Unit 5

Conditional statements

Summary

- The if-else and if statements
- Block of statements
- Conditional expression
- Comparison between objects
- The switch statement

5.1 Statements in Java

Till now we have seen two types of executable statements (without counting declarations):

- method invocation
- assignment

These are simple statements by means of which we can write programs

- constituted by sequences of simple statements;
- that do method calls, possibly nested.

Very often, when we have to solve a problem, we are interested in performing different actions depending on whether certain conditions are true or false.

5.2 Conditional statements

Java, like all other programming languages, is equipped with specific statements that allow us to check a condition and execute certain parts of code depending on whether the condition is true or false. Such statements are called conditional, and are a form of composite statement.

In Java, there are two forms of conditional statements:

- the if-else statement, to choose between two alternatives;
- the switch statement, to choose between multiple alternatives.

5.3 The if-else statement

The if-else statement allows us to select between two alternatives.

**Syntax**:

```java
if (condition)
    then-statement
else
    else-statement
```

- `condition` is an expression of type boolean, i.e., a conditional expression that is evaluated to true or false
- `then-statement` is a single statement (also called the then-branch of the if-else statement)
- `else-statement` is a single statement (also called the else-branch of the if-else statement)
Semantics:
First, the condition is evaluated. If the result of the evaluation is the value true, the then-statement is executed, otherwise the else-statement is executed. In both cases, the execution continues with the statement immediately following the if-else statement.

Example:
```java
int a, b;
...
if (a > b)
    System.out.println("bigger value = " + a);
else
    System.out.println("bigger value = " + b);
```

When this if-else statement is executed, the string "bigger value = " followed by the bigger one among a and b is printed on the output channel (the monitor).

5.4 Condition in an if-else statement

The condition in an if-else statement can be an arbitrary expression of type boolean, for example:
- a variable of type boolean;
  Example:
  ```java
  boolean finished;
  ... 
  if (finished)
  ...
  ```
- one of the comparison operators (==, !=, >, >=, or <=) applied to variables (or expressions) of a primitive type;
  Example:
  ```java
  int a, b, c;
  ... 
  if (a > b + c)
  ...
  ```
- a call to a predicate (i.e., a method that returns a value of type boolean);
  Example:
  ```java
  String answer;
  ... 
  if (answer.equalsIgnoreCase("YES"))
  ...
  ```
- a complex boolean expression, obtained by applying the boolean operators !, &&, and || to simpler expressions;
  Example:
  ```java
  int a, b, c, d;
  String answer;
  ... 
  if ((a > (b+c)) || (a == d) && !answer.equalsIgnoreCase("YES"))
  ...
  ```

5.5 The if variant

The else part of an if-else statement is optional. If it is missing, we have an if statement, which allows us to execute a certain part of code if a condition is satisfied (and do nothing otherwise).

if statement

Syntax:
if (condition)
    then-statement

• condition is an expression of type boolean
• then-statement is a single statement (also called the then-branch of the if statement)

Semantics:
First, the condition is evaluated. If the result of the evaluation is the value true, the then-statement is executed, and the execution continues with the statement immediately following the if statement. Otherwise, the execution continues directly with the statement following the if statement.

Example:

    boolean found;
    ...
    if (!found)
        System.out.println("element not found");

When this if statement is executed, the string "element not found" is printed on the output channel, provided the value of the boolean variable found is false.

5.6 Block of statements

The syntax of if-else allows us to have only a single statement in the then-branch (or the else-branch). If we want to execute more than one statement in the then-branch (or the else-branch), we have to use a block construct. A block of statements groups several statements in a single composite statement.

Block of statements

Syntax:

    {
        statement
        ...
        statement
    }

• statement is an arbitrary Java statement

Semantics:
The statements in the block are executed in sequence. The variables declared inside the block are not visible outside the block itself.

Example:

    int a, b, bigger;
    ...
    if (a > b) {
        bigger = a;
        System.out.println("smaller = " + b);
    }

5.7 Scope of variables defined in a block

A block of statements can contain variable declarations. The scope of a variable declared inside a block is the block itself, including other blocks contained in it, if present. This means that the variable is visible in the block and in all sub-blocks, but is not visible outside the block.

