Chapter 7

Repetition Structures

The while, do-while, and for-statements

Outline

The while-statement (while-loop)
String to integer conversion example
Square root and factorization examples
Sentinel controlled while-loops
Query controlled while-loops
Do-while statement (do-while loop)
General loop structures
For-statement (for-loop)
Computing big factorials
Loan repayment table example
Nested loops
Investment table example
Graphing a function
Recursion and loops
Recursive factorial and gcd examples
Repetition Structures

7.1 Introduction

In many algorithms we need to repeat the execution of a sequence of one or more statements several times. For a known small number of repetitions you could simply write down the statements to repeat several times. This is not practical if the number of repetitions is large and doesn’t work at all if the number of repetitions is not known in advance. Therefore all high-level languages provide one or more repetition structures that specify a sequence of statements to be repeated and a condition to continue or terminate the repetitions. A repetition structure is often called a loop and is the third and last of the basic programming structures. The first two are the sequential structure and the conditional structure.

In this chapter we introduce both the pseudo-code and Java forms of the three repetition structures: the while-statement, the do-while statement, and the for-statement. We will see that the for-statement and the do-while statement are just special types of the while-statement that are provided for convenience.

The while and do-while statements are normally used when the number of repetitions is not known in advance but depends on some condition. We discuss several important variations such as the sentinel controlled while-loop, and the query controlled while-loop.

The for-statement is normally used when the number of repetitions can be determined in advance, either as a constant or as the value of an expression.

The connection between loops and simple recursion is also discussed.

7.2 The while-statement (while-loop)

If the number of repetitions is not known in advance but depends on some condition, given by a boolean expression, then the while-statement (also called a while-loop) can be used. In pseudo-code it can be expressed as

\[
\text{WHILE BooleanExpression DO} \quad \text{Statements} \quad \text{END WHILE}
\]

In Java the while-loop has the structure shown in Figure 7.1. BooleanExpression is a boolean valued expression and Statements is a sequence of zero or more statements defining the block to be repeated. As in the case of the if-statement, the parentheses surrounding the boolean expression are necessary and the braces can be omitted only if there is one statement to be repeated.
When the while-loop is encountered BooleanExpression is evaluated. If the value is true then
the statements in the block are executed, otherwise the while-loop is ignored and control resumes
below it. Each time the statements in the loop are executed the boolean expression is re-evaluated
to determine if the statements should be executed again.

Once the loop is entered the only way out is to have the statements in the body of the loop
change the value of one or more of the variables defining the boolean expression, causing it’s value
to become false. Otherwise we have what is called an infinite loop.

A flowchart for the execution of the while-statement is shown in Figure 7.2. It clearly indicates
that to avoid an infinite loop there must be a statement in the body of the loop that eventually forces
the boolean expression Expr to become false.

Here are some simple examples of while-loops.

**Example 7.1 (Counting up with a while-loop)** The statements

```java
int count = 1;
while (count <= 10)
{
    System.out.print(count + " ");
    count = count + 1; // or use count++
}
```

display 1 2 3 4 5 6 7 8 9 10. Initially, the expression count <= 10 is true since count is
initialized to 1. Therefore the loop is entered and 1 is displayed. Since count is incremented
each time through the loop, the expression count <= 10 will become false when it reaches 11.
Therefore the last number displayed is 10. To try this example using the BeanShell workspace and
ditor choose “Capture System in/out/err” from the “File” menu.

**Example 7.2 (Counting down with a while-loop)** The statements
count backwards beginning at 10 and display 10 9 8 7 6 5 4 3 2 1. When count is decre-mented to 0 the boolean expression count \( \geq 1 \) becomes false and the loop terminates. Therefore the last number displayed is 1. To try this example using the BeanShell workspace and editor choose “Capture System in/out/err” from the “File” menu.

\[\text{EXAMPLE 7.3} \; \text{(Loop that may not terminate)} \] Given the integer \( n > 0 \) consider the loop

\[
\text{long } k = n;  \\
\text{System.out.print}(k);  \\
\text{while } (k > 1)  \\
\{  \\
\quad \text{if } (k \% 2 == 1) \; / / k \text{ is odd}  \\
\quad \{  \\
\quad \quad k = 3*k + 1;  \\
\quad \}  \\
\quad \text{else } \; / / k \text{ is even}  \\
\quad \{  \\
\quad \quad k = k / 2;  \\
\quad \}  \\
\text{System.out.print}(""," + k);  \\
\}
\]

Here the value of \( k \) is changing each time through the loop but it is not clear that \( k \) will eventually become 1 to stop the loop. In fact no one knows for arbitrary \( n > 0 \) if the loop will terminate. You might become famous if you can prove this for any integer \( n \). If \( n = 8 \) the loop displays 8, 4, 2, 1 and stops, if \( n = 7 \) the loop displays 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 and stops, and if \( n = 27 \), then 112 numbers are displayed ending in 1. This loop also ignores the fact that the calculation of \( 3 \times k + 1 \) may produce overflow. This can detected using the if-statement

\[
\text{if } (k > 3074457345618258602L)  \\
\text{System.out.println}(\text{"Overflow has occurred"});
\]

just before the calculation. The strange long integer literal here is the largest one such that \( 3 \times k + 1 \) does not produce overflow.

\[\text{EXAMPLE 7.4} \; \text{(Drawing some horizontal lines)} \] Suppose we want to draw 10 horizontal lines each separated by 20 pixels. Each line should begin at \( x = 10 \) and end at \( x = 200 \). The top line should have \( y = 10 \). Then the left end of line \( k \) has coordinates \((10, 10 + 20k)\) for \( k = 0, 1, \ldots, 9 \) and the right end has coordinates \((200, 10 + 20k)\). Assuming that \( g2D \) is the graphics context (see Chapter 5), the while-loop
7.2 The while-statement (while-loop)  

```java
int k = 0;
while (k <= 9)
{
    double y = 10.0 + 20.0*k; // y coordinate of line k
    g2D.draw(new Line2D.Double(10,y,200,y));
    k = k + 1; // or k++;
}
```

can be used to draw the lines. The temporary variable `y` is declared in the loop body so it is local to the loop body.

A common error in examples like these is to forget to initialize the boolean expression before entering the loop, or to forget to update it inside the loop. This often results in infinite loops.

7.2.1 Converting a digit string to an integer

Most computer languages provide methods for converting a numerical string to an integer or a floating point number. In Java we have the following static methods.

- `public int Integer.parseInt(String s)`
  Static method in the `Integer` class to convert `s` to an `int` value and return it.

- `public long Long.parseLong(String s)`
  Static method in the `Long` class to convert `s` to a `long` value and return it.

- `public float Float.parseInt(String s)`
  Static method in the `Float` class to convert `s` to a `float` value and return it.

- `public double Double.parseDouble(String s)`
  Static method in the `Double` class to convert `s` to a `double` value and return it.

Each of these methods throws a `NumberFormatException` if the string `s` is not a valid number of the appropriate type. The classes `Integer`, `Long`, `Double` and `Float` are called **wrapper classes**. They contain useful methods that operate on the corresponding primitive types.

As an example we show how to convert a digit string of the form `s = c_0 c_1 \ldots c_{n-1}` where the `c_k` are the digit characters to an integer of the form `d = d_0 d_1 \ldots d_{n-1}` where each digit `d_k = c_k - '0'` is obtained by subtracting the code of the character '0' from the code of `c_k`. For example, the digits '0' to '9' have codes 48 to 57 so subtracting the code for '0' (48) from each gives the integers 0 to 9. The integer value of the string is accumulated in a loop using the formula

\[ d = d_{n-1} + 10(d_{n-2} + \cdots + 10(d_1 + 10d_0)) \cdots \]

A pseudo-code algorithm is given in Figure 7.3.

**Example 7.5** (Converting a string to an integer) The following method
ALGORITHM stringToInt \((c_0, c_1, \ldots, c_{n-1})\)
\[
\begin{align*}
\text{value} & \leftarrow 0 \\
\text{k} & \leftarrow 0 \\
\text{WHILE} & \quad k < n \text{ DO} \\
& \quad \text{value} \leftarrow (c_k - '0') + 10 \times \text{value} \\
& \quad \text{k} \leftarrow \text{k} + 1 \\
\text{END WHILE} \\
\text{RETURN} & \quad \text{value}
\end{align*}
\]

Figure 7.3: Algorithm to convert a string to an integer

gives a Java implementation of the algorithm in Figure 7.3. In Java an automatic typecast is performed to convert the \textit{char} type to the \textit{int} type. The \texttt{stringToInt} method is easily tested using the \textit{BeanShell} workspace and editor.

Here is a simple Java class that can be used to test the method in BlueJ.

```java
package chapter7.conversion;

/**
 * A class to test the stringToInt method that converts a string to an int.
 */
public class StringToIntConverter {

    /**
     * convert a string of digits to an int
     * @param s the digit string to convert
     * @return int value of digit string
     */
    public int stringToInt(String s) {
        int numDigits = s.length();
        int value = 0;
        int k = 0;
        while (k < numDigits) {
            value = (s.charAt(k) - '0') + 10 * value;
            k = k + 1;
        }
        return value;
    }
}
```

Here is a runner class that can be used from the command-line or within BlueJ.

```java
class StringToIntRunner
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        StringToIntConverter converter = new StringToIntConverter();
        System.out.println("Enter digit string");
        String digitString = input.nextLine();
        int value = converter.stringToInt(digitString);
        System.out.println("int value is " + value);
    }
}
```

### 7.2.2 Square root algorithm using a while-loop

If we didn’t have the `Math.sqrt` function how could we calculate the square root of a non-negative number? There are many algorithms to do this. A famous one, although not the most efficient, for computing \( \sqrt{a} \) is to define the sequence of numbers \( x_0, x_1, \ldots, x_n, \ldots \), using the formula

\[
x_n = \frac{1}{2} \left( x_{n-1} + \frac{a}{x_{n-1}} \right), \quad \text{for } n = 1, 2, 3, \ldots
\]

This means that if we start with a value for \( x_0 \) and substitute \( n = 1 \) in this formula we get a value for \( x_1 \) defined in terms of \( x_0 \) by

\[
x_1 = \frac{1}{2} \left( x_0 + \frac{a}{x_0} \right)
\]

Then we can substitute \( n = 2 \) to get a value for \( x_2 \) defined in terms of \( x_1 \) by:

\[
x_2 = \frac{1}{2} \left( x_1 + \frac{a}{x_1} \right)
\]
So if we start with a value for $x_0$ we can compute the sequence of numbers. It can be shown that these numbers get closer and closer to $\sqrt{a}$, for any starting value $x_0 > 0$.

As an example, compute an approximation to $\sqrt{2}$ using $a = 2$, starting with $x_0 = 1$. Then $x_1 = (1/2)(1 + 2) = 3/2 = 1.5$, $x_2 = (1/2)(3/2 + 4/3) \approx 1.46667$, $x_3 \approx 1.41422$, and so on. These numbers get closer and closer to $\sqrt{2}$, which is approximately 1.41421.

A simple pseudo-code algorithm that uses the relative error (see Example 6.9) as a measure of the closeness of successive iterations is given in Figure 7.4.

**Example 7.6 (Square root method)** The following method

```java
public double squareRoot(double a) {
    double xOld = 1;
    double xNew = a;

    while (Math.abs((xNew - xOld) / xNew) > 1E-16 ) {
        xOld = xNew;
        xNew = 0.5 * (xOld + a / xOld);
    }
    return xNew;
}
```
gives a Java implementation of the algorithm in Figure 7.4. The `squareRoot` method is easily tested using the BeanShell workspace and editor.

