Repetition Structure

While loop

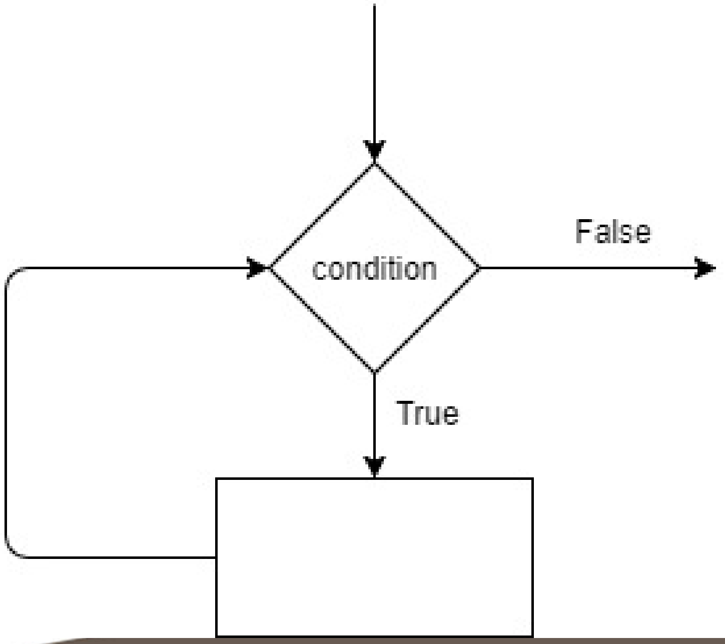
CAMT
College of Arts, Media and Technology
Chiang Mai University

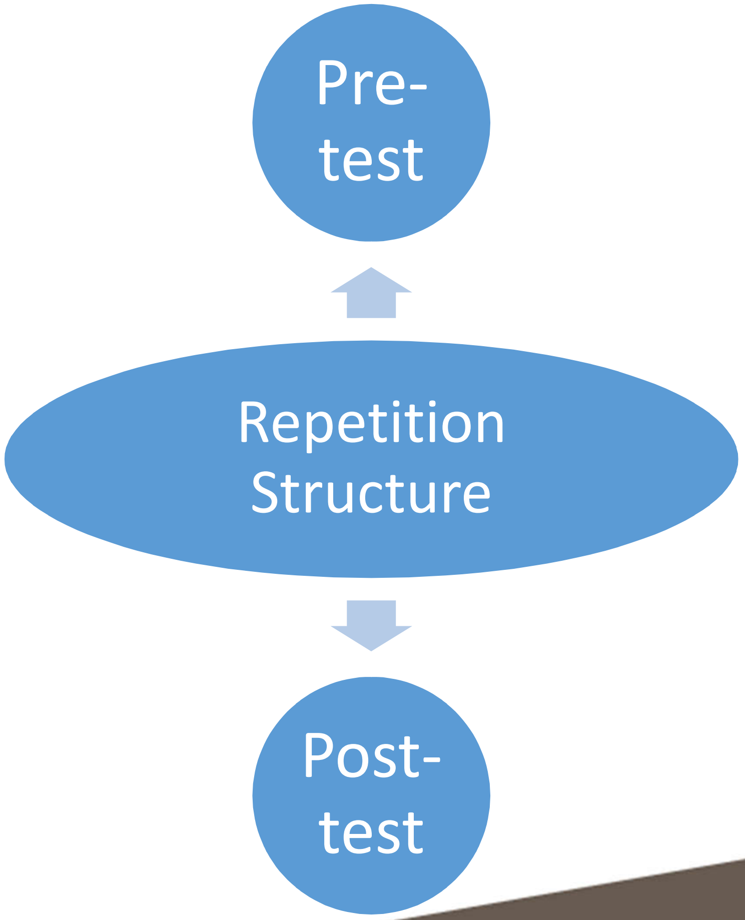
Repetition

- Doing the same statement
- Until some thing has been checked

Repetition Structure

- A *repetition* structure represents part of the program that repeats
- This type of structure is commonly known as a *loop*.
- The flowchart segment below shows a repetition structure expressed in Java as a *while* loop.