Example:

    public class ScopeInBlock {
        public static void main(String[] args) {
            String a = "Hello";
5.8 Use of blocks in an if-else statement

The then-branch or the else-branch of an if-else statement can be any Java statement, and in particular it can be a block.

*Example:* Given month and year, compute month and year of the next month.

```java
int month, year, nextMonth, nextYear;
...
if (month == 12) {
    nextMonth = 1;
    nextYear = year + 1;
} else {
    nextMonth = month + 1;
    nextYear = year;
}
```

5.9 Nested if’s

We have a nested if when the then-branch or the else-branch of an if-else statement is again an if-else or an if statement.

*Example:* Given day, month, and year, compute day, month, and year of the next day.

```java
int day, month, year, nextDay, nextMonth, nextYear;
...
if (month == 12) {
    if (day == 31) {
        nextDay = 1;
        nextMonth = 1;
        nextYear = year + 1;
    } else {
        nextDay = Day + 1;
        nextMonth = month;
    }
```
5.10 Nested if’s with mutually excluding conditions

A common use of nested if’s is when the conditions in the nested if’s are mutually excluding, i.e., no two of them can be simultaneously true.

Example: Based on the value of the temperature (an integer) print a message according to the following table:

<table>
<thead>
<tr>
<th>temperature $t$</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>$30 &lt; t$</td>
<td>hot</td>
</tr>
<tr>
<td>$20 &lt; t \leq 30$</td>
<td>warm</td>
</tr>
<tr>
<td>$10 &lt; t \leq 20$</td>
<td>fine</td>
</tr>
<tr>
<td>$t \leq 10$</td>
<td>cold</td>
</tr>
</tbody>
</table>

int temp;
...
if (30 < temp)
    System.out.println("hot");
else if (20 < temp)
    System.out.println("warm");
else if (10 < temp)
    System.out.println("fine");
else
    System.out.println("cold");

Observations:
• At the outermost level we have a single if-else statement.
• The order in which the conditions are specified is important.
• The second condition need not be composite, e.g., $(20 < temp) \&\& (temp <= 30)$, since it appears in the else-branch of the first condition. Hence, we already know that $(temp <= 30)$ is true.
• Each else refers to the if that immediately precedes it.

5.11 Ambiguity of the else in if-else statements

Consider the following code fragment:

```java
if (a > 0) if (b > 0) System.out.println("b positive");
else System.out.println("???");
System.out.println("???")
```

System.out.println("???") could in principle be the else-branch of:
• the first if: hence we should replace "???" with "a negative";
• the second if: hence we should replace "???" with "b negative".

The ambiguity is solved by considering that an else always refers to the nearest if without an associated else. In the above example we have:

```java
if (a > 0)
    if (b > 0)
        System.out.println("b positive");
    else
        System.out.println("b negative");
```

It is always possible to use a block (i.e., \{..\}) to disambiguate nested if-else statements. In particular, if we want that an else refers to an if that is not the immediately preceding one, we have to enclose the immediately preceding if in a block. For example:
if (a > 0) {
    if (b > 0)
        System.out.println("b positive");
    else
        System.out.println("a negative");
}

5.12 Example: type of a triangle

Given three values representing the lengths of the three sides of a triangle, determine whether the triangle is regular (all three sides are equal), symmetric (two sides are equal), or irregular (no two sides are equal).

A possible algorithm is the following: compare the sides two by two, until we have gathered sufficient information to decide the type of the triangle.

