

## The while Statement, Types of Loops, Looping Subtasks, Nested Loops

### I. *The while Statement*

Note: Loop - a control structure that causes a sequence of statement(s) to be executed repeatedly. The **while** statement is one of three looping statements in Java (the others will be covered in later chapters). Note: Only the **while** is needed, the other two are conveniences of the language.

A) While syntax:

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

or

```
while (Boolean expression)
{
    statement1;
    statement2;
    etc.
}
```

```
ex.1) int count;
      count = 1;
      while (count <= 10)
      {
          System.out.println(count + " ");
          count = count + 1;
      }
      System.out.println("Bye");
```

```
ex.2) int count;
      count = 20;
      while (count <= 10)
      {
          System.out.println(count + " ");
          count = count + 1;
      }
      System.out.println("Bye");
```

B) Semantic explanation

If TRUE do the following statement or block of statements and then loop back to the while Boolean expression, continuing to do this until the expression is FALSE (see next line).

If FALSE skip the following statement or block of statements and go right to the statement following the semicolon or brace.

Note: Make sure you understand the difference between the logic of an **if** statement and a **while** statement. Also, know when does a **while** act just like an **if**.

Answer: When the **while** is initially FALSE.

## *Infinite Loops:*

```
ex.1) int count;
      count = 1;
      while (count != 10)
      {
          System.out.println(count + " ");
          count = count + 2;
      }
      System.out.println("Bye");

ex.2) int count;
      count = 1;
      while (count < 3) ; // note the semicolon
      {
          System.out.println(count + " ");
          count = count + 1;
      }
      System.out.println("Bye");
```

Note: If a loop continues forever it is called an *infinite loop*.

## II. *Types of Loops*

Note: There are two different types of loops by categorizing them according to how the loop is terminated (stopped): count-controlled loops and event-controlled loops.

A) Count-Controlled Loop - a loop that executes a known (predetermined) number of times. This loop uses a numeric variable called the loop control variable which is used to count the number of times a loop is executed.

```
ex.1) int count;
      count = 1;
      while (count <= 10)
      {
          System.out.println(count + " ");
          count++;
      }
      System.out.println("Bye");
```

There are three parts to a count-controlled loop:

- 1) initialize the variable
- 2) test the variable
- 3) increment the variable

## II. *Types of Loops* - continued

B) Event-Controlled Loop - a loop that terminates when something happens inside the loop body to signal that the loop should be exited. There are three different types of event-controlled loops: sentinel-controlled, end-of-file-controlled, and flag (Boolean) controlled. We will only be concerned with the first type, sentinel-controlled.

Note: A sentinel (or trailer) is a special, predetermined value that is inputted from the keyboard (or a file). It should be chosen with care. It should not be a normal data value.

1) Sentinel-Controlled loop: a loop that continues as long as the data value inputted is *not* the special (sentinel) value.

```
ex.1) int day, month;
      while (day != 0)
      {
          String input;
          input = JOptionPane.showInputDialog("Type a day then <OK>");
          day = Integer.parseInt(input);
          input = JOptionPane.showInputDialog("Type a month then <OK>");
          month = Integer.parseInt(input);
          etc.;
      }
      System.out.println("Bye");
```

(Recall: **import** javax.swing.JOptionPane; needed to do the above)

There is a problem with this example. What?

Answer: The value of `day` is undefined. We need to give `day` an initial value. To do this we use an input before the loop. This is called a *priming* read.

Note: Use `<` or `<=` instead of `!=` for the end of loop boolean test especially when comparing decimal numbers

```
ex.2)  int day, month;
input = JOptionPane.showInputDialog("Type a day then <OK>");
day = Integer.parseInt(input);
input = JOptionPane.showInputDialog("Type a month then <OK>");
month = Integer.parseInt(input);
while (day != 0)
{
    etc.;
    input = JOptionPane.showInputDialog("Type a day then <OK>");
    day = Integer.parseInt(input);
    input = JOptionPane.showInputDialog("Type a month then <OK>");
    month = Integer.parseInt(input);
}
System.out.println("Bye");
```

Note how the body of the loop has been changed. When using this type of loop, the processing should be done at the beginning of the body of the loop using the priming read data and then the new data input is at the end of the body of the loop.