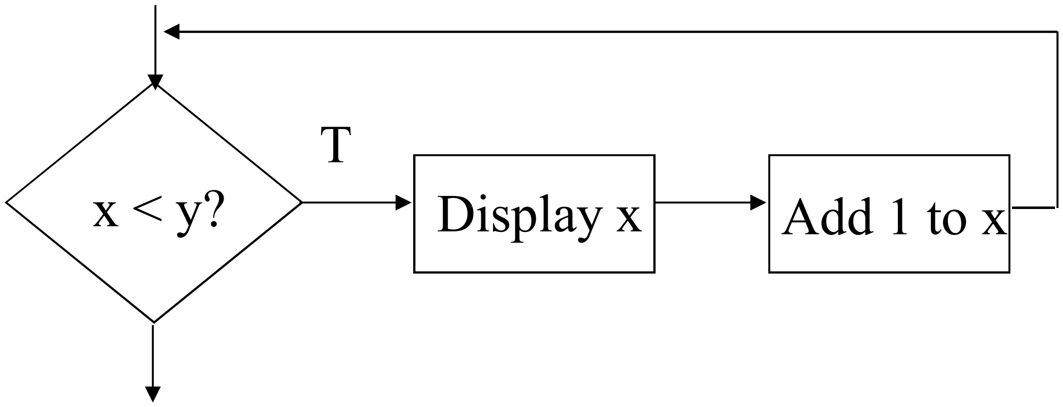
Pre-test

Repetition Structure

Post-test

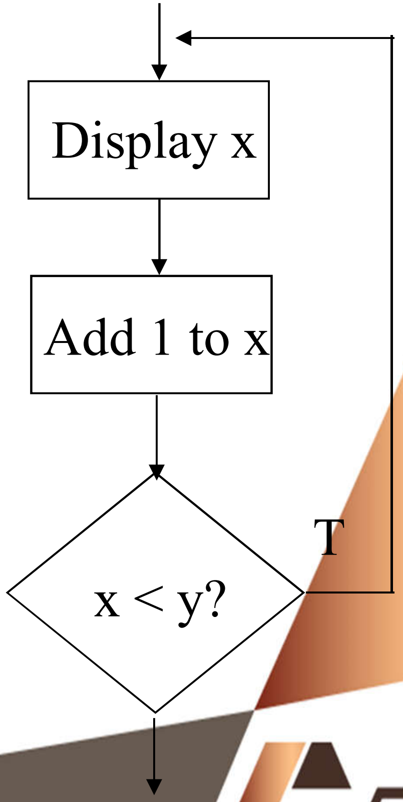
A Pre-Test Repetition Structure

- This type of structure is known as a *pre-test repetition* structure
- The condition is tested *BEFORE* any actions are performed.

