Chapter 6

Making Decisions

Conditional execution of statements

Outline

Simple boolean expressions
Relational and equality operators and expressions
Conditional operator
Comparison of floating point numbers
Simple if-else statement
If without else statement
Nested and multiple (N-way) if-statement
Common errors with if-statements
Compound boolean expressions and logical operators
Lexicographical ordering of strings using character codes
String comparison and equality
Error reporting using exceptions
Throwing and catching exceptions in BankAccount class
Paper, scissors, rock game
Complex roots of a quadratic equation
6.1 Introduction

In the programs we have written so far, statements are always executed one after the other in a sequential manner: the same sequence of statements is always executed. However, many algorithms are expressed in terms of conditions that can be true or false, depending on the input data for example, so the statements executed depend on these conditions. We need to be able to express the fact that one sequence of statements is to be executed if a certain condition is true and another sequence is to be executed if the condition is false.

For example, in the TriangleCalculator class from Chapter 3 (page 64) the constructor arguments were two sides of a triangle and the contained angle. We did not check that these values were positive. A better program would check using conditions such as \( a > 0 \) and \( b > 0 \) and only do the calculations if these conditions were true. Similarly, in Chapter 3 the QuadraticRootFinder class (page 67) was used to compute the roots of the quadratic equation \( ax^2 + bx + c = 0 \) in case both are real. The condition for real roots is \( b^2 - 4ac \geq 0 \) but we did not check this condition.

In this chapter we will see how the if-statement can be used for the conditional execution of statements. To express conditions we use boolean expressions, which can have true or false values, relational operators, which compare the values of arithmetic expressions to produce true or false values, and logical operators, which combine simple boolean expressions to obtain compound boolean expressions.

Boolean expressions can be tested in a program using an if-statement. When the expression has a true value one block of statements is executed and when it has a false value another block of statements is executed. This process of executing one block of statements or another, based on the value of a boolean expression, is called conditional execution. This will give our classes important decision making capabilities. A discussion of common if-statement errors is also included.

To illustrate these ideas we use the “Paper, Scissors, Rock” game. Also, we modify the BankAccount class, introduced in Chapter 4, to include error checking. The important concepts of an exception and throwing an exception are also introduced and illustrated using the BankAccount class. We also extend the QuadraticRootFinder class so that it finds both the real and complex roots of a quadratic equation.

6.2 Simple boolean expressions

Conditional execution is based on the evaluation of a condition. In Java the condition is a boolean expression which evaluates to one of the values true or false of the boolean data type (see Chapter 2). These two values are called boolean literals.

Just as there are many ways to form arithmetic expressions, the same applies to boolean expressions. The most common is to compare two arithmetic or boolean expressions using a binary comparison operator. These expressions are called comparison expressions and have the form

\[
\text{Expression1 ComparisonOperator Expression2}
\]

where ComparisonOperator is one of the six comparison operators shown in Table 6.1. This table also shows the standard mathematical notation for these operators. A double equal sign represents equality in Java since the single equal sign is already used for assignment. The two expressions are evaluated first before the comparison operator is applied.
### 6.2 Simple boolean expressions

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Mathematical Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>≥</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>≤</td>
<td>less than or equal</td>
</tr>
<tr>
<td>==</td>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>≠</td>
<td>not equal</td>
</tr>
</tbody>
</table>

Table 6.1: The six comparison operators

The six comparison operators fall into two groups: == and != are called **equality operators**, since they test expressions for equality or inequality, and the other four operators are called **relational operators**. The corresponding expressions have the form

\[ \text{ArithmeticExpression}_1 \ \text{RelationalOperator} \ \text{ArithmeticExpression}_2 \]

for **relational expressions**, and the form

\[ \text{Expression}_1 \ \text{EqualityOperator} \ \text{Expression}_2 \]

for **equality expressions**, where \( \text{Expression}_1 \) and \( \text{Expression}_2 \) can be either boolean or arithmetic expressions.

**Example 6.1** (Simple boolean expressions)

(a) The expression \( \text{month} == 3 \) is true only if \( \text{month} \) has the value 3.

(b) The expression \( k \% 2 == 0 \) is true only if the integer \( k \) is even, which is the case if the remainder on division by 2 is 0. The expression \( k \% 2 != 0 \) is true only if the integer \( k \) is odd. Generalizing, the expression \( k \% n == 0 \) is true only if \( k \) is divisible by the integer \( n \) (remainder on division by \( n \) is 0).

(c) The expression \( b*b - 4.0*a*c >= 0 \) is true only if the quadratic equation \( ax^2 + bx + c = 0 \) has real roots. For example it is false if \( a, b, \) and \( c \) all have the value 1, and true if \( a = 1, b = 3, \) and \( c = 2.\)

(d) If \( \text{playerChoice} \) is a variable of type \( \text{char} \), the expression \( \text{playerChoice} == 'P' \) is true only if the variable has the value ‘P’.

(e) In a turtle graphics system the turtle has a pen that can be either up or down. This condition can be represented by a variable called \( \text{penUp} \) of type \( \text{boolean} \), where a value of \( \text{true} \) indicates that the pen is up. A boolean variable such as \( \text{penUp} \) is a simple example of a boolean expression since it evaluates to a \( \text{true} \) or \( \text{false} \) value.

In examples (a) to (d) the expressions on either side of the comparison operator are evaluated before the comparison operator is applied.
6.3 If-statements

Conditional execution can be accomplished using an if-statement. Since the syntax varies with the computer language, it is useful to express it using the following language-independent algorithmic notation often called pseudo-code.

\[
\text{IF BooleanExpression THEN}
\begin{align*}
\text{Statements A} \\
\text{ELSE} \\
\text{Statements B} \\
\text{END IF}
\end{align*}
\]

Here BooleanExpression stands for any expression that evaluates to one of the values true or false. The statements labeled A are executed if BooleanExpression is true and the statements labeled B are executed if it is false. In Java, the corresponding if-statement has the structure shown in Figure 6.1. The parentheses enclosing BooleanExpression are necessary. A sequence of statements enclosed in braces is called a block so the if-statement defines two blocks, one for each value of BooleanExpression. The first block is called the if-block and the second block is called the else-block.

The statements in the two blocks are indented by an equal amount of space. Indentation has no effect on execution. It is there to improve the readability. We recommend using three spaces of indentation for the statements in a block.

The template in Figure 6.1 is a static diagram, designed to show the syntax and layout to use when writing if-statements. It does not show the flow of execution. A flowchart is a graphical representation of the flow of execution. The flowchart for the if-statement is shown in Figure 6.2. The downward arrow at the top indicates the flow before the if-statement is encountered. Then the diamond-shaped box represents the boolean expression to be evaluated. One of the outward arrows is chosen depending on the value of the expression. Rectangular boxes contain statements to be executed sequentially. To follow the flow, begin at the top and follow the arrows until you reach the bottom. In any case, exactly one of the two blocks A and B will be executed. The downward arrow at the bottom represents the flow after the if-statement.
**Example 6.2** (Calculating the absolute value) The absolute value $|x|$ of $x$ is defined to be $x$ if $x \geq 0$ and $-x$ if $x < 0$. It can be calculated using `Math.abs`. If we didn’t have this function in the `Math` class we could use the following method:

```java
double abs(double x)
{
    if (x >= 0)
    {
        return x;
    }
    else
    {
        return -x;
    }
}
```

which returns the absolute value of a `double` number.

**Example 6.3** (A cube root method) If $x$ is a real number then its cube root $x^{1/3}$ can be calculated using `Math.pow(x, 1.0/3.0)` but only if $x \geq 0$. If $x < 0$ we can write $x^{1/3} = -(-x)^{1/3}$ and use `-Math.pow(-x, 1.0/3.0)`. The method:

```java
double cubeRoot(double x)
{
    if (x >= 0)
    {
        return Math.pow(x, 1.0/3.0);
    }
```
uses an if-statement to return the cube root of $x$ in either case.

### 6.4 Real roots of a quadratic equation

In Chapter 3 we wrote a `QuadraticRootFinder` class (page 67) to find the real roots of a quadratic equation. We can now modify it to determine if there are real roots. To do this we add the data field

```java
private boolean realRoots;
```

This variable will be set to true if the equation has real roots and false otherwise. The “get” method

```java
public boolean hasRealRoots()
{
    return realRoots;
}
```

can be used to determine if the equation has real roots. We need to modify the `doCalculations` method to use an if-statement to give a value to this boolean variable depending on the sign of $b^2 - 4ac$.

#### 6.4.1 `QuadraticRootFinder` class

Here is the complete class with these modifications.

```java
package chapter6.root_finder; // remove this line if you're not using packages
/**
 * An object of this class can calculate the real roots of the
 * quadratic equation $ax^2 + bx + c = 0$ given the coefficients $a$, $b$, and $c$.
 * In this version there is a check for real roots.
 */
public class QuadraticRootFinder
{
    private double a, b, c;
    private double root1, root2;
    private boolean realRoots;
```
6.5 Block declaration of variables

/**
 * Construct a quadratic equation root finder given the coefficients
 * @param a first coefficient in ax^2 + bx + c
 * @param b second coefficient in ax^2 + bx + c
 * @param c third coefficient of ax^2 + bx + c
 */
public QuadraticRootFinder(double aCoeff, double bCoeff, double cCoeff)
{
    a = aCoeff;
    b = bCoeff;
    c = cCoeff;
    doCalculations();
}

private void doCalculations()
{
    double d1 = b*b - 4*a*c;
    if (d1 >= 0)
    {
        double d = Math.sqrt(d1);
        root1 = (-b - d) / (2.0 * a);
        root2 = (-b + d) / (2.0 * a);
        realRoots = true;
    }
    else
    {
        realRoots = false;
    }
}

/**
 * Returns true if real roots were found else false.
 * @return true if real roots were found else false
 */
public boolean hasRealRoots()
{
    return realRoots;
}

// getRoot1 and getRoot2 methods from Chapter 3 go here
// getA, getB, and getC methods from Chapter 3 go here
// setA, setB, and setC methods from Chapter 3 go here

6.5 Block declaration of variables

A block is any sequence of statements delimited by braces. Variables are defined in blocks and are said to have block scope. This means that they do not exist outside the block in which they are
declared. We have now seen three kinds of blocks:

- Data fields have the widest scope. They are defined in the class declaration block so they are available anywhere in the class. The variables \(a\), \(b\), and \(c\) in the QuadraticRootFinder class are examples.

- Local variables in a constructor or method are defined only in the block defining the constructor or method body. The variable \(d_1\) defined in the doCalculations method is an example.

- Variables declared in the if-block or the else-block of an if-statement are local to this block. The variable \(d\) defined in the if-block of the doCalculations method is an example.

### 6.6 If-statement with no else

When the else-part of the if-statement is not required, it can be omitted to give the pseudo-code statement

```
IF BooleanExpression THEN
    Statements
END IF
```

or the Java statement

```java
if (BooleanExpression)
{
    // statements
}
```

The if-block is executed only if the boolean expression is true. Otherwise it is skipped and execution resumes with any statements after the if-statement. The flowchart for the if-statement with no else-part is shown in Figure 6.3.

