

STUDENT OUTLINE

Allows us to REITERATE (REPEAT) PORTIONS OF OUR CODE

Lesson 9 – while Loops

INTRODUCTION:

APPROACH WITH SOME CAUTION

In many situations, the number of times a loop will occur is dependent on some changing condition within the loop itself. The **while** control structure allows us to set up a conditional loop, one that occurs for an indefinite period of time until some condition becomes false. In this lesson we will focus on **while** loops with some minor usage of nested **if-else** statements inside. Later on in the course you will be asked to solve very complex nesting of selection, iterative, and sequential control structures. The **while** loops are extremely useful but also very susceptible to errors. This lesson will cover key methodology issues that assist in developing correct loops.

The key topics for this lesson are:

- A. The **while** Loop
- B. Loop Boundaries
- C. The **break** Statement Variations
- D. Conditional Loop Strategies

VOCABULARY:

while
break
STATE

SENTINEL
BOUNDARY

DISCUSSION:

A. The **while** Loop

"... THIS IS TRUE..."

1. The general form of a **while** statement is:

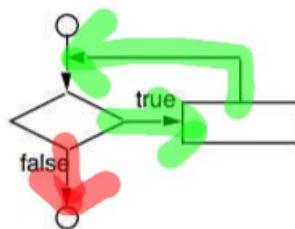
while (**expression**)
statement;

"... Do THIS."

- a. As in the if-else control structure, the Boolean expression must be enclosed in parentheses ().
 - b. The statement executed by the while loop can be a simple statement, or a compound statement blocked with braces {}.
- AN ENTIRE BLOCK OF CODE!**
2. If the expression is true the statement is executed. After execution of the statement, program control returns to the top of the **while** construct. The **while** statement will continue to be executed until the expression evaluates as false.

3. The following diagram illustrates the flow of control in a **while** loop:

while structure



- ENDS WHEN number IS !!*
- A.K.A.*
"THE Loop Body"
- "A.K.A.
"CONTROL"
or
counter"*
4. The following loop will print out the integers from 1-10.
- ```
int number = 1; // initialize
while (number <= 10) // loop boundary condition
{
 System.out.println(number);
 number++; // increment
}
```
5. The above example has three key lines that need emphasis:
- You must **initialize** the loop control variable (**lcv**). If you do not initialize **number** to 1, Java produces an error message.
  - The **loop boundary** conditional test (**number <= 10**) is often a source of error. Make sure that you have the **correct comparison** (**<, >, ==, <=, >=, !=**) and **that the boundary value is correct**.
  - There must be some type of increment or other statement that allows the **loop boundary** to eventually become **false**. Otherwise the program will get stuck in an **infinite loop**.
6. It is possible for the **while** loop to occur zero times. If the condition is **false** due to some initial value, the statement inside of the **while** loop will never happen. This is appropriate in some cases.

### B. Loop Boundaries

- The **loop boundary** is the **Boolean expression** that evaluates as **true** or **false**. We must consider two aspects as we devise the **loop boundary**:
  - It must **eventually become false**, which allows the **loop to exit**.
  - It must be **related to the task of the loop**. When the **task is done**, the **loop boundary** must become **false**.
- There are a variety of **loop boundaries** of which **two** will be discussed in this section.

**Student Answer:**  
**SEE NEXT PAGE**

3. The first is the idea of **attaining a certain count or limit**. The code in section A.4 is an example of a count type of bounds.

4. Sample problem: In the margin to the left, write a program fragment that prints the even numbers 2-20. Use a **while** loop.

