, commenting, naming conventions, etc.)
    - I've published the test program I'll use to grade your **UtilsFL.getAge()**
      - Your program's methods MUST run cleanly against this program!
      - Use this to CHECK your algorithm before submitting!

Lecture Topics

- **Last time**:
  - *if-else* and *switch* logic
- **Today**:
  - Anti-bugging and testing (end of last lecture)
  - Looping in general
  - Event-driven looping: the *while* loop

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Program**
  - Continue on the next assignment
  - Suggestion:
    - First, write your client class, using the *existing* version of the starter UtilsFL class
    - Then, update the readInt() method for Scanner/JOptionPane modes
    - Finally, work out the needed “age” logic
    - Test your program against the provided test driver program

Looping In Programs

- **Looping** in programming involves repeating the same set of instructions multiple times, but upon (usually) different data
- Looping may go by multiple terms
  - Looping, iteration, repetition, while-loops, for-loops, …
  - We will use the generic "looping"
- There are two general forms of looping in programs:
  - **Event-controlled looping** (***while*** loops, ***do-while*** loops)
    - Perform the same instructions, as long as a certain condition is true
    - We **DON'T** know ahead of times how many times we need to perform the instructions
  - **Count-controlled looping** (***for*** loops)
    - Perform the same instructions, for a proscribed number of times
    - We **DO** know ahead of time how many times to perform the instructions
- In order to implement looping, we need two things:
  - **Conditions**, the logic which tells us whether to continue
    - We've already seen conditions in the context of **selection**
  - **Looping structures**, which contain the statements to be executed each time

Looping Types Compared

- **Event-controlled looping** is used when we DON'T KNOW in advance how many loop iterations will be executed
  - Two types: ***while*** and ***do-while*** loops
  - Examples: read from keyboard input, read from file
- **Count-controlled looping** is used when we KNOW in advance exactly how many iterations will be executed
  - One type: ***for*** loops
  - Examples: giving the same raise percentage to all staff

Looping Example: The Grocery Cashier

- An example of a real-world looping scenario: a grocery cashier
- The SAME set of steps are performed for EACH customer encountered:
  - Reset the total cost to $0.00
  - As long as there is still an item on the belt, scan it and add it to the total
  - After all items are scanned, report the final total
- Something has to “happen” to tell us when we’re done
  - The divider bar (an “event”) signals the end of one customer’s order, and the beginning of the next customer’s order

Pseudocode for the Grocery Cashier

```
initialize total to $0.00
reach for first item
as long as item is not the divider bar {
    scan item
    add price to total
    reach for next item            // loop update
}   // end scanning items

// if we get to here, the next item was the divider bar
output the total price
```

What Is Event-Driven Looping?

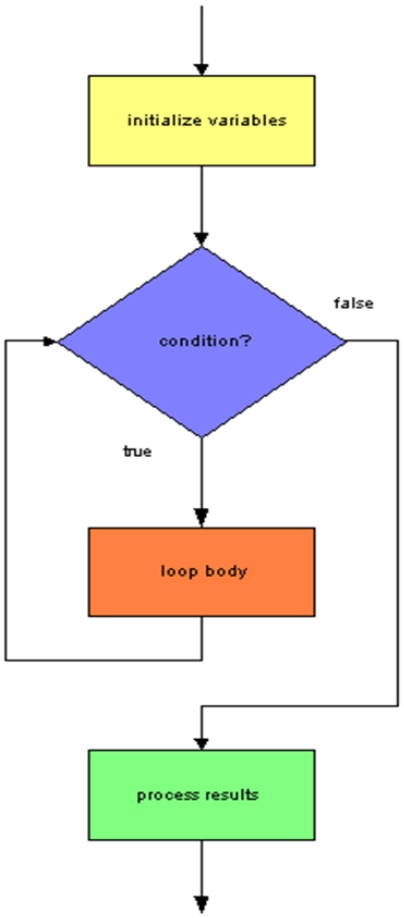
- **Event-driven looping** is another decision-making structure in code
  - Repeat the SAME set of instructions for EACH data input
  - But, we don't know in advance how many times we should do this: when are we "done"??
- An **event** is a signal, something that tells our code that we can STOP looping; for example:
  - A divider bar is encountered by the cashier
  - A special value is read (perhaps -1 or 'Q')
  - The end of a data file is reached
- We incorporate this event into the **condition** we check each time, to test whether or not to continue looping