### III. *Looping Subtasks*

Note: An iteration is an individual pass or repetition of a loop.

A) Counting (recall:  $x = x + 1$ ; and  $x++$ ; are the same thing.)

1) Iteration Counter: a counter variable that is incremented with each iteration of the loop

```
ex.1)  int count = 0;
char something;
something = JOptionPane.showInputDialog("Input something");
while (something != '?')
{
    count++;           //iteration counter
    something = JOptionPane.showInputDialog("Input something");
}
System.out.println("Bye");
```

How does the iteration counter compare to the number of times around the loop?

Answer: The same

Is the above example a counting loop?

Answer: No, it is a sentinel loop with a counter inside the body of it.

2) Event Counter: a variable that is incremented each time a particular event occurs

```
ex.1)  int count = 0;
        char something;
        something = JOptionPane.showInputDialog("Input something");
        while (something != '?')
        {
            if (something == '.')
                count++; //event counter
            something = JOptionPane.showInputDialog("Input something");
        }
        System.out.println("Bye");
```

Is the above example a counting loop?

Answer: No

How does the event counter compare to the number of times around the loop?

Answer: Not the same

```
ex.2)  int x = 2;
        int y = 0;
        while (x < 20)
        {
            x = x + 2; // loop control
            y = y + 1; // iteration counter
            System.out.println(x + " " + y);
        }
        System.out.println("Bye");
```

### III. Looping Subtasks continued

B) Summing Loop (accumulator or running total): Adding a value to a previous result. ex.) `sum = sum + number;`

```
ex.1) char gender;
int income;
int total = 0, count = 0, femaleCount = 0;
while (count < 100)
{
    gender = JOptionPane.showInputDialog("Enter M/F");
    String input = JOptionPane.showInputDialog("Enter income");
    income = Integer.parseInt(input);
    if (gender == 'F' || gender == 'f')
    {
        total = total + income;           // accumulator
        femaleCount++;
    }
    else
        System.out.println("Not a female.");
    count++;
}
System.out.println("Bye");
```

Is the above example a counting loop? Answer: Yes

What is femaleCount called? Answer: event counter

What is count++ called? Answer: iteration counter

### III. *Looping Subtasks* continued

C) Keeping track of a previous value: Remembering (storing) the next to last value inputted and the last value inputted i.e. being able to recall the previous value and the current value.

ex.1) Count the number of positive integers, negative integers, and zero's that have been entered, stopping when two zero's in a row have been inputted.

```
int previous, current;
int posCount = 0, negCount = 1, zeroCount = 0 ;
String input = JOptionPane.showInputDialog("Type a number");
previous = Integer.parseInt(input);
input = JOptionPane.showInputDialog("Type another number");
current = Integer.parseInt(input);
while (previous != 0 || current != 0)           //Why or and not and?
{
    if (current > 0)
        posCount++;
    else
        if (current < 0)
            negCount++;
        else
            zeroCount++;
    previous = current;
    input = JOptionPane.showInputDialog("Type a another number");
    current = Integer.parseInt(input);
}
System.out.println("Bye");
```

Note: There is a problem with the logic in the above example but not with the keeping track of a previous value or the **while** statement. What is it?

Answer: The count for the initial previous value is not done.

```
int previous, current;
int posCount = 0, negCount = 1, zeroCount = 0 ;
String input = JOptionPane.showInputDialog("Type a number");
previous = Integer.parseInt(input);
input = JOptionPane.showInputDialog("Type a another number");
current = Integer.parseInt(input);
if (previous > 0)
    posCount++;
else
    if (previous < 0)
        negCount++;
    else
        if (previous == 0 && current != 0)
            zeroCount++;
while (previous != 0 || current != 0)
    etc.

System.out.println("Bye");
```

### D) Loop and a Half

Sometimes termination condition of a loop can only be evaluated in the middle of the loop. Then, introduce a boolean variable to control the loop:

```
boolean done = false;
while (!done)
{
    Print prompt
    String input = read input;
    if (end of input indicated)
        done = true;
    else
    {
        // Process input
    }
}
```