**Example 6.4 (If-statement with no else-part)** In QuadraticRootFinder we could have written the doCalculations method as

```java
private void doCalculations()
{
    realRoots = false;
    double d1 = b*b - 4*a*c;
    if (d1 >= 0)
    {
        double d = Math.sqrt(d1);
        root1 = (-b - d) / (2.0 * a);
        root2 = (-b + d) / (2.0 * a);
        realRoots = true;
    }
}
```
6.6 If-statement with no else

Figure 6.3: Flowchart for an if-statement with no else-part

which initializes `realRoots` to `false` so no else-part is required.

**Example 6.5 (Assigning boolean expressions)** Another variation of Example 6.4 is

```java
private void doCalculations()
{
    double d1 = b*b - 4*a*c;
    realRoots = d1 >= 0;
    if (realRoots)
    {
        double d = Math.sqrt(d1);
        root1 = (-b - d) / (2.0*a);
        root2 = (-b + d) / (2.0*a);
    }
}
```

which uses a boolean assignment statement to assign the true or false value of the boolean expression `d1 >= 0` directly to the variable `realRoots`, which now becomes the condition in the if-statement.

**Example 6.6 (One line if-statement)** The following statements compute the maximum of the integer variables `x` and `y` and assign it to `max`.

```java
int max = x; // assume x is the maximum
if (y > max)
{
    max = y; // replace max with y if y is bigger than x
}
```
Since there is only a single statement in the if-block the braces can be omitted so you will often see the if-statement written as

```java
if (y > max)
    max = y;
```

or even as the so-called one line if-statement

```java
if (y > max) max = y;
```

The same rule applies to the else-block. Another variation of the maximum calculation is

```java
int max;
if (x > y)
    max = x;
else
    max = y;
```

If you find the use of braces more readable always use them. Several common errors can occur by not using braces (see Section 6.10).

### 6.7 Comparison of floating point numbers

Most floating point numbers cannot be stored exactly in computer memory. For example, the simple decimal number 0.1 cannot be stored exactly because its binary value has an infinite number of digits. This leads to truncation error. Computers also have a limited accuracy when performing computations. Each arithmetic operation may introduce a small roundoff error, and as more operations are carried out, roundoff error can accumulate. In practice we may consider the two numbers 1 and 0.99999999 to be “equal”, but they may not be equal to the computer. Therefore, it is advisable not to compare two expressions of type `double` directly, using the comparison operators, especially the equality operators. Instead, either the absolute error or the relative error can be used.

#### 6.7.1 Floating point tester class

Errors can be demonstrated using the following class that computes \( \pi^5 \) in two ways, once using `Math.pow`, and the other using four multiplications. The class can be run both inside and outside BlueJ.

```java
package chapter6.floating_point;

/**
 * Testing equality of floating point numbers using equality.
 */
```
public class FloatingPointTester1
{
    public void doTest()
    {
        double x = Math.pow(Math.PI, 5.0);
        System.out.println("1st approx is " + x);
        System.out.println("2nd approx is " + y);

        if (x == y)
            System.out.println("equal");
        else
            System.out.println("not equal");
    }

    public static void main(String[] args)
    {
        FloatingPointTester1 tester = new FloatingPointTester1();
        tester.doTest();
    }
}

■ Example 6.7 (Using == to compare floating point numbers) The FloatingPointTester1 class produces the output

1st approx is 306.0196847852814
2nd approx is 306.01968478528136
not equal

The pow method actually uses the formula $e^{5 \ln \pi} = \pi^5$ to compute its result rather than direct multiplication. The comparison finds them to be “not equal” since there will be differing amounts of round-off error in the two calculations. In fact $x$ is slightly greater than $y$. This example shows that we must be careful when comparing floating point numbers directly.

■ Example 6.8 (Absolute error for floating point comparison) If you want to compare two arithmetic expressions of type double for equality, or inequality, it may be better to use

$$\text{absoluteError} = |x - y|$$

as a measure of equality. This defines the absolute error between the two values $x$ and $y$ as the absolute value of the difference between the two numbers. We can use it to check if the absolute difference between two numbers is smaller than a very small number, say $1E-10$. Replace the if-statement in FloatingPointTester1 with

    if (Math.abs(x - y) <= 1E-10)  
        System.out.println("equal");
    else  
        System.out.println("not equal");
to get `FloatingPointTester2` and the “equal” message will be printed. This means that two floating point numbers should be considered equal if they are close enough to each other as defined by `1E-10`.

**Example 6.9** (Relative error for floating point comparison) In scientific calculations the relative error is often a better measure of the closeness of two floating point numbers `x` and `y`. If `x ≠ 0`, one definition is

\[ \text{relativeError} = \frac{x - y}{x} \]

We can replace the if-statement in `FloatingPointTester2` with

```java
double relativeError = (x - y) / x;
if (Math.abs(relativeError) <= 1E-10)
    System.out.println("equal");
else
    System.out.println("not equal");
```

...to get `FloatingPointTester3` and the “equal” message will be printed.

### 6.8 Conditional operator

There is a special operator in Java called the **conditional operator**, denoted by `?`, which can be used to write special if-statements in a compact fashion. It produces conditional expressions of the form

\[ \text{booleanExpression} ? \text{expressionA} : \text{expressionB} \]

The value of the conditional expression is `expressionA` if `booleanExpression` is true and `expressionB` otherwise. The value of the conditional expression can then be assigned to a variable. For example, if `expressionA` and `expressionB` evaluate to an `int` value then we can write a statement such as

```java
int v = booleanExpression ? expressionA : expressionB;
```

The conditional operator is not really necessary since you can achieve the same result with

```java
int v;
if (booleanExpression)
    v = expressionA;
else
    v = expressionB;
```

We will not use the conditional operator much since it can make programs harder to read.
6.9 Nested and multiple (N-way) if-statements

- **Example 6.10** (The conditional operator) The absolute value method in Example 6.2 can also be defined as
  
  ```java
double abs(double x)
{  
  return (x >= 0) ? x : -x;
}
```
  
  using the conditional operator.

- **Example 6.11** (The conditional operator) In Example 6.6 the maximum of two integer variables \( x \) and \( y \) was computed using an if-statement. This can also be done as
  
  ```java
  int max = (x >= y) ? x : y;
  ```
  
  using the conditional operator.

- **Example 6.12** (The conditional operator version of cubeRoot) The `cubeRoot` method from Example 6.3 can be expressed as
  
  ```java
double cubeRoot(double x)
{  
  return (x >= 0) ? Math.pow(x, 1.0/3.0) : -Math.pow(-x, 1.0/3.0);
}
```
  
  using the conditional operator.

6.9 Nested and multiple (N-way) if-statements

In the general if-statement in Figure 6.1 the two blocks can also contain other if-statements. If-statements within if-statements are said to be nested.

- **Example 6.13** (A nested if-statement) Suppose we have an amount of money \( a \geq 0 \) and the tax is 5% if \( 0 \leq a < 10000 \), 10% if \( 10000 \leq a < 100000 \), and 15% if \( a \geq 100000 \). The nested if-statement
  
  ```java
  double tax;
  if (a >= 10000)
  {  
  if (a < 100000)
  {  
  tax = 0.10 * a;
  }
  else // a >= 100000
  {  
  tax = 0.15 * a;
  }
  ```
shows one way to compute the tax using a nested if-statement inside the outer if-block. We have included comments on the else parts to improve readability.

The if-else-statement is designed for a two-way decision process. It can be generalized to a multiple if-statement, sometimes called an if-else-if-statement, that is a special kind of nested if-statement designed for a multi-way decision process. The template is a generalization of the one given in Figure 6.1 and is shown in Figure 6.4.

There are $N$ conditions, represented by $N$ boolean expressions, and $N + 1$ blocks. The $N$ conditions do not have to be mutually exclusive. There may be more than one condition that evaluates to true. But in this case only the block for the first of these conditions will be executed. The order of the tests is important in this case.

If the $N$ conditions are mutually exclusive, meaning that only one of them at a time can be true, then the order of the conditions is not important. In any case, exactly one of the $N + 1$ blocks of statements is executed. If none of the $N$ conditions is true, the default block is executed. A flowchart for the multiple if-statement is shown in Figure 6.5.
Example 6.14 (Calculating letter grades) The following if-statement assigns a letter grade for a given integer mark:

```java
String letterGrade;
if (mark < 0)
    letterGrade = "";
else if (mark > 100)
    letterGrade = "";
else if (mark >= 80)
    letterGrade = "A";
else if (mark >= 70)
    letterGrade = "B";
else if (mark >= 60)
    letterGrade = "C";
else if (mark >= 50)
    letterGrade = "D";
else
    letterGrade = "F";
```

Here marks outside the range 0 to 100 are assigned an empty string as a letter grade. The order of the conditions is important here since they are not mutually exclusive. For example, using marks >= 50 first will not work since any mark of 50 or more will result in a grade of D being assigned.

### 6.10 Common errors with if-statements

The following three examples show errors that can occur if you are not careful with braces. They can be avoided by always using braces.

Example 6.15 (Forgetting the braces) Consider the if-statement in Figure 6.6(a) The intent here is to assign the maximum of x and y to max and the minimum to min. However, because there are no braces in the else-block the compiler assumes that only the assignment to max belongs to this block. The indentation is misleading and has no effect on the compiler. Thus, min will always receive the value of x since this assignment statement is not part of the if-else statement. This is an example of a logical error. The if-statement in Figure 6.6(b) corrects the problem using braces.

Example 6.16 (Else without if) If you forget to use braces for the if-block, as in the if-statement in Figure 6.6(c) the Java compiler will now report an “else without if” error message. Since there are no braces in the if-block the compiler assumes that

```java
if (x >= y) 
    max = x;
```
if (x >= y) {
    max = x;
    min = y;
} else {
    max = y;
    min = x;
}

(a)

if (x >= y) {
    max = x;
    min = y;
} else {
    max = y;
    min = x;
}

(b)

if (x >= y) {
    max = x;
    min = y;
} else {
    max = y;
    min = x;
}

(c)

Figure 6.6: forgetting the braces
is a complete if-statement with no else part. Then the statement $\text{min} = y$ is a normal statement not inside an if-statement. Then the else is encountered with no matching if, and this is a syntax error.

\[\text{EXAMPLE 6.17 (Dangling else problem)}\]

Consider the following statement with a nested if-statement:

```java
if (mark >= 50)
    if (mark <= 100)
        System.out.println("Pass");
    else
        System.out.println("Fail");
```

The intent here is to display Pass in case the mark is in the range 50 to 100 and Fail otherwise. However, for marks less than 50 nothing is ever displayed and for marks greater than 100 the Fail message is displayed.

Again the indentation is misleading since it seems to associate the else part with the outer if. However, it is also possible to associate the else with the inner if and the results are not the same. This is called the "dangling else problem".