Here is a simple Java class that can be used to test the method in BlueJ.

```java
package chapter7.square_root; // remove this line if you're not using packages
/**

Class SquareRootCalculator

*/
```
7.2 The while-statement (while-loop)

* A simple class to test the square root algorithm.
*/
public class SquareRootCalculator
{
    /**
     * Calculate the square root of a number.
     * @param a the number to take square root of
     * @return the square root of a.
     */
    public double squareRoot(double a)
    {
        ...
    }
}

If you want to see the iterations and watch them converge insert the statement

    System.out.println(xNew);

after the assignment to xNew in the while-loop. To check the results you could also print the value of xNew * xNew just before the return statement to verify how close the result is to the input value a, or you can print the value of Math.sqrt(a).

Here is a runner class that can be used from the command-line. It squares the root as a check and also compares the root with Math.sqrt.

Class SquareRootRunner

```java
package chapter7.square_root; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * Runner class to test squareRoot method
 */
public class SquareRootRunner
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        SquareRootCalculator calculator = new SquareRootCalculator();
        System.out.println("Enter number");
        double a = input.nextDouble();
        input.nextLine(); // eat end of line
double root = calculator.squareRoot(a);
        System.out.println("Square root of " + a + " is " + root);
        System.out.println("Square root is " + root + root);
        System.out.println("Square root using Math.sqrt() is " + Math.sqrt(a));
    }

    public static void main(String[] args)
    {
        new SquareRootRunner().run();
    }
```

```
7.2.3 Double your money problem

Consider the following problem:

"How many months does it take to double an initial investment of x dollars if the annual interest rate is r %, and interest is compounded monthly."

To develop an algorithm we need to know that if you have an amount of money $V$, and it accumulates interest at a rate $r$ for a period of time, then the interest at the end of the period is $rV$, and the value of $V$ at the end of the period is $V + rV = V(1 + r)$. For our problem the annual rate of $r$ % percent is converted into the monthly rate $r/100/12$ as a fraction so at the end of a month the amount is $V(1 + r/1200)$ where $V$ is the value at the end of the previous month. A pseudo-code version of the doubling algorithm is shown in Figure 7.5.

**Example 7.7 (Double your money method)** The following method

```java
define doublingTime(initialValue, annualRate)
    double value = initialValue;
    double monthlyRate = annualRate / 100.0 / 12.0;
    int month = 0;
    while (value < 2.0 * initialValue)
    {
        month = month + 1;
        value = value * (1.0 + monthlyRate);
    }
    return month;
```

gives a Java implementation of the algorithm in Figure 7.5. The doublingTime method can be tested using the BeanShell workspace and editor.
Here is a simple Java class that can be used to test the method in BlueJ.

```java
package chapter7.money; // remove this line if you’re not using packages

/**
 * A simple class for the double your money problem.
 */
public class DoubleYourMoney
{
    /**
     * Return the number of months to double your money.
     * @param initialValue the initial investment amount
     * @param annualRate annual interest rate in percent
     * @return the number of months for initial amount to double
     */
    public int doublingTime(double initialValue, double annualRate)
    {
        ...
    }
}
```

Here is a runner class that can be used from the command-line. It also converts the total number of months for doubling into years and months.

```java
package chapter7.money; // remove this line if you’re not using packages
import java.util.Scanner;

/**
 * Runner class to test squareRoot method
 */
public class DoubleYourMoneyRunner
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        DoubleYourMoney calculator = new DoubleYourMoney();
        System.out.println("Enter initial investment amount");
        double amount = input.nextDouble();
        input.nextLine(); // eat end of line
        System.out.println("Enter annual rate in percent");
        double rate = input.nextDouble();
        input.nextLine(); // eat end of line
        int totalMonths = calculator.doublingTime(amount, rate);
        int years = totalMonths / 12;
        int months = totalMonths % 12;
        System.out.println("The amount doubles in ")
```
7.2.4 Factorization of an integer

The fundamental theorem of arithmetic states that every integer can be expressed uniquely as a product of prime numbers arranged in increasing order. Recall that a prime number has no factors other than itself and 1 (e.g., 18 is not prime since 6 is a factor but 17 is prime). This means that every integer $n$ can be expressed uniquely as the product

$$n = p_1^{e_1} \times p_2^{e_2} \times \cdots \times p_k^{e_k},$$

where $p_1 < p_2 < \cdots < p_k$ are primes.

For example

$$140931360 = 2^5 \times 3^3 \times 5 \times 17 \times 19 \times 101$$

$$140931369 = 3^2 \times 239 \times 65519$$

We can write a method to do this factorization using a while-loop. The largest factor $t$ that an integer $q$ can have is $\sqrt{q}$ so we can use this condition to terminate the while-loop. The algorithm begins by using $t = 2$ as the first trial factor and $q = n$ as the first quotient. If $t$ is a factor of $q$ then we save the factor $t$ and divide it out of $q$ to get the next quotient.

This gives the pseudo-code algorithm in Figure 7.6. To obtain the next trial factor we choose 3 if $t$ is 2 otherwise $t$ is odd and we can choose the next odd factor $t + 2$.

To write a Java method for this algorithm we can keep track of factors by appending them to a string as they are obtained so that the output for $n = 140931360$ is expressed as the string

```
+ years + " years and " + months + " months";
```

```
public static void main(String[] args)
{
    new DoubleYourMoneyRunner().run();
}
```

```xml
Figure 7.6: Pseudo-code factorization algorithm
```

```
ALGORITHM factor(n)
    q ← n // initial quotient
    t ← 2 // initial trial factor
    WHILE t ≤ √q DO
        IF t divides q THEN
            Save t as a factor of q
            q ← q div t
        ELSE
            t ← next trial factor
        END IF
    END WHILE
    Save q as last factor
```
EXAMPLE 7.8 (Factorization method) The following method

```java
public String factor(int n)
{
    int q = n; // initial quotient
    int t = 2; // initial trial factor
    String factors = "<"; // string to hold factors

    // a factor cannot be larger than square root of quotient
    while (t <= q / t)
    {
        if (q % t == 0)
        {
            // t is a factor so append it to string and
            // divide it out of quotient

            factors = factors + t + ",";
            q = q / t;
        }
        else
        {
            // t is not a factor so get the next trial factor.
            // After 2 all trial factors will be odd.

            t = (t == 2) ? 3 : t + 2;
        }
    }

    factors = factors + q + ">";
    return factors;
}
```

implements the algorithm in Figure 7.6 by returning the factors of \( n \) as a string. The `factor` method can be tested using the BeanShell workspace and editor.

Here is a simple Java class that can be used to test the method in BlueJ.

### Class Factorizer

```
package chapter7.factors; // remove this line if you’re not using packages
/**
 * A class to test the factor method that finds all the prime factors
 * of a number and returns them in a string.
 */
public class Factorizer
```
Repetition Structures

```java
/**
 * Find all the factors of a number.
 * @param n the number to factor
 * @return the string containing the factors
 */
public String factor(int n)
{
 ...
}
```

The following class can be used within BlueJ or from the command-line to display the factorization of 10 consecutive integers given the first one.

```java
package chapter7.factors; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * A runner class to test Factorizer by displaying the factorization
 * of all numbers in a given range.
 */
public class FactorizerRunner
{
 /** Factorize 10 numbers starting with a given number
 * @param n the given number
 * @return string of form <f1,f2,...,fn>
 */
 public void displayFactors(int n)
{
 Factorizer f = new Factorizer();

 int k = n;
 while (k <= n + 9)
 {
    String factors = f.factor(k);
    System.out.println(k + " = " + factors);
    k++;
 }
}

public static void main(String[] args)
{
 Scanner input = new Scanner(System.in);
 System.out.println("Enter first value of n");
 int n = input.nextInt();
 input.nextLine(); // eat end of line
 new FactorizerRunner().displayFactors(n);
}
```

Some typical output is

```java
java FactorizerRunner
Enter first value of n
140931360
140931360 = <2,2,2,2,2,3,3,3,5,17,19,101>
140931361 = <227,383,1621>
140931362 = <2,11,11,13,44797>
140931363 = <3,31,1515391>
140931364 = <2,2,7,157,32059>
140931365 = <5,571,49363>
140931366 = <2,3,23488561>
140931367 = <353,399239>
140931368 = <2,2,2,17616421>
140931369 = <3,3,239,65519>
```

### 7.3 Sentinel-controlled while-loops

Let us write a program that read a series of student marks using console input and calculates their average. Assume that the number of marks entered by the user is not known in advance. Since the value of a valid mark ranges from 0 to 100, we can design the program so that it stops asking the user for numbers when the user enters a negative value to indicate the end of the input. This kind of fictitious value, used to indicate the end of the input data, is often called a **sentinel value**. If the first mark entered is negative, then the while-loop is never executed. In this case we assign an average of zero. The following method computes the average mark.

**Example 7.9 (Sentinel-controlled while-loop)** The following method computes the average of a list of marks and displays it.

```java
public void averageMark()
{
    Scanner input = new Scanner(System.in);
    double sum = 0.0;
    int numberOfMarks = 0;
    double mark;

    System.out.println("Enter mark (negative to quit)");
    mark = input.nextDouble();
    input.nextLine(); // eat end of line

    while (mark >= 0.0)
    {
        if (mark <= 100.0)
        {
            sum = sum + mark;
            numberOfMarks = numberOfMarks + 1;
        }
        mark = input.nextDouble();
    }

    if (numberOfMarks > 0)
    {
        double average = sum / numberOfMarks;
        System.out.println("Average mark: "+average);
    }
}
```
A common mistake in loops like this is to forget to read a new mark at the bottom of the loop before returning to the top again. The result is an infinite loop.

### 7.3.1 AverageMarkCalculator class

Here is a simple class to test the method from the command line or from within BlueJ.

```java
public class AverageMarkCalculator {
    /**
     * A class to illustrate a sentinel-controlled while loop
     */
    public class AverageMarkCalculator {
        /**
         * Read marks and compute average mark
         */
        public void averageMark()
        {
            ...}

        public static void main(String[] args)
        {
            new AverageMarkCalculator().averageMark();
        }
    }
}
```

### 7.4 Query-controlled while-loops

In a query-controlled interactive loop the user is asked at the end of every iteration if there is more data to enter. If the answer is yes, another iteration is performed by reading new data and
processing it. This kind of loop is needed when there is no sentinel value that can be used. For example, a program that reads an arbitrary series of numbers cannot use any one of them as a sentinel value.

For interactive console input a boolean-valued method, similar to the following one, can be used to test for-loop termination in a query-controlled loop.

```java
public boolean moreValues()
{
    System.out.println("Do you want to enter another value [Y/N ?]");
    String reply = input.nextLine();
    return reply.equals("") || reply.toUpperCase().charAt(0) == 'Y';
}
```

Here we assume that `input` is a `Scanner` object. The method returns `true` if the user enters a blank line or types something that begins with `y` or `Y`, indicating that the user wants to enter another value. Short-circuit evaluation is used for the boolean expression

```
reply.equals("") || reply.toUpperCase().charAt(0) == 'Y'
```

If the user enters a blank line, `reply` is empty and `reply.equals("")` is true, so the right operand of the “or expression” is never evaluated. Therefore, the evaluation of `charAt` is never attempted for an empty string.

Using this method, the query-controlled while-loop can be written as

```
while (moreValues())
{
    // read a value here
    // process the value here
}
```

This loop works even if there are no values to enter.

### 7.4.1 BankAccount example

We want to write a class that uses a query-controlled while-loop to find the bank account with the maximum balance from a list of accounts entered using the console. The heart of the class is the method

```java
public BankAccount findMaxBalance()
{
    BankAccount maxAccount = readAccount();
    while (moreAccounts())
    {
        BankAccount next = readAccount();
        if (next.getBalance() > maxAccount.getBalance())
        {
            maxAccount = next;
        }
    }
```
that finds the account with the maximum balance and returns a reference to it. This reference starts out as a reference to the first account and each time an account with a larger balance is read the maxAccount reference is updated to refer to this account. When the method exits all the account objects, except the one with the maximum balance referenced by maxAccount, will be orphans (no references to them) so they will be garbage collected.

Here is a Java class using this method that can be run within BlueJ or from the command-line.

