4. Flow of Control: Loops

[ITP20003] Java Programming

Agenda

- Java Loop Statements
- Programming with Loops
- Graphics Supplement

Java Loop Statements

- A portion of a program that repeats a statement or a group of statements is called a *loop*.
- The statement or group of statements to be repeated is called the *body* of the loop.

Java Loop Statements

- The `while` statement
- The `do-while` statement
- The `for` Statement

The `while` Statement

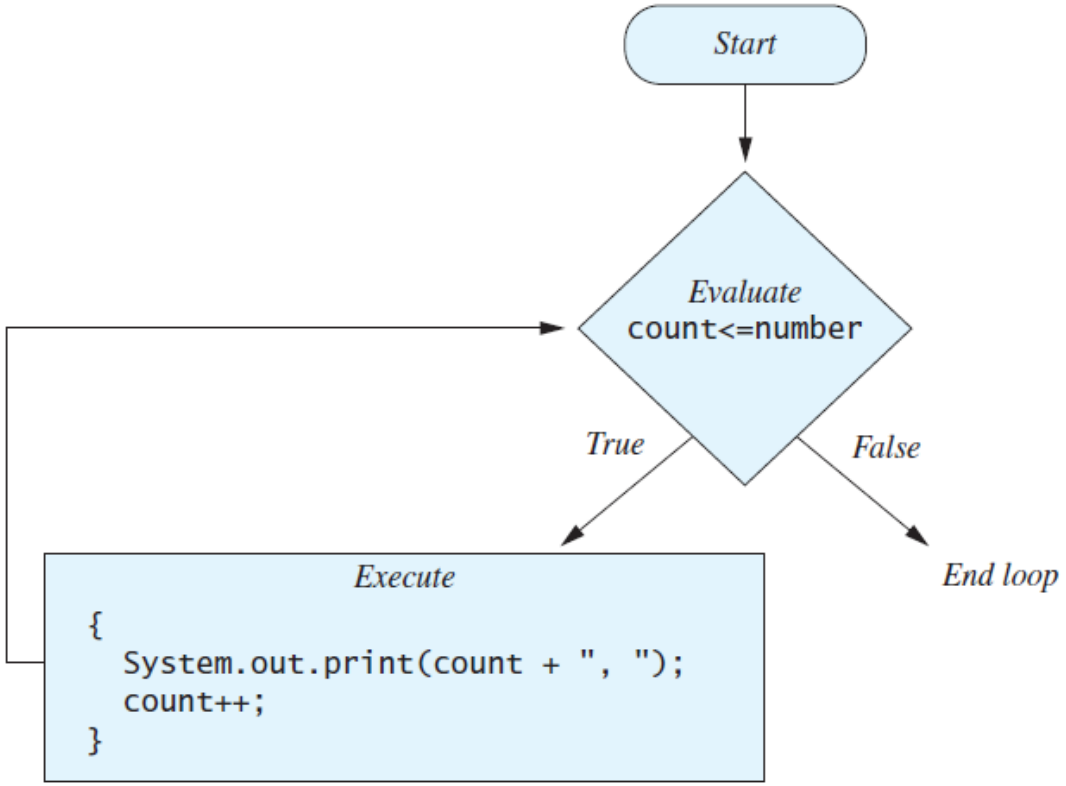
- Also called a `while` loop
- A `while` statement repeats while a controlling boolean expression remains true
- The loop body typically contains an action that ultimately causes the controlling boolean expression to become false.

while Demo

```
import java.util.Scanner;
public class WhileDemo
{
    public static void main (String [] args)
    {
        int count, number;
        System.out.println ("Enter a number");
        Scanner keyboard = new Scanner (System.in);
        number = keyboard.nextInt ();
        count = 1;                              // loop variable
        while (count <= number)                 // control expression
        {
            System.out.print (count + ", ");
            count++;                            // eventually, it makes 'count <= number' false
        }
        System.out.println ();
        System.out.println ("Buckle my shoe.");
    }
}
```