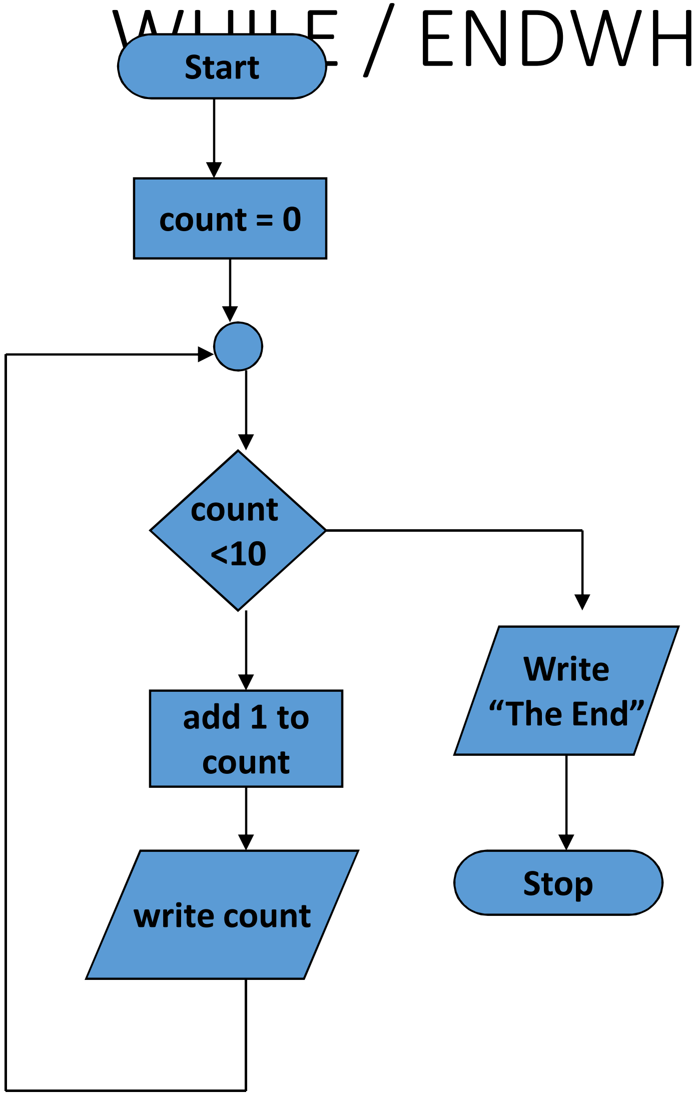


A Post-Test Repetition Structure

- This flowchart segment shows a *post-test* repetition structure.
- The condition is tested *AFTER* the actions are performed.
- A post-test repetition structure *always* performs its actions at least *once*.



The Pre test structure WHILE / ENDWHILE



```
count = 0
WHILE count < 10
        ADD 1 to count
        WRITE count
ENDWHILE
WRITE “The End”
```

Mainline

```
count = 0
WHILE count < 10
        DO Process   ← Modular
ENDWHILE
WRITE “The End”
```

Process

```
ADD 1 to count
WRITE count
```

The while Statement

- We can write a program that adds exactly 5 integers and a different application that sums exactly 50 integers
-
- But, can we write a program flexible enough to add 5 integers, 50 integers, 50,000 integers or even 50,000,000 integers?

The while Statement

```
while (condition){
        statement 1;
        statement 2;
        statement 3;
        ….
        statement n;

}
```

The while Statement

- The following segment performs addition of 50 numbers with just a few lines of code
- There is nothing special about 50 and we can just as easily add 500,000 numbers:

```
int sum = 0;
int count = 0;
while(count < 50)
{
     sum = sum + input.nextInt();
     count++;
}
System.out.print(“Sum is “ + sum);
```

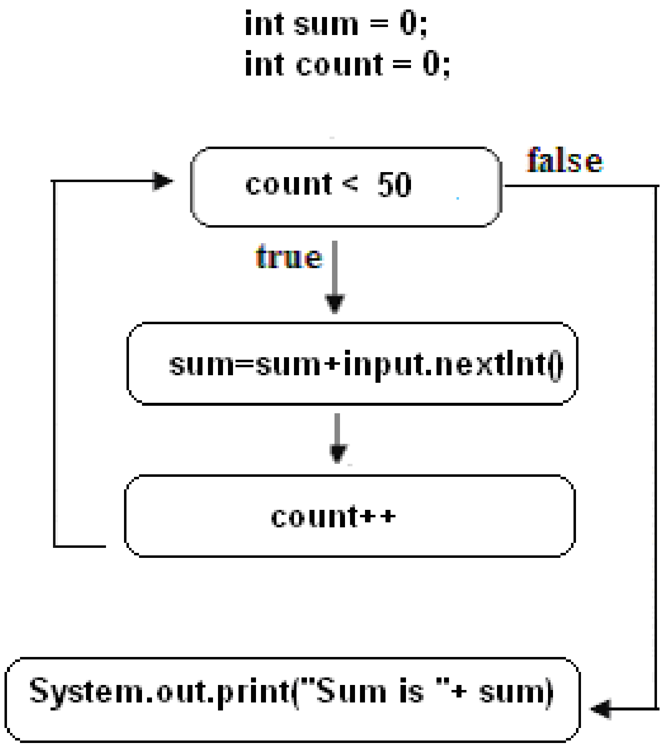
The while Statement

- The assignment statement:

```
sum = sum + input.nextInt()    // line 5
```

- executes 50 times
- From while condition

Adding 50 integers



int sum = 0;
int count = 0;

count < 50

false

true

sum=sum+input.nextInt()

count++

System.out.print("Sum is "+ sum)

The while statement

- **Problem Statement**
  Write a program that sums of a list of integers supplied by a user.

- The list can be of any SIZE (last time fixed number of time is in the code)

- Hint : The program should prompt the user for the number of data.