Types of Event-Driven Looping

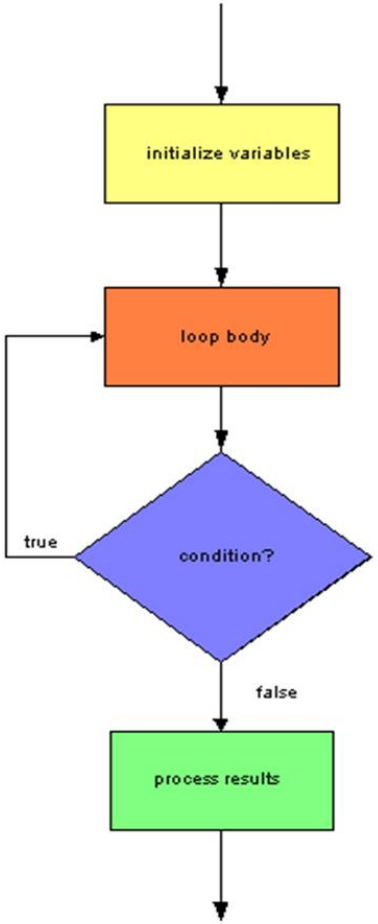
- There are two types of event-driven looping in Java
  - The ***while*** loop
  - The ***do-while*** loop
- Both types are very similar, with one key difference
  - ***while*** loop
    - Condition checking is done before beginning each loop
    - The statements in the loop body may or may NOT get executed
  - ***do-while*** loop
    - Condition checking is done at the END of each loop
    - The statements in the loop body are guaranteed to be executed at least once

Event-Driven Looping Comparison

***while* loop**



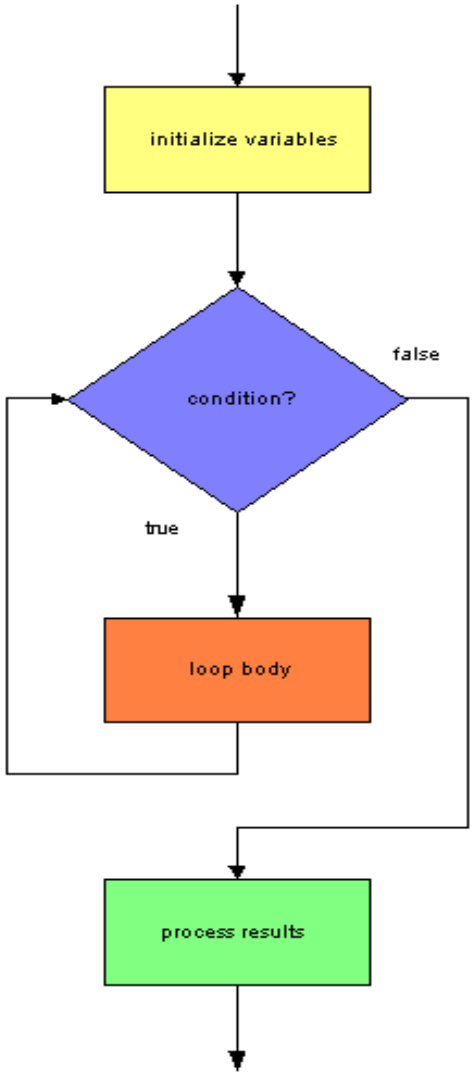
***do-while* loop**



Some Looping Terminology

- Before looking at *while* and *do-while* loops in detail, some terminology:
  - **Loop body**
    - The block of statements to be executed on each pass thru the loop
  - **Iteration**
    - One execution pass thru the statements of the loop body
  - **Priming read**
    - The initial read of data before an iteration, to get the loop underway
  - **Loop update**
    - One or more statements that could lead to the loop continuation condition to evaluate to false, and thus terminate the loop iteration
  - **Sentinel value**
    - A special value to help determine continuation/termination (i.e., 0, -1, -99, 'Q', etc.)
  - **Loop continuation condition**
    - Some *boolean* condition/expression which tells a loop to continue iterating
    - loopContinuationCondition = !(loopTerminationCondition)
  - **Loop termination condition**
    - Some *boolean* condition/expression which causes a loop to stop iterating
    - loopTerminationCondition = !(loopContinuationCondition)