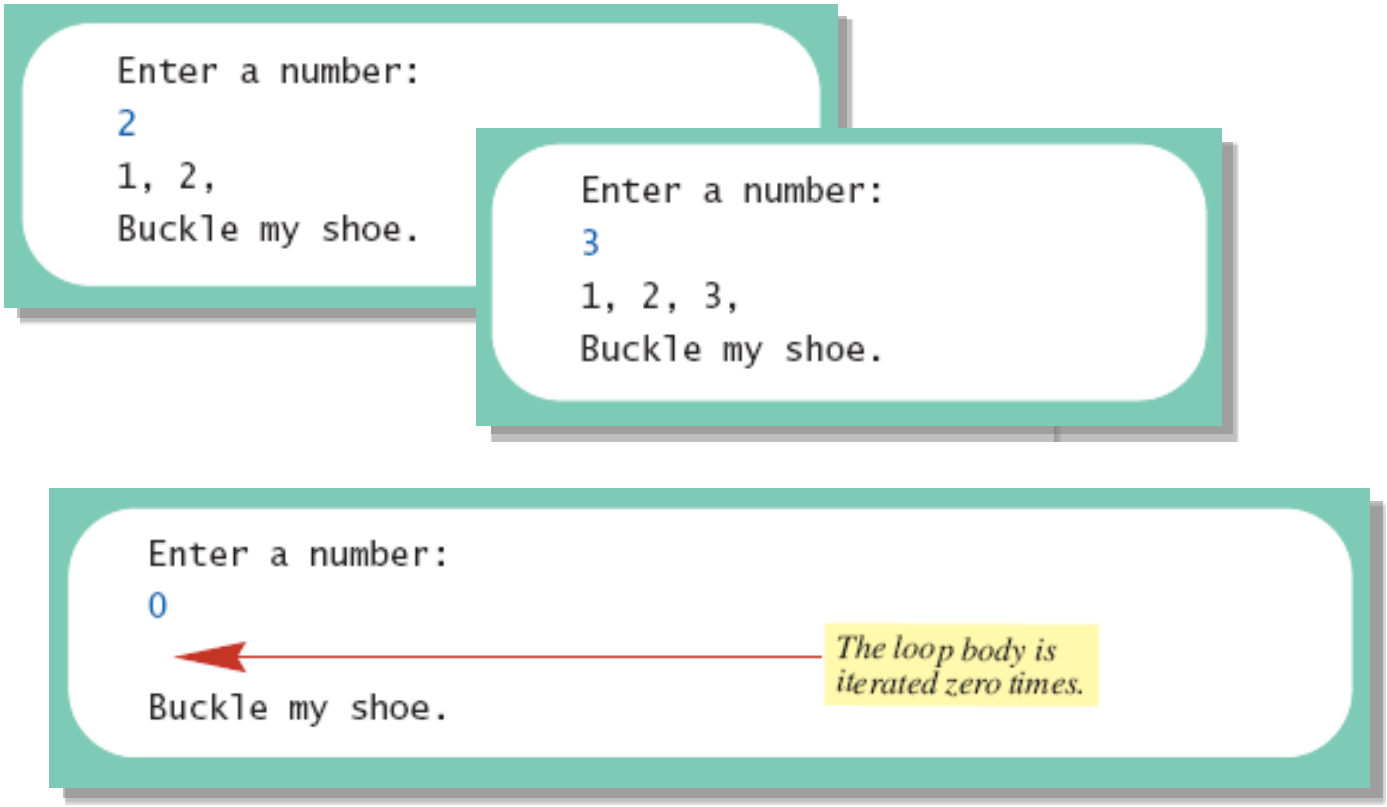
The while Statement

```
while (count <= number)
{
   System.out.print(count + ", ");
   count++;
}
```