Solution

- The following application utilizes three variables:

```
size, sum, and count
```

- `size` is the number of data;
- `sum` holds a running sum of the numbers supplied by the user so that each time the user enters a number
  - that number is added to sum
- `count` keeps track of the number of data

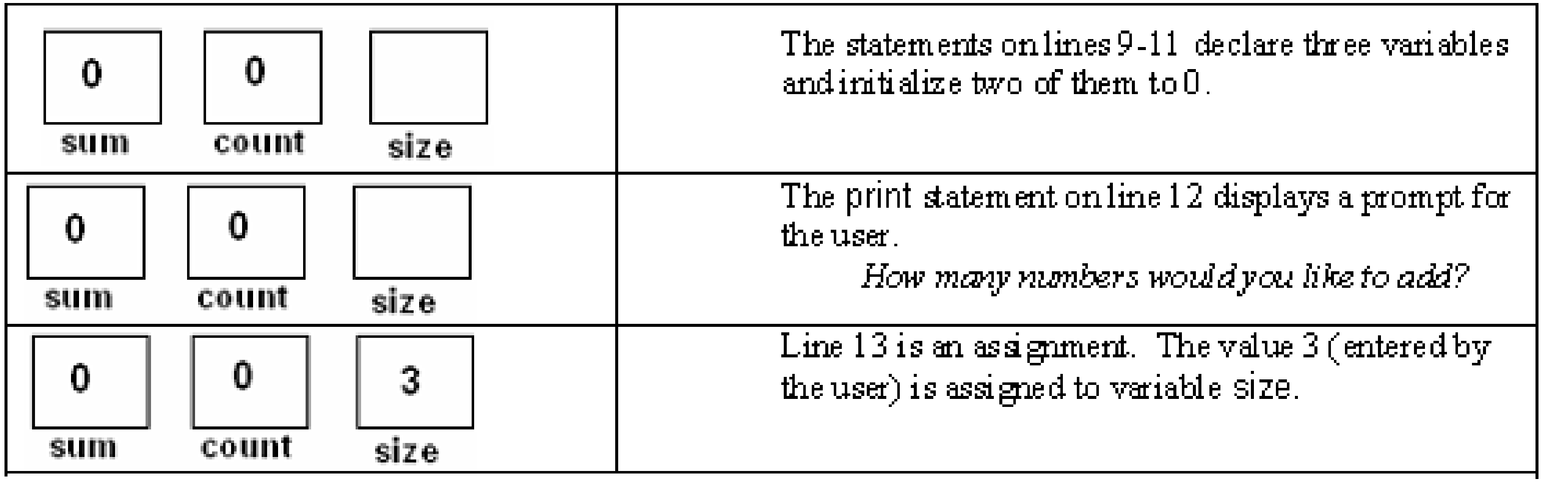
```
import java.util.*;
public class AddEmUp
 {
     // adds an arbitrarily long list of integers
    // the user first supplies the size of the list
    public static void main (String[] args)
    {
        Scanner input =new Scanner(System.in);
        int sum = 0;          // Running sum
        int count = 0;        //Keeps track of the number of integers
        int size ;                           // Size of the list
        System.out.print("How many numbers would you like to add? ");
        size = input.nextInt();
        System.out.println("Enter the "+size+" numbers");

        while (count < size)  // while number of data  < size repeat:
        {
           sum  = sum + input.nextInt(); // read next integer, add to sum
           count++;    // keep track of the number of data, so far
         }
         System.out.println(“Sum:  “+ sum);
  }
```

Output

```
How many numbers would you like to add? 3
Enter the 3 numbers
5
7
9
Sum: 21
```

