



# D06

# PROGRAMMING with JAVA

## Ch7 – Iteration

# Chapter Goals

---

- To be able to program loops with the **while**, **for**, and **do** statements
- To avoid **infinite loops** and **off-by-one errors**
- To understand **nested loops**
- To learn how to process input
- To implement simulations

# while Loops

- The **while** statement executes a statement or a block statement repeatedly while the **condition** is true:

```
while (condition)
{
    statements;
}
```

- Example (calculating the growth of an investment): Invest \$10,000, 5% interest, compounded annually

Year	Balance
0	\$10,000
1	\$10,500
2	\$11,025
3	\$11,576.25
4	\$12,155.06
5	\$12,762.82

When has the bank account reached a particular balance?

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

# File Investment.java

```
01: /**
02:     A class to monitor the growth of an investment that
03:     accumulates interest at a fixed annual rate.
04: */
05: public class Investment
06: {
07:     /**
08:         Constructs an Investment object from a starting balance
09:         and interest rate.
10:         @param aBalance the starting balance
11:         @param aRate the interest rate in percent
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
19:
```

*Continued...*

# File Investment.java

```
20:    /**
21:        Keeps accumulating interest until a target balance has
22:        been reached.
23:        @param targetBalance the desired balance
24:    */
25:    public void waitForBalance(double targetBalance)
26:    {
27:        while (balance < targetBalance)
28:        {
29:            years++;
30:            double interest = balance * rate / 100;
31:            balance = balance + interest;
32:        }
33:    }
34:
35:    /**
36:        Gets the current investment balance.
37:        @return the current balance
38:    */
```

*Continued...*

# File Investment.java

```
39:     public double getBalance()
40:     {
41:         return balance;
42:     }
43:
44:     /**
45:      Gets the number of years this investment has
46:      accumulated interest.
47:      @return the number of years since the start of the
         investment
48:     */
49:     public int getYears()
50:     {
51:         return years;
52:     }
53:
54:     private double balance;
55:     private double rate;
56:     private int years;
57: }
```

# File InvestmentTester.java

```
01: /**
02:     This program computes how long it takes for an investment
03:     to double.
04: */
05: public class InvestmentTester
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         Investment invest
12:             = new Investment(INITIAL_BALANCE, RATE);
13:         invest.waitForBalance(2 * INITIAL_BALANCE);
14:         int years = invest.getYears();
15:         System.out.println("The investment doubled after "
16:             + years + " years");
17:     }
18: }
```

**Output**

The investment doubled after 15 years

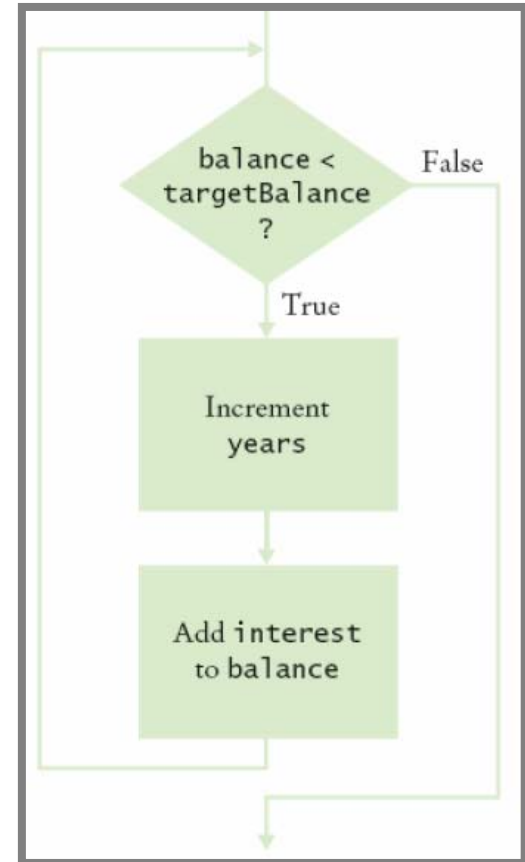
# while Loop Flowchart

- The **while** statement is often called a **loop** . If you draw a flowchart, you will see that the control loops backwards to the test after every iteration