## IV. *Nested Loops* - one loop inside another

ex.1)

```
int outsideCounter, insideCounter;
outsideCounter = 1;
while (outsideCounter <= 3)
{
    System.out.println("$$$$");
    insideCounter = 1;
    while (insideCounter <= 2)
    {
        System.out.println(" # # ");
        insideCounter++;
    }
    System.out.println( );
    outsideCounter++;
}
System.out.println("Bye");
```

### Memory Simulation:

<u>Output:</u>	<u>outsideCounter</u>	<u>insideCounter</u>
\$\$\$\$	1	1
# #		2
# #		3
	2	
\$\$\$\$		1
# #		2
# #		3
	3	
\$\$\$\$		1
# #		2
# #		3
	4	
Bye		

## AP Programming - Chapter 6 Lecture

How many times is the line `System.out.println(" # # ");` done? *Answer: 6*  
Is there a fast way of getting this answer by just reading the code?

ex.2) How many times is the line `System.out.println(" # # ");` done in code below?

```
int outsideCounter, insideCounter;
outsideCounter = 1;
while (outsideCounter <= 4)
{
    insideCounter = 1;
    while (insideCounter <= 6)
    {
        System.out.println(" # # ");
        insideCounter++;
    }
    outsideCounter++;
}
```

II. **for** loop - a loop that is used as a convenience to code count-controlled **while** loops.

A) It puts the initialization, the Boolean test condition, and the counter (increment) all in one (heading) line:

```
ex.1) int y = 0;
      int x;
      for (x = 1; x <= 5; x++)
      {
          y = y + 10;
          System.out.println(x + " " + y);
      }
```

```
ex.2) x = 1;
      y = 0;
      for ( ; x <= 5; x++)
      {
          y = y + 10;
          System.out.println(x + " " + y);
      }
```

What would happen if x were initialized to 6 in ex.2) above?

```
ex.3) x = 1;
      y = 0;
      for ( ; x <= 5; )
      {
          y = y + 10;
          System.out.println(x + " " + y);
          x++;
      }
```

Note: There is no semicolon after the counter, x++, nor after the **for** parenthesis.

Note: There must be two and only two semicolons, ;, in the **for** parenthesis.

B) Infinite Loop: **for** (x = 1; ; x++)

C) Null Statement: **for** (x = 1; x <= 5; x++); // Note the semicolon at the end.

ex.4) Scope

```
int y = 0;
for (int x = 1; x <= 5; x++)
{
    y = y + 10;
}
System.out.println(y);
System.out.println(x);
```

The above code does not work due to where x is declared. Fix by doing:

```
int x;
int y = 0;
for (x = 1; x <= 5; x++)
{
    y = y + 10;
}
System.out.println(y);
System.out.println(x);
```

ex.5) More Scope: Which statement below is illegal?

```
int i, j, k;
for (j = 1; j <= 10; j++)
{
    k = j;
    k = i;
    i = j;
    k = i;
}
```

Answer: the first `k = i;` statement since there is no value for `i` to assign to `k`

## III. **break** and **continue**

**break** statement - causes an immediate exit from the innermost loop or switch where it is located

**continue** statement - causes a branch to the bottom of the loop and the loop then continues if the logic dictates it to

What is the output below of the code fragments with inputs of: 3 6 0 9 -5 2

ex.1)

```
for (count = 1; count <= 5; count++)
{
    String input = JOptionPane.showInputDialog("Type a #");
    int number = Integer.parseInt(input);
    if (number <= 0)
        break;
    number = number + 2;
}
System.out.println(number);
```

**MEMORY:**  
count    number

ex.2)

```
for (count = 1; count <= 5; count++)
{
    String input = JOptionPane.showInputDialog("Type a #");
    int number = Integer.parseInt(input);
    if (number <= 0)
        continue;
    number = number + 2;
}
System.out.println(number);
```

**MEMORY:**  
count    number

IV. **do while** - a looping structure in which the loop condition (Boolean expression) is tested at the end (bottom) of the loop called a posttest condition.  
(Note: Not an AP topic)

A) Syntax:

```
do
    statement;
while (Boolean expression);    // Note the semicolon at the end of this while!
```

or

```
do
{
    statement1;
    statement2;
    etc.
} while (Boolean expression);    // Note the semicolon at the end of this while!
```

Note: The above **while** could have been one line lower but by convention that is where to put it.