Discussion

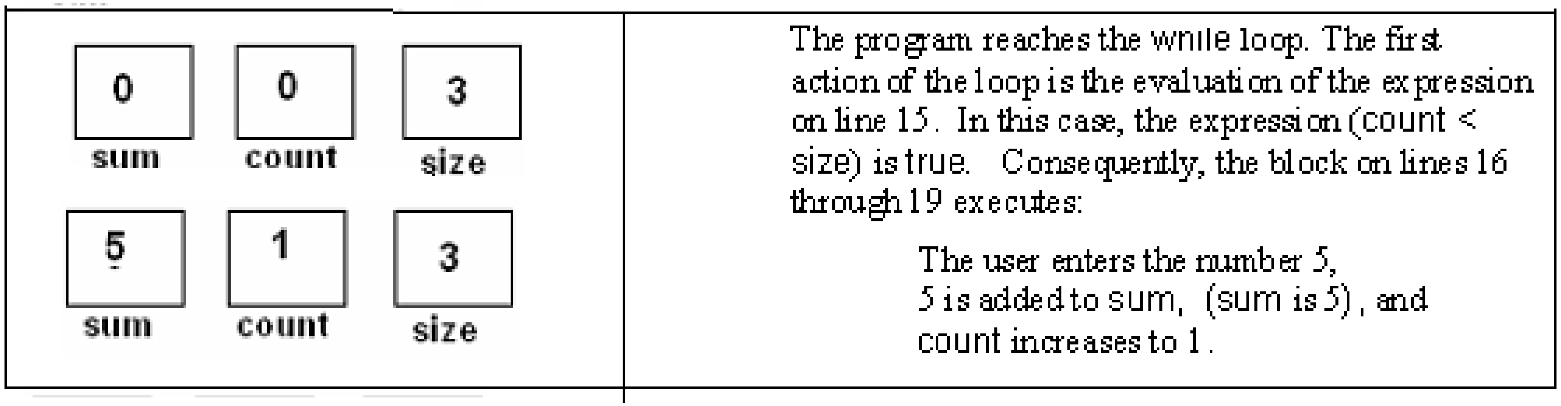
```
9.    int sum = 0;
10.   int count = 0;
11.   int size ;
12.   System.out.print("How many numbers would you like to add? ");
13.   size = input.nextInt();
```



0 sum	0 count	 size	The statements on lines 9-11 declare three variables and initialize two of them to 0.
0 sum	0 count	 size	The print statement on line 12 displays a prompt for the user. *How many numbers would you like to add?*
0 sum	0 count	3 size	Line 13 is an assignment. The value 3 (entered by the user) is assigned to variable size.

Discussion

```
while (count < size)// while number of data  < size
// repeat:
{
   sum  = sum + input.nextInt(); // read next integer,
add                          //to sum
     count++;                 // keep track of the
               // number of data, so far
```



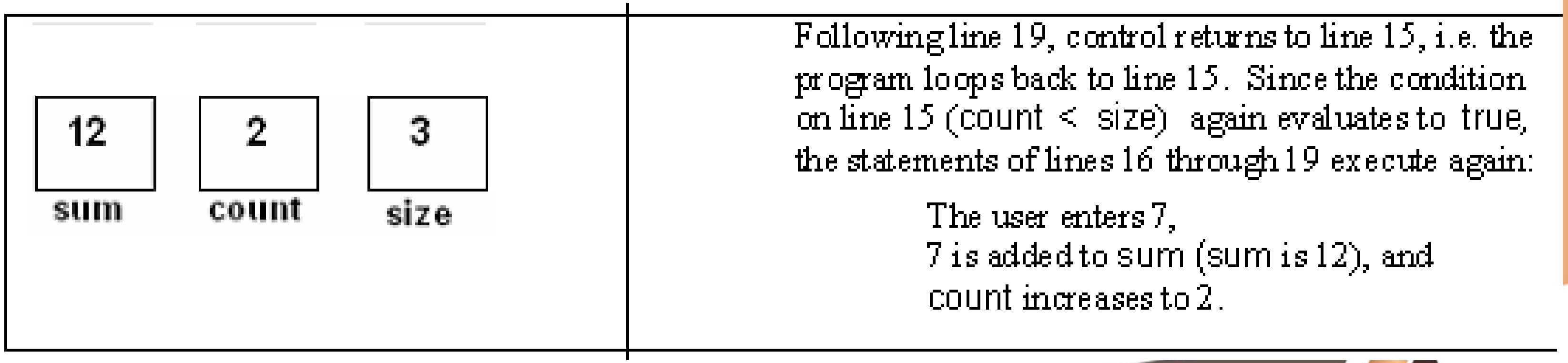
0	0	3
sum	count	size
5	1	3
sum	count	size

The program reaches the while loop. The first action of the loop is the evaluation of the expression on line 15. In this case, the expression (count < size) is true. Consequently, the block on lines 16 through 19 executes:

The user enters the number 5,
5 is added to sum, (sum is 5), and
count increases to 1.