```java
package chapter7.loops; // remove this line if you’re not using packages
import java.util.Scanner;

/**
 * A class to illustrate the query-controlled while loop by reading
 * a list of accounts from the console and finding the one that
 * has the maximum balance.
 */
public class MaxBalanceCalculator
{
    Scanner input = new Scanner(System.in);

    /**
     * Read accounts from console
     * @return reference to account having the maximum balance
     */
    public BankAccount findMaxBalance()
    {
        BankAccount maxAccount = readAccount();
        while (moreAccounts())
        {
            BankAccount next = readAccount();
            if (next.getBalance() > maxAccount.getBalance())
            {
                maxAccount = next;
            }
        }
        return maxAccount;
    }

    private boolean moreAccounts()
    {
        System.out.println("Do you want to enter another account [Y/N ?]");
        String reply = input.nextLine();
        return reply.equals("") || reply.toUpperCase().charAt(0) == 'Y';
    }

    // method body
}
```
private BankAccount readAccount()
{
    System.out.println("Enter account number");
    int number = input.nextInt();
    input.nextLine(); // eat end of line

    System.out.println("Enter owner name");
    String name = input.nextLine();

    System.out.println("Enter balance");
    double balance = input.nextDouble();
    input.nextLine(); // eat end of line
    return new BankAccount(number, name, balance);
}

public static void main(String[] args)
{
    MaxBalanceCalculator calc = new MaxBalanceCalculator();
    System.out.println("Account with maximum balance is " + calc.findMaxBalance());
}

In BlueJ use “Add class from files” to add the BankAccount class from the custom-classes project. The class also uses two private methods to read an account and do the query-controlled test for another account.

7.5 Do-while statement (do-while loop)

In the while-loop the test for loop termination is always done at the top of the loop. There are cases when we would like to do the test at the bottom of the loop. Several variations occur in programming languages. In pseudo-code we could use the structure

    REPEAT
    Statements
    WHILE BooleanExpression

which repeats Statements while the boolean expression is true. Alternatively we could use the negated form

    REPEAT
    Statements
    UNTIL BooleanExpression

which repeats Statements until the boolean expression is true (or equivalently, while it is not true). In either case, unlike the while-loop, the statements in the loop are always executed at least once. In practice the do-while loop is not as common as the while-loop.

Languages such as Pascal and Modula-2 have a repeat-until statement. Others, C, C++ and Java for example, have a repeat-while statement (called do-while) shown in Figure 7.7. The do-while statement needs a semi-colon at the end of the while-part.

A flowchart for the execution of the do-while statement is shown in Figure 7.8.
Repetition Structures

```java
do {
    Statements
} while (BooleanExpression);
```

Figure 7.7: A template for the do-while statement

![Flowchart](image)

Figure 7.8: A flowchart for the execution of a do-while statement

**EXAMPLE 7.10** (Counting up with a do-while loop) The statements

```java
int count = 1;
do {
    System.out.print(count + " ");
    count = count + 1;
} while (count <= 10);
```

display 1 2 3 4 5 6 7 8 9 10. After 10 is displayed count is incremented to 11, the boolean expression `count <= 10` becomes false and the loop exits. Try this example using the BeanShell workspace and editor. Also choose “Capture System in/out/err” from the “File” menu.

**EXAMPLE 7.11** (Counting down with a do-while loop) The statements

```java
int count = 10;
do {
    System.out.print(count + " ");
    count = count - 1;
} while (count >= 0);
```
7.6 General loop structures

The while-loop makes the test at the top of the loop and the do-while loop makes it at the bottom of the loop. It is possible to generalize and make the test somewhere in the middle of the loop. In pseudo-code we could invent a general loop-structure such as

\[
\text{LOOP} \\
\text{StatementsA} \\
\text{IF BooleanExpression THEN EXIT} \\
\text{StatementsB} \\
\text{END LOOP}
\]

Here an IF statement with a special EXIT statement is used to exit the loop somewhere in the middle if BooleanExpression is true. The while-loop is the special case that there is no StatementA block and the repeat-until loop (do-while in negated form) is the special case when there is no StatementB block.

General loops like this can be hard to read since there could be several if-exit statements in the loop. If possible you should always try to write your loops in the while or do-while forms or using the for-statement discussed next.

Java does not have a LOOP statement but it has a break statement corresponding to EXIT so we can write a general loop using the while-loop shown in Figure 7.9.

```java
while(true)
{
    StatementsA
    if (BooleanExpression) break;
    StatementsB
}
```

Figure 7.9: A template for a general loop structure.
7.7 For-statement (for-loop)

The for-statement (for-loop) is the last of the three repetition statements. It is used to repeat one or more statements a fixed number of times, determined in advance, either as a constant or as the value of an expression.

7.7.1 Pseudo-code for-loops for counting in steps

In pseudo-code the for-statement can be expressed as

```plaintext
FOR k ← start TO end BY step DO
    Statements
END FOR
```

or

```plaintext
FOR k ← start TO end BY −step DO
    Statements
END FOR
```

where we assume that \( step > 0 \). In the first case the BY part can be omitted in the most common case that \( step = 1 \).

Here \( k \) is called the loop variable (or the loop counter). It is initialized to the value \( start \) and is incremented or decremented automatically each time the Statements block is executed. The value of \( step \) determines how much is added to or subtracted from the loop counter each time the block is executed. The two cases define an upward counting loop and a downward counting loop and can be described as follows.

- **BY \( step \):** In this case the values of \( k \) are \( start, start + step, start + 2 \cdot step, \) and so on, increasing and ending with the last value such that \( k \leq end \). If \( start > end \) the for-loop is ignored.

- **BY \( −step \):** In this case the values of \( k \) are \( start, start − step, start − 2 \cdot step, \) and so on, decreasing and ending with the last value such that \( k \geq end \). If \( start < end \) the for-loop is ignored.

The important special cases occur when \( step = 1 \) for which the loops count upward from \( start \) to \( end \) in steps of 1 in the first case or downward from \( start \) to \( end \) in steps of 1 in the second case.

In Java the for-statement is needlessly complicated (to please C and C++ programmers) but we can easily use it to model the pseudo-code versions even though there are more general versions. A template is shown in Figure 7.10. As in the case of the while and do-while statements, the body of the for-statement must be enclosed in braces if it contains more than one statement.

The **Initialization** part normally involves a numeric variable declaration with initialization. This variable is the loop counter.

The **Test** part is a boolean expression which depends on the loop counter. If the value of the condition is true, the statements in the body of the loop are executed.

If the statements in the loop are executed (because **Test** was true initially) the **Update** statement is then executed. It’s purpose its to modify the loop counter (increase or decrease it). Then **Test** is
7.7 For-statement (for-loop)

for (Initialization ; Test ; Update )
{
    Statements
}

Figure 7.10: A template for the for-statement.

Figure 7.11: A flowchart for the execution of a for-statement

evaluated again to see if the loop statements should be executed again. Eventually Update should modify the counter so that Test will be false and terminate the loop.

The flowchart for the for-loop is shown in Figure7.11. The loop counter is commonly a variable of type int although a variable of type double can also be used.

7.7.2 For-loops for counting in steps

The first case of the pseudo-code for-statement can be expressed in Java as

```java
for (int k = start; k <= end; k = k + step)
{
    // loop statements
}
```

If step is positive the values of k count up from start, in steps of step, ending at the largest value of k less than or equal to end. The initialization part is just an initialized variable declaration state-
Repetition Structures

...
7.8 Computing factorials

We can use a for-loop to compute \( n! \). For values of type \( \text{int} \) we can do this before overflow only if \( 0 \leq n \leq 12 \). This range can be extended using the \( \text{long} \) data type (see Exercise 7.1). We can also calculate large factorials using objects from the \( \text{BigInteger} \) class in package \( \text{java.math} \) that represent arbitrarily large integers and perform arithmetic operations on them limited only by the amount of memory available.

7.8.1 Computing the factorial of an integer

For a non-negative integer \( n \), the factorial of \( n \), denoted by \( n! \), is defined by

\[
    n! = \begin{cases} 
        1 \times 2 \times \cdots \times n, & n > 0 \\
        1, & n = 0 
\end{cases}
\]

We can write a method that uses a for-loop to compute \( n! \). The algorithm is simple: initialize a product variable to 1, multiply it by 2, multiply it by 3, and so on, until it is multiplied by \( n \). There will be \( n - 1 \) multiplications:

\[
\text{Example 7.17 (Method for \( n! \)) The following method}
\]

```java
int factorial(int n)
{
    int product = 1;
    for (int k = 2; k <= n; k++)
```

Using the for-loop.

**Example 7.16 (Computing \( 1 + 2 + \cdots + n \))** Given a value for \( n \) the for-loop

```java
int sum = 0;
for (int k = 1; k <= n; k++)
{
    sum = sum + k;
}
```

computes the sum of the first \( n \) integers.

We shall see in the remainder of this Chapter and in Chapter 8 that the for-loop has many applications.
Repetition Structures

```java
product = product * k;
}
return product;
}
```
calculates \( n! \) and returns it. The `product` variable successively takes on the values \( 2 = 1 \times 2, 6 = 2 \times 3, 24 = 6 \times 4, \) etc. For \( n = 0, \) or \( n = 1, \) the loop will not be executed even once, because \( k \leq n \) is false if \( k \) is 2, so the final value of `product` will be 1. This agrees with the definition of \( n! \).

The method can easily be tested using the `BeanShell` editor and workspace. In particular you can verify that \( n \) must be in the range \( 0 \leq n \leq 12. \)

Here is a simple tester class that can be used in `BlueJ` using a modified method that throws an exception if \( n \) is outside the range \( 0 \leq n \leq 12. \)

**Class FactorialCalculator**

```java
package chapter7.factorial; // remove this line if you're not using packages
/**
 * A simple class to test the factorial method.
 */
public class FactorialCalculator
{
    /**
     * Calculate \( n! \).
     * @param n value for \( n! \)
     * @return \( n! \)
     * @throws IllegalArgumentException if \( n \) is outside the range
     * \( 0 \leq n \leq 12. \)
     */
    public int factorial(int n)
    {
        if (n < 0 || n > 12)
        {
            throw new IllegalArgumentException("\( n! \) is only defined for \( n = 0..12 \)");
        }
        int product = 1;
        for (int k = 2; k <= n; k++)
        {
            product = product * k;
        }
        return product;
    }
}
```

For command-line testing outside `BlueJ` the following class can be used.

**Class FactorialRunner**

```java
package chapter7.factorial;
```
7.8 Computing factorials

```java
package chapter7.factorial; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing factorial method from command line
 */
public class FactorialRunner{
  public void run()
  {
    Scanner input = new Scanner(System.in);
    FactorialCalculator calc = new FactorialCalculator();
    System.out.println("Enter value of n");
    int n = input.nextInt();
    System.out.println(n + "! = " + calc.factorial(n));
  }
  public static void main(String[] args)
  {
    new FactorialRunner().run();
  }
}
```

### 7.8.2 Computing factorials using the BigInteger class

Java has a `BigInteger` class in the `java.math` package which represents arbitrarily large integers and arithmetic operations limited only by the amount of available memory. We can use it to calculate large factorials. Each integer is represented as a `BigInteger` object. To convert a normal `int` or `long` integer to a `BigInteger` object there is the static method called `valueOf` with prototype