To resolve this ambiguity the compiler always associates an else with the nearest if so the if-statement is interpreted as

```java
if (mark >= 50)
{
    if (mark <= 100)
        System.out.println("Pass");
    else
        System.out.println("Fail");
}
```

To obtain the desired meaning, and associate the else with the outer if, we need to use braces:

```java
if (mark >= 50)
{
    if (mark <= 100)
        System.out.println("Pass");
}
else
{
    System.out.println("Fail");
}
```

which now displays the Fail message for marks less than 50.
6.11 Compound boolean expressions

A boolean expression is an expression which can have a true or false value. The simplest boolean expression is just a boolean variable itself, for example realRoots in Example 6.5. Boolean expressions can also be obtained using the comparison operators in Table 6.1. A compound boolean expression consists of two or more boolean expressions connected together by one or more logical operators. These operators can be used to express conditions that can be true in more than one way.

The mathematical and Java notations for the three basic logical operators are shown in Table 6.2. In mathematics, \( \wedge \) is often called “wedge” and \( \vee \) is often called “vee”. There are several mathematical notations for negation. Two are shown in the table, namely \( \sim \), which is called “tilde”, and \( \neg \). Both are called “not”. We will use \( \sim \). In pseudo-code you can use either the mathematical notation or the names AND, OR, and NOT.

The symbols \( \wedge \) and \( \vee \) are not available on keyboards so Java uses \( \& \& \) and \( || \) instead. For negation the exclamation mark is used.

6.11.1 Writing expressions using AND, OR, and NOT

If \( b_1, b_2, ..., b_n \) are \( n \) boolean expressions then

\[
b_1 \wedge b_2 \wedge \ldots \wedge b_n \quad (b_1 \& \& b_2 \& \& \ldots \& \& b_n, \text{ in Java})
\]

is the compound expression obtained by “and”ing together the \( n \) expressions. It is called the “logical and”. The value of this compound expression is true only if all \( n \) expressions \( b_1 \) to \( b_n \) are true. If any expression is false then the compound expression is false. A truth table gives the values of a compound boolean expression in terms of all possible values of the simple boolean expressions it contains. In the case of two simple expressions, \( p \) and \( q \), the truth table for \( p \wedge q \) is shown in Figure 6.3(a). For example, the first row of this table tells us that if \( p \) is false and \( q \) is false, then \( p \wedge q \) is also false. The last column of the table clearly shows that \( p \wedge q \) is true only if both \( p \) and \( q \) are true.

Similarly, we can perform the \( \vee \) operation and

\[
b_1 \vee b_2 \vee \ldots \vee b_n \quad (b_1 \mid \mid b_2 \mid \mid \ldots \mid \mid b_n, \text{ in Java})
\]

is the compound expression obtained by “or”ing together the \( n \) expressions. It is called the “logical or”. The value of this compound expression is true if any one of the \( n \) expressions \( b_1 \) to \( b_n \) is true.

\[\text{Table 6.2: The three basic logical operators}\]

<table>
<thead>
<tr>
<th>Java</th>
<th>Mathematical Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; &amp;</td>
<td>( \wedge )</td>
<td>logical “and” (looks like an “A”)</td>
</tr>
<tr>
<td>| |</td>
<td>( \vee )</td>
<td>logical “or” (looks like an “r”)</td>
</tr>
<tr>
<td>!</td>
<td>( \sim ), ( \neg )</td>
<td>logical “not” (negation)</td>
</tr>
</tbody>
</table>

\[1\text{Double symbols are used since \& and \| have other meanings in Java which we will not discuss (bitwise operators).}\]
6.11 Compound boolean expressions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$q$</td>
<td>$p \land q$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$q$</td>
<td>$p \lor q$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$\neg p$</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

(a) (b) (c)

Table 6.3: Truth tables for AND, OR, and NOT

It is false only if all $n$ expressions are false. In the case of two simple expressions $p$ and $q$ the truth table for $p \lor q$ is shown in Figure 6.3(b). The last column of the table clearly shows that $p \lor q$ is false only if both $p$ and $q$ are false.

The operators $\land$ and $\lor$ are binary operators because they operate on two boolean expressions. The $\neg$ operator is an example of a unary operator. It operates on only one boolean expression to give the opposite truth value (negation) of the expression. The truth table for $\neg p$ is shown in Figure 6.3(c).

Operator precedence rules

Among the unary operators $-$, $+$, $!$, the binary arithmetic operators $\ast$, $/$, $\%$, $+$, $-$, the comparison operators $<$, $\leq$, $>$, $\geq$, $==$, $!=$, and the binary logical operators $\&\&$, $||$, we have the following precedence rules in the order from highest to lowest:

1. Parentheses ( ) have the highest precedence.
2. The unary operators $-$, $+$, and the unary negation operator $!$, have equal precedence and they are right associative (they are applied in right to left order).
3. The binary arithmetic operators $\ast$, $/$, and $\%$ have equal precedence and they are left associative (they are applied in left to right order).
4. The binary arithmetic operators $+$ and $-$ have equal precedence and they are left associative.
5. The relational operators $<$, $\leq$, $>$, and $\geq$ have equal precedence and are not associative.
6. The equality operators $==$ and $!=$ have equal precedence and are left associative.
7. The binary logical “and” operator $\&\&$ is left associative.
8. The binary logical “or” operator $||$ is left associative.

**Example 6.18** (Applying precedence rules) In expression $\text{mark} \geq 0 \&\& \text{mark} \leq 100$ the expressions $\text{mark} \geq 0$ and $\text{mark} \leq 100$ are evaluated first and then $\&\&$ is applied. To emphasize this the expression could be written using parentheses as $(\text{mark} \geq 0) \&\& (\text{mark} \leq 100)$. 
Assuming that \(d\) and \(p\) are of type \(\text{int}\), the compound expression \(! d == 0 || p == 0\) results in the compiler error “can’t convert int to boolean”. The \(!\) operator is being applied to \(d\) since it has a higher precedence than \(==\). Parentheses are needed to obtain \(! (d == 0) || p == 0\). Now \(d == 0\) is evaluated and then \(!\) is applied. Then \(p == 0\) is evaluated and finally \(||\) is applied.

**Example 6.19 (Mixed boolean expressions)** The \(\land\) and \(\lor\) operators can be mixed together. In the expression \(a \land b \lor c\) the expression \(a \land b\) is evaluated first according to the precedence rules, then \(\lor\) is applied. To evaluate \(b \lor c\) first it is necessary to use parentheses and write \(a \land (b \lor c)\).

**Example 6.20 (Truth table for \((a \land b) \lor c\) and \(a \land (b \lor c)\))** The truth table requires 8 rows, since there are \(2^3 = 8\) possible combinations for the values of \(a\), \(b\), and \(c\). This gives the truth table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>(a \land b)</th>
<th>(b \lor c)</th>
<th>((a \land b) \lor c)</th>
<th>(a \land (b \lor c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

where the results for \((a \land b) \lor c\) and \(a \land (b \lor c)\) in the final two columns show that the expressions are not the same.

**DeMorgan’s laws**

If \(a\) and \(b\) are boolean expressions, we have the laws

\[
\sim (a \land b) = (\sim a) \lor (\sim b) \\
\sim (a \lor b) = (\sim a) \land (\sim b)
\]

for negating compound expressions. These laws are called deMorgan’s laws and can be easily verified using truth tables. The generalizations to \(n\) expressions \(b_1, b_2, \ldots, b_n\) are

\[
\sim (b_1 \land b_2 \land \ldots \land b_n) = (\sim b_1) \lor (\sim b_2) \lor \ldots \lor (\sim b_n) \\
\sim (b_1 \lor b_2 \lor \ldots \lor b_n) = (\sim b_1) \land (\sim b_2) \land \ldots \land (\sim b_n)
\]

**Testing numerical ranges**

Often it is necessary to test if a variable \(n\) has a value in the range \(a\) to \(b\). In mathematics this is expressed as \(a \leq n \leq b\). In programming languages we cannot use the expression \(a <= n <= b\). Instead, it is expressed as \(a <= n && n <= b\) using \(||\) to connect two relational expressions.
### Example 6.21 (Numerical ranges)

In Example 6.14 a letter grade was assigned to a mark using a multiple if-statement. The conditions were not mutually exclusive so the order was important. Using compound logical expressions involving `&&`, we can make the conditions mutually exclusive as follows:

```java
if (80 <= mark && mark <= 100)
    letterGrade = "A";
else if (70 <= mark && mark < 80)
    letterGrade = "B";
else if (60 <= mark && mark < 70)
    letterGrade = "C";
else if (50 <= mark && mark < 60)
    letterGrade = "D";
else if (0 <= mark && mark < 50)
    letterGrade = "F";
else
    letterGrade = ""; // invalid mark case
```

Now the order of the conditions is not important. The else-block is used to trap errors. When possible it is best to write the conditions in a multiple if-statement in mutually exclusive form, since these conditions are easier to understand, and use the else-block to trap invalid cases.

### Example 6.22 (Numerical ranges)

Using compound boolean expressions we can rewrite Example 6.13 as

```java
double tax;
if (0 <= a && a < 10000)
{
    tax = 0.05 * a;
}
else if (10000 <= a && a <= 100000)
{
    tax = 0.10 * a;
}
else // a >= 100000
{
    tax = 0.15 * a;
}
```

which is much easier to read.

### 6.11.2 Leap year problem

In a leap year February has 29 days instead of 28. There are two statements that define when a year is a leap year:

\[
\begin{align*}
    s_1 &= \text{the year is divisible by 4 but not by 100 (for example, 1988 or 1992)} \\
    s_2 &= \text{the year is divisible by 400 (for example, 1600 or 2000)}
\end{align*}
\]
If either of these statements is true then the year is a leap year, so if we let \( s \) be the statement “the year is a leap year” then \( s \) is the compound statement

\[
s = \text{the year is a leap year} = s_1 \lor s_2
\]

We need to translate \( s_1 \) and \( s_2 \) into logical expressions. They contain the following three simpler logical expressions:

\[
\begin{align*}
  a & = \text{year is divisible by 4} \\
  b & = \text{year is not divisible by 100} \\
  c & = \text{year is divisible by 400}
\end{align*}
\]

Therefore, we can write (interpreting “but” as “and”)

\[
\begin{align*}
  s_1 & = a \land b, \\
  s_2 & = c, \\
  s & = s_1 \lor s_2 = (a \land b) \lor c.
\end{align*}
\]

Using these expressions we can write the pseudo-code algorithm shown in Figure 6.7. In this pseudo-code algorithm we use the left arrow symbol to indicate assignment. In Java, if \( \text{year} \) is an integer variable then

\[
\begin{align*}
  a & \text{ translates to } \text{year} \% 4 == 0 \\
  b & \text{ translates to } \text{year} \% 100 \neq 0 \\
  c & \text{ translates to } \text{year} \% 400 == 0
\end{align*}
\]

and we obtain the boolean leap year expression

\[
(\text{year} \% 4 == 0) \land (\text{year} \% 100 \neq 0) \lor (\text{year} \% 400 == 0)
\]

The \( \land \) operator will be performed first, since it has a higher precedence than \( \lor \lor \).

**Example 6.23 (Leap year test)** To test the leap year expression try the following statements in the BeanShell workspace editor (select it from the File menu).
6.12 String comparison and equality

```java
int year = 2004;
if ((year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0))
    print(year + " is a leap year");
```

From the editor’s “Evaluate” menu select “Eval in workspace” and you will see the message “2004 is a leap year” in the workspace. Try other years and select “Eval in workspace” each time.