while Flowchart and General Form




```
// initialize variables

while (condition) {
    // loop body:
    // statement(s) to execute if true
    // loop update statement(s)
}
// otherwise, go here if false

// process results
// continue with program…
```

while Loop Operation

- Initialize any data needed by the loop ("**priming read**")
- Evaluate the **loop condition**
  - If the loop condition is true:
    - Execute all the statements in the loop
    - Perform the **loop update** statement(s)
      - Give the loop condition a chance to evaluate to false
      - Sentinel value encountered? End-of-file encountered? Etc…
    - Re-evaluate the loop condition, and repeat
  - If the loop condition is false:
    - Terminate the loop iteration

Example: Sentinel-Controlled User Input

- Establish **sentinel value**
  - Value for which looping terminates
- Perform **priming read**
  - Outside and before the loop begins
- Check **loop condition**
  - Is user input NOT EQUAL to sentinel?
- Perform **update read**
  - Do this INSIDE the loop
- Perform the closing action(s)
  - Do this outside and after the loop

- See ***LoopingWhileSentinel.java*** in **Example Source Code**

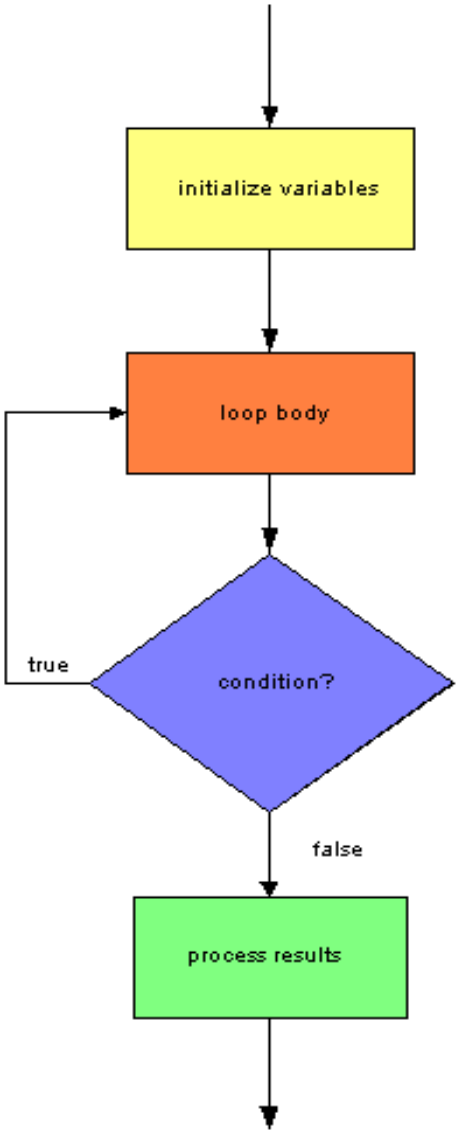
```
13 public class LoopingWhileSentinel {
14
15     public static void main(String [] args) {
16
17         // declarations
18         final int SENTINEL = -1;
19         String prompt = "Enter integer, or " +
20                         SENTINEL + " to terminate > ";
21         int num;
22
23         // priming read
24         num = UtilsFL.readInt(prompt);
25
26         // loop body
27         while (num != SENTINEL) {
28             System.out.println("You entered: " + num);
29
30             // update read
31             num = UtilsFL.readInt(prompt);
32
33         } // end while
34
35         // SENTINEL was encountered
36         System.out.println("Termination commanded, goodbye!");
37
38     } // end main
39
40 } // end class
```

```
----jGRASP exec: java LoopingWhileSentinel

Enter integer, or -1 to terminate > 4
You entered: 4
Enter integer, or -1 to terminate > 5
You entered: 5
Enter integer, or -1 to terminate > 42
You entered: 42
Enter integer, or -1 to terminate > -1
Termination commanded, goodbye!

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

do-while Flowchart and General Form




```
// initialize variables

do {
   // loop body:
   // statement(s) to execute if true
   // loop update statement(s)
} while (condition);
// otherwise, go here if false

