CHAPTER: 03

Selection

Objectives:

By the end of this chapter you should be able to:

- *explain the difference between **sequence** and **selection**;*
- *use an* `if` *statement to make a single choice in a program;*
- *use an* `if`…`else` *statement to make a choice between two options in a program;*
- *use nested* `if`…`else` *statements to make multiple choices in a program;*
- *use a* `switch` *statement to make multiple choices in a program.*

3.1 Introduction

One of the most rewarding aspects of writing and running a program is knowing that *you* are the one who has control over the computer. But looking back at the programs you have already written, just how much control do you actually have? Certainly, it was you who decided upon which instructions to include in your programs but *the order in which these instructions were executed* was *not* under your control. These instructions were always executed in **sequence**, that is one after the other, from beginning to the end of the `main` method. You will soon find that there are numerous instances when this order of execution is too restrictive and you will want to have much more control over the order in which instructions are executed.

3.2 Making choices

Very often you will want your programs to make *choices* among different courses of action. For example, a program processing requests for airline tickets could have the following choices to make:

- display the price of the seats requested;
- display a list of alternative flights;
- display a message saying that no flights are available to that destination.

A program that can make choices can behave *differently* each time it is run, whereas programs in which instructions are just executed in sequence behave the *same way* each time they are run.

As we have already mentioned, unless you indicate otherwise, program instructions are always executed in sequence. **Selection**, however, is a method of program control in which a choice can be made about which instructions to execute.

For example, consider the following program, which welcomes customers queuing up for a roller-coaster ride:

Program 3.1

```
import java.util.*;

public class RollerCoaster
{
  public static void main(String[] args)
  {
    // declare variables
    int age;
    Scanner keyboard = new Scanner (System.in);

    // four instructions to process information
    System.out.println("How old are you?");
    age = keyboard.nextInt();
    System.out.println("Hello Junior!");
    System.out.println("Enjoy your ride");
  }
}
```

As you can see, following the variable declarations, there are *four* remaining instructions in this program. Remember that at the moment these instructions will be executed in sequence, from top to bottom. Consider the following interaction with this program:

```
How old are you?
10
Hello Junior!
Enjoy your ride
```

This looks fine but the message "Hello Junior!" is only meant for children. Now let's assume that someone older comes along and interacts with this program as follows:

```
How old are you?
45
Hello Junior!
Enjoy your ride
```

The message "Hello Junior!", while flattering, might not be appropriate in this case! In other words, it is not always appropriate to execute the following instruction:

```
System.out.println("Hello Junior!");
```

What is required is a way of deciding (while the program is running) whether or not to execute this instruction. In effect, this instruction needs to be *guarded* so that it is only executed *when appropriate*. Assuming we define a child as someone under 13 years of age, we can represent this in pseudocode as follows:

```
DISPLAY "How old are you?"
ENTER age
IF age is under 13
BEGIN
    DISPLAY "Hello Junior!"
END
DISPLAY "Enjoy your ride"
```

In the above, we have emboldened the lines that have been added to guard the "Hello Junior!" instruction. The emboldened lines that have been added are not to be read as *additiona*l instructions; they are simply a means to *control the flow* of the *existing* instructions. The emboldened lines say, in effect, that the instruction to display the message "Hello Junior!" should only be executed if the age entered is under 13.

This, then, is an example of the form of control known as selection. Let's now look at how to code this selection in Java.

3.3 The 'if' statement

The particular form of selection discussed above is implemented by making use of Java's **`if`** statement. The general form of an **`if`** statement is given as follows:

```
if ( /* a test goes here */)
{
   // instruction(s) to be guarded go here
}
```

As you can see, the instructions to be guarded are placed inside the braces of the **`if`** statement. A **test** is associated with the **`if`** statement. A test is any expression that produces a result of **`true`** or **`false`**. For example **`x>100`** is a test as it is an expression that either gives an answer of **`true`** or **`false`** (depending upon the value of `x`). We call an expression that returns a value of **`true`** or **`false`** a **boolean expression**, as **`true`** and **`false`** are **`boolean`** values. Examples of tests in everyday language are:

- this password is valid;
- there is an empty seat on the plane;
- the temperature in the laboratory is too high.