6.11.3 Short circuit evaluation

In Java the boolean expression \( p \land q \) is evaluated using what is called short circuit evaluation. The idea is that if \( p \) is evaluated and it is false, the entire expression will be false, so there is no need to evaluate \( q \). However, if \( p \) is true then it is necessary to evaluate \( q \) to obtain the truth value of \( p \land q \). A similar rule is used for \( p \lor q \). Here \( p \) is evaluated first and if it has the value true the entire expression is true so there is no need to evaluate \( q \). In Java the order of expressions in the compound boolean expressions \( p \land q \) and \( p \lor q \) may be important even though in mathematics the order is unimportant since \( p \land q = q \land p \) and \( p \lor q = q \lor p \).

**Example 6.24** (Short-circuit evaluation) If \( x \) and \( y \) are variables of type int the if-statement

```java
if (x != 0 && y/x > 2) {...}
```

does not result in a division by zero when \( x \) is zero. The expression \( x \neq 0 \) will be evaluated first to false in this case, and the second expression will not be attempted. However, when the order of the expressions is reversed, as in the if-statement

```java
if (y/x > 2 && x != 0) {...}
```

an error will result when \( x \) happens to be zero.

6.12 String comparison and equality

The String class was introduced in Chapter 4. We now consider string comparison methods. In Table 6.1 we introduced the four relational operators \(<\), \(\leq\), \(\geq\), and \(>\) comparing two arithmetic expressions, and the two equality operators, \(==\) and \(!=\). We cannot use these operators to compare two strings. In particular the \(==\) operator compares two string references for equality not the string objects that they reference.

**Example 6.25** (Incorrect use of == for string comparison) To test this try the following statements in the BeanShell workspace editor (select it from the File menu).

```java
String s1 = "hello";
String s2 = "hello";
if (s1 == s2)
    print("equal");
else
    print("not-equal");
```
and evaluate it in the work space. The result “not-equal” will be displayed no matter what strings are used. The reason is that the variables s1 and s2 are not String objects. They are references to String objects and will always have different values, even if they reference the same string object (see Part (b) of Figure 4.13). It is the references that are being compared by the == operator not the characters in the string objects.

6.12.1 Equals method for string comparison

In order to compare String objects themselves for equality, the String class has an instance method called equals with the prototype

```java
public boolean equals(Object obj)
```

The argument is an Object type rather than a String type for technical reasons that we will discuss in Chapter 10. when we study inheritance and polymorphism. An object of any class is of type Object and in particular a string is also of type Object so we can use a string as an argument. For now just assume that the argument is of type String.

The prototype tells us that we can send the equals message to a string asking if it is equal to the argument string. The result will be true if the two string objects are equal, which means they have the same length and they contain the same characters. The comparison is case sensitive so the string "abc" is different from the string "aBc".

**Example 6.26** (String equality using equals method) If we change Example 6.25 to use equals to obtain

```java
String s1 = "hello";
String s2 = "hello";
if (s1.equals(s2))
  print("equal");
else
  print("not-equal");
```

then the if-statement works as expected. To test for inequality you can use the negation in an if-statement of the form

```java
if (! s1.equals(s2)) {...}
```

Here s1.equals(s2) is evaluated and ! is applied to negate the result.

**Example 6.27** (Selecting menu choices) In an interactive program you may display a menu of choices and then ask the user to select one of the choices. Suppose the choices are the strings "addition", "subtraction", "multiplication", or "division". The statements

```java
if (choice.equals("addition"))
  // process addition choice here
else if (choice.equals("subtraction"))
  // process subtraction choice here
```
else if (choice.equals("multiplication"))
    // process multiplication choice here
else if (choice.equals("division"))
    // process division choice here
else
    // process invalid choice here

can be used to process the different choices.

### 6.12.2 Lexicographical ordering of strings

Sometimes we want to check two strings not to see if they are equal but to determine which one comes first in the **lexicographical ordering** of strings. This ordering is defined by comparing characters from the strings one at a time.

Each character in Java is internally represented by a 16-bit integer, called a **character code**, using the Unicode system. The usual North American subset for the English language (punctuation, digits '0' to '9', uppercase letters 'A' to 'Z', and lowercase letters 'a' to 'z'), occupy the first 128 positions in this 16-bit code. This 128 character subset is called the ASCII code. The next 128 codes contain various accented characters, used by languages such as German and French, and the first 256 Unicode characters form what is called the ISO-LATIN1 character set.

The following simple class can be used to find out the codes corresponding to various characters.

```java
package chapter6.strings; // remove this line if you’re not using packages

/**< *
 * Finding the code for a given character
 */
public class CharacterDecoder
{
    /**< *
     * Return integer code of a character
     * @param c the character to decode
     * @return the integer code of c
     */
    public int code(char c)
    {
        int code = (int) c;
        return code;
    }
}
```

Here a typecast is used to convert the character `ch` to an integer value. The results for the printable characters in the range 0 to 127 are shown in Table 6.4. To find the ASCII code for a character in the table, add the number at the beginning of its row to the number at the top of its column. For
example, the code for ‘Z’ is $80 + 10 = 90$. The first 32 codes (0 to 31) are not shown, since they are non-printable control codes. For example, carriage return has code 13 and is denoted by ‘\r’, line feed has code 10 and is denoted by ‘\n’, and the tab character has code 9 and is denoted by ‘\t’.

In this code the digits ‘0’ to ‘9’ have smaller codes than uppercase letters, which in turn have smaller codes than the lowercase letters. The codes for ‘0’ to ‘9’ are increasing, the codes for ‘A’ to ‘Z’ are increasing, and the codes for ‘a’ to ‘z’ are increasing. This means that we can apply the six comparison operators to characters in a relational expression and Java will use their codes. This imposes an ordering on the character set so we can say, for example, that ‘a’ < ‘f’, ‘A’ < ‘C’, ‘a’ < ‘A’, ‘5’ < ‘A’, ‘5’ < ‘6’, and so on.

### Using character codes to order strings

A string ordering can now be defined using the ordering of the string characters based on their character codes. This is called the lexicographical order. You start comparing the two strings one character at a time until either of the following conditions are true:

1. the character in one string is different from the corresponding one in the other string
2. one of the strings ends before the other

In case 1 the string whose character has the smaller code at the first position where they differ is said to precede the other (is “smaller than” the other) in the lexicographical ordering. For example, “Frank” precedes “Fred”, since the “a” at index 2 of “Frank” has a smaller code than the “e” at index 2 of “Fred”. As another example, “Bobby” comes after “Bob”. Their first three characters match but then there are no more characters in “Bob” so we say that “Bobby” follows “Bob” (or “Bob” precedes “Bobby”) in the lexicographical ordering. This is case 2, and the shorter string always precedes the longer one.

### 6.12.3 compareTo method for string comparison

We have seen that we cannot use == and != to compare strings for equality and inequality. Instead we use the equals method. Similarly we cannot use <, <=, >, or >= to compare two strings. Instead there is a compareTo instance method in the String class. It has the prototype

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>’</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: ASCII codes for characters
### 6.12 String comparison and equality

<table>
<thead>
<tr>
<th>Boolean expression</th>
<th>Meaning if true</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1.compareTo(s2) &lt; 0</td>
<td>s1 precedes s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) &lt;= 0</td>
<td>s1 precedes or is equal to s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) &gt; 0</td>
<td>s1 follows s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) &gt;= 0</td>
<td>s1 follows or is equal to s2</td>
</tr>
<tr>
<td>s1.compareTo(s2) == 0</td>
<td>s1 is equal to s2 (same as s1.equals(s2))</td>
</tr>
<tr>
<td>s1.compareTo(s2) != 0</td>
<td>s1 is not equal to s2 (same as !s1.equals(s2))</td>
</tr>
</tbody>
</table>

Table 6.5: Lexicographical string comparison using `compareTo`

```java
public int compareTo(String s)
{
    int result = s1.compareTo(s2);
    if (result < 0)
        return s1 + " precedes " + s2;
    else if (result == 0)
        return s1 + " equals " + s2;
    else
        return s1 + " follows " + s2;
}
```

The return value indicates the result of the comparison. A negative value for `s1.compareTo(s2)` means that `s1` precedes `s2`, a zero value means that `s1` and `s2` are equal, and a positive value means that `s1` follows `s2`. The boolean expressions that are used to compare strings are shown in Table 6.5. Here is a simple string comparison class you can try in BlueJ.

```java
public class StringComparer
{
    public String compare(String s1, String s2)
    {
        int result = s1.compareTo(s2);
        if (result < 0)
            return s1 + " precedes " + s2;
        else if (result == 0)
            return s1 + " equals " + s2;
        else
            return s1 + " follows " + s2;
    }
}
```

You can also try the following example using the BeanShell workspace editor.

**Example 6.28** (String comparison using `compareTo` method) Using the BeanShell editor type in the `compare` method from the `StringComparer` class and evaluate it in the workspace. Now try the statements

```bash
bsh % show();
<true>
```
bsh % compare("one", "two");
<one precedes two>
bsh % compare("two", "one");
<two follows one>
bsh %
to experiment with the compareTo method.

### 6.12.4 Case insensitive string comparison

Sometimes it is useful to do a case-insensitive string comparison. This means that lower case letters are considered equivalent to their upper case counterparts. One way to do this is to convert both strings to lower case and then use compareTo as in the following expression

\[
s1.toLowerCase()(s2.toLowerCase());
\]

However, this is not necessary since there is a compareToIgnoreCase method in the String class with prototype

\[
\text{public int compareToIgnoreCase(String s)}
\]

that does this case-insensitive comparison.

### 6.13 Boolean valued methods

Methods that return boolean values are called **boolean valued methods**. The equals method in the String class is a good example. There are other such methods in the String class.

**EXAMPLE 6.29 (Boolean valued String methods)** The equalsIgnoreCase method is like the equals method but it ignores the case of the letters. It has the prototype

\[
\text{public boolean equalsIgnoreCase(String s)}
\]

Other examples are the startsWith and endsWith instance methods that have the prototypes

\[
\begin{align*}
\text{public boolean startsWith(String prefix)} \\
\text{public boolean endsWith(String suffix)}
\end{align*}
\]

The first returns true if the string receiving the message starts with the given prefix, and the second returns true if the string receiving the message ends with the given suffix.

We can also write our own boolean valued methods as the following examples show.

**EXAMPLE 6.30 (An isLeapYear method)** For the leap year Example 6.28 we can write a boolean valued method called isLeapYear that takes the year as a formal argument and returns true or false, depending on whether the year is a leap year. The method prototype is

\[
\text{public boolean isLeapYear(int y)}
\]
where the formal argument $y$ stands for the year. The method declaration is

```java
public boolean isLeapYear(int y)
{
    return (y % 4 == 0) && (y % 100 != 0) || (y % 400 == 0);
}
```

This method could easily be tested in BeanShell as follows

```bash
bsh % show();
<true>
bsh % boolean isLeapYear(int y)
{ return (y % 4 == 0) && (y % 100 != 0) || (y % 400 == 0); }
bsh % isLeapYear(2000);
<true>
bsh % isLeapYear(2003);
<false>
bsh % isLeapYear(2004);
<true>
bsh %
```

assuming that `show()` is on.