// process results
// continue with program…
```

do-while loop operation

- Initialize any data needed by the loop ("**priming read**")
- Evaluate the loop body one time (guaranteed)
- Evaluate the **loop condition**
  - If the loop condition is true:
    - Execute all the statements in the loop
    - Perform the **loop update** statement(s)
      - Give the loop condition a chance to evaluate to false
      - Sentinel value encountered? End-of-file encountered? Etc…
    - Re-evaluate the loop condition, and repeat
  - If the loop condition is false:
    - Terminate the loop iteration

Example: Validating User Input

- This example uses the *do-while* loop to validate that user input is within a specified range [1-10]
- Re-prompt upon error is within the loop, so the condition must reflect the <u>invalid</u> case
- Also illustrates <u>loop counting</u>

```
13 public class LoopingWhileBoundsCheck {
14
15     public static void main(String [] args) {
16
17         // declarations
18         final int MINVAL = 1;
19         final int MAXVAL = 10;
20         int num;
21         int ntimes = 0;
22         String prompt = "Enter a number over the range [" +
23                         MINVAL + "-" + MAXVAL + "] > ";
24
25         // loop body
26         do {
27             if (ntimes == 1) {
28                 // make the prompt a bit more pointed if repeated
29                 prompt = "Seriously? " + prompt;
30             }
31
32             // update read
33             num = UtilsFL.readInt(prompt);
34             ntimes++;
35
36         } while ((num < MINVAL) || (num > MAXVAL));
37
38         // closing actions
39         if (ntimes == 1) {
40             System.out.println("Thanks, you entered " + num);
41         }
42         else {
43             System.out.println("After " + ntimes + " tries, " +
44                                "you FINALLY entered: " + num);
45         }
46
47     } // end main
48
49 } // end class
```

```
----jGRASP exec: java LoopingWhileBoundsCheck

Enter a number over the range [1-10] > 11
Seriously? Enter a number over the range [1-10] > -8
Seriously? Enter a number over the range [1-10] > 42
Seriously? Enter a number over the range [1-10] > 5
After 4 tries, you FINALLY entered: 5

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

See ***LoopingWhileBoundsCheck.java*** in **Example Souce Code**

Looping Applications

- Looping is a common component for many common programming applications
- Standard application patterns exist for examples such as the following:
  - Accumulation
  - Counting
  - Averaging
  - Finding min/max values
  - Animation (which we won't cover…)

Looping Application: **Accumulation**

- **Objective**: calculate a total
- **Approach**:
  - Set a total to 0
  - Read each value, add to total
  - Read all values in the loop
  - Total is complete when there are no more values to read (sentinel value detected)

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

// **pseudocode**:

```
Set total to 0;      // or else wrong answer!
Read a number;   // priming read

while (number != sentinel) {
   total += number;
   read next number;   // update read
}

Output the total
```

Looping Application: **Counting**

- **Objective**: count all items
- **Approach**:
  - Set a count to 0
  - Read a value (*optionally, also test its value*)
  - Increment count (*optionally, if it passes some test*)
  - Read all values in a loop
  - Count is complete when there are no more values to read (sentinel value detected)

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

// **pseudocode**:

```
Set count to 0;      // or else wrong answer!
Read a number;   // priming read

while (number != sentinel) {
   if (some test passes) {
      count++;
   }
   read next number;   // update read
}
```

Output the count

Looping Application: **Averaging**

- **Objective**: calculate an average
- **Approach**:
  - Combine accumulation AND counting
  - Set both total and count to 0
  - Read each value, add to total and update the count
  - Read all values in a loop until no more values to be read
  - Calculate average = total/count (guard against divide by zero?)

