STUDENT OUTLINE

Lesson 9 – while Loops

INTRODUCTION: 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`	SENTINEL
`break`	BOUNDARY
STATE	

DISCUSSION:

A. The **`while`** Loop

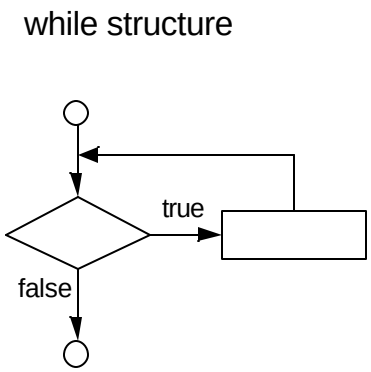
1. The general form of a **`while`** statement is:

```
while (expression)
    statement;
```

   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 {}.

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 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:



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:

   a. You must initialize the loop control variable (lcv). If you do not initialize `number` to 1, Java produces an error message.
   b. 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.
   c. 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 endless 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

1. The loop boundary is the Boolean expression that evaluates as true or false. We must consider two aspects as we devise the loop boundary:

   a. It must eventually become false, which allows the loop to exit.
   b. It must be related to the task of the loop. When the task is done, the loop boundary must become false.

2. There are a variety of loop boundaries of which two will be discussed in this section.

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.

Student Answer:

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

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.getInt();
  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.getInt();

  if (number < 0)
    break;

  total += number;  // this does not get executed if number < 0
}
```

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 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.

4. The **`switch`** structure (to be covered in a later lesson) will require the use of the **`break`** statement.

D. Conditional Loop Strategies

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.

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

4. And now the code:

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

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

  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.getInt(); // 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");
}
```

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:

   65 23 81 17 45 -1

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.

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*

LAB EXERCISE

LoanTable

Background:

When buying a home, a very important financial consideration is getting a loan from a financial institution. Interest rates can be fixed or variable and there are service charges called points for taking out a loan. One point is equal to 1% of the loan amount borrowed. Taking out a loan of $150,000 with a 2 point charge will cost you $3,000 before you ever make your first house payment! Some banks offer lower interest rates but higher points, and vice versa. It is helpful to know what the monthly house payment will be for a given loan amount over different interest rates.

The monthly payment on a loan is determined using three inputs:

1. The amount of the loan (principal).

2. The number of years for the loan to be paid off.

3. The annual interest rate of the loan.

The formula for determining payments is:

$$a = \frac{(p * k * c)}{(c - 1)}$$

p = principal, amount borrowed
k = monthly interest rate (annual rate/12.0)
n = number of monthly payments (years * 12)
$c = (1 + k)^n$
a = monthly payment (interest and principal paid)

Assignment:

1. Write a program that prompts the user for the following information:

    a. The amount of the loan
    b. The length of the loan in years
    c. A low interest rate in %
    d. A high interest rate in %

2. Print out the monthly payment for the different interest rates from low to high, incremented by 0.25%.

3. A sample run output is given below:

```
Mortgage problem

Principal = $100000.00
Time = 30 years
Low rate = 11%
High rate = 12%

Annual Interest Rate   Monthly Payment

     11.00                   952.32
     11.25                   971.26
     11.50                   990.29
     11.75                  1009.41
     12.00                  1028.61
```

4. Your program should make use of the built-in `pow` function located in the Math class.

5. Your program must make use of separate methods for the data input section and the printing section of the assignment.

6. Your program must use a **`while`** loop to solve the problem.

Instructions:

1. Write the program. Confirm that it works to the screen using the above sample output.

2. Solve 2 run outputs to the printer. Use the following sets of inputs.

Run 1:	Principal	187450.00
	Low rate	8.00
	High rate	12.00
	Years	30

Run 2:	Principal	12000.00
	Low rate	10.00
	High rate	12.00
	Years	5

LAB EXERCISE

FunLoops

Background:

1. Magic square problem:

   a. Some perfect squares have unique mathematical properties. For example, 36 is:

      - a perfect square, 6^2
      - and the sum of the integers 1 to 8 (1+2+3+4+5+6+7+8 = 36)
      - so let us call a "magic square" any number that is both a perfect square AND equal to the sum of consecutive integers beginning with 1.

   b. The next magic square is 1225:

      - $35^2 = 1225$
      - 1225 = sum of 1 to 49

   c. Write a method that prints the first n magic squares.

2. Reversing an integer problem:

   a. Write a method that reverses the sequence of digits in an integer value.

      - 123 ---> 321
      - 1005 ---> 5001
      - 2500 ---> 52 {you will not have to print out any leading zeroes, such as 0052}

3. Least Common Multiple problem:

   a. write a method that determines the Least Common Multiple of two integers. For example, the LCM of the following pairs:

2,3	LCM = 6
4,10	LCM = 20
12,15	LCM = 60
7,70	LCM = 70

Assignment:

1. Code separate methods to solve each problem.
2. Test each method using the following instructions.
3. You will need to work with long integers for problems 1 & 2.

Instructions:

1. Find the first four magic squares. The first one is the integer 1.

2. Solve these values for reverse the digits:

   12345 10001 1200 5

3. Find the LCM of the following pairs of values:

   15, 18
   40, 12
   2, 7
   100, 5

4. You may use the following form for the `main` method

```
public static void main(String[] args)
{
  FunLoops fun = new FunLoops();

  fun.magicsquare(4);
  System.out.println("12345 reversed ---> " + fun.reverse (12345));
  System.out.println("10001 reversed ---> " + fun.reverse (10001));
  System.out.println("1200 reversed ---> " + fun.reverse (1200));
  System.out.println("5 reversed ---> " + fun.reverse (5));
  System.out.println();
  System.out.println("LCM (15,18) = " + fun.lcm (15,18));
  System.out.println("LCM (40,12) = " + fun.lcm (40,12));
  System.out.println("LCM (2,7) = " + fun.lcm (2,7));
  System.out.println("LCM (100,5) = " + fun.lcm (100,5));
}
```

5. Try finding the first 10 magic squares. This is one of the few programs in this course that take so long that you can accurately time them. Experiment with this program by running it on computers with different clock speed (number of megahertz).