**Figure 1:**  
**Flowchart of a while Loop**

- The following is an **infinite loop**. It executes the statement block over and over, without terminating:

```
while (true)
{
    statements;
}
```



# Syntax 7.1: The `while` Statement

```
while (condition)  
    statement
```

## Example:

```
while (balance < targetBalance)  
{  
    year++;  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

## Purpose:

To repeatedly execute a statement as long as a condition is true

# Self Check

---

1. How often is the statement in the loop

```
while (false) statement;
```

executed?

2. What would happen if `RATE` was set to 0 in the `main` method of the `InvestmentTester` program?

# Answers

---

1. Never
2. The `waitForBalance` method would never return due to an infinite loop

# Common Error: Infinite Loops

- An **infinite loop** is a loop that runs forever and can be stopped only by killing the program or restarting the computer
- A common reason for infinite loops is forgetting to advance the variable that controls the loop:

```
int years = 0;
while (years < 20)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

- Another common reason for an infinite loop is accidentally incrementing a counter that should be decremented:

```
int years = 20;
while (years > 0)
{
    years++; // Oops, should have been years--
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```


# Common Error: Off-By-One Errors

- Consider the computation of the number of years that are required to double an investment:

```
int years = 0;
while (balance < 2 * initialBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
System.out.println("The investment reached the target after "
    + years + " years.");
```

Should `years` start at 0 or 1?

Should the test be `<` or `<=`?

-  **Off-by-one errors** are easy to avoid, simply by thinking through a couple of test cases and using the information from the test cases to come up with a rationale for the correct loop condition

# do Loops

- Sometimes you want to execute the body of a loop at least once and perform the loop test after the body was executed. The **do loop** serves that purpose:

```
do
    statement while (condition);
```

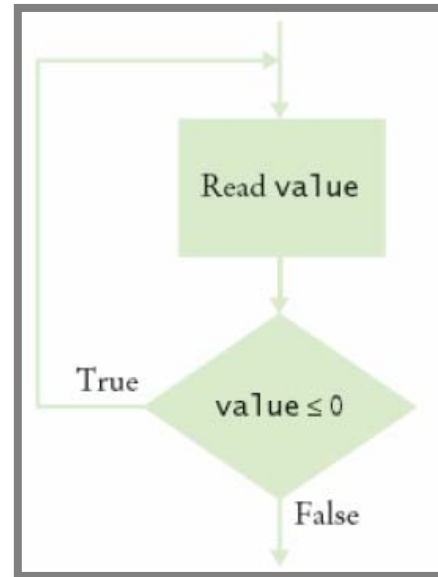
- The statement is executed while the condition is true. The condition is tested after the statement is executed, so the statement is executed at least once:

```
double value;
do
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
}
while (value <= 0);
```

*Continued...*

# do Loops

Figure 2:  
Flowchart of a do Loop

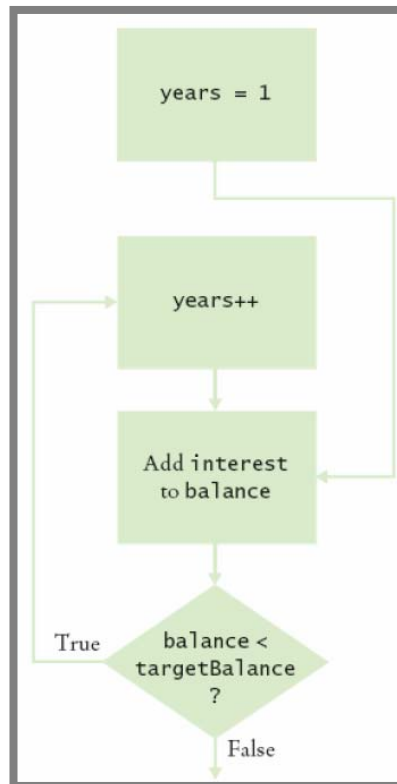


- You can always replace a **do** loop with a **while** loop, by introducing a Boolean control variable:

```
boolean done = false;
while (!done)
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
    if (value > 0) done = true;
}
```

# Spaghetti Code

- Because the lines denoting `goto` statements weave back and forth in complex flowcharts, the resulting code is named `spaghetti code`



**Figure 3:**  
**Spaghetti Code**

# for Loops

- One of the most common **loop** types has the form:

```
i = start;
while (i <= end)
{
    ...
    i++;
}
```

- Because this loop is so common, there is a special form for it, the **for** loop:

```
for (i = start; i <= end; i++)
{
    ...
}
```

- You can also declare the loop counter variable inside the loop header. That convenient shorthand restricts the use of the variable to the body of the loop

```
for (int i = start; i <= end; i++)
{
    ...
}
```

*Continued...*

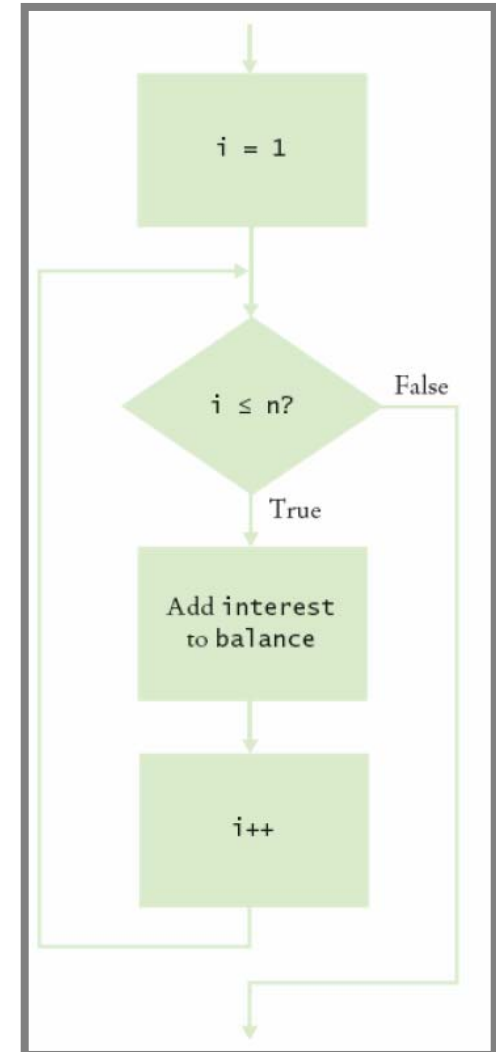
# for Loops

- Examples:

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

```
for (years = n; years > 0; years--) . . .
```

```
for (x = -10; x <= 10; x = x + 0.5) . . .
```



**Figure 4:**  
Flowchart of a for Loop

# Syntax 7.2: The `for` Statement

```
for (initialization; condition; update)  
    statement
```

## Example:

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

## Purpose:

To execute an initialization, then keep executing a statement and updating an expression while a condition is true

# File Investment.java

```
01: /**
02:     A class to monitor the growth of an investment that
03:     accumulates interest at a fixed annual rate
04: */
05: public class Investment
06: {
07:     /**
08:         Constructs an Investment object from a starting
09:         balance and interest rate.
10:         @param aBalance the starting balance
11:         @param aRate the interest rate in percent
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
```

*Continued...*

# File Investment.java

```
19:
20:     /**
21:         Keeps accumulating interest until a target balance
22:         has been reached.
23:         @param targetBalance the desired balance
24:     */
25:     public void waitForBalance(double targetBalance)
26:     {
27:         while (balance < targetBalance)
28:         {
29:             years++;
30:             double interest = balance * rate / 100;
31:             balance = balance + interest;
32:         }
33:     }
34:
```

*Continued...*

# File Investment.java