The algorithm can be implemented as follows:

```java
double first, second, third;
...
if (first == second) {
    if (second == third)
        System.out.println("regular");
    else
        System.out.println("symmetric");
} else {
    if (second == third)
        System.out.println("symmetric");
    else if (first == third)
        System.out.println("symmetric");
    else
        System.out.println("irregular");
}
```

5.13 Shortcut evaluation of a complex condition

The condition in an if-else statement can be a complex boolean expression, in which the logical operators &&, ||, and ! may appear. We have seen that Java performs a shortcut evaluation of such expressions. In other words, the subexpressions to which the operators are applied are evaluated from left to right as follows:

- when evaluating \((e_1 \&\& e_2)\), if the evaluation of \(e_1\) returns false, then \(e_2\) is not evaluated.
- when evaluating \((e_1 \mid\| e_2)\), if the evaluation of \(e_1\) returns true, then \(e_2\) is not evaluated.

Consider the case of \((e_1 \&\& e_2)\). If the value of \(e_1\) is false, then the value of the whole expression \((e_1 \&\& e_2)\) is false, independently of the value of \(e_2\). This justifies why in Java \(e_2\) is not even evaluated. Similar considerations hold for \((e_1 \mid\| e_2)\).

In general, the fact that Java performs a shortcut evaluation of boolean expressions has to be taken into account and cannot be ignored, since the correctness of the code may depend on that.

Example:

```java
String s;
...
if (s != null && s.length() > 0) {
    System.out.println(s);
}
```

In this case, when the value of \(s\) is null then \(s.length()>0\) is not evaluated and the method \(length()\) is not called. Note that, if Java evaluated \(s.length()>0\) also when \(s\) is null, then the above code would be wrong, since it would cause trying to access via \(s\) a nonexistent object.

Note: In general, if-else statements that make use of complex boolean conditions could be rewritten by making use of nested if-else statements. However, to do so, it may be necessary to duplicate code. We illustrate this in the following separately for \&\& and \|\|.

5.14 Eliminating the conjunction operator \&\& in a complex condition

The code fragment

©Diego Calvanese
Lecture Notes for Introduction to Programming
A.A. 2005/06
```java
if ((x < y) && (y < z))
    System.out.println("y is between x and z");
else
    System.out.println("y is not between x and z");
```
corresponds to

```java
if (x < y)
    if (y < z)
        System.out.println("y is between x and z");
    else
        System.out.println("y is not between x and z");
else
    System.out.println("y is not between x and z");
```
In this case, by eliminating the complex condition, the code of the else-branch must be duplicated.

Note that, due to shortcut evaluation, the second condition in the `&&` is not evaluated if the first condition is false. And this holds also for the corresponding nested `if-else` statements.

### 5.15 Eliminating the disjunction operator `||` in a complex condition

The code fragment

```java
if ((x == 1) || (x == 2))
    System.out.println("x equal to 1 or to 2");
else
    System.out.println("x different from 1 and from 2");
```
corresponds to

```java
if (x == 1)
    System.out.println("x equal to 1 or to 2");
else if (x == 2)
    System.out.println("x equal to 1 or to 2");
else
    System.out.println("x different from 1 and from 2");
```
In this case, by eliminating the complex condition, the code of the then-branch must be duplicated.

Again, due to shortcut evaluation, the second condition in the `||` is not evaluated if the first condition is true. And this holds also for the corresponding nested `if-else` statements.

### 5.16 Conditional expression

Java is equipped with a selection operator that allows us to construct a conditional expression. The use of a conditional expression can in some cases simplify the code with respect to the use of an `if-else` statement.

**Syntax:**

```
condition ? expression-1 : expression-2
```

- `condition` is a boolean expression
- `expression-1` and `expression-2` are two arbitrary expressions, which must be of the same type

**Semantics:**

Evaluate `condition`. If the result is `true`, then evaluate `expression-1` and return its value, otherwise evaluate `expression-2` and return its value.

**Example:**

```java
System.out.println("bigger value = " + (a > b)? a : b);
```
The statement in the example, which makes use of a conditional expression, is equivalent to:

```java
if (a > b)
    System.out.println("bigger value = " + a);
else
    System.out.println("bigger value = " + b);
```

Note that the selection operator is similar to the `if-else` statement, but it works at a different syntactic level:
- The selection operator combines expressions and returns another expression. Hence it can be used wherever an expression can be used.
- The `if-else` statement groups statements, and the result is a composite statement.