**EXAMPLE 6.31 (A realRoots method)** Given the quadratic equation $ax^2 + bx + c = 0$, the method

```java
public boolean realRoots(double a, double b, double c)
{
    return b*b - 4.0*a*c >= 0;
}
```

returns `true` if the equation has real roots and `false` otherwise.

## 6.14 Error checking techniques

A simple version of the bank account class was developed in Chapter 4, page 106. In that version there were no checks for illegal values of the data fields. For example, the account number must be positive, and the balance must not be negative. Also, it must not be possible to withdraw more than the current balance, or deposit a negative amount.

We can use an if-statement to test for these illegal conditions. In general there are several ways to do the error handling. For the class designer the easiest way is to do no error checking at all! This places the entire burden of error checking on the user of the class. For example, to ensure that a withdrawal is legal, the user would have to write statements such as

```java
BankAccount myAccount = new BankAccount(123, "Peter Pascoe", 4050.00);
...
if (myAccount.getBalance() >= 5000)
```
On the other hand the programmer could simply decide to use

```java
myAccount.withdraw(5000);
```
without checking, and put the account in an illegal state with a negative balance.

It is never a good idea to leave error checking to the user of the class. It is the responsibility of the class designer to ensure that a bank account object can never be in an inconsistent state (bad account number or negative balance). This principle is called **data encapsulation** and it ensures data integrity. The ability to encapsulate data within an object and protect it is one of the most important benefits of object-oriented programming.

Ensuring data integrity in a class can be accomplished by following some simple rules in designing classes:

1. Make all data fields private.
2. Check the validity of all arguments in a constructor and report an error if any of them are illegal.
3. Check the validity of method arguments, especially for methods that can change the values of one or more data fields, and report an error if any of them are illegal.
4. Do not return references to private data fields. This is the unwanted side-effect problem illustrated in Chapter 4 using the `MPoint` and `MCircle` classes.

### 6.14.1 Reporting errors

There are also several ways to report errors:

1. If an error occurs inside a constructor, such as an illegal argument,
   
   (a) display an error message and exit the program,
   
   (b) have the constructor “throw an exception” (to be discussed later)

2. If an error occurs inside a method, such as an illegal argument, leave the data fields unchanged and either
   
   (a) display an error message and exit the program,
   
   (b) have the method “throw an exception” (to be discussed later),
   
   (c) return a boolean value or other error indicator that can be checked by the user,
   
   (d) do nothing and exit the method.
6.14 Error checking techniques

6.14.2 Using boolean return values and exit to report errors

We can use the BankAccount class to illustrate error reporting. For example, the original withdraw method is

```java
public void withdraw(double amount)
{
    balance = balance - amount;
}
```

which does no error handling. It could be replaced by the boolean valued method

```java
public boolean withdraw(double amount)
{
    boolean amountValid = (0 <= amount) && (amount <= balance);
    if (amountValid)
    {
        balance = balance - amount;
    }
    return amountValid;
}
```

which returns true if the amount is valid, and false otherwise. This method changes the balance only if there is enough money. This ensures data integrity, but leaves the processing of the error to the user. For example, the user could write statements such as

```java
BankAccount myAccount = new BankAccount(123, "Andy Dalziel", 4050.00);
...
boolean ok = myAccount.withdraw(5000);
if (! ok)
{
    // report error here
    // ask for a new withdrawal amount or cancel withdrawal
}
```

The important idea here is that the program can recover from the error by either asking for a new withdrawal amount or canceling the withdrawal. Even if the user forgets to do the error checking, by ignoring the return value as in

```java
myAccount.withdraw(5000);
```

the data integrity of the object is maintained. Similarly, the original deposit method

```java
public void deposit(double amount)
{
    balance = balance + amount;
}
```

should not change the balance if a negative amount is used, so it could be replaced by
public boolean deposit(double amount) {
    boolean amountValid = amount >= 0;
    if (amountValid) {
        balance = balance + amount;
    }
    return amountValid;
}

which does not change the balance, in case amount is negative.

### 6.15 Error reporting using exceptions

We now show how to throw an exception to indicate illegal arguments in a constructor or method. This is the most important error-handling technique.

#### 6.15.1 Exception classes and objects

An exception is an object from an exception class that has error information in it. To throw one means to suspend execution of the program and either process the error or signal the caller of the constructor or method that an error condition has occurred. The throw statement is used to throw exceptions. It has the syntax

    throw exceptionObject;

where exceptionObject is an exception object constructed from one of the exception classes. There are many kinds of exception classes in Java and we can even write our own.

#### 6.15.2 Throwing exceptions in the BankAccount class

We now illustrate exceptions for illegal method and constructor arguments using the BankAccount class. The exception class is called IllegalArgumentException and it already exists in package java.lang.

In our first attempt at the BankAccount class (page 106) the constructor declaration is

    public BankAccount(int accountNumber, String ownerName, double initialBalance) {
        number = accountNumber;
        name = ownerName;
        balance = initialBalance;
    }

We need to check for account numbers that are 0 or negative, names that have no object associated with them or are empty strings, and negative initial balances. This can be done by throwing an exception if one of these errors occur.
Here is a version of the bank account constructor that constructs exception objects and throws them. An exception constructor can take one argument, a string that specifies an appropriate error message.

```java
public BankAccount(int accountNumber, String ownerName, double initialBalance) {
    if (accountNumber <= 0)
        throw new IllegalArgumentException("Account number must be positive");
    if (ownerName.equals("") || ownerName == null)
        throw new IllegalArgumentException("Owner name not defined");
    if (initialBalance <= 0)
        throw new IllegalArgumentException("Balance must be non-negative");

    number = accountNumber;
    name = ownerName;
    balance = initialBalance;
}
```

Each if-statement checks one of the three arguments. We don’t need to use else here: when an exception is thrown control is immediately transferred out of the constructor to the method that called it. Since `ownerName` is a reference to a `String` object, we can check if an object has been defined, by comparing the reference with the special value `null`. If `ownerName` is `null` this means that the caller has forgotten to initialize the reference for the string.

We can also use this technique to write the following versions of the `withdraw` and `deposit` methods:

```java
public void deposit(double amount) {
    if (amount < 0)
        throw new IllegalArgumentException("Invalid amount for deposit");
    balance = balance + amount;
}

public void withdraw(double amount) {
    if (amount < 0 || amount > balance)
        throw new IllegalArgumentException("Invalid amount for withdraw");
    balance = balance - amount;
}
```

When an exception is thrown the Java interpreter immediately stops executing the method or constructor in which the exception occurred and looks for an error processing block called a catch block. If it doesn’t find one it looks for a catch block in the caller of this method or constructor and so on. Eventually, if the exception is not caught, the Java interpreter will catch it, display the error message, and terminate the program.

In any case when throwing exceptions as in the `BankAccount` example the important idea is that we do not need to concern ourselves with who catches the exception or how it is processed.
We simply provide an informative error message. Here is the complete version of the bank account class with error checking using exceptions.

```java
package chapter6.bank_account; // remove this line if you’re not using packages

/**
 * A bank account object encapsulates the account number, owner name, and
 * current balance of a bank account.
 * This version checks for illegal method and constructor arguments.
 */
public class BankAccount
{
    private int number;
    private String name;
    private double balance;

    /**
     * Construct a bank account with given account number,
     * owner name and initial balance.
     * @param accountNumber the account number
     * @param ownerName the account owner name
     * @param initialBalance the initial account balance
     * @throws IllegalArgumentException if account number is negative,
     *         owner name is null or empty, or if balance is negative.
     */
    public BankAccount(int accountNumber, String ownerName, double initialBalance)
    {
        if (accountNumber <= 0)
            throw new IllegalArgumentException("Account number must be positive");
        if (ownerName.equals("") || ownerName == null)
            throw new IllegalArgumentException("Owner name not defined");
        if (initialBalance < 0)
            throw new IllegalArgumentException("Balance must be non-negative");
        number = accountNumber;
        name = ownerName;
        balance = initialBalance;
    }

    /**
     * Deposit money in the account.
     * @param amount the deposit amount. If amount <= 0 the
     * account balance is unchanged.
     * @throws IllegalArgumentException if deposit amount is negative
     */
    public void deposit(double amount)
    {
        if (amount < 0)
            throw new IllegalArgumentException("Invalid amount for deposit");
        balance = balance + amount;
    }
}
```

```
/**
 * Withdraw money from the account.
 * If account would be overdrawn the account balance is unchanged.
 * @param amount the amount to withdraw.
 * @throws IllegalArgumentException if withdraw amount is invalid
 */
public void withdraw(double amount)
{
    if (amount < 0 || amount > balance)
        throw new IllegalArgumentException("Invalid amount for withdraw");
    balance = balance - amount;
}

/**
 * Return the account number.
 * @return the account number.
 */
public int getNumber()
{
    return number;
}

/**
 * Return the owner name.
 * @return the owner name.
 */
public String getName()
{
    return name;
}

/**
 * Return the account balance.
 * @return the account balance.
 */
public double getBalance()
{
    return balance;
}

/**
 * string representation of this account.
 * @return string representation of this account.
 */
public String toString()
{
    return "BankAccount[" + number + ", " + name + ", " + balance + "]";
}
To test the error processing in this version of BankAccount we can use a simple tester class such as

```java
package chapter6.bank_account; // remove this line if you’re not using packages
/**
 * Showing uncaught exception messages;
 */
public class ExceptionTester {
    public void doTest() {
        BankAccount b = new BankAccount(123, "Fred", 100);
        b.withdraw(200);
    }

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

If you execute this class in BlueJ the appropriate throw statement will be highlighted in the BankAccount source code. The following output shows what happens when the Java interpreter processes an illegal argument exception.

```
Exception in thread "main" java.lang.IllegalArgumentException:
  Invalid amount for withdraw
at chapter6.bank_account.BankAccount.withdraw(BankAccount.java:58)
at chapter6.bank_account.ExceptionTester.doTest(ExceptionTester.java:10)
at chapter6.bank_account.ExceptionTester.main(ExceptionTester.java:15)
```

Our custom error message is displayed and there is useful information concerning the location of the error. The error occurred on line 58 in the BankAccount class within the body of the withdraw method which was called from line 10 in the ExceptionTester class, and this is within the body of the doTest method. Finally, the doTest method was called from line 15 which is within the body of the main method. Thus, the exception processing traces the flow of execution until the exception occurs.

### 6.15.3 Catching exceptions

Having the Java interpreter catch exceptions and terminate our program is rarely satisfactory, unless we cannot recover from the error. To catch and process exceptions ourselves we use what is called a **try-catch statement**. The simplest form is given by

```java
try
```
The idea here is that we put any statements that can throw exceptions in the try block. If no exception is thrown the catch block is ignored and control passes to the other statements below the try-catch statement as though the try-catch block was not there. However, if an exception is thrown execution immediately leaves the try block and the statements in the catch block are executed.

For our BankAccount example the catch block will have the form

```java
catch (IllegalArgumentException e)
{
    // statements to execute when try fails
}
```

Exceptions, like almost everything in Java, are objects. This catch block specifies the class to which the exception belongs, and a name for the exception object. Multiple catch blocks can be used if there is more than one type of exception. We can find out more about the exception that occurred by invoking the `getMessage` method on `e`. A string containing the error message will be returned.