```
35:    /**
36:        Keeps accumulating interest for a given number of years.
37:        @param n the number of years
38:    */
39:    public void waitYears(int n)
40:    {
41:        for (int i = 1; i <= n; i++)
42:        {
43:            double interest = balance * rate / 100;
44:            balance = balance + interest;
45:        }
46:        years = years + n;
47:    }
48:
49:    /**
50:        Gets the current investment balance.
51:        @return the current balance
52:    */
```

Continued...

# File Investment.java

```
53:     public double getBalance()
54:     {
55:         return balance;
56:     }
57:
58:     /**
59:      Gets the number of years this investment has
60:      accumulated interest.
61:      @return the number of years since the start of the
           investment
62:     */
63:     public int getYears()
64:     {
65:         return years;
66:     }
67:
```

*Continued...*

# File Investment.java

---

```
68:     private double balance;  
69:     private double rate;  
70:     private int years;  
71: }
```

# File InvestmentTester.java

```
01: /**
02:     This program computes how much an investment grows in
03:     a given number of years.
04: */
05: public class InvestmentTester
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         final int YEARS = 20;
12:         Investment invest = new Investment(INITIAL_BALANCE, RATE);
13:         invest.waitYears(YEARS);
14:         double balance = invest.getBalance();
15:         System.out.printf("The balance after %d years is %.2f\n",
16:             YEARS, balance);
17:     }
18: }
```

**Output**

The balance after 20 years is 26532.98

# Self Check

---

3. Rewrite the `for` loop in the `waitYears` method as a `while` loop
4. How many times does the following `for` loop execute?

```
for (i = 0; i <= 10; i++)  
    System.out.println(i * i);
```

# Answers

---

3.

```
int i = 1;
while (i <= n)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
    i++;
}
```

4. 11 times

# Common Errors: Semicolons



A **semicolon** that shouldn't be there:

```
sum = 0;
for (i = 1; i <= 10; i++);
    sum = sum + i;
System.out.println(sum);
```

- Occasionally all the work of a loop is already done in the loop header. If you do run into a loop without a body, it is important that you make sure the **semicolon** is not forgotten

```
for (years = 1; (balance = balance + balance * rate / 100) < targetBalance; years++)
System.out.println(years);
```

- You can avoid this error by using an empty block **{ }** instead of an empty statement

# Scope of variables in a **for** loop

- It is legal in Java to declare a variable in the header of a **for** loop:

```
for (int i = 1; i <= n; i++)
{
    ...
}
// i no longer defined here
```

- The **scope** of the variable extends to the end of the **for** loop. Therefore, **i** is no longer defined after the loop ends
- In the loop header, you can declare multiple variables, as long as they are of the same type and you can include multiple update expressions, separated by commas. Better than that, make the **for** loop control a single counter, and update the other variable explicitly

```
for (int i = 0, j = 10; i <= 10; i++, j--)
```

```
int j = 10;
for (int i = 0; i <= 10; i++)
{
    ...
    j--;
}
```


# Nested Loops

- Sometimes, the body of a loop is again a loop. We say that the inner loop is **nested** inside an outer loop. This happens often when you process two-dimensional structures, such as tables
- Example: create triangle pattern (loop through rows)

```
[ ]  
[ ][ ]  
[ ][ ][ ]  
[ ][ ][ ][ ]
```

```
for (int i = 1; i <= n; i++)  
{  
    // make triangle row  
}
```

```
for (int j = 1; j <= i; j++)  
    r = r + "[ ]";  
r = r + "\n";
```



# File Triangle.java

```
01: /**
02:     This class describes triangle objects that can be
03:     displayed as shapes like this:
04:     []
05:     [][]
06:     [][][]
07: */
08: public class Triangle
09: {
10:     /**
11:         Constructs a triangle.
12:         @param aWidth the number of [] in the last row of
13:             the triangle.
14:     */
15:     public Triangle(int aWidth)
16:     {
17:         width = aWidth;
18:     }
19: }
```

*Continued...*

# File Triangle.java