Discussion

```
while (count < size)// while number of data  < size
// repeat:
{
   sum  = sum + input.nextInt(); // read next integer, add
                                  //to sum
     count++;                     // keep track of the
                                  // number of data, so far
 }
```

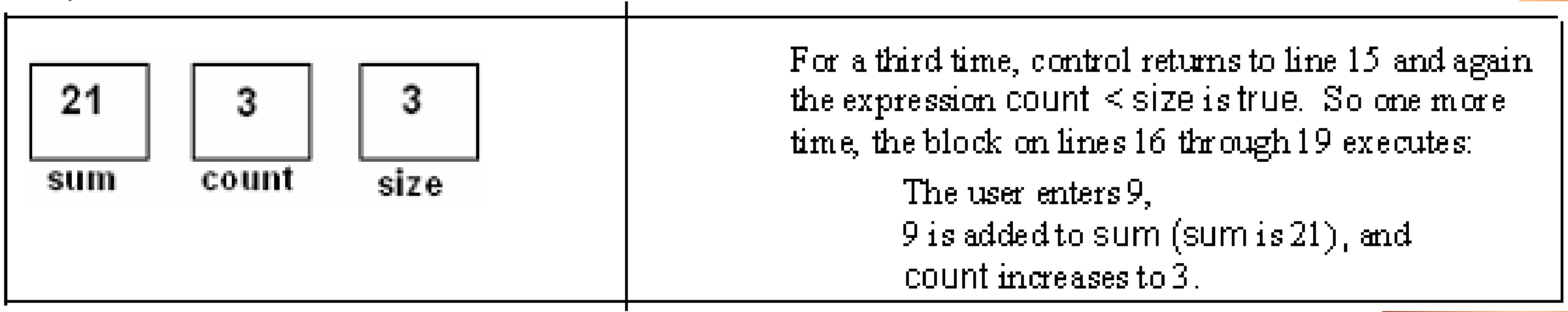

12
sum
2
count
3
size

Following line 19, control returns to line 15, i.e. the program loops back to line 15. Since the condition on line 15 (count < size) again evaluates to true, the statements of lines 16 through 19 execute again:

The user enters 7,
7 is added to sum (sum is 12), and
count increases to 2.

Discussion

```
while (count < size)// while number of data  < size
// repeat:
{
   sum  = sum + input.nextInt(); // read next integer, add
                                  //to sum
     count++;                     // keep track of the
                                  // number of data, so far
 }
```

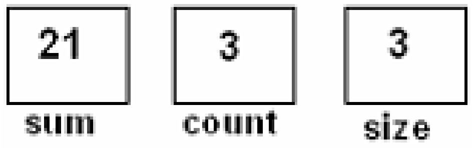


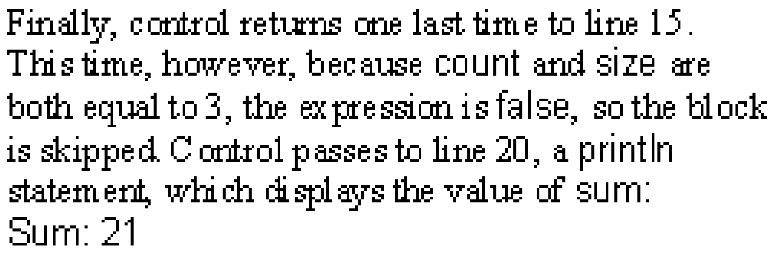
For a third time, control returns to line 15 and again the expression count < size is true. So one more time, the block on lines 16 through 19 executes:

The user enters 9,
9 is added to sum (sum is 21), and
count increases to 3.

Discussion

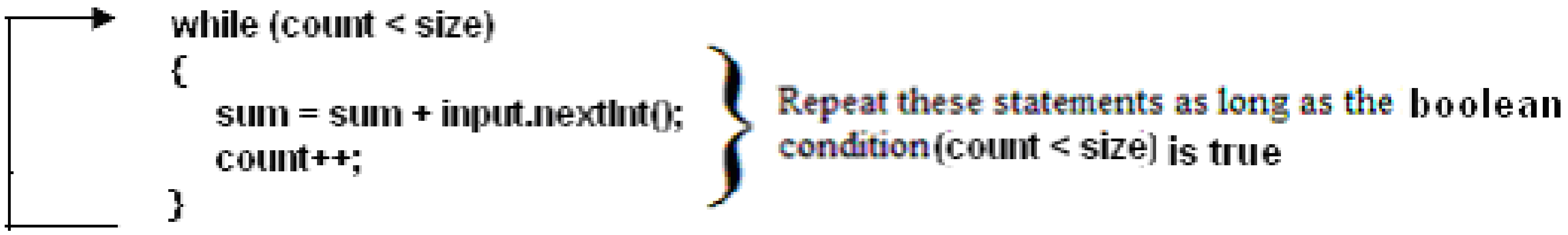
```
while (count < size)// while number of data  < size
// repeat:
{
   sum  = sum + input.nextInt(); // read next integer, add
                                  //to sum
     count++;                     // keep track of the
                                  // number of data, so far
 }
```