As a simple illustration the following class uses a try-catch block to display the error message for the exception thrown by the `withdraw` method.

```java
package chapter6.bank_account; // remove this line if you're not using packages
/**
 * Catching an exception
 */
public class ExceptionCatcher
{
    public void doTest()
    {
        try
        {
            BankAccount b = new BankAccount(123, "Fred", 100);
            b.withdraw(200);
        }
        catch (IllegalArgumentException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```
public static void main(String[] args) {
    new ExceptionCatcher().doTest();
}

Here the method call expression b.withdraw(200) will throw an exception.
In this simple example we just display the error message using e.getMessage(): When the
doTest method is executed within BlueJ our error message

    Invalid amount for withdraw

is displayed in the terminal window. If you also want to trace the location of the exception then
replace the println statement with the statement

    e.printStackTrace();

In more realistic programs we could try to recover from the error and ask for another withdrawal amount.

6.16  Paper, scissors, rock game (PSR)

As an example of if-statements, boolean-valued methods, and exception processing we consider
the paper, scissors, rock (PSR) game. This is a game for two players, called Player 1 and Player 2.
We can imagine that each player has a piece of paper (P), a pair of scissors (S), and a rock (R). At
the signal, each player will present one of these three items. The rules are simple.

6.16.1  Rules of the game

- If Player 1 chooses paper and Player 2 chooses rock, Player 1 wins (paper covers rock)
- If Player 1 chooses paper and Player 2 chooses scissors, Player 2 wins (scissors cut paper)
- If Player 1 chooses rock and Player 2 chooses scissors, Player 1 wins (rock breaks scissors)
- If both players choose the same item, then the game is a draw.

The nine possible combinations are shown in Table 6.6. The three rows correspond to Player 1’s
choices, and the columns correspond to Player 2’s choices.

6.16.2  Object-oriented PSR game

We can write an OOP version of this game using two classes: one called PSRPlayer that represents
a player, and one called PSRGame that represents the game.
Designing the **PSRPlayer** class

We can represent the player choices using the characters ’P’ for paper, ’S’ for scissors, and ’R’ for rock. Each player object represents the current choice so we can provide methods for getting and setting the choice. This gives the class design:

```java
public class PSRPlayer {
    private char choice;
    public PSRPlayer() {...}
    public char getChoice() {...}
    public void setChoice(char choice) {...}
}
```

Designing the **PSRGame** class

The **PSRGame** class needs to construct a game for two players and their choices (aggregation), play a single round and report who won. Internally the class will determine the winner based on the rules of the game. The class design is:

```java
public class PSRGame {
    private PSRPlayer p1, p2;

    public static final int DRAW = 0;
    public static final int WIN_PLAYER_ONE = 1;
    public static final int WIN_PLAYER_TWO = 2;

    public PSRGame(PSRPlayer p1, PSRPlayer p2) {...}
    public int playRound() {...}
}
```

Here we have used the `int` return type on `playRound` since there are three possible outcomes of a round: player 1 wins, player 2 wins, or the round is a draw. We have defined three constants in the class to represent these three outcomes, and `playRound` will return one of them.

**PSRPlayer** implementation

The **PSRPlayer** implementation is simple:
**Class PSRPlayer**

```java
package chapter6.psr_game; // remove this line if you’re not using packages
/**
 * This class represents a player in the PSR game
 */
public class PSRPlayer
{
    private char choice; // the player’s choice

    /**
     * Construct a PSR game player
     */
    public PSRPlayer()
    {
    }

    /**
     * get the player’s choice
     * @return the player’s choice character
     */
    public char getChoice()
    {
        return choice;
    }

    /**
     * Set the choice made by the player
     * @param choice the choice character
     */
    public void setChoice(char choice)
    {
        this.choice = Character.toUpperCase(choice);
    }
}
```

Here we have converted the player’s character to upper case to make the character choices case-insensitive.

**PSRGame implementation**

The constructor implementation is simple and the `playRound` method can be expressed in terms of a boolean valued `isWin` method as

```java
public int playRound()
{
    // throw exceptions here for invalid choices

    if (isWin(p1, p2))
    {
```
Here the `isWin` method returns true if the first player wins over the second player so it is called twice to determine if a player wins. It can be implemented as a static method that uses Table 6.6 to determine who wins. The `playRound` method is also a good place to check for illegal input characters for the choices. We can do this by throwing an exception. The appropriate exception in this case is called `IllegalStateException`.

Finally, Table 6.6 shows that there are three ways that Player 1 can win, so we can simply use a "logical or" to combine them to obtain the condition.

\[
(p1 == 'P' && p2 == 'R') \lor (p1 == 'S' && p2 == 'P') \lor (p1 == 'R' && p2 == 'S')
\]

where `p1` and `p2` represent the choices of Player 1 and Player 2.

Here is the complete implementation.

```java
package chapter6.psr_game;

public class PSRGame {
    public static final int WIN_PLAYER_ONE = 1;
    public static final int WIN_PLAYER_TWO = 2;
    public static final int DRAW = 0;

    private PSRPlayer p1, p2;

    public PSRGame(PSRPlayer p1, PSRPlayer p2) {
    }
```
Making Decisions

```java
this.p1 = p1;
this.p2 = p2;
}

/**
 * Return the result of one round of the game as one of
 * the three public constants.
 * @return outcome as one of the three public constants
 */
public int playRound()
{
    if (isInvalidChoice(p1))
    {
        throw new IllegalStateException("Player 1: invalid choice");
    }

    if (isInvalidChoice(p2))
    {
        throw new IllegalStateException("Player 2: invalid choice");
    }

    if (isWin(p1, p2))
    {
        return WIN_PLAYER_ONE;
    }
    else if (isWin(p2, p1))
    {
        return WIN_PLAYER_TWO;
    }
    else
    {
        return DRAW;
    }
}

private static boolean isWin(PSRPlayer p1, PSRPlayer p2)
{
    char pc1 = p1.getChoice();
    char pc2 = p2.getChoice();
    return (pc1 == 'P' && pc2 == 'R') || (pc1 == 'S' && pc2 == 'P') ||
            (pc1 == 'R' && pc2 == 'S');
}

private static boolean isInvalidChoice(PSRPlayer p)
{
    char choice = p.getChoice();
    return choice != 'P' && choice != 'S' && choice != 'R';
}
```
Testing the class with BlueJ

You can play the game in BlueJ as follows:

1. Construct two player objects called p1 and p2.
2. Construct a PSRGame object called game, using p1 and p2 as arguments.
3. Select the setChoice method from the PSRPlayer object menu for each player and make a choice.
4. Select the playRound choice from the PSRGame object menu. The result 0, 1, or 2 will be shown in a method result box.
5. Repeat steps 3 and 4 for another round.

Testing the class with BeanShell

The following example shows how the class can be tested using BeanShell.

EXAMPLE 6.32 (PSR game) Try the statements in BeanShell

```bash
bsh % addClassPath("c:/book-projects/chapter6/psr_game");
bsh % PSRPlayer p1 = new PSRPlayer();
bsh % PSRPlayer p2 = new PSRPlayer();
bsh % PSRGame game = new PSRGame(p1, p2);
bsh % p1.setChoice('p');
bsh % p2.setChoice('s');
bsh % print(game.playRound());
  2
bsh % p1.setChoice('s');
bsh % p2.setChoice('p');
bsh % print(game.playRound());
  1
bsh % p1.setChoice('p');
bsh % p2.setChoice('p');
bsh % print(game.playRound());
  0
bsh %
```

to play three rounds of the game.

Running the game using a main method

To play a round of this game using a main method in BlueJ, or outside BlueJ from the command line, requires that we learn how to get console (terminal) input from the user. We need to write a special kind of user interface class called a console interface to play one round of the game from the command line as follows:
Player 1, enter your choice: P, S, or R
p
Player 2, enter your choice: P, S, or R
s
Player 2 wins!

Here each player types a character and presses return in response to a prompt.

### 6.17 Console Input Using a Scanner object

Before Java 1.5 getting console input was not trivial. Most people wrote a special class to do this. Console input has been standardized in Java 1.5 with the introduction of the Scanner class. It is in a package called java.util and can be imported into any class by using the import statement

```java
import java.util.Scanner;
```

#### 6.17.1 Some useful Scanner methods

This class has a constructor and methods with the following prototypes

- **Scanner input = new Scanner(System.in);**
  
  Construct a Scanner object called input and connect it for reading from the console. There are many other types of constructors but this is the only one we need. The System.in object is used to get input from the keyboard in the console window just as System.out is used to display output in the console window.

- **public int nextInt()**
  
  Read the next number typed in the console window and return it as an int value.

- **public long nextLong()**
  
  Read the next number typed in the console window and return it as a long value.

- **public float nextFloat()**
  
  Read the next number typed in the console window and return it as a float value.

- **public double nextDouble()**
  
  Read the next number typed in the console window and return it as a double value.

- **public String nextLine()**
  
  Read the rest of a line typed in the console window and return it as a String.

There are many other methods but these are the only ones we need.
6.17 Console Input Using a **Scanner** object

### 6.17.2 One input per line input model

If you are not careful there are some pitfalls when using the **Scanner** class for interactive input. For example, the statements

```java
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
System.out.println("Enter your name");
String name = input.nextLine();
```

do not work as you may expect since `name` will be the empty string. This is so because when you enter the age and press the Enter key the newline is not read by the `nextInt` method. Instead it is read by the `nextLine` method and the result is the empty string being assigned to `name`. The name that you typed is left unread in the input buffer.

The following statements avoid this

```java
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
input.nextLine(); // eat the end of line
System.out.println("Enter your name");
String name = input.nextLine();
```

Following each numeric input method call by the statement

```java
input.nextLine();
```

will eat the new line character (throw it away). This is called the one input per line interactive input model.

### 6.17.3 Console interface class for the PSR game

Using the **Scanner** class we can write a console-interface class for the PSR game which can be run both inside BlueJ and outside BlueJ from the command line.