```
19:    /**
20:        Computes a string representing the triangle.
21:        @return a string consisting of [] and newline
           characters
22:    */
23:    public String toString()
24:    {
25:        String r = "";
26:        for (int i = 1; i <= width; i++)
27:        {
28:            // Make triangle row
29:            for (int j = 1; j <= i; j++)
30:                r = r + "[";
31:            r = r + "\n";
32:        }
33:        return r;
34:    }
35:
36:    private int width;
37: }
```

# File TriangleTester.java

```
01: /**
02:     This program tests the Triangle class.
03: */
04: public class TriangleTester
05: {
06:     public static void main(String[] args)
07:     {
08:         Triangle small = new Triangle(3);
09:         System.out.println(small.toString());
10:
11:         Triangle large = new Triangle(15);
12:         System.out.println(large.toString());
13:     }
14: }
```

# Self Check

---

5. How would you modify the nested loops so that you print a square instead of a triangle?
6. What is the value of n after the following nested loops?

```
int n = 0;
for (int i = 1; i <= 5; i++)
    for (int j = 0; j < i; j++)
        n = n + j;
```

# Answers

---

5. Change the inner loop to

```
for (int j = 1; j <= width; j++)
```

6. 20

# Processing Sentinel Values

- One common method for indicating the end of a data set is a **sentinel value**, a value that is not part of the data. Instead, the sentinel value indicates that the data has come to an end
- Some programmers choose numbers such as **0** or **-1** as sentinel values. A better idea is to use an input that is not a number, such as the letter **Q**
- Of course, we need to read each input as a string, not a number. Once we have tested that the input is not the letter **Q**, we convert the string into a number:

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
    We are done
else
{
    double x = Double.parseDouble(input);
    . . .
}
```

# Symmetric and Asymmetric Bounds

- It is easy to write a **symmetric** loop with **i** going from **1** to **n**:

```
for (i = 1; i <= n; i++)
```

- But, when traversing the characters in a string, the bounds are **asymmetric**

```
for (i = 0; i < str.length(); i++)
```

- The values for **i** are bounded by  $0 \leq i < \text{str.length}()$ , with a  $\leq$  comparison to the left and a  $<$  comparison to the right
- It is not a good idea to force symmetry artificially:

```
for (i = 0; i <= str.length() - 1; i++)
```

# 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
    }
}
```

# The **break** and **continue** statements

- In addition to breaking out of a **switch** statement, a **break** statement can also be used to exit a **while**, **for**, or **do** loop
- In Java, there is a second form of the **break** statement that is used to break out of a nested statement. The statement **break label;** immediately jumps to the end of the statement that is tagged with a label
- Finally, there is another **goto**-like statement, the **continue** statement, which jumps to the end of the current iteration of the loop

```
while (!done)
{
    String input = in.next();

    if (input.equalsIgnoreCase("Q"))
    {
        done = true;
        continue; // Jump to the end of the loop body
    }
    double x = Double.parseDouble(input);

    data.add(x);

    // continue statement jumps here
}
```

# File InputTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program computes the average and maximum of a set
05:     of input values.
06: */
07: public class InputTester
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         DataSet data = new DataSet();
13:
14:         boolean done = false;
15:         while (!done)
16:         {
```

*Continued...*

# File InputTester.java

```
17:         System.out.print("Enter value, Q to quit: ");
18:         String input = in.next();
19:         if (input.equalsIgnoreCase("Q"))
20:             done = true;
21:         else
22:         {
23:             double x = Double.parseDouble(input);
24:             data.add(x);
25:         }
26:     }
27:
28:     System.out.println("Average = " + data.getAverage());
29:     System.out.println("Maximum = " + data.getMaximum());
30: }
31: }
```

# File DataSet.java

```
01: /**
02:     Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:     /**
07:         Constructs an empty data set.
08:     */
09:     public DataSet()
10:     {
11:         sum = 0;
12:         count = 0;
13:         maximum = 0;
14:     }
15:
16:     /**
17:         Adds a data value to the data set
18:         @param x a data value
19:     */
```