**5.17 Note on comparisons: equality between strings**

To test whether two strings (i.e., two objects of the class `String`) are equal, we have to use the `equals()` method. It would be wrong to use `==`.

*Example:*

```java
String input;
...
if (input == "YES") { ... } // This is WRONG!!!
if (input.equals("YES")) { ... } // This is CORRECT!!!
```

Indeed, `==` tests the equality between two references to an object, and this corresponds to test the identity of the two objects (i.e., that the two objects are in fact the same object). Instead, `equals()` tests that the two objects have the same content (i.e., that the two strings are constituted by the same sequences of characters).

*Example:*

```java
String s = new String("pippo");
String t = new String("pippo");
String w = s;
System.out.println("s == w? " + s == w); // TRUE
System.out.println("s == t? " + s == t); // FALSE
System.out.println("s equals t?" + s.equals(t)); // TRUE
```

**5.18 Note on comparisons: lexicographic comparison between strings**

A string `s` precedes a string `t` in lexicographic order if
- `s` is a prefix of `t`, or
- if `c` and `d` are respectively the first character of `s` and `t` in which `s` and `t` differ, then `c` precedes `d` in character order.

*Note:* For the characters that are alphabetical letters, the character order coincides with the alphabetical order. Digits precede letters, and uppercase letters precede lowercase ones.

*Example:*

- `house` precedes `household`
- `Household` precedes `house`
- `composer` precedes `computer`
- `H2O` precedes `HOTEL`

To verify whether a string precedes another one in lexicographic order, we use the `compareTo()` method. Given two strings `s` and `t`, `s.compareTo(t)` returns
- a negative integer, if `s` precedes `t`;
- 0, if `s` is equal to `t`, i.e., if `s.equals(t)` returns true;
- a positive integer, if `s` follows `t`. 

©Diego Calvanese

Lecture Notes for Introduction to Programming

A.A. 2005/06
5.19 Note on comparisons: equality between objects

As for strings, to test whether two objects of a class are equal, we cannot use ==, because it tests the equality of the references to the two objects. Instead, we have to define a suitable method that takes into account the structure of the objects of the class.

Example:

Referring to the class BankAccount developed in Unit 4, we could add to such a class the definition of a predicate equalTo(), which we can use to compare two bank accounts:

```java
public boolean equalTo(BankAccount ba) {
    return this.name.equals(ba.name) &&
    this.surname.equals(ba.surname) &&
    this.balance == ba.balance;
}
```

In other words, two bank accounts are considered equal if they coincide in the name and surname of the owner (this is checked using the equals() method for strings) and if they have the same balance (this is checked using ==, since balance is an instance variable of the primitive type float).

Example of usage:

```java
BankAccount ba1 = new BankAccount("Mario", "Rossi"); // balance is 0
BankAccount ba2 = new BankAccount("Carla", "Verdi"); // balance is 0
BankAccount ba3 = new BankAccount("Carla", "Verdi"); // balance is 0
BankAccount ba4 = ba2;

System.out.println("ba1 and ba2 equal? " + ba1.equalTo(ba2)); // NO
System.out.println("ba2 and ba3 equal? " + ba2.equalTo(ba3)); // YES
System.out.println("ba2 and ba3 coincide? " + ba2 == ba3); // NO
System.out.println("ba2 and ba4 equal? " + ba2.equalTo(ba4)); // YES
System.out.println("ba2 and ba4 coincide? " + ba2 == ba4); // YES
```

Note: Actually, there is a standard technique to define an equality predicate that is based on overriding of the method equals() inherited from Object. This technique will be shown in later courses.

5.20 Note on comparisons: comparison with null

We recall that a variable of type reference to object that does not refer to any object has value null. The comparison with null can be used in the condition of an if-else statement. To compare with null we have to use == and not the equals() method.