while Demo

- Result



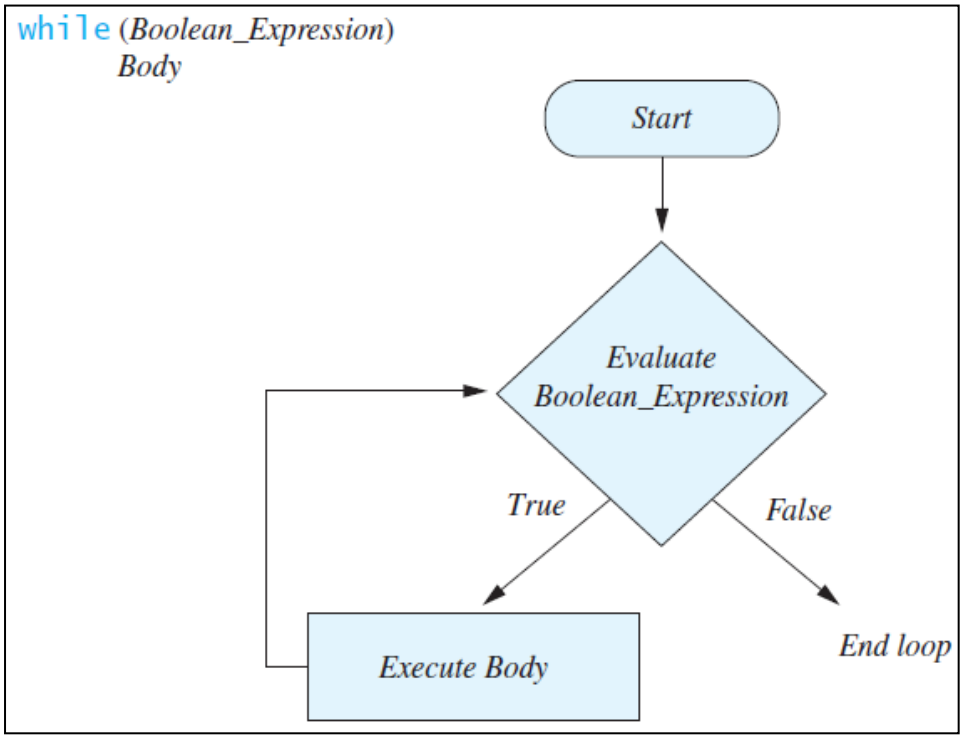
The while Statement

- Syntax

```
while (Boolean_Expression)
    Body_Statement
```

Or

```
while (Boolean_Expression)
{
    First_Statement
    Second_Statement
    …
}
```



The do-while Statement

- Also called a do-while loop
- Similar to a while statement, except that the loop body is executed at least once
- Syntax

```
do
        Body_Statement
while (Boolean_Expression);
```

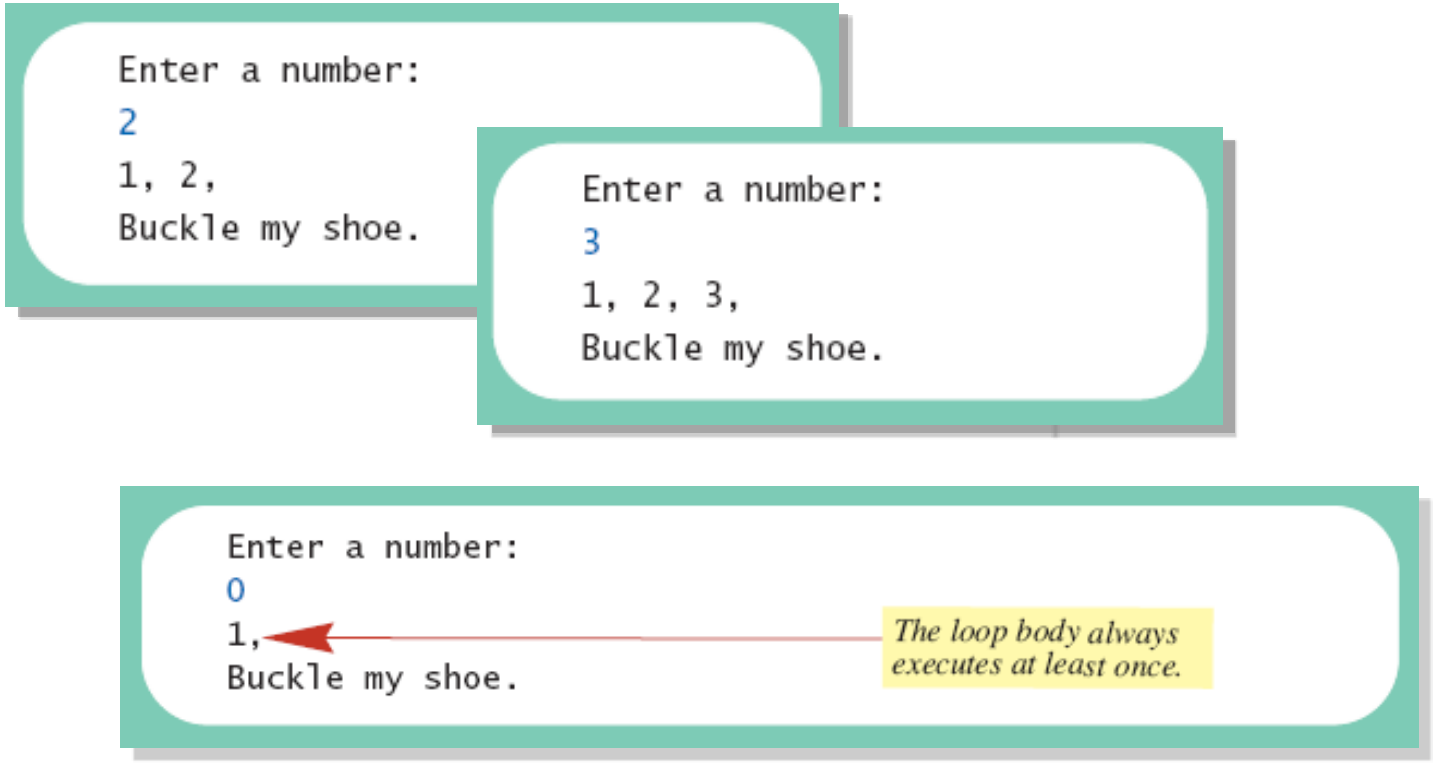
  - Don't forget the semicolon!
    - cf) for loop and while loop.