Finally, control returns one last time to line 15. This time, however, because count and size are both equal to 3, the expression is false, so the block is skipped. Control passes to line 20, a println statement, which displays the value of sum:
Sum: 21

The actions of a *while* loop



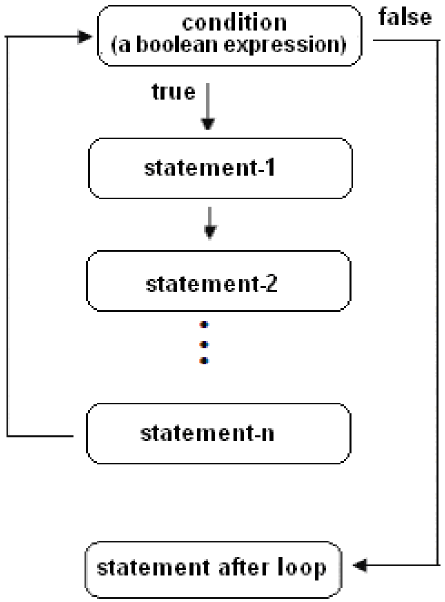
```
while (count < size)
{
    sum = sum + input.nextInt();
    count++;
}
```

Repeat these statements as long as the boolean condition(count < size) is true

while Semantics

- *condition*, a boolean expression, is evaluated.
- If *condition* evaluates to true,
    - *statement-1*, *statement-2*, …, *statement-n* execute.
    - Program control returns to the top of the loop.
    - The process *repeats* (go to step 1).
- If *condition* evaluates to *false*,
    - *statement-1, statement-2,…, statement-n* are *skipped*
    - Program control passes to the first statement after the loop.

while Semantics



The semantics of the *while* statement

Example

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

You must use while –statement to complete the task.

Example

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

You must use while –statement to complete the task.

Example

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

You must use while –statement to complete the task.

Example

- Write a program to read 10 integers from user and display the average.

- You must use while –statement to complete the task.

Example

- Write a program to read 10 integers from user and display the max and min.

- You have to use the while-statement.
- You may use the if-statement to help.

Example

```
1
12
123
1234
12345
123456
```

You must use while –statement to complete the task.

Example

```
Input : 7
1
12
123
1234
12345
123456
1234567
```

You have to read the input from user.

You must use while –statement to complete the task.

The flag

- Another mechanism used to terminate a loop is a *flag* or *sentinel*.
- A *flag* or *sentinel* is a value appended to a data collection that signals the end of the data.
-
- A sentinel cannot be a number that is a *feasible* data value. For example, if all data are positive integers, you might use -1 as a flag
- a list of data might have the form 234, 564, 567, 128, 123*, -1*.

The flag

- **Problem Statement**

- Write a program that computes the sum a list of integers that is supplied by a user.
- The end of data is signaled by the value -999.

- This value is used only as a *flag* and is *not included* in the sum.

Solution

```
import java.util.*;
public class AddEmUpAgain
 {
    // adds an arbitrarily long list of integers
    // -999 signals the end of data
    public static void main (String[] args)
   {
        Scanner input =new Scanner(System.in);
        final int FLAG = -999;      // signals the end of data
        int sum = 0;               // Running sum
        int number;               // holds the next integer to be added
        System.out.println("Enter the numbers.  End with "+FLAG);

        number = input.nextInt();
        while (number != FLAG)   // FLAG signals the end of data
        {
            sum += number;      // add the current integer to sum
            number = input.nextInt();  // read the next integer
         }
        System.out.println("Sum: "+ sum);
  }
}
```

Output

```
Enter the numbers.  End with -999
5 6 7 -999
Sum: 18
```

Discussion

```
9.          final int FLAG = -999;
10.         int sum = 0;
11.          int number;
12.          System.out.println("Enter the numbers.  End with "+FLAG);

13.         number = input.nextInt();
14.         while (number != FLAG)
15.         {
16.             sum += number;
17.             number = input.nextInt();
18.          }
```

The constant FLAG (line 9) serves as a sentinel that *signals* the end of data.