Note: A **do while** loop is always done at least once as compared to **while** or **for** loops (which may not loop at all).

Note: There is no need for a *priming read* in a **do while** loop.

ex.1a)

```
int n = 0;
do
{
    String input = JOptionPane.showInputDialog("Type a number");
    n = Integer.parseInt(input);
    n = n + 2;
    System.out.println (n + " ");
} while (n != -9999);
```

ex.1b)

```
int n = 0;
String input = JOptionPane.showInputDialog("Type a number"); // priming read
n = Integer.parseInt(input);
while (n != -9999)
{
    n = n + 2;
    System.out.println(n + " ");
    String input = JOptionPane.showInputDialog("Type a number");
    n = Integer.parseInt(input);
}
```

ex.2)

```
int n = 0;
do
{
    String input = JOptionPane.showInputDialog("Type a number");
    n = Integer.parseInt(input);
    if (n == -9999)
        break;
    n = n + 2;
    System.out.println(n + " ");
} while (n != -9999);
```

ex.3) What is wrong with the following?

```
int x, y;
y = Stdin.readInt( ); // this is a Ready shortcut for keyboard integer input
while (y > 10)
{
    x = 20;
}
System.out.println(x);
```

Answer:  $y > 10$  may be false, therefore the statement  $x = 20$ ; may never be done thus causing the computer to say error.

Solution: Make it a **do while** loop.

## V. *Nested Loops*

ex.1)

```
*****  
*****  
****  
***  
**  
*
```

```
for (x = 6; x >=1; x- -)  
{  
    for (y = 1; y <= x; y++)  
    {  
        System.out.print("*");  
    }  
    System.out.println( );  
}
```

ex.2)

```
for (a = 1 ; a <= 3 ; a++)  
{  
    for (b = 1 ; b <= 2 ; b++)  
    {  
        for (c = 1 ; c <= 4 ; c++)  
        {  
            System.out.print("#");  
        }  
        System.out.println( );  
    }  
    System.out.println( );  
}
```

MEMORY:  
a    b    c

OUTPUT:

### VI. *Random Numbers*

- A) Random Numbers using the static **random( )** method found in the Math class (new AP topic in 2006-2007)

Math.random( ) produces a *pseudorandom* random decimal number between 0 (inclusive) and n (exclusive)

The following line would generate a random number between *Big* number and *Small* number inclusive:

**(int)**(Math.random( )\*(*Big* - *Small* + 1)) + *Small* or  
**(int)**(Math.random( )\*(*Big* - *Small* + 1) + *Small*)

- ex.1) Generate a random number between 5 and 10 inclusive

Note 1: *Big* is the larger number, 10 in this example

Note 2: *Small* is the smaller number, 5 in this example

Note 3: the formula needs a typecasting of **int**

Answer: **(int)**(Math.random( )\*(10 - 5 + 1) + 5) or **(int)**(Math.random( )\*(6) + 5)

- ex.2) generate a random number between 15 and 25 inclusive and store in x

Answer: **int** x = **(int)**(Math.random( )\*(11) + 15);

- ex.3) **(int)**(Math.random( )\*(100) + 10) will produce a random number between what two numbers?

Answer: between 10 and 109 inclusive (work: 100 = last - 10 + 1)

- B) Random Numbers using the Random Class (AP topic)

The Random Class (found in **import** java.util.Random; ) has two methods in it that can be used to generate *pseudorandom* random numbers:

nextInt(n) - a random integer between 0 (inclusive) and n (exclusive)

nextDouble( ) - a random floating-point number between 0 (inclusive) and 1 (exclusive)

The following two lines would generate a random number between *Big* number and *Small* number inclusive:

Random generator = **new** Random( );

**int** x = generator.nextInt(*Big* - *Small* + 1) + *Small*;

- ex.1) generate a random number between 1 and 5 inclusive and store in x:

Random die = **new** Random( );