```java
package chapter6.psr_game; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * A console interface for one round of the PSRGame
 */
public class PSRGameRunner {
    public void run()
    {
```
Scanner input = new Scanner(System.in);
PSRPlayer p1 = new PSRPlayer();
PSRPlayer p2 = new PSRPlayer();
PSRGame game = new PSRGame(p1, p2);

System.out.println("Player 1, enter your choice: P, S, or R");
char player1Choice = input.nextLine().charAt(0);
p1.setChoice(player1Choice);

System.out.println("Player 2, enter your choice: P, S, or R");
char player2Choice = input.nextLine().charAt(0);
p2.setChoice(player2Choice);

int result = 0; // any initial value will do

try
{
    result = game.playRound();
}
catch (IllegalStateException e)
{
    System.out.println("Illegal input");
    result = PSRGame.DRAW;
}

if (result == PSRGame.WIN_PLAYER_ONE)
    System.out.println("Player 1 wins!");
else if (result == PSRGame.WIN_PLAYER_TWO)
    System.out.println("Player 2 wins!");
else
    System.out.println("It’s a draw");

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


In the run method we first construct the objects needed to play a round. Then we use the nextLine method and the setChoice method to set each player’s character to the first character of the input line.

If playRound throws an exception, we display a message and set the outcome to a draw. It is necessary to declare the result variable outside the try block, otherwise it would be local to the try block and undefined in the if-statement that displays the result.

This PSRGameRunner class can be run from the command line or inside BlueJ using either the main method or by constructing a PSRGameRunner object and choosing its run method.
6.18 Complex roots of a quadratic equation

We have written a QuadraticRootFinder class (page 256) that calculates the roots only if they are real. We now generalize and find the roots in any case, real or complex.

Recall that a complex number has the form \( a + bi \) where \( a \) and \( b \) are called the real and imaginary part of the complex number and \( i = \sqrt{-1}, \ i^2 = -1 \). Given the quadratic equation \( ax^2 + bx + c = 0 \) the roots have non-zero imaginary part if \( b^2 - 4ac < 0 \) and are given by

\[
 r_1 = c + di, \quad r_2 = c - di \quad \text{where} \quad c = -\frac{b}{2a}, \quad d = \frac{\sqrt{|b^2 - 4ac|}}{2a}
\]

An interesting way to find the roots is to first write a class called Complex that represents complex numbers and then use it in a class called ComplexQuadraticRootFinder to find the roots.

A simple design for the Complex class is

```java
public class Complex
{
    private double realPart;
    private double imagPart;

    public Complex(double real, double imag) {...}
    public double getRealPart() {...}
    public double getImagPart() {...}
    public String toString() {...}
}
```

6.18.1 Complex class

Except for the toString method the implementation is straightforward:

```java
package chapter6.root_finder; // remove this line if you're not using packages
/**
 * A simple class whose objects represent complex numbers by their
 * real and imaginary parts.
 */
public class Complex
{
    private double realPart;
    private double imagPart;

    /** Construct complex number with given real and imaginary parts.
     * @param real the real part of the complex number
     * @param imag the imag part of the complex number
     */
    public Complex(double real, double imag)
```
Making Decisions

```java
{ 
    realPart = real;
    imagPart = imag;
}

/**
 * Returns real part of this complex number.
 * @return the real part of this complex number
 */
public double getRealPart()
{
    return realPart;
}

/**
 * Returns imaginary part of this complex number.
 * @return the imaginary part of this complex number
 */
public double getImagPart()
{
    return imagPart;
}

/** Returns a string representation of this complex number.
 * @return a string representation of this complex number
 * of the form a + b i, a - b i, or a in case b = 0
 */
public String toString()
{
    if (imagPart > 0)
        return realPart + " + " + imagPart + " i";
    else if (imagPart < 0)
        return realPart + " - " + Math.abs(imagPart) + " i";
    else
        return realPart + ";"
}
```

The `toString` method displays the complex number in the form \( a + bi \) or \( a - bi \) depending on the sign of \( b \).

Now the `QuadraticRootFinder` class (page 6.4.1) can easily be modified to give the complex version:

```java
package chapter6.root_finder; // remove this line if you're not using packages

/**
 * An object of this class can calculate the complex roots of the
 * quadratic equation \( ax^2 + bx + c = 0 \) given the coefficients \( a, b, \) and \( c \).
 */
```
public class ComplexQuadraticRootFinder
{
    private double a, b, c;
    private Complex root1, root2;

    /**
     * Construct a quadratic equation root finder given the coefficients
     * @param aCoeff first coefficient in ax^2 + bx + c
     * @param bCoeff second coefficient in ax^2 + bx + c
     * @param cCoeff third coefficient of ax^2 + bx + c
     */
    public ComplexQuadraticRootFinder(double aCoeff, double bCoeff, double cCoeff)
    {
        a = aCoeff;
        b = bCoeff;
        c = cCoeff;
        doCalculations();
    }

    private void doCalculations()
    {
        double d1 = b*b - 4*a*c;
        double d2 = Math.sqrt(Math.abs(d1));
        if (d1 >= 0)
        {
            // real root case
            double realPart1 = (-b - d2) / (2.0 * a);
            double realPart2 = (-b + d2) / (2.0 * a);
            root1 = new Complex(realPart1, 0.0);
            root2 = new Complex(realPart2, 0.0);
        }
        else
        {
            // complex root case
            double realPart = -b / (2.0 * a);
            double imagPart = d2 / (2.0 * a);
            root1 = new Complex(realPart, imagPart);
            root2 = new Complex(realPart, -imagPart);
        }
    }

    /**
     * Return the first root as a complex number.
     * @return the first root as a complex number
     */
    public Complex getRoot1()
    {
        return root1;
    }
}
/**
 * Return the second root as a complex number.
 * @return the second root as a complex number
 */
public Complex getRoot2()
{
    return root2;
}

/**
 * Return the coefficient of x^2.
 * @return the coefficient of x^2
 */
public double getA()
{
    return a;
}

/**
 * Return the coefficient of x.
 * @return the coefficient of x
 */
public double getB()
{
    return b;
}

/**
 * Return the constant coefficient.
 * @return the constant coefficient
 */
public double getC()
{
    return c;
}

/**
 * Change the value of the coefficient of x^2.
 * @param value the new value for the coefficient of x^2
 */
public void setA(double value)
{
    a = value;
    doCalculations();
}

/**
 * Change the value of the coefficient of x.
 * @param value the new value for the coefficient of x
 */
public void setB(double value)
{
6.18 Complex roots of a quadratic equation

```java
b = value;
doCalculations();
}

/**
 * Change the value of the constant coefficient.
 * @param value the new value for the constant coefficient.
 */
public void setC(double value)
{
    c = value;
doCalculations();
}
```

The return type of the `getRoot1` and `getRoot2` methods is now `Complex`.

**Testing the class with BlueJ**

To test the class in BlueJ perform the following steps.

1. Construct a `ComplexQuadraticRootFinder` object called `finder` with values for `a, b, c`.
2. From the object menu select `getRoot1()` and click on `<object-reference>`.
3. Click “Get” button and name the `Complex` object `root1`. It will appear on the object bench.
4. From the object menu select `getRoot2()` and click on `<object-reference>`.
5. Click “Get” button and name the `Complex` object `root2`. It will appear on the object bench.
6. From the object menu of `root1` and `root2` select the `toString` method to see the roots.

**Testing the class with BeanShell**

The following example shows how the class can be tested using BeanShell.

```bash
EXAMPLE 6.33 (Complex roots) Try the statements in BeanShell

bsh % addClassPath("c:/book-projects/chapter6/root_finder");
bsh % ComplexQuadraticRootFinder finder = new ComplexQuadraticRootFinder(3,4,5);
bsh % Complex r1 = finder.getRoot1();
bsh % Complex r2 = finder.getRoot2();
bsh % print(r1);
-0.6666666666666666 + 1.1055415967851332 i
bsh % print(r2);
-0.6666666666666666 - 1.1055415967851332 i
bsh %
```

To find the complex roots of a quadratic equation.
Console interface

We can use `Scanner` to write the following console interface called `ComplexRunner` for the complex root finding class:

```java
package chapter6.root_finder; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * A simple runner class for finding complex roots of a quadratic equation
 */
public class ComplexRunner
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter coefficient a");
        double a = input.nextDouble();
        input.nextLine(); // eat the end of line

        System.out.println("Enter coefficient b");
        double b = input.nextDouble();
        input.nextLine(); // eat the end of line

        System.out.println("Enter coefficient c");
        double c = input.nextDouble();
        input.nextLine(); // eat the end of line

        ComplexQuadraticRootFinder finder = new ComplexQuadraticRootFinder(a,b,c);
        Complex root1 = finder.getRoot1();
        Complex root2 = finder.getRoot2();

        System.out.println("Root 1 is " + root1);
        System.out.println("Root 2 is " + root2);
    }

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

This class can be run inside BlueJ using the `run` method and the terminal window, or from the command line using the `main` method. Typical command-line output is

```
Enter coefficient a
3
```
6.19 Review exercises

◮ Review Exercise 6.1 Define the following terms and give examples of each.

- conditional execution
- boolean expression
- boolean literal
- comparison expression
- comparison operator
- equality expression
- equality operator
- relational expression
- relational operator
- block
- flowchart
- absolute error
- relative error
- conditional operator
- compound boolean expression
- truth table
- short circuit evaluation
- lexicographical ordering
- character code
- boolean valued method
- exception
- throwing an exception
- catch block

◮ Review Exercise 6.2 Develop a set of test data for program class PSRGameTester that will guarantee that (1) every branch of the PSRGame program is tested, and (2) for all legal inputs the program produces the correct output. This is called exhaustive testing and it constitutes a proof that the program is correct. In most cases programs are too complicated for exhaustive testing.

◮ Review Exercise 6.3 Write truth tables that verify deMorgan’s laws

\[
\neg(a \land b) = (\neg a) \lor (\neg b) \\
\neg(a \lor b) = (\neg a) \land (\neg b)
\]

◮ Review Exercise 6.4 The operation \(p \lor q\) is true if either of \(p\) and \(q\) is true or both are true. The exclusive or of \(p\) and \(q\), denoted by \(p \oplus q\), excludes the case that both are true. Its truth table is shown in Table 6.7. Using a truth table show that

\[
p \oplus q \equiv (p \lor q) \land \neg (p \land q)
\]

Table 6.7: Truth table for the exclusive or

<table>
<thead>
<tr>
<th></th>
<th>(p)</th>
<th>(q)</th>
<th>(p \oplus q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

where \(a \equiv b\) means that \(a\) and \(b\) are logically equivalent (they have the same truth table). This shows that the exclusive or can be expressed in terms of the logical operators \(\land\), \(\lor\), and \(\neg\).
Review Exercise 6.5  For the CircleCalculator class from Chapter 3 (page 63) what modifications would you make to include error processing using exceptions.

Review Exercise 6.6  For the TriangleCalculator class from Chapter 3 (page 64) what modifications would you make to include error processing using exceptions.

6.20  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.

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 6.1  Evaluate the abs method in Example 6.2 in the editor and try statements such as the following in the workspace

```bash
bsh % double result = abs(3);
bsh % print(result);
3.0
bsh % result = abs(-3);
bsh % print(result);
3.0
```

BeanShell Exercise 6.2  Do BeanShell Exercise 6.1 using the abs method in Example 6.10

BeanShell Exercise 6.3  Repeat BeanShell Exercise 6.1 for the cubeRoot method in Example 6.3

BeanShell Exercise 6.4  Do BeanShell Exercise 6.3 using the cubeRoot in Example 6.12

BeanShell Exercise 6.5  Using the statements in Example 6.6 write a method in the editor called max that takes two double values and returns their maximum. Test the method using the workspace as in BeanShell Exercise 6.1

BeanShell Exercise 6.6  Do BeanShell Exercise 6.5 using the statement in Example 6.11

BeanShell Exercise 6.7  Convert Example 6.13 into a method called calculateTax that has one argument, the amount a, and returns the tax calculated. Test your method as in BeanShell Exercise 6.1

BeanShell Exercise 6.8  Do BeanShell Exercise 6.7 using the if-statement in in Example 6.22
BeanShell Exercise 6.9 Convert Example 6.14 into a method called letterGrade that has one argument for the mark, and returns the letter grade as a String. Test your method as in BeanShell Exercise 6.1.