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

```
// pseudocode:

Set total to 0;    // or else wrong answer!
Set count to 0;  // or else wrong answer!
Read a number; // priming read

while (number != sentinel) {
   total += number;
   count++;
   read next number;   // update read
}

average = (double) total/count;  // casting!
Output the average
```

Looping Application: **Finding Minimum**

- **Objective**: find the minimum
- **Approach**:
  - (Finding maximum is similar)
  - Set initial minimum to FIRST value (NOT to some arbitrary constant)
  - Read each value, check against current minimum
  - Update current minimum if greater
  - Read all values in a loop until no more values to be read (sentinel value detected)

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

// **pseudocode**:

```
Read a number;   // priming read
Set minimum to first value

while (number != sentinel) {
   if (number < minimum) {
      minimum= number;
   }
   read next number;   // update read
}

Output the minimum
```

Loop Conditions

- For sentinel-controlled while loops:
  - **Loop termination condition**: (input == sentinel)
    - This is when we want to STOP looping
  - **Loop continuation condition**: (input != sentinel)
    - This is when we want to CONTINUE looping
  - So the above two conditions are inverses of each other:
    - (loop continuation) = !(loop termination)

Constructing Loop Conditions

- A *while* loop requires the condition under which looping is to **continue** (condition == *true*)
  - But often it is more "natural" to express the condition under which the looping is to **terminate** (condition == *false*)
- To systematically construct a proper *while* loop condition:
  - Define the logic for the loop termination condition
  - Define the continuation condition logic by **negating** the termination condition:

    (loop continuation) = !(loop termination)
  - Simplify the condition expression by **applying DeMorgan's Laws** where possible
  - See details on next slide(s)

Negating Expressions

expression	! (expression)
a == b	a != b
a != b	a = = b
a < b	a >= b
a >= b	a < b
a > b	a <= b
a <= b	a > b

DeMorgan's Laws:

- DeMorgan's Laws were introduced in the Conditions lecture
- The following expressions are logically equivalent:

  !(A && B) ←→ (!A) || (!B)

  !(A || B) ←→ (!A) && (!B)

- A and B are themselves simple conditional expressions

Incorrect Example: Menu Program

- Assume an application with case-insensitive keyboard user input
  - Either ‘S’ or ‘s’ will stop (end) the program
  - So we have TWO sentinel values: ‘S’ and ‘s’
- First guess at a continuation condition:

  while ((option != ‘S’) || (option != ‘s’))

  - If user enters ‘S’ → (option != ‘s’) is *true*
  - If user enters ‘s’ → (option != ‘S’) is *true*
  - If user enters any other character, condition is also *true*
- So this leads to an endless loop!
  - At least one of the expressions is ALWAYS *true*

Improved Menu Program

1. Define the **loop termination condition**:

   (option == ‘S' || option == ‘s‘) **(A || B)**

2. Create the **loop continuation condition** by applying the ! operator:

   ! ((option == 'S‘) || (option == 's‘)) **!(A || B)**

3. Simplify by applying DeMorgan's Laws:

   ((option != 'S‘) && (option != 's’)) **(!A && !B)**

   This condition is correct!

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

The Endless Loop Problem

- An **endless loop**, or an **infinite loop**, occurs when the loop body executes continuously without end
- Symptoms:
  - If no output: program appears to "hang"
  - If output: the output is "spewed" endlessly
  - In either case, the user must terminate execution somehow
- Possible causes ("gotchas"):
  - Putting a semicolon after the *while* condition
    - This creates an empty loop body
    - The subsequent {code block} (within braces) never gets executed
  - Forgetting to add an update read
    - The loop condition never gets a chance to become *false*

Endless Loop Examples

```
while (tempF >= 75.0) ;
// the above is an endless loop
// tempF never gets updated

while (itemCost > 0.0) {
   total += itemCost;
   // endless loop, "hangs"
   // because no update read
}
```

```
while (true) {
   // condition is always true

   // some statements…

   // this is a common structure
   // in embedded systems
   // in C/C++
}
```

To terminate any of these loops, you need to hit "End" in jGRASP's command window

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

Testing Techniques for *while* Loops

- Clean compilation and execution are a great start for code, but they're not the whole story
  - Does the program produce correct results with a set of known input values?
    - Check outputs against some (known) hand-calculated values
    - Check boundary values (highest and/or lowest expected values)
    - Check above/below/at the "edge" values of *while* statements
  - Does the program produce correct results if the sentinel value if the first and only input?
    - "Do exactly nothing, gracefully" - Donald Knuth, noted CS author
  - Does the program deal appropriately with invalid input?
    - Enter some invalid data
    - Are there any exceptions? Incorrect actions?