do-while Demo

```
import java.util.Scanner;
public class DoWhileDemo
{
    public static void main (String [] args)
    {
        int count, number;
        System.out.println ("Enter a number");
        Scanner keyboard = new Scanner (System.in);
        number = keyboard.nextInt ();
        count = 1;
        do
        {
            System.out.print (count + ", ");
            count++;
        } while (count <= number);
        System.out.println ();
        System.out.println ("Buckle my shoe.");
    }
}
```

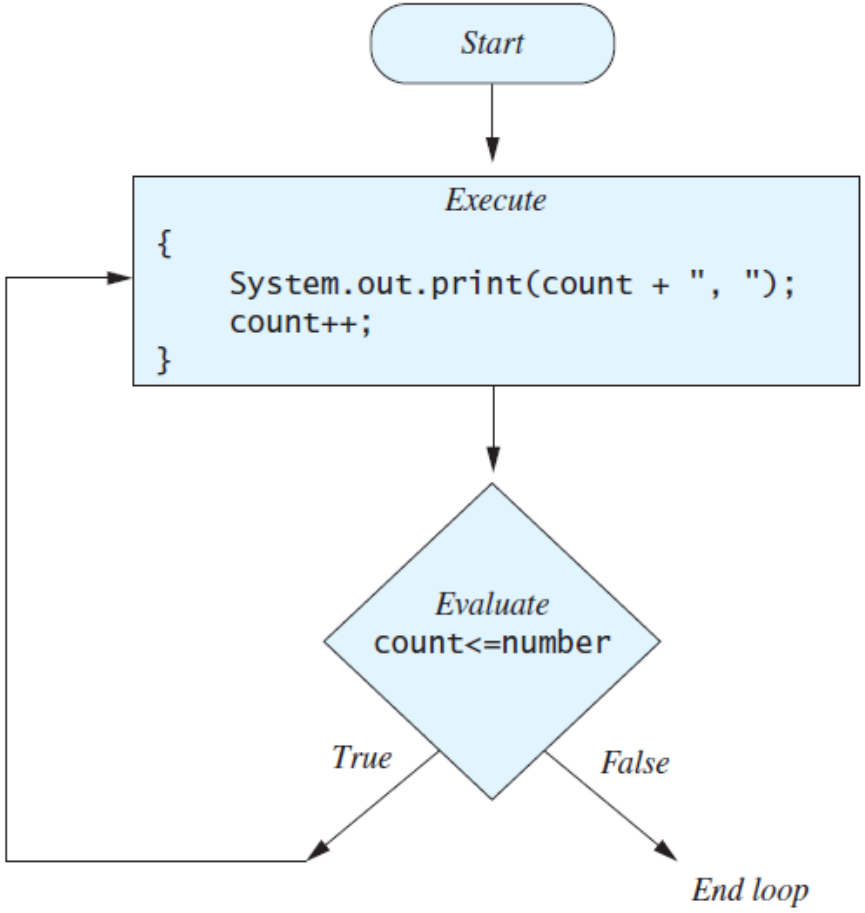
do-while Demo

- Result



The do-while Statement

```
do
{
   System.out.print(count + ", ");
   count++;
} while (count <= number);
```