Discussion

```
9.             final int FLAG = -999;
10.            int sum = 0;
11.            int number;
12.            System.out.println("Enter the numbers.  End with "+FLAG);
13.            number = input.nextInt();
14.            while (number != FLAG)
15.            {
16.                sum += number;
17.                number = input.nextInt();
18.             }
```

The first datum is read *outside* the while loop (line 13).
If the statement on line 13 is omitted, the compiler generates an error message on line 14:

variable number might not have been initialized.

If the first datum happens to be *FLAG*, the program *never* enters the loop and correctly determines that the sum is 0.

Discussion

```
final int FLAG = -999;
int sum = 0;
 int number;
 System.out.println("Enter the numbers.  End with "+FLAG);
number = input.nextInt();
while (number != FLAG)
{
    sum += number;
    number = input.nextInt();
 }
```

The last action of the loop is an input statement.
Consequently, when the user enters -999, the sentinel value is read but *not included* in the sum.

Discussion

- A more general program might prompt the user for the sentinel value *rather than* forcing the use of –999
- This improvement is easily accomplished by replacing:

```
final int FLAG = -999;
```

with

```
System.out.println("Enter sentinel value: ");
final int FLAG = input.nextInt();
```

Loops: A Source of Power; A Source of Bugs

- Two common bugs:
  - The infinite loop.
  - The “off by one” error.

The Infinite Loop

- An infinite loop continues *forever*.
  - An infinite while loop exists if the loop's terminating condition fails to evaluate to false:

```
while (count < size)
{
     sum  = sum + input.nextInt();
}
```

- The problem is a failure to *increment count*.

A loop may never execute

- A loop may *never* execute:

```
 count = 0;
while (count > size)
{
number = input.nextInt();
sum += number;
count++;
}
```

- Since count is initialized to 0, and the user presumably enters a *positive* integer for size
  - then the expression count > size evaluates to *false*
  - the statements of the loop never execute.

The “Off by One” Error

- This error occurs if a loop executes one *too many* or one *too few times*.

The “Off by One” Error

- The following *erroneous* program is intended to calculate
- the sum of the first n positive integers: 1+2+3+ …+ n.
- The user supplies a value for n:

The “Off by One” Error

```
import java.util.*;
public class AddUpToN  // with an error!
{
    public static void main (String[] args)
    {
        Scanner input =new Scanner(System.in);
        int sum = 0;          // Cumulative  sum
        int number;           // find sum 1+2+...+ number
        int count = 1;        //counts 1 to number
        System.out.print("Enter a positive integer: ");
        number = input.nextInt();      // read the next integer
        while (count < number)          //here's the bug
        {
            sum += count;
            count++;
        }
        System.out.println("The sum of the first "+number+
                                    " positive integers is "+ sum);
    }
}
```

(Erroneous) output

```
Enter a positive integer: 5
The sum of the first 5 positive integers is 10
```

- The loop executes *four* rather than five times. The error *lies* in the condition,
  - which should be

    count <= number
  - rather than

    count < number.

The “Off by One” Error

- The following program is supposed to calculate the average of a list of numbers terminated by the sentinel value -999.

- The program does not work correctly. It mistakenly includes the sentinel as *part* of the data:

The “Off by One” Error

```
public class Average   // PRODUCES FAULTY OUTPUT!
{
    public static void main (String[] args)
    {
        Scanner input =new Scanner(System.in);
        final int FLAG = -999;
        double sum = 0;              // running sum
        double number;                // holds the next integer to be added
        int count = 0;               // counts the number of data
        double average;
        System.out.println("Enter the numbers end with "+FLAG);
        number = input.nextDouble();   // read the next number
        while (number != FLAG)
        {
            count++;
            number = input.nextDouble();    // read the next number
            sum += number;                  // add the current integer to sum
        }
        average = sum/count;
        System.out.println("Average: "+ average);
    }
}
```

(Erroneous) output

```
Enter the numbers end with -999
1
2
3
-999
Average: -331.3333333333333
```

- The sentinel value (-999) is *included* in the sum and the first number (1) is *not*,
  - i.e., sum is computed with the values 2, 3, and -999.
- Reversing the last two lines of the loop corrects the problem:

```
while (number != FLAG)
{
        count++;
        sum += number;
        number = input.nextDouble
}
```