**int** x = die.nextInt(6) + 1

### VII. *String Tokenization (nonAP topic)*

There is a `StringTokenizer` class that allows several of items of data to be typed all on one line separated by white spaces and then one data item at a time can be extracted for use by using the `nextToken()` method. Another useful method in the `StringTokenizer` class is `hasMoreTokens()`

```
ex1.) String input = JOptionPane.showInputDialog("Enter data:");
StringTokenizer ex1 = new StringTokenizer(input)
while (ex1.hasMoreTokens( ))
{
    String token = ex1.nextToken( );
    System.out.println(token); // or do something else with each token
}
```

### VIII. String Class (automatically imported as part of the java.lang package)

#### A) Instantiating (declaring) a string

\*Note: A string is really an object of the String class

Method 1: String x; x = **new** String( ); x = "hello there";

Method 2: String x; x = "hello there";

Method 3: String x = "hello there";

#### B) String Class Methods

1) toLowerCase( ) - returns the string as all lower case letters

2) toUpperCase( ) - returns the string as all upper case letters

3) length( ) - returns the length of the string including spaces etc.

4) concat(stg2) - makes one longer string from the two

5) compareTo( ) – see last Chapter

6) charAt(#) – extract an individual character, at position #, from a string and convert it to a **char** type  
(remember: the starting position in strings is 0)

ex.1)

```
String string1 = "WHEATON";
```

```
String string2 = "warrenville";
```

```
String string3 = "Wheaton Warrenville South";
```

```
string1.toLowerCase( )           result: wheaton
```

```
string2.toUpperCase( )          result: WARRENVILLE
```

```
string3.length( )              result: 25
```

```
string1.concat(string2)         result: WHEATONwarrenville
```

#### *Traversing Strings (nonAP topic)*

In the String class is a method that will extract an individual character from a string and convert it to a **char** type.

```
ex1.) String myString = "abcd";  
    for(int x = myString.length( ) - 1; x > 0; x--)  
    {  
        char ex1 = myString.charAt(x);  
        System.out.println(ex1);  
    }
```

Output: dcba

### IX. Shortcut Console Input

Our lab book author has created a console input shortcut. In the *hsa* package is a class called `Stdin`

```
Import java.io.IOException;
import hsa.Stdin;
public class ex
{
    public static void main (String[ ] args) throws IOException
    {
        int x;
        System.out.println("Type in an integer then <Enter>");
        x = Stdin.readInt( );
        System.out.println("Your input is: " + x);
    }
}
```

The line `x = Stdin.readInt( );` could be:

<b>double</b> input = Stdin.readDouble( );	if an double number is to be input
<b>char</b> input = Stdin.readChar( );	if one character is to be input
<b>boolean</b> input = Stdin.readBoolean( );	if <i>true</i> or <i>false</i> is to be input
String input = Stdin.readString( );	if one word is to be input
<b>int</b> input = Stdin.readLine( );	if a string (multiple words) is to be input

**\*\*Note:** When trying to read a numeric value but the data has nonnumeric characters in it the computer will throw an exception error. The same thing happens when trying to parse data that has nonnumeric characters. (More about throwing exceptions in a later chapter.)

### Chapter 6 Terminology:

loop - A control structure that causes a sequence of statements to be executed repeatedly is called this.

priming read - An input statement located before the **while** loop that processes its input data.

loop exit - This is the moment that repetition of the loop body ends and control passes to the first statement following the loop.

termination condition - The condition that causes a loop to be exited.

iteration - This is an individual pass through, or repetition of, the body of a loop.

count-controlled loop - A loop in which the number of repetitions is known in advance.

event-controlled loop - A loop that terminates when something happens inside the loop body to signal that the loop should be exited.

sentinel-controlled loop - An event-controlled loop whose event is the input of a special value.

flag - An event-controlled loop that uses a Boolean (TRUE/FALSE) variable to record its event.

iteration counter - A counter variable that is incremented with each iteration of a loop.

event counter - A counter variable that is incremented each time a particular event occurs.

pretest - the general name given to a loop in which the loop test is positioned before the loop body.

posttest - the general name given to a loop in which the loop test is positioned after the loop body.