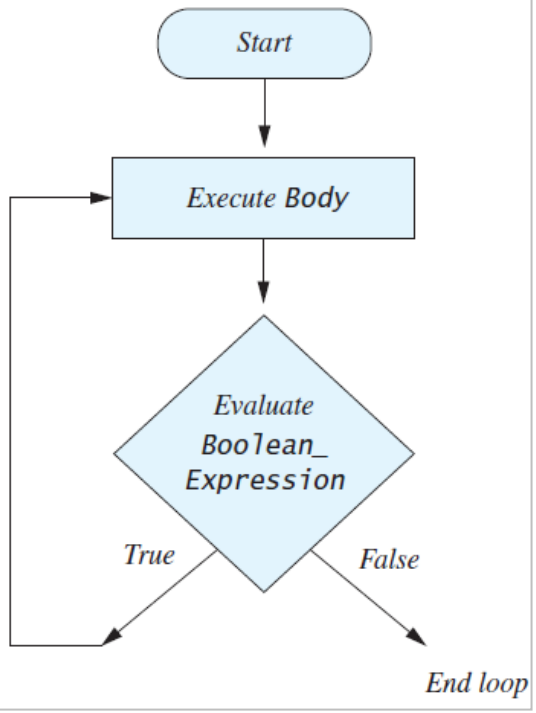


The `do-while` Statement

- First, the loop body is executed.
- Then the boolean expression is checked.
  - As long as it is true, the loop is executed again.
  - If it is false, the loop ends.
- Equivalent `while` statement

```
Statement(s)_S1
while (Boolean_Condition)
    Statement(s)_S1
```

```
do
    Body
while (Boolean_Expression)
```



Programming Example: Bug Infestation

- Given
  - Volume a roach: 0.002 cubic feet
  - Starting roach population
  - Rate of increase: 95% /week
  - Volume of a house
- Find
  - Number of weeks to exceed the capacity of the house
  - Number and volume of roaches

Programming Example: Bug Infestation

- Algorithm for roach population program (rough draft)
  1. Get volume of house.
  2. Get initial number of roaches in house.
  3. Compute number of weeks until the house is full of roaches.
  4. Display results.

Programming Example: Bug Infestation

- Variables needed
  - GROWTH_RATE —weekly growth rate of the roach population (a constant 0.95)
  - ONE_BUG_VOLUME —volume of an average roach (a constant 0.002)
  - houseVolume — volume of the house
  - startPopulation —initial number of roaches
  - countWeeks —week counter
  - population —current number of roaches
  - totalBugVolume —total volume of all the roaches
  - newBugs —number of roaches hatched this week
  - newBugVolume —volume of new roaches

Detailed Algorithm

- Algorithm for roach population program

1. Read houseVolume
2. Read startPopulation
3. population = startPopulation
4. totalBugVolume = population * ONE_BUG_VOLUME
5. countWeeks = 0
6. while (totalBugVolume < houseVolume)

```
{
    newBugs = population * GROWTH_RATE
    newBugVolume = newBugs * ONE_BUG_VOLUME
    population = population + newBugs
    totalBugVolume = totalBugVolume + newBugVolume
    countWeeks = countWeeks + 1
}
```

7. Display startPopulation, houseVolume, countWeeks, population, and totalBugVolume

Programming Example: Bug Infestation

- Result

```
Enter the total volume of your house
in cubic feet: 20000
Enter the estimated number of
roaches in your house: 100
Starting with a roach population of 100
and a house with a volume of 20000.0 cubic feet,
after 18 weeks,
the house will be filled with 16619693 roaches.
They will fill a volume of 33239 cubic feet.
Better call Debugging Experts Inc.
```

Infinite Loops

- A loop which repeats without ever ending is called an *infinite loop*.
  - If the controlling boolean expression never becomes false, a `while` loop or a `do-while` loop will repeat without ending.

    Ex) A negative growth rate in the preceding problem causes `totalBugVolume` always to be less than `houseVolume`

Nested Loops

- The body of a loop can contain any kind of statements, including another loop.