```
public static BigInteger valueOf(long val)
```

For example,

```
BigInteger bigI = BigInteger.valueOf(1);
```

converts the integer value 1 to a big integer. There is also a constructor that takes a string as an argument and uses it to construct a big integer object. It has the prototype

```
public BigInteger(String val)
```

For example,

```
BigInteger bigI = new BigInteger("111111111111111111111111111111111");
```

converts the given number string (too big even for type `long`) to a big integer.

There are `add`, `subtract`, `multiply` and `divide` methods for big integers. The `multiply` method has prototype

```
public BigInteger multiply(BigInteger val)
```

For example, the following statements multiply two big integer objects `b1` and `b2` to produce a new object which is assigned to `b3`. 
Finally, we need a way to convert a big integer result to a string, so that we can display it. To do this there is a `toString` method with prototype

```java
public String toString()
```

There are many other methods in the `BigInteger` class (see Java class documentation) but we don’t need them to compute big factorials.

To define and initialize the `product` variable to 1 we can use the static `valueOf` method:

```java
BigInteger product = BigInteger.valueOf(1);
```

To obtain a big integer object representing the integer loop counter `k` we can use the `valueOf` method:

```java
BigInteger bigK = BigInteger.valueOf(k);
```

We don’t need to make the loop counter `k` into a big integer. To multiply `product` by `bigK` in the loop use

```java
product = product.multiply(bigK);
```

**Example 7.18** (Big integer method for \( n! \)) The following method

```java
BigInteger bigFactorial(int n) {
    BigInteger product = BigInteger.valueOf(1);
    for (int k = 2; k <= n; k++) {
        BigInteger bigK = BigInteger.valueOf(k);
        product = product.multiply(bigK);
    }
    return product;
}
```

is the `BigInteger` version of \( n! \).

To test this method we need a way to display the large answers. For example, 200! has 375 digits (`toString` returns a string of 375 characters). To do this we can break the string into blocks of a given number of characters per line using the method

```java
private void displayLongString(String s, int width) {
    int length = s.length();
    int numberOfLines = length / width;
    for (int k = 0; k < numberOfLines; k++) {
        System.out.println(s.substring(k * width, (k + 1) * width));
    }
}
```
7.8 Computing factorials

System.out.println(s.substring(k * width, (k+1) * width));
if (length % width != 0) // display a final partial line
    System.out.println(s.substring(numberOfLines * width));

which displays width characters of s per line by extracting substrings.

Here is a class that uses this method to test the bigFactorial method.

```java
public class BigFactorialCalculator
{
    public void displayFactorial(int n)
    {
        String s = bigFactorial(n).toString();
        System.out.println("Number of digits is "+ s.length());
        System.out.println(n + "! = ");
        displayLongString(s, 60);
    }

    private BigInteger bigFactorial(int n)
    {
        ...
    }

    private void displayLongString(String s, int width)
    {
        ...
    }
}
```

For command-line testing the following class can be used.

```java
public class BigFactorialRunner
{
    public static void main(String[] args)
    {
        BigFactorialCalculator calculator = new BigFactorialCalculator();
        calculator.displayFactorial(10); // Example usage
    }
}
```
package chapter7.factorial; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing big factorial method from command line
 */
public class BigFactorialRunner
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        BigFactorialCalculator calc = new BigFactorialCalculator();
        System.out.println("Enter value of n");
        int n = input.nextInt();
        calc.displayFactorial(n);
    }

    public static void main(String[] args)
    {
        new BigFactorialRunner().run();
    }
}

The output for 200! is

```
java BigFactorialRunner
Enter value of n
200
200! =
788657867364790503552363213932185062295135977687173263294742
533244359449963403342920304284011984623904177212138919638830
257642790242637105061926624952829931113462857270763317237396
988943922445621451664240254033291864131227428294853277524242
407573903240321257405579568660226031904170324062351700858796
17892222789623703897374720000000000000000000000000000000000
00000000000
```

7.9 Expressing the for-loop as a while-loop

The for-loop shown in Figure 7.10 can be expressed as the while-loop shown in Figure 7.12, but the for-loop is simpler in these cases.

- **EXAMPLE 7.19** (Comparing the for and while-loops) The for-loop and while-loop for calculating factorials are

Using a for-loop
```
product = 1;
for (int k = 2; k <= n; k++)
{
    product = product * k;
}
```

Using a while-loop
```
product = 1;
int k = 2;
while (k <= n)
{
    product = product * k;
```
7.10 Loan repayment table

We want to write a program that solves the following problem

“Given the amount of a loan (principal), the number of years to pay back the loan, the number of payments per year, and the annual interest rate, produce a loan repayment table for each payment period showing the principal repaid and the principal remaining.”

We need some financial mathematics to solve this problem. Let us define the following quantities

\[ A \quad \text{the amount of the loan (principal)} \]
\[ y \quad \text{the number of years to pay back the loan} \]
\[ m \quad \text{the number of payments per year (periods per year)} \]
\[ j \quad \text{the annual interest rate as a decimal number} \]

Using these quantities we need to compute the following quantities.

\[ n = my, \quad \text{the total number of payments}, \]
\[ i = \frac{j}{m}, \quad \text{the interest rate per payment period as a decimal number}, \]
\[ R = \frac{A}{a(n,i)}, \quad \text{the payment made at the end of each payment period}, \]

where

\[ a(n,i) = \frac{1 - (1+i)^{-n}}{i} \]
The loan repayment table has the form shown in Table 7.1. The following properties of the table can be used as checks:

1. \( P_1, P_2, P_3, \ldots \), satisfy \( P_2/P_1 = P_3/P_2 = \cdots = 1 + i \)
2. In each row the entries in columns 3 and 4 sum to \( R \)
3. The total of column 2 is the total amount paid
4. The total of column 3 is the total interest paid
5. The total of column 4 is \( A \), the amount of the loan
6. The entry in column 5 should be 0 after \( n \) payments

A pseudo-code algorithm for producing the table is shown in Figure 7.13.

7.10.1 Right justifying numbers in a field of given width

We want to produce a nicely formatted table that looks like
with the columns right-justified in fields and numbers displayed with two digits after the decimal point. We can use the `String.format` method from Chapter 4.2.5 to do this. The format code `%7d` can be used to format the first column of integers and the code `%12.2f` can be used to format the remaining fields.

### 7.10.2 `StringBuilder` class

We want to make our class general so we do not produce the table using `System.out.print` statements. Instead we will return the entire table as a big formatted string. That way when we discuss graphical user interfaces (GUI’s) in a later Chapter, where `System.out.print` has no meaning, we can use our loan repayment class unchanged.

Since there are a lot of string manipulations it is more appropriate to use the `StringBuilder` class in package `java.lang`. This is a mutable version of the `String` class that is more efficient to use when performing a lot of string manipulations. The constructor and method prototypes from this class that we need are

- **public `StringBuilder`(int size)**
  
  Construct an empty string buffer with space for size characters initially. The size will expand as needed.

- **public `void append(String s)`**
  
  Append the given string s to the end of the buffer (like + for string concatenation). There are several versions of `append` that have `int` and `double` and other types of arguments. In any case the argument is converted to a string and appended to the string buffer.

- **public `String toString()`**
  
  Convert the string buffer to a `String` object. We usually do this when we are finished creating the string buffer.

### 7.10.3 Loan repayment table class

The public interface of our class is

```java
public class LoanRepaymentTable {
    public LoanRepaymentTable(double a, int y, int p, double r) {...}
    public String toString() {...}
}
```
Here $a$ is the amount of the loan, $y$ is the number of years for repayment, $p$ is the number of payments per year and $r$ is the annual rate in percent. The `toString` method will return the table as one big string. The complete class is

```java
package chapter7.loan_repayment; // remove this line if you’re not using packages

/**
 * A class to compute a loan repayment table, given the loan amount, the
 * number of years to repay the loan, the number of payments made per year,
 * and the annual interest rate in percent.
 */
public class LoanRepaymentTable
{
    private double loanAmount; // initial amount of the loan
    private int years; // years to pay back the loan
    private int paymentsPerYear;
    private double annualRate; // as a fraction
    private String table; // the loan repayment table

    /**
     * Construct a loan repayment table.
     * @param a the given amount of the loan
     * @param y the number of years to pay it back
     * @param p the number of payments per year
     * @param r the annual interest rate in percent
     */
    public LoanRepaymentTable(double a, int y, int p, double r)
    {
        loanAmount = a;
        years = y;
        paymentsPerYear = p;
        annualRate = r / 100.0; // convert percent to a fraction
        computeTable();
    }

    /**
     * Return the loan repayment table.
     * @return the loan repayment table.
     */
    public String toString()
    {
        return table;
    }

    /* Construct the entire table as one big string that
     contains newlines to break the table into lines
     */
    private void computeTable()
    {
        // Implementation...
    }
}
```
int n = paymentsPerYear * years;
double i = annualRate / paymentsPerYear;
double payment = loanAmount / a(n,i);
double principalRemaining = loanAmount;
double interest;
double principalRepaid;

// append headings
StringBuilder buffer = new StringBuilder(1000);

buffer.append("Payment Number Payment Paid Interest Repaid Principal Remaining\n");
buffer.append("------- -------- ------ ------- ---- ---- \n");

// Calculate table and append rows to buffer

double totalInterestPaid = 0.0;
double totalPrincipalPaid = 0.0;
for (int paymentNumber = 1; paymentNumber <= n; paymentNumber++)
{
    interest = principalRemaining * i;
    principalRepaid = payment - interest;
    principalRemaining = principalRemaining - principalRepaid;
    buffer.append(String.format("%7d", paymentNumber));
    buffer.append(String.format("%12.2f", payment));
    buffer.append(String.format("%12.2f", interest));
    buffer.append(String.format("%12.2f", principalRepaid));
    buffer.append(String.format("%12.2f", principalRemaining));
    buffer.append("\n");
    totalInterestPaid += interest;
    totalPrincipalPaid += principalRepaid;
}

double totalLoanCost = totalInterestPaid + totalPrincipalPaid;
buffer.append("\n");
buffer.append("Total interest paid is " +
    String.format("%.2f", totalInterestPaid) + "\n");
buffer.append("Total premium paid is " +
    String.format("%.2f", totalPrincipalPaid) + "\n");
buffer.append("Total cost of loan is " +
    String.format("%.2f", totalLoanCost) + "\n");
table = buffer.toString();
}

private double a(int n, double i)
{
    return (1.0 - Math.pow(1.0 + i, -n)) / i;
}
Most of the work is done by the `computeTable` method which is called by the constructor. It initializes a `StringBuilder` object called `buffer` and appends rows of the table and new line characters. When the calculations are complete the buffer is returned from the method as the instance data field `table` which can be obtained using the `toString` method.

### 7.10.4 Console user interface

Here is a runner class that can be used in BlueJ and from the command line.