Example: showInputDialog returns null if the user presses the “cancel” button:

```java
String input = JOptionPane.showInputDialog("...");
if (input != null) { ... }
```

5.21 Exercise: modification of the class BankAccount

Specification:

Modify the class to handle bank accounts developed in Unit 4 in such a way that the a withdrawal is only done if enough money is available. Add also the following methods:

- a method sameOwner() to verify whether two objects of the class BankAccount have the same owner (i.e., the same name and surname);
- a method transferTo() that, given a bank account and an amount, transfers the amount to the bank account, handling also the case where not enough money is available for the transfer;
- a method transferFrom() that, given a bank account and an amount, transfers the amount from the bank account, handling also the case where not enough money is available for the transfer.

Example of usage:

```java
public class TestBankAccount {
    public static void main(String[] args) {
        public class TestBankAccount {
            ```
BankAccount ba1 = new BankAccount("Mario", "Rossi");
BankAccount ba2 = new BankAccount("Carla", "Verdi");
BankAccount ba3 = new BankAccount("Carla", "Verdi");
System.out.println("Do ba1 and ba2 have the same owner? " + ba1.sameOwner(ba2));
System.out.println("Do ba2 and ba3 have the same owner? " + ba2.sameOwner(ba3));
ba1.deposit(1000);
ba2.deposit(500);
ba3.deposit(750);
System.out.println("Before the transfer ...");
System.out.println(ba1);
System.out.println(ba2);
ba1.transferTo(ba2, 250);
System.out.println("After the transfer...");
System.out.println(ba1);
System.out.println(ba2);
}
}

5.22 A class to handle the time of the day

Specification:
Design a class containing utilities to handle the time of the day, represented through hours, minutes, and, seconds. The class should define the following methods:

- methods to add and subtract two times of the day,
- predicates to compare two times of the day,
- a method to print a time of the day.

Methods of the class TimeOfDay:

// constructor
public TimeOfDay(int h, int m, int s)
// adds a time
public void add(TimeOfDay t)
// subtracts a time
public void subtract(TimeOfDay t)
// predicate to test for precedence
public boolean precedes(TimeOfDay t)
// predicate to test for equality
public boolean equalTo(TimeOfDay t)
// method toString
public String toString()

Example of usage:

public class TestTimeOfDay {
    public static void main(String[] args) {
        TimeOfDay t1 = new TimeOfDay(10,45,15);
        TimeOfDay t2 = new TimeOfDay(15,00,00);
        if (t1.precedes(t2)) {
            t1.add(new TimeOfDay(0,30,00));
        } else {
            t2.subtract(new TimeOfDay(0,30,00));
        }
        System.out.println("TimeOfDay 1 = " + t1);
        System.out.println("TimeOfDay 2 = " + t2);
        System.out.println("TimeOfDay 1 equal to TimeOfDay 2 ? " + t1.equalTo(t2));
    }
}
5.23 Solution of the exercise on the class TimeOfDay

```java
class TimeOfDay {
    // we represent a time of the day by the total number of seconds since midnight
    private int totsec;

    public TimeOfDay(int h, int m, int s) {
        totsec = h*3600 + m*60 + s;
    }

    public void add(TimeOfDay t) {
        totsec += t.totsec;
        if (totsec > 24*3600)
            totsec -= 24*3600;
    }

    public void subtract(TimeOfDay t) {
        totsec -= t.totsec;
        if (totsec < 0)
            totsec += 24*3600;
    }

    public boolean precedes(TimeOfDay t) {
        return totsec < t.totsec;
    }

    public boolean equalTo(TimeOfDay t) {
        return totsec == t.totsec;
    }

    public String toString() {
        int h = totsec / 3600;
        int m = (totsec - h*3600) / 60;
        int s = (totsec - h*3600 - m*60);
        String hh = (h < 10) ? " " + h : Integer.toString(h);
        String mm = (m < 10) ? " " + m : Integer.toString(m);
        String ss = (s < 10) ? " " + s : Integer.toString(s);
        return hh + ":" + mm + ":" + ss;
    }
}
```

5.24 The switch statement

If we have to realize a multiple-choice selection we can use several nested if-else statements. Java has also a specific statement that can be used in specific cases to realize in a simpler way a multiple-choice selection.

**switch statement (version with break)**

**Syntax:**