BeanShell Exercise 6.10 Do BeanShell Exercise 6.9 using the if-statement in in Example 6.21.

BeanShell Exercise 6.11 Verify some of the results in Table 6.4 using BeanShell statements such as

```java
bsh % print((int) 'a');
  97
bsh % print((int) 'A');
  65
```

6.21 Programming exercises

Exercise 6.1 (A marks converter class)
Write a class called MarksConverter that uses a method with prototype

```java
public String letterGrade(int mark)
```

based on the if-statement in Example 6.21 to convert a mark to a letter grade. Indicate how to test your class in BlueJ and BeanShell.

Exercise 6.2 (A tax calculator class)
Write a class called TaxCalculator that uses a method with prototype

```java
public double calculateTax(double amount)
```

based on the if-statement in Example 6.22, to calculate the tax on a given amount of money. Indicate how to test your class in BlueJ and BeanShell.

Exercise 6.3 (A better ChangeHelper class)
Rewrite the ChangeHelper class in Exercise 3.6, Chapter 3 so that zero amounts are not displayed. You should also check that the amount received is not smaller than the amount due.

Exercise 6.4 (A better CircleCalculator class)

(a) Write a new version of the CircleCalculator class from Chapter 3 (page 63) that uses an exception in case there is an illegal argument in the constructor.

(b) Write a console-interface called CircleCalculatorRunner that shows how to test this class using the Scanner class.

Exercise 6.5 (A better TriangleCalculator class)

(a) Write a new version of the TriangleCalculator class from Chapter 3 (page 64) that uses exceptions in case there are illegal arguments.
(b) Write a console-interface called TriangleCalculatorRunner that shows how to test this class using the Scanner class.

▶ Exercise 6.6 (Maximum of three numbers)
Write a class called MaxThreeCalculator that computes the maximum of three double numbers using the following class design.

```java
public class MaxThreeCalculator
{
    private double maximum; // maximum of x1, x2, and x3

    public MaxThreeCalculator(double x1, double x2, double x3) {...}
    public double getMaximum() { return maximum; }
}
```

Use two private methods with prototypes

```java
private double max2(double n1, double n2)
private double max3(double n1, double n2, double n3)
```

It doesn’t matter whether these methods are static or not since they don’t access any instance data fields. The max2 method should return the maximum of two numbers and the max3 method should use max2 to return the maximum of three numbers. Finally, max3 can be called in the constructor to calculate the value of the data field maximum which can be returned by the getMaximum method.

Give a set of test data that you would use to verify the correctness of your program.

▶ Exercise 6.7 (Finding the smallest of three strings)
(a) Write a class called StringSorter that takes three strings and arranges them in increasing lexicographical order using the compareTo method in the String class. Use the class design

```java
public class StringSorter
{
    private String first, second, third;
    public StringSorter(String x1, String x2, String x3) {...}
    public String getFirst() { return first; }
    public String getSecond() { return first; }
    public String getThird() { return first; }
}
```

The constructor should sort the three strings x1, x2, and x3 and assign them in sorted order to first, second, and third.

(b) Give a set of test data that you would use to verify the correctness of your program.

(c) Write a console-interface called StringSorterTester that shows how to test this class using the Scanner class.
Exercise 6.8 (Finding the maximum balance for three bank accounts)
Write a class called MaxMinAccount that finds the minimum and maximum balance for three BankAccount objects. Use the class design:

```java
public class MaxMinAccount {
    private BankAccount min, max;
    public MaxMinAccount(BankAccount b1, BankAccount b2, BankAccount b3) {...}
    public BankAccount getMin() { return min; }
    public BankAccount getMax() { return max; }
}
```

Here min is a reference to the account with the minimum balance among b1, b2, and b3, and max is a reference to the account with the maximum balance. The “get” methods return references to these accounts.

Exercise 6.9 (Multiple if-statement and decision tables)
(a) Multiple if-statements directly correspond to a decision table and vice versa. Such a table lists a set of mutually exclusive rules for calculating some quantity. As an example, consider the calculation of the sales commission received by a real estate agent for selling a house. Here is the table:

<table>
<thead>
<tr>
<th>Selling price $p$</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq p \leq $100,000$</td>
<td>3 percent</td>
</tr>
<tr>
<td>$$100,000 &lt; p \leq $250,000$</td>
<td>5 percent</td>
</tr>
<tr>
<td>$$250,000 &lt; p \leq $500,000$</td>
<td>7 percent</td>
</tr>
<tr>
<td>$p &gt; $500,000$</td>
<td>10 percent</td>
</tr>
</tbody>
</table>

Write a class called SalesCommissionCalculator having the structure:

```java
public class SalesCommissionCalculator {
    private double sellingPrice;
    private double commission;

    public SalesCommissionCalculator(double sellingPrice) {...}

    public double getSellingPrice() { return sellingPrice; }
    public double getCommission() { return commission; }
}
```

that has a constructor that uses the selling price to construct a sales commission object and calculate the commission. The two enquiry methods can be used to retrieve the selling price and commission. The constructor should throw an exception if the selling price is negative. Do the calculation of the commission from the selling price using a method with prototype...
public double commission(double sellingPrice)

that returns the commission. Now the SalesCommission constructor can simply call this method.

(b) Use the following console-interface class to test your class.

```java
import java.util.Scanner;
public class SalesCommissionRunner {
    public void run() {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter selling price");
        double price = input.nextDouble();
        input.nextLine();

        try {
            SalesCommission sc = new SalesCommission(price);
            System.out.println("Selling price: " + sc.getSellingPrice());
            System.out.println("Commission: " + sc.getCommission());
        }
        catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
    public static void main(String[] args) {
        new SalesCommissionRunner().run();
    }
}
```

Exercise 6.10 (Calculating federal income tax)

(a) Using the previous exercise as a guide, write a FederalTaxCalculator class that computes the Federal tax, given two amounts: the taxable income and the total non-refundable tax credits. Here are the mutually exclusive rules:

1. If the taxable income is not greater than $29,590.00, the federal tax is 17 percent of the taxable income.
2. If the taxable income is greater than $29,590.00 but not greater than $59,180.00, the federal tax is $5030.00 on the first $29,590.00 and 26 percent on the remainder.
3. If the taxable income is greater than $59,180.00, the federal tax is $12,724.00 on the first $59,180.00 and 29 percent on the remainder.
From the amount calculated, subtract the total non-refundable tax credits to obtain the total federal tax payable. If this amount is negative, the total federal tax payable is zero.

(b) Write a console-interface called TaxCalculatorRunner that reads the taxable income and the total non-refundable tax credits, computes the total federal tax payable, and displays the result.

**Exercise 6.11 (Calculating roots of quadratic equation)**

Modify the ComplexQuadraticRootFinder class to use in the real root case, the following formulas

1. If $b \geq 0$ define $r_1 = -\frac{1}{2a} \left( b + \sqrt{b^2 - 4ac} \right)$
2. If $b \leq 0$ define $r_1 = -\frac{1}{2a} \left( b - \sqrt{b^2 - 4ac} \right)$
3. In either case $r_1 r_2 = c/a$ so the second root is $r_2 = c/(ar_1)$

Your class should also deal with special cases which arise when some of the coefficients are zero or the roots are not real. For example, if $a = 0$ and $b = 0$ then there is no equation, if $a = 0$ and $b \neq 0$ then the equation is linear and there is one root. The class can provide boolean-valued methods hasRealRoots, isLinear, isInvalidEquation that can be used by a runner class to determine what to display.

Give a set of test cases for the various paths through your class and write a suitable console-interface class to test your class

**Exercise 6.12** Roman numerals are still in use. In the motion picture industry the year a film is released is given in roman numerals. Write a program called RomanNumeralConverter to convert year numbers in the range 1 to 3999 to a string of roman numerals. Use an instance method with prototype

```java
public String roman(int year)
```

For example the year 1998 is MCMXCVIII in roman numerals and the year 2003 is MMIII.

Hint: Use / and % to determine the thousands, hundreds, tens and units digits of the number and convert each part to roman numerals.

**Exercise 6.13 (Solving cubic equations)** A mathematician has given you the pseudo-code algorithm for finding the real roots of the cubic equation $ax^3 + bx^2 + cx + d = 0$ shown in Figure 6.8. According to the algorithm the equation has either one real root or three real roots. Write a program class called CubicSolver for this algorithm that has the following structure

```java
public class CubicSolver {
    private double a, b, c, d; // ax^3 + bx^2 + cx + d = 0
    private double root1, root2, root3; // roots of equation
    private boolean oneRealRoot; // true in one real root case
```
ALGORITHM CubicSolver($a, b, c, d$)
$p \leftarrow \frac{1}{3a} \left[ 3ac - b^2 \right], \quad q \leftarrow \frac{1}{27a^2} \left[ 2b^3 - 9abc + 27a^2d \right]$
$\Delta \leftarrow \frac{p^3}{27} + \frac{q^2}{4}, \quad s \leftarrow \frac{b}{3a}$

**IF** $\Delta > 0$ **THEN**
$f_1 \leftarrow -\frac{q}{2} + \sqrt{\Delta}, \quad f_2 \leftarrow -\frac{q}{2} - \sqrt{\Delta}$
$y_1 \leftarrow (f_1)^{1/3} + (f_2)^{1/3}$
$x_1 \leftarrow y_1 - s$
**RETURN** $x_1$

**ELSE IF** $(\Delta = 0) \cap (q = 0)$ **THEN**
$x_1 \leftarrow -s, \quad x_2 \leftarrow -s, \quad x_3 \leftarrow -s$
**RETURN** $x_1, x_2, x_3$

**ELSE**
$m \leftarrow 2\sqrt{-\frac{p}{3}}$
$\theta \leftarrow \frac{1}{3} \arccos \left( \frac{3q}{pm} \right)$
$y_1 \leftarrow mcos\theta, \quad y_2 \leftarrow mcos\left( \theta + \frac{2\pi}{3} \right), \quad y_3 \leftarrow mcos\left( \theta + \frac{4\pi}{3} \right)$
$x_1 \leftarrow y_1 - s, \quad x_2 \leftarrow y_2 - s, \quad x_3 \leftarrow y_3 - s$
**RETURN** $x_1, x_2, x_3$

**END IF**

Figure 6.8: Pseudo-code algorithm for roots of a cubic equation

```java
public CubicSolver(double aa, double bb, double cc, double dd ) {...}
public boolean hasOneRealRoot() { return oneRealRoot; }
public boolean isCubic() { return a != 0.0; }
public double getRoot1() { return root1; }
public double getRoot2() { return root2; }
public double getRoot3() { return root3; }
public double getA() { return a; }
public double getB() { return b; }
public double getC() { return c; }
public double getD() { return d; }
private double cubeRoot(double x) {...}
```

where the `cubeRoot` method is from Example 6.12. The QuadraticSolver class can be used if `isCubic` returns false (coefficient of $x^3$ is zero).

Write a runner class to test the program and develop some test data. Include a check by also displaying the value of $ax^3 + bx^2 + cx + d$ for each root found (the value should be close to zero).