```java
package chapter7.loan_repayment; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Class for running LoanRepaymentTable from console.
 */
public class LoanRepaymentTableRunner {
    public void run()
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter loan amount");
        double a = input.nextDouble();
        input.nextLine();
        System.out.println("Enter number of years");
        int y = input.nextInt();
        input.nextLine();
        System.out.println("Enter number of payments per year");
        int p = input.nextInt();
        input.nextLine();
        System.out.println("Enter annual interest rate in percent");
        double r = input.nextDouble();
        input.nextLine();

        LoanRepaymentTable table = new LoanRepaymentTable(a, y, p, r);
        System.out.println(table);
    }

    public static void main(String[] args)
    {
        new LoanRepaymentTableRunner().run();
    }
}
```

The entire table is displayed in the console or terminal window by the single statement

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

The console output for a $10,000 loan, at 10% per year, for a period of 5 years, with payments twice a year is
7.11 Nested loops

One of the statements in the body of a loop could be another loop statement. We say that the inner loop is nested within the outer loop. The inner loop will be executed for each iteration of the outer one. Nested for-loops are quite common for processing data which has a two-dimensional representation as a number of rows and columns. We consider several simple examples:

**Example 7.20** (5 rows of 10 circles) Suppose we want to draw the grid of circles shown in Figure 7.14. Assuming that the radius of each circle is 20, then the top left corner coordinates of the bounding boxes are \((40 \times \text{column}, 40 \times \text{row})\) where \(\text{row}\) goes from 0 to 4 and \(\text{column}\) goes from 0 to 9. The width of each box is 40. Therefore, the following nested loop draws the circles, assuming that \(\text{g2D}\) is a Graphics2D reference.

```java
double size = 40.0;
for (int row = 0; row <= 4; row++)
{
    double yTopLeft = size * row;
    for (int column = 0; column <= 9; column++)
    {
        double xTopLeft = size * column;
        g2D.draw(new Ellipse2D.Double(xTopLeft, yTopLeft, size, size));
    }
}
```

```java
java LoanRepaymentTableRunner
Enter loan amount
10000
Enter number of years
5
Enter number of payments per year
2
Enter annual interest rate in percent
10
Payment Payment Interest Principal Principal
Number Paid Repaid Remaining
------- ------- -------- --------- ---------
1 1295.05 500.00 795.05 9204.95
2 1295.05 460.25 834.80 8370.16
3 1295.05 418.51 876.54 7493.62
4 1295.05 374.68 920.36 6573.25
5 1295.05 328.66 966.38 5606.87
6 1295.05 280.34 1014.70 4592.17
7 1295.05 229.61 1065.44 2408.02
8 1295.05 176.34 1118.71 1233.38
9 1295.05 120.40 1174.64 1233.38
10 1295.05 61.67 1233.38 0.00
Total interest paid is 2950.46
Total premium paid is 10000.00
Total cost of loan is 12950.46
```
For each value of row, the inner loop draws an entire row of 10 circles indexed by the value of column.

**Example 7.21** (Square pattern) You can experiment with nested loops using console output. For example, the nested loop statement

```java
for (int row = 1; row <= 4; row++)
{
    for (int column = 1; column <= 10; column++)
    {
        System.out.print("*");
    }
    System.out.println();
}
```

displays the rectangular pattern

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

with 4 rows of 10 asterisks. Try this example using the BeanShell workspace and editor.

**Example 7.22** (Triangular pattern) In Example 7.20 and Example 7.21, the inner loop index column did not depend on the outer loop index row. In the nested loop

```java
for (int row = 1; row <= 4; row++)
{
    for (int column = 1; column <= row; column++)
    {
        System.out.print("*");
    }
    System.out.println();
}
```
the inner loop index depends on the outer one and the following triangular pattern is displayed.

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

For the first iteration of the outer loop (row = 1), the inner loop index goes from 1 to 1 so one asterisk is displayed. For the second iteration of the outer loop (row = 2), the inner loop index goes from 1 to 2, so two asterisks are displayed. Thus, each row contains one more asterisk than the preceding one. The total number of asterisks displayed is \(1 + 2 + \cdots + n = n(n + 1)/2\), which is 10 in this case. Try this using the BeanShell workspace and editor.

**Example 7.23** (Doubly-nested loop for computing powers) The following loop structure computes the second to fifth powers of the numbers one to 10. For a given \(n\) defining a row of the table the row contains the numbers \(n, n^2, n^3, n^4\) and \(n^5\). Thus \(n\) is the outer loop row index and \(p\), the power, is the inner loop column index. The double loop is given by

```java
int val;
for (int n = 1; n <= 10; n++)
{
    System.out.printf("%5d", n);
    val = n;
    for (int p = 2; p <= 5; p++)
    {
        val = val * n;
        System.out.printf("%8d", val);
    }
    System.out.println();
}
```

The output is

```
  1   1   1   1   1
  2   4   8  16  32
  3   9  27  81 243
  4  16  64 256 1024
  5  25 125 625 3125
  6  36 216 1296 7776
  7  49 343 2401 16807
  8  64 512 4096 32768
  9  81 729 6561 59049
 10 100 1000 10000 100000
```

For row \(n\) each value in the inner loop is obtained from the one to its left by multiplying by \(n\). We never need to use the `Math.pow` method.
7.11.1 Investment table

Let us write a program to print a table showing the value of an investment for different interest rates and different numbers of years. More specifically,

“Given an initial investment amount, compute a future value table for different rates from a minimum rate of 4% to a maximum rate of 10% in steps of 0.5%, and for an investment time of 5 to 30 years in steps of 5 years. The rows of the table correspond to the rates, and the columns correspond to the number of years.”

For example, if the initial investment is $1000 the following table shows the value of the investment, with some rows omitted:

<table>
<thead>
<tr>
<th>RATE</th>
<th>5 YEARS</th>
<th>10 YEARS</th>
<th>15 YEARS</th>
<th>20 YEARS</th>
<th>25 YEARS</th>
<th>30 YEARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00</td>
<td>1221.00</td>
<td>1490.83</td>
<td>1820.30</td>
<td>2222.58</td>
<td>2713.77</td>
<td>3313.50</td>
</tr>
<tr>
<td>4.50</td>
<td>1251.80</td>
<td>1566.99</td>
<td>1961.56</td>
<td>2455.47</td>
<td>3073.74</td>
<td>3847.70</td>
</tr>
<tr>
<td>9.50</td>
<td>1605.01</td>
<td>2576.06</td>
<td>4134.59</td>
<td>6636.06</td>
<td>10650.94</td>
<td>17094.86</td>
</tr>
<tr>
<td>10.00</td>
<td>1645.31</td>
<td>2707.04</td>
<td>4453.92</td>
<td>7328.07</td>
<td>12056.94</td>
<td>19837.40</td>
</tr>
</tbody>
</table>

For example, at 4.5% an investment of $1000 is worth $3073.74 after 25 years. This table can be produced by a nested for-loop. The outer loop goes over the rows of the table, and the inner loop goes over the columns. We can generalize and use variables for the investment amount, the minimum rate, the maximum rate, the rate step from one row to the next, the minimum number of years, the maximum number of years, and the year step from one column to the next. The nested loop structure has the form

```java
double small = 0.00001;
for (double rate = minRate; rate <= maxRate + small; rate += rateStep)
{
    ...
    for (int years = minYears; years <= maxYears; years += yearStep)
    {
        ...
    }
    ...
}
```

The outer loop goes over the rows of the table and the inner loop goes across the columns in a row. We add a small constant to `maxRate` in case there is roundoff error to force `maxRate` to be reached.

The formula for the future value $F$ of an amount $A$, with a yearly rate of $r$ percent, compounded monthly, for $n$ years is

$$F = A \left( 1 + \frac{r}{1200} \right)^{12n}$$

We can use this formula to write the following method for calculating the future value.

```java
private double futureValue(double presentValue, double yearlyRate, int years)
```
As in the LoanRepaymentTable class we use a StringBuilder to accumulate the table as a string and we use the String.format method to line everything up in columns. The complete class is given by

```java
package chapter7.investment; // remove this line if you’re not using packages
/**
 * A class that produces an investment table for a given initial investment.
 */
public class InvestmentTable
{
    private double minRate, maxRate, rateStep; // range and step for rows
    private int minYears, maxYears, yearStep; // range and step for columns
    private double initialValue; // initial value of investment
    private String table; // the investment table

    /**
     * Construct table for given initial investment, rate range, and year range.
     * @param minRate rate (percent per year) for first table row
     * @param maxRate rate (percent per year) for last table row
     * @param rateStep step size in percent between table rows
     * @param minYears number of years for first column
     * @param maxYears number of years for last column
     * @param yearStep step size in years between table columns
     * @param initialValue initial value of the investment
     */
    public InvestmentTable(double minRate, double rateStep, double maxRate,
                           int minYears, int yearStep, int maxYears, double initialValue)
    {
        this.minRate = minRate;
        this.maxRate = maxRate;
        this.rateStep = rateStep;
        this.minYears = minYears;
        this.maxYears = maxYears;
        this.yearStep = yearStep;
        this.initialValue = initialValue;
        computeTable();
    }

    /**
     * Return the investment table.
     * @return the table
     */
    public String getTable()
    { return table; }
}
```
public String toString()
{
    return table;
}

private void computeTable()
{
    /* append table heading */

    StringBuilder buffer = new StringBuilder(1000);
    buffer.append(" RATE");
    for (int years = minYears; years <= maxYears; years += yearStep)
    {
        String head = String.format("%2d YEARS", years);
        buffer.append(String.format("%12s", head));
    }
    buffer.append("\n");

    /* Calculate and append rows of table */

    double small = 0.00001;
    for (double rate = minRate; rate <= maxRate + small; rate += rateStep)
    {
        buffer.append(String.format("%6.2f", rate));
        for (int years = minYears; years <= maxYears; years += yearStep)
        {
            double value = futureValue(initialValue, rate, years);
            buffer.append(String.format("%12.2f", value));
        }
        buffer.append("\n");
    }
    table = buffer.toString();
}

private double futureValue(double presentValue, double yearlyRate, int years)
{
    double monthlyRate = yearlyRate / 100.0 / 12.0;
    return presentValue * Math.pow(1.0 + monthlyRate, 12 * years);
}

7.11.2 Console user interface
Here is a runner class that can be used in BlueJ and from the command line.

Class InvestmentTableRunner

package chapter7.investment; // remove this line if you’re not using packages
import java.util.Scanner;
/**
* Console runner class for InvestmentTable.
* This version allows all table parameters to be specified.
*
public class InvestmentTableRunner
{
  public void run()
  {
    Scanner input = new Scanner(System.in);

    System.out.println("Enter minimum rate in percent");
    double minRate = input.nextDouble();
    input.nextLine();
    System.out.println("Enter table rate step");
    double rateStep = input.nextDouble();
    input.nextLine();
    System.out.println("Enter maximum rate in percent");
    double maxRate = input.nextDouble();
    input.nextLine();

    System.out.println("Enter minimum number of years");
    int minYears = input.nextInt();
    input.nextLine();
    System.out.println("Enter table year step");
    int yearStep = input.nextInt();
    input.nextLine();
    System.out.println("Enter maximum number of years");
    int maxYears = input.nextInt();
    input.nextLine();

    System.out.println("Initial investment");
    double amount = input.nextDouble();
    input.nextLine();
    InvestmentTable table = new InvestmentTable(minRate, rateStep, maxRate,
                                                 minYears, yearStep, maxYears, amount);
    System.out.println(table);
  }
}

public static void main(String[] args)
{
  new InvestmentTableRunner().run();
}

Typical console output is

    java InvestmentTableRunner
    Enter minimum rate in percent
    2
    Enter table rate step
    0.5
    Enter maximum rate in percent
Repetition Structures

Enter minimum number of years
1
Enter table year step
1
Enter maximum number of years
5
Initial investment
1000

<table>
<thead>
<tr>
<th>RATE</th>
<th>1 YEARS</th>
<th>2 YEARS</th>
<th>3 YEARS</th>
<th>4 YEARS</th>
<th>5 YEARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>1020.18</td>
<td>1040.78</td>
<td>1061.78</td>
<td>1083.21</td>
<td>1105.08</td>
</tr>
<tr>
<td>2.50</td>
<td>1025.29</td>
<td>1051.22</td>
<td>1077.80</td>
<td>1105.06</td>
<td>1133.00</td>
</tr>
<tr>
<td>3.00</td>
<td>1030.42</td>
<td>1061.76</td>
<td>1094.05</td>
<td>1127.33</td>
<td>1161.62</td>
</tr>
<tr>
<td>3.50</td>
<td>1035.57</td>
<td>1072.40</td>
<td>1110.54</td>
<td>1150.04</td>
<td>1190.94</td>
</tr>
<tr>
<td>4.00</td>
<td>1040.74</td>
<td>1083.14</td>
<td>1127.27</td>
<td>1173.20</td>
<td>1221.00</td>
</tr>
<tr>
<td>4.50</td>
<td>1045.94</td>
<td>1093.99</td>
<td>1144.25</td>
<td>1196.81</td>
<td>1251.80</td>
</tr>
<tr>
<td>5.00</td>
<td>1051.16</td>
<td>1104.94</td>
<td>1161.47</td>
<td>1220.90</td>
<td>1283.36</td>
</tr>
</tbody>
</table>

7.12 Plotting the graph of a function

As another example of a for-loop let us write a graphics program to plot a function \(f(x)\) for \(x_L \leq x \leq x_R\). To draw the graph we have to approximate the curve by many line segments and draw each line segment. If we choose small enough line segments the graph will look smooth. Therefore, we divide the interval \([x_L, x_R]\) into \(n\) equal size subintervals using the points

\[x_0 = x_L, x_1 = x_L + dx, \ldots, x_i = x_L + i dx, \ldots, x_n = x_R = x_L + n dx,
\]

where \(dx = (x_R - x_L)/n\). The graph between \(x_i\) and \(x_{i+1}\) is shown in Figure 7.15. It shows the curve and the line segment that is used to approximate it. On this interval we need to draw a line from the point \((x_i, f(x_i))\) to the point \((x_{i+1}, f(x_{i+1}))\). A pseudo-code algorithm for drawing the graph of \(f(x)\) is shown in Figure 7.16. Here each time a line segment is drawn its right end point becomes the left end point of the next line segment. This is accomplished with the assignment \((x_0, y_0) \leftarrow (x, y)\).

Figure 7.15: Approximating part of a function with a line segment
7.12 Plotting the graph of a function

Figure 7.16: Pseudo-code graph drawing algorithm

As an example, let us write a class to draw the graph of sin x from $x_L = -2\pi$ to $x_R = 2\pi$. The natural world coordinate system is one that has this range on the x-axis, and the range $-1$ to $1$ on the y-axis.

We can use the GraphicsFrame class and the worldTransform method from the BarGraph3 class in Chapter 5 (page 239). The bounding box for the world coordinate system can be defined using