*Continued...*

# File DataSet.java

```
20:     public void add(double x)
21:     {
22:         sum = sum + x;
23:         if (count == 0 || maximum < x) maximum = x;
24:         count++;
25:     }
26:
27:     /**
28:      * Gets the average of the added data.
29:      * @return the average or 0 if no data has been added
30:      */
31:     public double getAverage()
32:     {
33:         if (count == 0) return 0;
34:         else return sum / count;
35:     }
36:
```

*Continued...*

# File DataSet.java

```
37:     /**
38:         Gets the largest of the added data.
39:         @return the maximum or 0 if no data has been added
40:     */
41:     public double getMaximum()
42:     {
43:         return maximum;
44:     }
45:
46:     private double sum;
47:     private double maximum;
48:     private int count;
49: }
```

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

Output

# Self Check

---

7. Why does the `InputTester` class call `in.next` and not `in.nextDouble`?
8. Would the `DataSet` class still compute the correct maximum if you simplified the update of the `maximum` field in the `add` method to the following statement?

```
if (maximum < x) maximum = x;
```

# Answers

---

7. Because we don't know whether the next input is a number or the letter Q.
8. No. If *all* input values are negative, the maximum is also negative. However, the `maximum` field is initialized with 0. With this simplification, the maximum would be falsely computed as 0.

# Random Numbers and Simulations

- In a **simulation**, you generate random events and evaluate their outcomes
- The **Random** class of the Java library implements a **random number generator**, which produces numbers that appear to be completely random. To generate random numbers, you construct an object of the **Random** class, and then apply one of the following methods, **nextInt(n)** or **nextDouble()**:

```
Random generator = new Random();  
int n = generator.nextInt(a); // 0 <= n < a  
double x = generator.nextDouble(); // 0 <= x < 1
```

- Example: throw die (random number between 1 and 6)

```
int d = 1 + generator.nextInt(6);
```

# File Die.java

```
01: import java.util.Random;
02:
03: /**
04:     This class models a die that, when cast, lands on a
05:     random face.
06: */
07: public class Die
08: {
09:     /**
10:         Constructs a die with a given number of sides.
11:         @param s the number of sides, e.g. 6 for a normal die
12:     */
13:     public Die(int s)
14:     {
15:         sides = s;
16:         generator = new Random();
17:     }
18:
```

*Continued...*

# File Die.java

```
19:     /**
20:         Simulates a throw of the die
21:         @return the face of the die
22:     */
23:     public int cast()
24:     {
25:         return 1 + generator.nextInt(sides);
26:     }
27:
28:     private Random generator;
29:     private int sides;
30: }
```

# File DieTester.java

```
01: /**
02:     This program simulates casting a die ten times.
03: */
04: public class DieTester
05: {
06:     public static void main(String[] args)
07:     {
08:         Die d = new Die(6);
09:         final int TRIES = 10;
10:         for (int i = 1; i <= TRIES; i++)
11:         {
12:             int n = d.cast();
13:             System.out.print(n + " ");
14:         }
15:         System.out.println();
16:     }
17: }
```

Output

6 5 6 3 2 6 3 4 4 1

Second Run

3 2 2 1 6 5 3 4 1 2

# Self Check

---

9. How do you use a random number generator to simulate the toss of a coin?

# Answers

---

9. `int n = generator.nextInt(2); // 0 = heads, 1 = tails`

# Chapter Summary

---

- A **while** statement executes a block of code repeatedly. A condition controls how often the loop is executed
- An **off-by-one** error is a common error when programming loops. Think through simple test cases to avoid this type of error
- You use a **for** loop when a variable runs from a starting to an ending value with a constant increment or decrement
- Loops can be **nested**. A typical example of nested loops is printing a table with rows and columns
- Sometimes, the termination condition of a loop can only be evaluated in the middle of a loop. You can introduce a **sentinel** (Boolean) variable to control such a loop
- Make a choice between symmetric and **asymmetric** loop bounds
- Count the number of iterations to check that your loop is correct
- In a **simulation**, you repeatedly generate random numbers and use them to simulate an activity