**A&A. "FLAG"**

5. A second type of boundary construction involves the use of a **sentinel value**. In this category, the while loop continues until a specific value is entered as **input**. The loop **watches out for this sentinel value**, continuing to execute until this special value is **input**. For example, here is a loop that keeps a running total of positive integers, terminated by a negative value.

```
int total = 0;
int number = 1; // set to an arbitrary value
 // to get inside the loop
while (number >= 0)
{
 System.out.print ("Enter a number (-1 to quit) --> ");
 number = console.readInt();
 if (number >= 0)
 total += number;
}
System.out.println("Total = " + total);

a. Initialize number to some positive value.
b. The if (number >= 0) expression is used to avoid adding the sentinel
 value into the running total.
```

#### C. The **break** Statement Variations

1. Java provides a **break** command that forces an immediate end to a control structure (**while**, **for**, **do**, and **switch**).
2. The same problem of **keeping a running total of integers** provides an example of using the **break** statement:

```
ConsoleIO console = new ConsoleIO();
total = 0;
number = 1; /* set to an arbitrary value */

while (number >= 0)
{
 System.out.print("Enter a number (-1 to quit) --> ");
 number = console.readInt();
 if (number < 0)
 break;
 total += number; // this does not get executed if number < 0
}
```

**AAACK!**  
**DON'T USE break**  
**IN LOOPS!!**  
**BAD FORM!!**

```
x Undo Cut Copy Paste Find... Close
int number = 2; // initialize
while (number <= 20) // loop boundary condition
{
 System.out.print(number + " ");
 number += 2; // increment by 2
}

number = 1; // initialize
System.out.println();
System.out.println();
while (number <= 10) // loop boundary condition
{
 System.out.print(number*2 + " ");
 number++; // increment
}

number = 1; // initialize
System.out.println();
System.out.println();
while (number <= 20) // loop boundary condition
{
 if (0 == number % 2) // check for even
 {
 System.out.print(number + " ");
 }
 number++; // increment
```

- a. As long as (`number >= 0`), the `break` statement will not occur and `number` is added to `total`.
  - b. When a negative number is typed in, the `break` statement will cause program control to immediately exit the `while` loop.
3. The keyword `break` causes program control to immediately exit out of a `while` loop. This contradicts the rule of structured programming that states that a control structure should have only one entrance and one exit point. It is accepted for a switch/case structure which is not part of the AP subset and will not be covered in this course.

#### D. Conditional Loop Strategies

ndb

- 1. This section will present a variety of strategies that assist the novice programmer in developing correct `while` loops. The problem to be solved is described first.

Problem statement:

A program will read integer test scores from the keyboard until a negative value is typed in. The program will drop the lowest score from the total and print the average of the remaining scores.

- 2. One strategy to utilize in the construction of a `while` loop is to think about the following four sections of the loop: initialization, loop boundary, contents of loop, and the state of variables after the loop.
  - a. Initialization - Variables will usually need to be initialized before you get into the loop. This is especially true of `while` loops that have the boundary condition at the top of the control structure.
  - b. Loop boundary - You must construct a Boolean expression that becomes false when the problem is done. This is the most common source of error in coding a `while` loop. Be careful of off-by-one errors that cause the loop to happen one too few or one too many times.
  - c. Contents of loop - This is where the problem is solved. The statement of the loop must also provide the opportunity to reach the loop boundary. If there is no movement toward the loop boundary you will get stuck in an endless loop.
  - d. State of variables after loop - To ensure the correctness of your loop you must determine on paper the status of key variables used in your loop. This involves tracing of code, which is demanding but necessary.

A.K.A.  
"THE CONDITIONAL"

A.K.A.  
"THE loop BODY"

Post

3. We now solve the problem by first developing pseudocode.

Pseudocode:

```

initialize total and count to 0
initialize smallest to INT_MAX
get first score
while score is not a negative value
 increment total
 increment count
 change smallest if necessary
 get next score
 subtract smallest from total
 calculate average

```

?? GUARANTEES  
smallest will  
BE OVERWRITTEN

4. And now the code:

```

public static void main (String[] args)
{
 ConsoleIO console = new ConsoleIO();
 int total=0;
 int smallest = Integer.MAX_VALUE;
 int score;
 int count = 0;
 double avg;

 System.out.print("Enter a score (-1 to quit) ---> ");
 score = console.readInt();

 while (score >= 0) // loop boundary
 {
 total += score;
 count++;
 if (score < smallest)
 smallest = score; // maintain state of smallest
 System.out.print("Enter a score (-1 to quit) --> ");
 score = console.readInt(); // allows us to approach boundary
 }

 if (count > 1)
 {
 total -= smallest;
 avg = (double)total/(count-1);
 System.out.println("Average = " + avg);
 }
 else
 System.out.println("Insufficient data to average");
}