```java
private double xLeft = -2 * Math.PI;
private double xRight = 2 * Math.PI;
private double yBottom = -1.0;
private double yTop = 1.0;
```

The paintComponent method has the basic structure

```java
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    Graphics2D g2D = (Graphics2D) g;

    int w = getWidth();
    int h = getHeight();

    int numPoints = 100;
    double b = 1.1;
    AffineTransform world = worldTransform(xLeft, xRight, yBottom*b, yTop*b, w, h);
    g2D.transform(world);

    // choose a line thickness here
    // Draw the x and y axes here
    // Draw the graph of sin x here
}
```

Here $b$ provides a 10% border in the y direction so the graph doesn’t touch the edge of the window.

Recall that any transformation of user space involving a scaling also transforms the line thickness. We want our lines to be one pixel in size so we need to calculate the width and height of a pixel in the world system and use the smallest of them as our pixel size. Since $h$ pixels vertically
Repetition Structures

correspond to $y_{Top} - y_{Bottom}$ units in the world and $w$ pixels horizontally correspond to $x_{Right} - x_{Left}$ units we obtain

```java
double pixelHeight = (yTop - yBottom) / h;
double pixelWidth = (xRight - xLeft) / w;
double pixelSize = Math.min(pixelHeight, pixelWidth);
```

Now we can set the line size using

```java
g2D.setStroke(new BasicStroke((float)pixelSize));
```

Alternatively, if we want a one pixel line we can use 0F as the argument of BasicStroke (see Chapter 5).

The axes can be drawn in blue using

```java
Line2D.Double xAxis = new Line2D.Double(xLeft, 0, xRight, 0);
Line2D.Double yAxis = new Line2D.Double(0, yBottom, 0, yTop);
g2D.setPaint(Color.blue);
g2D.draw(xAxis);
g2D.draw(yAxis);
```

Finally, the graph can be drawn in black using `Point2D.Double` objects called `p0` and `p1` for the points $(x_0, y_0)$ and $(x, y)$ shown in the pseudo-code algorithm:

```java
double dx = (xRight - xLeft) / numPoints;
Point2D.Double p0 = new Point2D.Double(xLeft, Math.sin(xLeft));
g2D.setPaint(Color.black);
for (int i = 1; i <= numPoints; i++)
{
    double x = xLeft + i*dx;
    double y = Math.sin(x);
    Point2D.Double p1 = new Point2D.Double(x, y);
    g2D.draw(new Line2D.Double(p0, p1));
    p0 = p1;
}
```

### 7.12.1 SineGraph class

Here is the complete class. The graph is shown in Figure 7.17.

```java
class SineGraph
```

```
package chapter7.sine_graph; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
```
// Use the GraphicsFrame class to draw the graph
// of a sine curve from -2*Math.PI to 2*Math.PI.
// An affine transformation is used to transform the coordinate system.
*/
public class SineGraph extends JPanel
{
    // define bounding box for graph in world coordinate system
    private double xLeft = -2 * Math.PI;
    private double xRight = 2 * Math.PI;
    private double yBottom = -1.0;
    private double yTop = 1.0;

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;

        int w = getWidth();
        int h = getHeight();

        int numPoints = 100;

        // Make a world coordinate system to go from xLeft*b to xRight*b in the
        // x direction and yBottom*b to yTop*b in the y direction, where b is
        // chosen to leave a 10 percent border in y direction
        double b = 1.1;
        AffineTransform world = worldTransform(xLeft, xRight,
                                             yBottom*b, yTop*b, w, h);
        g2D.transform(world);

        // find out the size of a pixel and use it to scale the
        // line thickness to one pixel. pixelHeight and pixelWidth
        // are the pixel width and height in the world system.
// Choose their minimum as the width of lines.

double pixelHeight = (yTop - yBottom) / h;
double pixelWidth = (xRight - xLeft) / w;
double pixelSize = Math.min(pixelHeight, pixelWidth);
g2D.setStroke(new BasicStroke((float)pixelSize));

// Another way to get one pixel lines is to use 0F as the brush size

// Draw the x and y axes

Line2D.Double xAxis = new Line2D.Double(xLeft, 0, xRight, 0);
Line2D.Double yAxis = new Line2D.Double(0, yBottom, 0, yTop);
g2D.setPaint(Color.blue);
g2D.draw(xAxis);
g2D.draw(yAxis);

// Determine the distance on the x-axis between successive points

double dx = (xRight - xLeft) / numPoints;

// Set the starting point on the curve to the leftmost point.

Point2D.Double p0 = new Point2D.Double(xLeft, Math.sin(xLeft));

// Draw numPoints line segments. The x coordinates of their right end
// points are xLeft+dx, xLeft+2*dx, ... and so on. After drawing a
// segment reset the starting point p0 for the next line segment.

g2D.setPaint(Color.black);
for (int i = 1; i <= numPoints; i++)
{
    double x = xLeft + i*dx;
    double y = Math.sin(x);
    Point2D.Double p1 = new Point2D.Double(x, y);
    g2D.draw(new Line2D.Double(p0, p1));
    p0 = p1;
}

private AffineTransform worldTransform(double xMin, double xMax,
double yMin, double yMax, int w, int h)
{
    double sx = (w-1) / (xMax - xMin);  // scale factor in x direction
    double sy = (h-1) / (yMax - yMin);  // scale factor in y direction
    AffineTransform at = new AffineTransform();
at.translate(0, h-1);  // move origin to bottom left corner of JPanel
    at.scale(sx, -sy);  // -sy reverses y axis
    at.translate(-xMin, -yMin);  // make (xMin, yMin) the lower left corner
    return at;
}
7.13 Recursion and loops

7.13.1 What is recursion?

Recursion is a problem solving technique that expresses a problem (algorithm) in terms of one or more smaller versions of itself. These smaller versions are, in turn, expressed in terms of smaller versions of themselves, and so on. The smaller versions of the problem at each stage are called the recursive cases. This process continues until we arrive at one or more cases which can be solved directly. These cases are called base cases.

7.13.2 Examples of recursive definitions

The simplest forms of recursion have a close connection with loops. Many functions in mathematics have non-recursive definitions expressed in terms of loops and recursive definitions that do not have explicit loops: the looping process is managed by the recursion process itself, as it breaks the problem into smaller subproblems.

We give three examples.

■ EXAMPLE 7.24 (Recursive definition of n!)

In Section 7.8.1 we gave a non-recursive definition of n!. A recursive definition is

\[
0! = 1, \quad 1! = 1 \\
n! = n(n - 1)! \quad (\text{recursive cases, } n > 1)
\]

The base cases are 0! and 1!. The recursive cases express \( n! \) in terms of \( (n - 1)! \), a smaller version of itself.

■ EXAMPLE 7.25 (Recursive definition of the Fibonacci numbers)

The Fibonacci numbers \( F_n, n = 0, 1, \ldots \) are defined by

\[
F_0 = 0, \quad F_1 = 1 \quad (\text{base cases, } n = 0, 1) \\
F_n = F_{n-1} + F_{n-2} \quad (\text{recursive cases, } n > 1)
\]

Here there are two simple base cases and the recursive cases express \( F_n \) as the sum of two smaller versions, \( F_{n-1} \) and \( F_{n-2} \).
EXAMPLE 7.26 (Recursive definition of the greatest common divisor) The greatest common divisor of two integers \( m \) and \( n \) is denoted by \( \text{gcd}(m,n) \). It is the largest integer that divides both \( m \) and \( n \). We can assume that \( m \geq 0 \) and \( n \geq 0 \) since \( \text{gcd}(m,n) = \text{gcd}(|m|,|n|) \). Under these assumptions a recursive definition for \( \text{gcd}(m,n) \) is

\[
\begin{align*}
\text{gcd}(m,0) &= m & \text{(base cases, } n = 0) \\
\text{gcd}(m,n) &= \text{gcd}(n,m \mod n) & \text{(recursive cases, } n > 0) 
\end{align*}
\]

We could also assume that \( m \geq n \) since \( \text{gcd}(m,n) = \text{gcd}(n,m) \), although the recursive definition works in either case. The base cases, when \( n = 0 \), do not involve recursion. The recursive cases involve smaller versions of \( \text{gcd}(m,n) \) since \( m \mod n \) is smaller than \( n \) so the second argument, \( n \), decreases until the base case is reached at \( n = 0 \). For example

\[
\begin{align*}
\text{gcd}(2436,1015) &= \text{gcd}(1015,2436 \mod 1015) = \text{gcd}(1015,406) \\
&= \text{gcd}(406,1015 \mod 406) = \text{gcd}(406,203) \\
&= \text{gcd}(203,406 \mod 203) = \text{gcd}(203,0) \\
&= 203
\end{align*}
\]

so \( \text{gcd}(2436,1015) = 203 \). □

7.13.3 Recursive factorial method

To see the connection with loops recall that the non-recursive version of the factorial function in program FactorialCalculator (page 338) involved a for-loop. Using the recursive definition in Example 7.24 the recursive version of this function is given in the following example.

EXAMPLE 7.27 (Recursive method for \( n! \))