```java
switch (expression) {
    case label-1: statements-1
        break;
    ...
    case label-n: statements-n
        break;
    default: default-statements
}
```

- **expression** is an expression of an integer type or of type char
- **label-1, ..., label-n** are constant integer (or character) expressions; in other words, the expressions can contain only integer (or character) literals or constants that are initialized with constant expressions; the
values of two different labels cannot coincide

- \textit{statements}-i, …, \textit{statements}-n and \textit{default-statements} are arbitrary sequences of statements
- the \textit{default} part is optional

\textbf{Semantics:}

1. First, \textit{expression} is evaluated.
2. Then, the first \textit{i} is found for which the value of \textit{label-i} is equal to the value of \textit{expression}.
3. If there is such an \textit{i}, then \textit{statements}-i are executed;
   otherwise \textit{default-statements} are executed.
4. The execution continues with the statement immediately following the \textit{switch} statement.

\textbf{Example:}

```java
int i;
...
switch (i) {
    case 0: System.out.println("zero"); break;
    case 1: System.out.println("one"); break;
    case 2: System.out.println("two"); break;
    default: System.out.println("less than zero or greater than two");
}
```

When \textit{i} is equal to 0 (respectively, 1, 2) then "zero" (respectively "one", "two") is printed; when \textit{i} is less than 0 or greater than two, then "less than zero or greater than two" is printed.

\textit{Note:} If we have more than one values of the \textit{expression} for which we want to execute the same statements, we can group together different \textit{cases}:

```java
    case label-1: case label-2: statements
                  break;
```

\subsection*{5.25 Example of switch statement}

Computing the days of the month.

```java
int month, daysOfMonth;
...
switch (month) {
    case 4: case 6: case 9: case 11:
        daysOfMonth = 30;
        break;
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        daysOfMonth = 31;
        break;
    case 2:
        daysOfMonth = 28;
        break;
    default:
        daysOfMonth = 0;
        System.out.println("Month is not valid");
}
System.out.println("Days: " + daysOfMonth);
```

\subsection*{5.26 Observation on the switch statement}

The \textit{expression} used for the selection can be an arbitrary Java expression that returns an \textit{integer} or \textit{character} value (but not a floating point value).
The values specified by the various case labels must be constant expressions, i.e., their value must be known at compile time. In particular, they cannot be expressions that refer to variables.

The following code fragment is wrong:

```java
int a;
...
switch (a) {
    case a<0: System.out.println("negative"); // ERROR! a<0 is not a constant expression
    case 0: System.out.println("zero");
    case a>0: System.out.println("positive"); // ERROR! a>0 is not a constant expression
}
```

It follows that the usefulness of the switch statement is limited.

### 5.27 Omission of break (optional)

Actually, Java does not require that in each case of a switch statement the last statement is a break.

Hence, in general, the form of a switch statement is the following.

#### switch statement (general version)

**Syntax:**

```java
switch (expression) {
    case label-1: statements-1
    ...
    case label-n: statements-n
    default: default-statements
}
```

- *expression* is an expression of an integer type or of type char
- *label-1,..., label-n* are constant integer (or character) expressions; in other words, the expressions can contain only integer (or character) literals or constants that are initialized with constant expressions; the values of two different labels cannot coincide
- *statements-1,..., statements-n* and *default-statements* are arbitrary sequences of statements
- the default part is optional

**Semantics:**

1. First, *expression* is evaluated.
2. Then, the first *i* is found for which the value of *label-* is equal to the value of *expression*.
3. If there is such an *i*, then *statements-*-*i*, *statements-*-*i+1*,... are executed in sequence until the first break statement or the end of the switch statement; otherwise *default-statements* are executed.
4. The execution continues with the statement immediately following the switch statement.

**Example:** More cases of a switch statement executed in sequence.