- to calculate the average score of each exam
  - The average score can be computed using a loop.
  - This loop is placed inside another loop (for exams), so the process could be repeated for other sets of exam scores.
    - see the next slide

Nested Loops

```
do
{
    System.out.println();
    System.out.println("Enter all the scores to be averaged.");
    System.out.println("Enter a negative number after");
    System.out.println("you have entered all the scores.");
    sum = 0;
    numberOfStudents = 0;
    next = keyboard.nextDouble();
    while (next >= 0)                                  // the average of one exam
    {
       sum = sum + next;
       numberOfStudents++;
       next = keyboard.nextDouble();
    }
    if (numberOfStudents > 0)
        System.out.println("The average is " +
                           (sum / numberOfStudents));
    else
        System.out.println("No scores to average.");

    System.out.println("Want to average another exam?");
    System.out.println("Enter yes or no.");
    answer = keyboard.next();
} while (answer.equalsIgnoreCase("yes"));     // the next exam?
```

Nested Loops: result

```
This program computes the average of
a list of (nonnegative) exam scores.

Enter all the scores to be averaged.
Enter a negative number after
you have entered all the scores.
100
90
100
90
-1
The average is 95.0
Want to average another exam?
Enter yes or no.
yes

Enter all the scores to be averaged.
Enter a negative number after
you have entered all the scores.
90
70
80
-1
The average is 80.0
Want to average another exam?
Enter yes or no.
no
```

The *for* Statement

- A *for* statement executes the body of a loop a fixed number of times.

- Syntax

  for(*Initialization*; *Condition*; *Update*)

  Body_Statement

  - Body_Statement can be either a simple statement or a compound statement.

  Ex) for (count = 1; count < 3; count++)

  System.out.println(count);

- Corresponding *while* statement

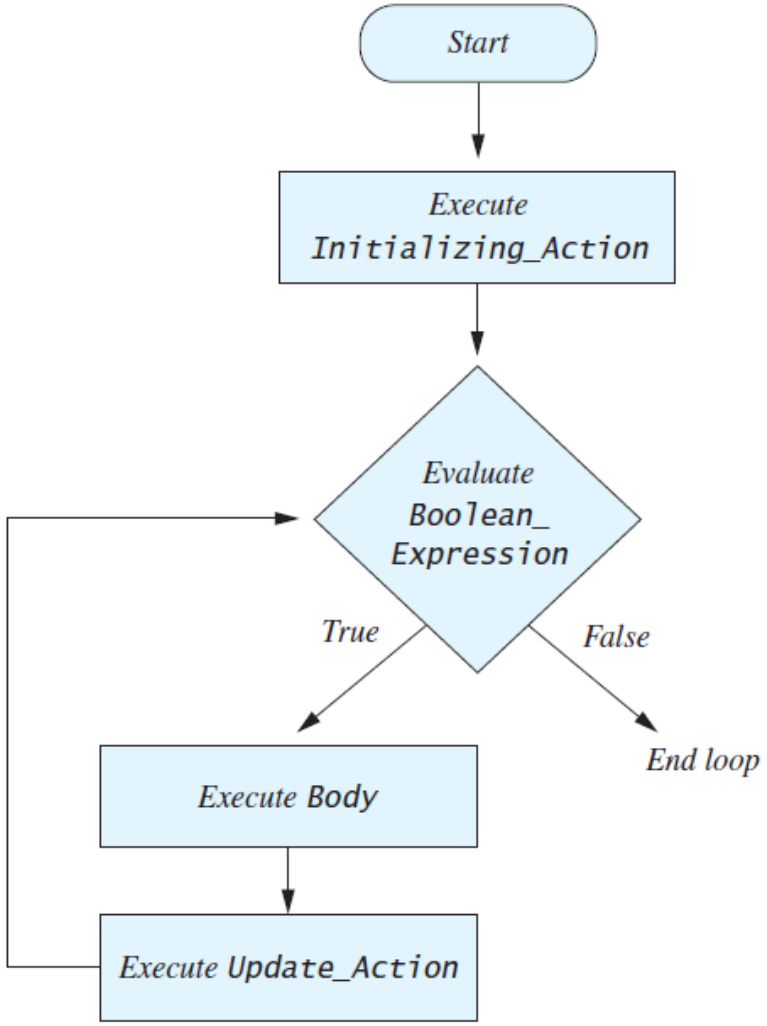
  Initialization

  while (*Condition*)