```c
int factorial(int n)
{
    if (n == 0 || n == 1) // base cases
        return 1;
    else // recursive case
        return n * factorial(n-1);
}
```

Notice that the function calls itself but with a smaller version of the argument. Eventually the function will be called with 1 as a argument and the base case will stop the recursion. There is no loop in this version of the factorial function. The recursive process itself does the looping: each of the pending returns does one of the multiplications. Of course, if the base case is omitted then we have what is called infinite recursion, corresponding to an infinite loop. □

The following class can be used to test the recursive factorial function.
### Class FactorialCalculator

```java
package chapter7.recursion; // remove this line if you’re not using packages

/**
 * A simple class to test recursive version of factorial method.
 */
public class FactorialCalculator {

/**
 * Calculate n factorial using recursive algorithm.
 * @param n value for n factorial
 * @return n factorial
 * @throws IllegalArgumentException if n is outside the range
 * 0 <= n <= 12.
 */
public int factorial(int n) {
    if (n < 0 || n > 12) {
        throw new IllegalArgumentException("n! is only defined for n = 0..12");
    }
    if (n == 0 || n == 1) // base cases
        return 1;
    else // recursive cases
        return n * factorial(n-1);
}
}
```

A runner class for testing the method from the command line is given by

### Class FactorialRunner

```java
package chapter7.recursion; // remove this line if you’re not using packages
import java.util.Scanner;

/**
 * Testing recursive factorial method from command line
 */
public class FactorialRunner {

    public void run() {
        Scanner input = new Scanner(System.in);
        FactorialCalculator calc = new FactorialCalculator();
        System.out.println("Enter value of n");
        int n = input.nextInt();
        System.out.println(n + "! = " + calc.factorial(n));
    }
}
```
```java
public static void main(String[] args) {
    new FactorialRunner().run();
}
```

To see how the recursive process works let us calculate 4!. The steps are

\[
4! \rightarrow 4 \cdot 3!
\]

\[
\rightarrow 4 \cdot 3 \cdot 2!
\]

\[
\rightarrow 4 \cdot 3 \cdot 2 \cdot 1! \text{ base case}
\]

\[
\rightarrow 4 \cdot 3 \cdot 2
\]

\[
\rightarrow 4 \cdot 6
\]

\[
\rightarrow 24
\]

The recursive calls end with the base case, then the pending returns do all the multiplications.

### 7.13.4 Recursive gcd method

The following example gives a method for the recursive gcd algorithm in Example 7.26.

```java
int gcd(int m, int n) {
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

In case \(m = n = 0\) the method returns 0 even though \(gcd(0, 0)\) is not normally defined. Insert the statements

```java
m = Math.abs(m);
n = Math.abs(n);
```

if you want the method to also work in case one or both of \(m\) and \(n\) are negative.

The following class can be used to test the recursive gcd function.

```
```
package chapter7.recursion; // remove this line if you’re not using packages
/**
 * A simple class to test recursive version of gcd method.
 */
public class GcdCalculator
{
    /**
     * Calculate gcd(m,n) using recursive algorithm.
     * m >= 0 and n >= 0. Algorithm produces gcd(0,0) = 0
     * even though gcd(0,0) is undefined.
     * @return gcd(m,n)
     */
    public int gcd(int m, int n)
    {
        if (n == 0)
            return m;
        else
            return gcd(n, m % n);
    }
}

A runner class for testing the method from the command line is given by

**Class GcdRunner**

package chapter7.recursion; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing recursive gcd method from commmand line
 */
public class GcdRunner
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        GcdCalculator calc = new GcdCalculator();
        System.out.println("Enter value of m");
        int m = input.nextInt();
        input.nextLine();
        System.out.println("Enter value of n");
        int n = input.nextInt();
        input.nextLine();

        System.out.println("gcd(" + m + ", " + n + ") = " + calc.gcd(m,n));
    }
}

public static void main(String[] args)
{
    new GcdRunner().run();
}
7.13.5 Non-recursive and recursive sum methods

As another example of the connection between loops and recursion consider the problem of writing a method to compute the sum

\[ S(a, b) = a + (a+1) + \cdots + (b-1) + b \]

of the integers between \( a \) and \( b \) for \( a \leq b \). Pretending that we don’t know the answer

\[ S(a, b) = \frac{b(b+1)}{2} - \frac{(a-1)a}{2} = \frac{(b-a+1)(b+a)}{2} \]

the non-recursive solution is to use a for-loop as in the simple method

```java
public int sum(int a, int b)
{
    int s = 0;
    for (int k = a; k <= b; k++)
    {
        s = s + k;
    }
    return s;
}
```

We can also give a recursive definition of this sum in terms of smaller sums:

“These sum of the numbers from \( a \) to \( b \) is the first number plus the sum of the remaining numbers.”

The smaller version of the problem is “sum of the remaining numbers”, since there is one less number in this sum, and the base case occurs when there is a single number to sum. Assuming that \( a \leq b \) we have the following recursive definition

\[ S(a,a) = a \] (base cases, \( b = a \))
\[ S(a,b) = a + S(a+1,b) \] (recursive cases, \( b > a \))

Here is a recursive method to compute the sum:

```java
public int sum(int a, int b)
{
    if (a == b) // base case
        return a;
    else
        return a + sum(a+1, b);
}
```

7.14 Common loop errors

There are several errors that commonly occur when writing loops.
7.14.1 Misplaced semi-colon

Referring to Example 7.1, consider the simple while-loop

```java
int count = 1;
while (count <= 10);
{
    System.out.print(count + " ");
    count = count + 1; // or use count++
}
```

Instead of displaying 1 2 3 4 5 6 7 8 9 10 nothing is displayed and the program doesn’t stop. There is a semi-colon at the end of the line containing while. This is not a syntax error! This line is a complete while-loop with no body. Since count <= 10 is true it will always be true and the empty loop will never exit. The statements enclosed in braces are not part of the loop and will never be executed.

7.14.2 Off by one errors

It is important to test loops to make sure the loop variables begin and end with the required values. It is easy to be “off by one” and have a loop that executes one less time or one more time. These logical errors can be difficult to find.

7.15 BeanShell exercises

The following BeanShell exercises can be done using the Workspace Editor. First run BeanShell, then choose “Workspace Editor” from the “File” menu to open the editor. If you want to use `System.out.println` then it is also necessary to choose “Capture System in/out/err” from the “File” menu.

Now you can type statements into the editor and they won’t be executed as they are entered. When you have finished entering statements choose “Evaluate in Workspace” from the “Evaluate” menu. Now the statements will be executed. You can edit the statements and evaluate them again, and so on.

This is useful for testing static methods. Type in the method, evaluate it then test it interactively using the workspace.

**BeanShell Exercise 7.1** Write some statements to compute the sum $S_n = 1 + 2 + \cdots + n$ using

(a) a for-loop that counts up,
(b) a for-loop that counts down,
(c) a while-loop that counts up,
(d) a while-loop that counts down,
(e) a do-while loop that counts up,
(f) a do-while loop that counts down.
Test your statements using the BeanShell editor and workspace.

**BeanShell Exercise 7.2** Repeat Exercise 7.1 for the sum \( O_n = 1 + 3 + 5 + \cdots + (2n-1) \) of the first \( n \) odd numbers.

**BeanShell Exercise 7.3** Repeat Exercise 7.1 for the sum \( E_n = 2 + 4 + 6 + \cdots + 2n \) of the first \( n \) even numbers.

**BeanShell Exercise 7.4** Write a method with prototype

\[
\text{int power(int m, int k)}
\]

that uses a for-loop to compute \( m^k \) for \( k \geq 0 \). Test your method using the BeanShell editor and workspace.

**BeanShell Exercise 7.5** Write a method with prototype

\[
\text{double power(double m, int k)}
\]

that uses a single for-loop to compute \( m^k \) where \( k \) is any integer (positive or negative). Hint: distinguish the cases \( k < 0 \) and \( k \geq 0 \), multiplying \( m \) by itself \( k - 1 \) times if \( k > 0 \) but multiplying \( 1/m \) by itself \( -k \) times if \( k < 0 \). Test your method using the BeanShell editor and workspace.

### 7.16 Programming exercises

**Exercise 7.1 (Computing factorials with the long data type)**
Write a version of FactorialCalculator called LongFactorialCalculator that uses the long data type for calculating \( n! \). Also write a runner class called LongFactorialRunner. What is the largest value of \( n \) that can be used before overflow occurs?

**Exercise 7.2 (Number digits in \( n! \))**
The number of digits \( d \) in \( n! \) is \( 1 + \lfloor p \rfloor \) where \( n! = 10^p \). Taking logarithms to base 10 gives the formula

\[
d = 1 + \left\lfloor \log_{10} 2 + \log_{10} 3 + \cdots + \log_{10} n \right\rfloor
\]

Write a method called factorialDigits and a tester class for this formula and test it using BigFactorialRunner. In the Math class there is the function floor with prototype

\[
\text{public static long floor(double n)}
\]

and there is the logarithm function to base \( e \) with prototype

\[
\text{public static double log(double n)}
\]

To get logarithms to base 10 use the formula \( \log_{10} n = \frac{\log_e n}{\log_e 10} \).
Exercise 7.3 (Trailing zeros in \( n! \))

It can be shown that the number of trailing zeros in \( n! \) is

\[
n \div 5 + n \div 5^2 + n \div 5^3 + \cdots + n \div 5^k + \cdots
\]

Terms in the sum with \( 5^k > n \) do not contribute since \( n \div 5^k \) is zero. Write a class called TrailingZeros that uses the int data type to compute this sum. Use a while-loop with condition \( 5^k \leq n \) in a method with prototype

```java
int zeros(int n)
```

that returns the number of zeros. Also use a method with prototype

```java
public int power(int m, int k)
```

that computes \( m^k \) \( (k > 0) \) using a for-loop and can be used as `power(5, k)` by the `zeros` method. You can use BigFactorialTester to test this formula.

Exercise 7.4 (Powers of two)

Write a class called PowersOfTwoCalculator that computes and displays, for a given value of \( n \), the first \( n \) powers of 2, namely \( 2^1, 2^2, \ldots, 2^n \). Do not use the `Math.pow` function. The program output for \( n = 4 \) should look like

```
Enter the largest power
4
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
```

What is the largest power that you can compute as an int?

Exercise 7.5 (Powers of two using BigInteger objects)

Try the previous exercise by writing a class called BigPowersOfTwo that uses BigInteger objects.

Exercise 7.6 (Computing integer powers)

Write a class called IntegerPowerCalculator that tests the method in BeanShell Exercise 7.5 for computing integer powers. Also write a runner class IntegerPowerRunner that can be run from the command line.