```java
int sides; // maximum number of sides of a polygon (at most 6)
...
System.out.print("Polygons with at most " + sides + " sides: ");
switch (sides) {
    case 6: System.out.print("hexagon, ");
    case 5: System.out.print("pentagon, ");
    case 4: System.out.print("rectangle, ");
    case 3: System.out.println("triangle");
        break;
    case 2: case 1: System.out.println("none");
        break;
    default: System.out.println;
```

©Diego Calvanese
Lecture Notes for Introduction to Programming
A.A. 2005/06
System.out.println("Please input a value <= 6.");
}

If the value of sides is equal to 5, then the preceding code prints:

pentagon, rectangle, triangle

Note: When we omit the break statements, the order in which the various cases are written becomes important. This can easily cause errors.

Hence, it is a good programming practice to insert break as the last statement in each case.

Exercises

Exercise 5.1. Write a program that reads a real number and prints a message according to the following table:

<table>
<thead>
<tr>
<th>alcohol content $g$</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>$40 &lt; g$</td>
<td>extra strong liquor</td>
</tr>
<tr>
<td>$20 &lt; g \leq 40$</td>
<td>strong liquor</td>
</tr>
<tr>
<td>$15 &lt; g \leq 20$</td>
<td>liquor</td>
</tr>
<tr>
<td>$12 &lt; g \leq 15$</td>
<td>strong vine</td>
</tr>
<tr>
<td>$10.5 &lt; g \leq 12$</td>
<td>normal vine</td>
</tr>
<tr>
<td>$g \leq 10.5$</td>
<td>light vine</td>
</tr>
</tbody>
</table>

Exercise 5.2. Write a program that reads from input the lengths of the three sides of a triangle and determines the type of the triangle according to the following algorithm:

compare each pair of sides and count how many pairs are equal
if (the number of equal pairs is 0)
  it is irregular
else if (the number of equal pairs is 1)
  it is symmetric
else
  it is regular

Exercise 5.3. Write a program that reads from input the lengths of the three sides of a triangle and determines the type of the triangle by using if-else statements with complex conditions.

Exercise 5.4. Realize a Java class to represent triangles, on which the following functionalities are defined:

• creation of a triangle, given the lengths of the three sides;
• return of the length of the longest side, the intermediate side, and the shortest side;
• test whether the three sides can actually be the sides of a triangle; i.e., they respect the triangular inequality, which states that the longest side is shorter than the sum of the other two;
• return of the perimeter of the triangle;
• return of the area of the triangle; notice that, given the lengths $a$, $b$, and $c$ of the three sides of a triangle, the area $A$ can be computed according to $A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$, where $s = (a+b+c)/2$ is the semiperimeter;
• return of a string representing the type of the triangle, which may be either regular, symmetric, or irregular.

Write also a program to test all functionalities of the class representing triangles.

Exercise 5.5. Write a program that reads from input the coefficients $a$, $b$, $c$ of the quadratic equation $a \cdot x^2 + b \cdot x + c = 0$ and computes the zeroes of the equation.

Depending on the sign of the discriminant $b^2 - 4 \cdot a \cdot c$, the program should print the two distinct real solutions, the real double solution, or the two complex solutions.
Exercise 5.6. Write a program that reads from input a line of text containing a YES/NO question (without final question mark) and prints an answer to the question according to the following rules:

1. if the line starts with a vocal, the answer is "MAYBE".
2. if the last letter of the line is "a", "i", or "u", the answer is "YES";
3. if the last letter of the line is "e" or "o", the answer is "NO";
4. if the last letter of the line is a character different from "a", "e", "i", "o", "u", the answer is "DON’T KNOW";

Note: When two rules can be applied, the answer is obtained by concatenating the answers for the two rules.

Exercise 5.7. Realize a Java class to represent dates, on which the following functionalities are defined:

- creation of a date, given day, month, and year;
- return of day, month, and year;
- test whether two dates are equal;
- test whether a date precedes another date;
- test whether the year of a date is a leap year;
- compute the date of the next day.

Write also a program to test all functionalities of the class representing dates.