  Body_Statement_Including_Update

The *for* Statement

```
for (Initializing_Action; Boolean_Expression; Update_Action)
    Body
```



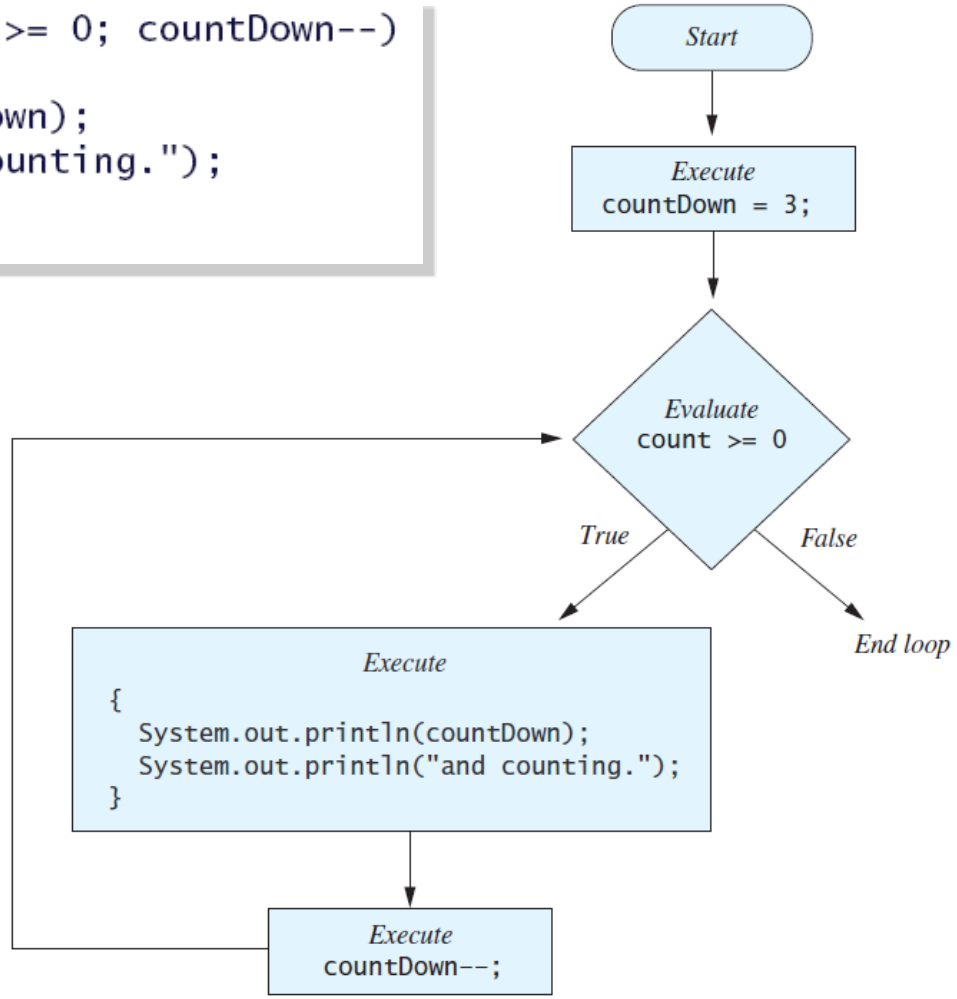
for Demo

```
public class ForDemo
{
    public static void main (String [] args)
    {
        int countDown;
        for (countDown = 3 ; countDown >= 0 ; countDown--) {
            System.out.println (countDown);
            System.out.println ("and counting.");
        }
        System.out.println ("Blast off!");
    }
}
```

```
3
and counting.
2
and counting.
1
and counting.
0
and counting.
Blast off!
```

The *for* Statement

```
for (countDown = 3; countDown >= 0; countDown--)
{
    System.out.println(countDown);
    System.out.println("and counting.");
}
```



The *for* Statement

- Possible to declare variables within a for statement

```
int sum = 0;
for (int n = 1; n <= 10; n++)
    sum = sum + n * n;
```

- Note that variable *n* is local to the loop

The *for* Statement

- A comma separates multiple initializations

```
for (n = 1, product = 1; n <= 10; n++)
    product = product * n;
```

- Only one boolean expression is allowed, but it can consist of &&s, ||s, and !s.