Exercise 7.7 (Computing \( e \) using a series)

The base of the natural logarithms has the representation

\[
e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{k!} + \cdots
\]

as an infinite sum. The \( n \)-th partial sum of this series is defined as

\[
S_n = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}, \quad n \geq 0
\]
As \( n \) increases these partial sums get closer and closer to the value of \( e \). The \( n \)th term in this series is \( t_n = 1/n! \). Write a class called \texttt{ExpCalculator} that uses a for-loop to compute these sums for \( 0 \leq n \leq nMax \). Where the value of \( nMax \) is input by the user. You do not need to compute any factorials. Instead use the fact that

\[
\frac{t_k}{t_{k-1}} = \frac{(k-1)!}{k!} = \frac{(k-1)!}{k(k-1)!} = \frac{1}{k}
\]

to write \( t_k = t_{k-1}/k \), where \( t_0 = 1 \). This expresses each term in terms of the preceding one. Then the partial sum \( S_k \) can be expressed in terms of the preceding one using \( S_k = S_{k-1} + t_k \) and \( S_0 = 1 \). Now you can use a loop based on the pseudo-code algorithm in Figure 7.18 which displays the partial sum number (0, 1, 2, \ldots, \( nMax \)) and the partial sum. Here \( s \) denotes a partial sum and \( t \) denotes a term. Your program should also display the approximate value of \( e \) using the constant \texttt{Math.E}.

\begin{algorithm}
\textbf{ALGORITHM} ExpCalculator\((nMax)\)
\begin{algorithmic}
\STATE \( s \leftarrow 1.0 \)
\STATE \textbf{OUTPUT} 0, \( s \)
\STATE \( t \leftarrow 1.0 \)
\FOR {\( k \leftarrow 1 \) \TO \( nMax \)}
\STATE \( s \leftarrow s + t \)
\STATE \( t \leftarrow t/k \)
\STATE \textbf{OUTPUT} \( k, s \)
\END FOR
\end{algorithmic}
\end{algorithm}

Figure 7.18: Partial sum algorithm for \( e \)

\[ e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots \]

\textbf{Exercise 7.8 (Computing} \( e^x \)\textbf{ using a series)}
Adapt the pseudo-code algorithm of Exercise 7.7 to compute partial sums of the series for \( e^x \) given by

\textbf{Exercise 7.9 (Computing} \( \sin x \)\textbf{ and} \( \cos x \)\textbf{ using a series)}
Adapt the pseudo-code algorithm of Exercise 7.7 to compute partial sums of the series for \( \sin x \) and \( \cos x \) given by

\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots + \frac{(-1)^k x^{2k+1}}{(2k+1)!} + \cdots
\]

using a class called \texttt{ExpXCalculator}. The input is now \( x \) and \( nMax \). Compare your results by displaying \texttt{Math.exp(x)} after the last partial sum is displayed. Consider cases such as \( x = 0.01, x = 0.1, x = 1, \) and \( x = 10 \) and determine how many terms in the sum are needed to get agreement with \texttt{Math.exp(x)}. 
\[
\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + \frac{(-1)^k x^{2k}}{(2k)!} + \cdots
\]

using classes called SinXCalculator and CosXCalculator. The input is now \(x\) and \(nMax\). Compare your results by displaying \(\text{Math.sin}(x)\) and \(\text{Math.cos}(x)\) after the last partial sum is displayed. Consider cases such as \(x = 0.01, x = 0.1, x = 1,\) and \(x = 10\) and determine how many terms in the sum are needed to get agreement with \(\text{Math.sin}(x)\) and \(\text{Math.cos}(x)\).

**Exercise 7.10 (A triangle of asterisks)**
Write a class called TriangleRight that inputs a value of \(n\) defining the number of rows in the triangle and displays output like

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

which is the case \(n = 4\).

**Exercise 7.11 (Another triangle of asterisks)**
Write a class called TriangleCenter that inputs a value of \(n\) defining the number of rows in the triangle and displays output like

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

which is the case \(n = 4\).

**Exercise 7.12 (Reversing a string)**
Write a program class called ReverseString that tests a method called reverse with prototype

```java
public String reverse(String s)
```

The method returns a string that is the reverse of \(s\). For example, if \(s\) is "Help" the string returned is "pleH". Use a for-loop that implements the steps shown in the table

<table>
<thead>
<tr>
<th>(i)</th>
<th>reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;h&quot; ← &quot;h&quot; + &quot;n&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;eh&quot; ← &quot;e&quot; + &quot;h&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;leh&quot; ← &quot;l&quot; + &quot;eh&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;pleh&quot; ← &quot;p&quot; + &quot;leh&quot;</td>
</tr>
</tbody>
</table>

in the case that \(s\) is "Help".
Exercise 7.13 (Nested while-loops)
Write a complete class called MarksCalculator that reads a series of integer marks for a number of students using a negative mark to signal the end of the marks for each student. The program then calculates and prints the average of the marks for each student as a double number. After calculating and printing each average, the program asks if the user wants to enter marks for another student. A reply of “N” or “n” terminates the program. For any other reply the program continues with the next student. Use the following loop structure:

```java
while (moreStudents)
{
    ...
    while (mark >= 0)
    {
        ...
    }
}
```

Some typical program output is

```
Enter marks for a student terminated by a negative mark
65
85
90
-3
The average for this student is 80
Do you want to enter marks for another student [Y/N]?
Y
Enter marks for a student terminated by a negative mark
55
75
70
-2
The average for this student is 66.67
Do you want to enter marks for another student [Y/N]?
N
```

Exercise 7.14 (Recursive calculation of Fibonacci numbers)
The Fibonacci numbers were defined recursively in Example 7.25. Write a method with prototype

```java
public int fibonacci(int n)
```
to compute Fibonacci numbers using this recursive definition. Put your method in a class called RFibonacciCalculator to test the method.

Exercise 7.15 (Non-recursive calculation of Fibonacci numbers)
Do the previous exercise using a non-recursive method. Hint: each Fibonacci number is just the sum of the previous two numbers so keep track of two successive numbers so you can calculate the next one.
Exercise 7.16 (Recursive binary gcd algorithm)
Here is an interesting recursive definition of gcd(m, n) for m ≥ n ≥ 0 that uses only simple subtraction and division by 2 operations.

(a) If m < n then swap m and n.
(b) If m = 0 then gcd(m, n) = n (base case).
(c) If n = 0 then gcd(m, n) = m (base case).
(d) If m and n are both even then gcd(m, n) = 2 gcd(m/2, n/2).
(e) If m is odd and n is even then gcd(m, n) = gcd(m/2, n).
(f) If m is even and n is odd then gcd(m, n) = gcd(m/2, n).
(g) If m and n are both odd then gcd(m, n) = gcd((m − n)/2, n).

Repeat the previous exercise using this definition and write a class called BinaryGcdCalculator to test your method.

Exercise 7.17 (A non-recursive gcd algorithm)
Using the non-recursive pseudo-code algorithm for calculating gcd(m, n) shown in Figure 7.19, write a method with prototype

```
ALGORITHM gcd(m, n)
  a ← m
  b ← n
  r ← a mod b
  WHILE r ≠ 0 DO
    a ← b
    b ← r
    r ← a mod b
  END WHILE
  RETURN b
```

Figure 7.19: Pseudo-code non-recursive gcd algorithm

Write a program called GcdTester, similar to FactorialTester, to test the method.

Exercise 7.18 (A non-recursive gcd algorithm using subtraction)
Another interesting non-recursive pseudo-code algorithm for calculating gcd(m, n) that only uses subtraction is shown in Figure 7.20, where m > 0 and n > 0. Write a method with prototype

```
public int gcd(int m, int n)
```


that uses this algorithm. Write a class called GcdSubtractTester, similar to GcdTester in the preceding exercise to test the method.

**Exercise 7.19 (Ackermann’s function)**

Ackermann’s function \( A(m,n) \), for integers \( m,n \geq 0 \), has the recursive definition

\[
A(m,n) = \begin{cases} 
  n + 1, & \text{if } m = 0 \\
  A(m-1,1), & \text{if } n = 0, m > 0 \\
  A(m-1,A(m,n-1)), & \text{if } m,n > 0 
\end{cases}
\]

Write a method with prototype

\[
\text{public int Ackermann(int m, int n)}
\]

for this function. Use a class called AckermannCalculator, similar to the FactorialCalculator class, so that you can test it. Evaluate \( A(2,5) \) and \( A(3,3) \). What happens when you try to evaluate \( A(4,4) \)?

**Exercise 7.20 (Pythagorean triples)**

Pythagorean triples have the form \( (x,y,z) \) where \( x, y, \) and \( z \) are positive integers satisfying \( x^2 + y^2 = z^2 \). They correspond to right angled triangles with integer sides \( x \) and \( y \), containing the right angle, and hypotenuse \( z \). Write a class called PythagoreanTriples that displays all triples having \( x \leq 100, y \leq 100, \) and \( z \leq 100 \). Generate only the triples in ordered form such that \( x < y < z \). For example, \( (3,4,5) \) will be generated but \( (4,3,5) \) will not be generated.

**Exercise 7.21 (Unique Pythagorean triples)**

In the previous exercise the triples \( (3,4,5), (6,8,10), \) and \( (9,12,15) \) are generated. Dividing the numbers in the second triple by 2 gives the first triple and dividing the numbers in the third triple by 3 also gives the first triple. For the triple \( (3,4,5) \) we have \( \text{gcd}(3,4) = 1 \), for the second triple \( \text{gcd}(6,8)=2 \), and for the third triple \( \text{gcd}(9,12) = 3 \). There are many other cases like this.

Rewrite the program of the preceding exercise to generate only the triples \( (x,y,z) \) such that \( \text{gcd}(x,y) = 1 \). You can use one of the gcd methods from the preceding exercises.
Exercise 7.22 (Generating prime numbers)
A prime number is an integer \( n \geq 2 \) having no factors other than itself and 1. The first five prime numbers are 2, 3, 5, 7, 11. Write a class called PrimeGenerator that takes two numbers \( nMin \) and \( nMax \) as input and displays all prime numbers \( p \) such that \( nMin \leq p \leq nMax \).

Exercise 7.23 (Drawing a grid of lines)
Write a graphics program called GridMaker that draws horizontal and vertical lines to form a 10 by 10 array of cells (as in a spreadsheet program). If the output window is resized the grid should expand or contract to fill it. An example is show in Figure 7.21. Here the cells are square but in general they will be rectangles.

Exercise 7.24 (Drawing concentric circles)
Write a graphics program called ConcentricCircles that draws concentric circles in the output window, each with a different color. Use console input to get the number of circles from the user. The largest circle should just fit the window even if the window is resized. Color the circles (draw them from largest to smallest) using random colors obtained using the method

```java
public Color randomColor()
{
    float red = (float) Math.random();
    float green = (float) Math.random();
    float blue = (float) Math.random();
    return new Color(red, green, blue);
}
```

which returns a random Color object for setPaint.

You may find the transformation

```java
AffineTransform at = new AffineTransform();
at.translate(xMax/2.0, yMax/2.0);
g2D.transform(at);
```
useful for moving the coordinate origin to the center of the window. A sample output window is shown in Figure 7.22.

◮ Exercise 7.25 (Drawing a regular polygon)
The pentagon and hexagon were considered in Chapter 5. Write a RegularPolygon class that draws a regular \( n \)-sided polygon (all sides equal). Use console input to get the value of \( n \) from the user. The polygon should appear centered in the window. Fill the polygon with yellow and draw it with a brush 2 pixels wide. Hint: Assuming a center at \((0,0)\), the \( n \) vertices of the polygon are \((x_k, y_k), k = 0, \ldots n - 1\), where \( x_k = r \cos ka, y_k = r \sin ka, \) \( r \) is the radius and the angle \( a \) is given by \( a = 2\pi/n \) radians. If \( n \) is large enough the polygon will look like a circle.

◮ Exercise 7.26 (Drawing a clock face)
Write a graphics program called ClockFace that draws a circular clock face. The circle should be the largest one that fits the window. Draw tick marks every minute and double length tick marks every five minutes.

◮ Exercise 7.27 (Drawing a ruler)
Write a graphics program called RulerMaker that draws a picture of a ruler 10 cm wide, with centimeter marks in red, shorter 0.5 centimeter marks in blue and shorter millimeter marks in black.

◮ Exercise 7.28 (Graphing functions)
Using SineGraph as a model write a class called SineCosGraph that draws \( \sin x \) and \( \cos x \) on the same graph using different colors.

◮ Exercise 7.29 (User interface for the PSRGame)
In Chapter 6 the PSRGameRunner class was a console user interface for one round of the game. Write a better user interface that uses a query controlled while loop to play rounds of the game.
until the users decide to quit. When the game ends display how many wins there were for each player.