```

UPDATE THE  
STATE VARIABLE

5. Tracing code is best done in a chart or table format. It keeps your data organized instead of marking values all over the page. We now trace the following sample data input:

*SEE NEXT PAGE*

**USE THE DEBUGGER**

| score     | score >= 0 | total | count | smallest |
|-----------|------------|-------|-------|----------|
| undefined | undefined  | 0     | 0     | INT_MAX  |
| 65        | true       | 65    | 1     | 65       |
| 23        | true       | 88    | 2     | 23       |
| 81        | true       | 169   | 3     | 23       |
| 17        | true       | 186   | 4     | 17       |
| 45        | true       | 231   | 5     | 17       |
| -1        | false      |       |       |          |

When the loop is terminated the three key variables (*total*, *score*, and *smallest*) contain the correct answers.

6. Another development tool used by programmers is the concept of a state variable. The term state refers to the condition or value of a variable. The variable *smallest* maintains state information for us, that of the smallest value read so far. There are three aspects to consider about state variables:

- a. A state variable must be initialized.
- b. The state will be changed as appropriate.
- c. The state must be maintained as appropriate.

In the chart above, *smallest* was initialized to the highest possible integer. As data was read, *smallest* was changed only if a newer smaller value was encountered. If a larger value was read, the state variable did not change.

7. When analyzing the correctness of state variables you should consider three things.

Is the state initialized?  
Will the state find the correct answer?  
Will the state maintain the correct answer?

As first-time programmers, students will often initialize and find the state, but their code will lose the state information as the loop continues on. Learn to recognize when you are using a state variable and focus on these three parts: initialize, find, and maintain state.

8. Later on in the year we will add the strategy of developing loop boundaries using DeMorgan's law from Boolean algebra. This advanced topic will be covered in Lesson 18.

ActivInspire File Edit View Insert Tools Help

BlueJ: Terminal Window - L09 Sample

Enter a score (-1 to quit) ---> 87

Page5 - L09 Sample

Page5 X

Compile Undo Cut Copy Paste Find... Close

```
22 while (score >= 0) // loop boundary
23 {
24 total += score;
25 count++;
26
27 if (score < smallest)
28 smallest = score; // maintains the lowest score
29
30 System.out.print("Enter a score (-1 to quit) ---> ");
31 score = console.readInt(); // allows user input
32 }
33
34 if (count > 1)
35 {
36 total -= smallest;
37 avg = (double)(total)/(count-1);
38 System.out.println("Average = " + avg);
39 }
```

Threads main (at b...)

Call Sequence Page5.main

Static variables

Instance variables

Local variables

String[] args = <object>

ConsoleIO console = <object>

int total = 0

int smallest = 2147483647

int count = 0

int score = 87

STOP Halt Step Step Into

The screenshot shows the BlueJ IDE interface with several windows open. The terminal window displays the output of the program: 'Enter a score (-1 to quit) ---> 87'. The code editor window shows the Java code for calculating the average of user input scores. The debugger window on the right shows the current state of variables: args (String[]), console (ConsoleIO), total (int), smallest (int), count (int), and score (int). The score variable is currently set to 87. The debugger also shows control flow information like threads, call sequences, and step controls.

**SUMMARY/  
REVIEW:**

This lesson provides both syntax and strategies needed to build correct `while` loops. The terminology of loop construction will give us tools to build and debug conditional loops. We can use terms such as "off-by-one" errors or "failure to maintain state." This is a critical topic, one that takes much time and practice to master.

**ASSIGNMENT:**

Lab Exercise, L.A.9.1, *LoanTable*  
Lab Exercise, L.A.9.2, *FunLoops*