- Multiple update actions are allowed, too.

```
for (n = 1, product = 1; n <= 10; product = product * n, n++);
```

The *for-each* Statement

- Possible to step through values of an enumeration type

- Example

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
for (Suit nextSuit : Suit.values())
        System.out.print(nextSuit + " ");
System.out.println();
```

- `output:`

```
CLUBS DIAMONDS HEARTS SPADES
```

Agenda

- Java Loop Statements
- **Programming with Loops**
- Graphics Supplement

The Loop Body

- To design the loop body, write out the actions the code must accomplish.

  Ex) Read numbers from the user and compute the sum of them

1. Display instructions to the user.
2. Initialize variables.
3. Read a number into the variable `next`.
4. `sum = sum + next`
5. Display the number and the sum so far.
6. Read another number into the variable `next`.
7. `sum = sum + next`
8. Display the number and the sum so far.
9. Read another number into the variable `next`.
10. `sum = sum + next`
11. Display the number and the sum so far.
12. Read another number into the variable `next`.
13. and so forth.

The Loop Body

- Then, look for a repeated pattern.
  - The repeated pattern will form the body of the loop.

1. Display instructions to the user.
2. Initialize variables.
3. Repeat the following for the appropriate number of times:

```
{
    Read a number into the variable next.
    sum = sum + next
    Display the number and the sum so far.
}
```

Initializing Statements

- Some variables need to have a value before the loop begins.
- Other variables get values only while the loop is iterating.

Controlling Number of Loop Iterations

- If the number of iterations is known before the loop starts, the loop is called a count-controlled loop.
  - Use a *for* loop.
- Asking the user before each iteration if it is time to end the loop is called the ask-before-iterating technique.
  - Appropriate for a small number of iterations
  - Use a *while* loop or a *do-while* loop.

Controlling Number of Loop Iterations

- For large input lists, a *sentinel value* can be used to signal the end of the list.
  - The sentinel value must be different from all the other possible inputs.
  - A negative number following a long list of nonnegative exam scores could be suitable.

```
90
0
10
-1
```

Controlling Number of Loop Iterations

Ex) Reading a list of scores followed by a sentinel value

```
int next = keyboard.nextInt();
while (next  >= 0)
{
    Process_The_Score
    next = keyboard.nextInt();
}
```

BooleanDemo

```
import java.util.Scanner;
public class BooleanDemo
{
    public static void main (String [] args)
    {
        System.out.println ("Enter nonnegative numbers.");
        System.out.println ("Place a negative number at the end");
        System.out.println ("to serve as an end marker.");
        int sum = 0;
        boolean areMore = true;
        Scanner keyboard = new Scanner (System.in);
        while (areMore)
        {
            int next = keyboard.nextInt ();
            if (next < 0)
                areMore = false;
            else
                sum = sum + next;
        }
        System.out.println ("The sum of the numbers is " + sum);
    }
}
```

BooleanDemo

- Result

```
Enter nonnegative numbers.
Place a negative number at the end
to serve as an end marker.
1 2 3 -1
The sum of the numbers is 6
```

The *break* Statement in Loops

- A *break* statement can be used to end a loop immediately.
- The *break* statement ends only the innermost loop or switch statement that contains the break statement.
- Use *break* statements sparingly (if ever).
  - *break* statements make loops more difficult to understand.

The *break* Statement in Loops

- Ending a loop with a break statement,

```
while (itemNumber <= MAX_ITEMS)
{
    . . .
    if (itemCost <= leftToSpend)
    {
        . . .
        if (leftToSpend > 0)
            itemNumber++;
        else
        {
            System.out.println("You are out of money.");
            break;
        }
    }
    else
        . . .
}

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

The *continue* Statement in Loops

- A *continue* statement
  - Ends current loop iteration
  - Begins the next one
- Text recommends avoiding use
  - Introduce unneeded complications

Tracing Variables

- Tracing variables means watching the variables change while the program is running.
  - Simply insert temporary output statements in your program to print of the values of variables of interest
  - Or, learn to use the debugging facility that may be provided by your system.

Loop Bugs

- Common loop bugs
  - Unintended infinite loops
  - Off-by-one errors
  - Testing equality of floating-point numbers

- Subtle infinite loops
  - The loop may terminate for some input values, but not for others.

    Ex) You can’t get out of debt when the monthly penalty exceeds the monthly payment.

questions or comments?

hchoi@handong.edu