The test must follow the **`if`** keyword and be placed in round brackets. When the test gives a result of **`true`** the instructions inside the braces of the **`if`** statement are executed. The program then continues by executing the instructions after the braces of the **`if`** statement as normal. If, however, the **`if`** test gives a result of **`false`** the instructions inside the **`if`** braces are *skipped* and not executed.

We can rewrite program 3.1 by including an appropriate **`if`** statement around the "Hello Junior!" message with the test (age < 13) as follows:

Program 3.2

```
import java.util.*;

// This program is an example of the use of selection in a Java program

public class RollerCoaster2
{
  public static void main(String[] args)
```

```
    {
      int age;
      Scanner keyboard = new Scanner (System.in);
      System.out.println("How old are you?");
      age = keyboard.nextInt();
      if (age < 13) // test controls if the next instruction is executed
      {
          System.out.println("Hello Junior!");
      }
      System.out.println("Enjoy your ride");
    }
  }
```

Now the message "Hello Junior!" will only be executed if the test (`age<13`) is **`true`**, otherwise it will be skipped (see figure 3.1).

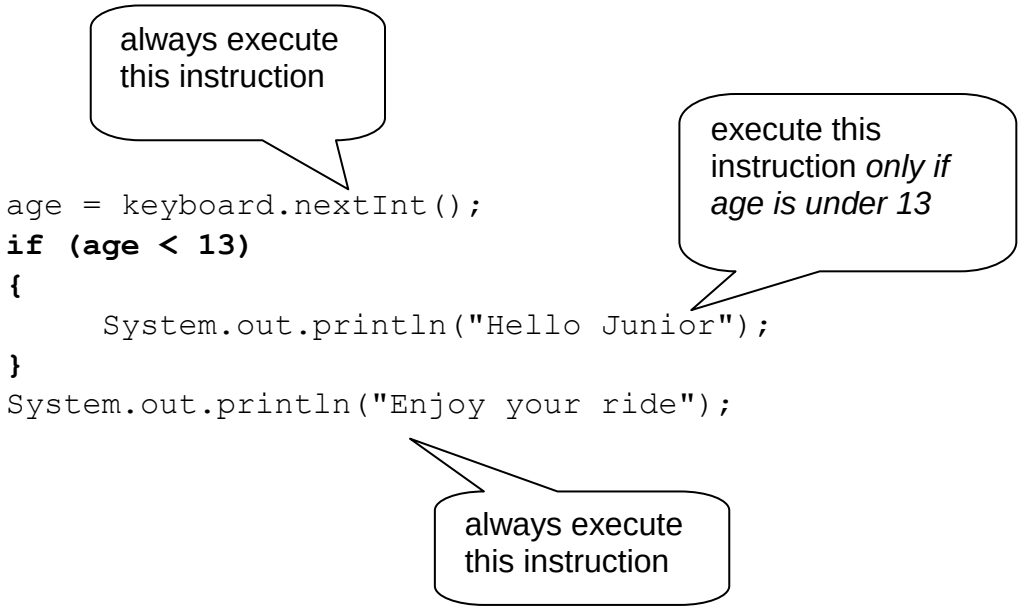

Fig 3.1. The *if* statement allows a choice to be made in programs

Let's assume we run program 3.2 with the same values entered as when running program 3.1 before. First, the child approaches the ride:

```
How old are you?
10
Hello Junior!
Enjoy your ride
```

In this case, the **`if`** statement has allowed the "Hello Junior!" message to be displayed as the age entered is less than 13. Now the adult approaches the ride:

```
How old are you?
45
Enjoy your ride
```

In this case the **if** statement has not allowed the given instruction to be executed as the associated test was not true. The message is skipped and the program continues with the following instruction to display "Enjoy your ride".

In this program there was only a *single* instruction inside the **if** statement.

```
age = keyboard.nextInt();
if (age < 13)
{
    System.out.println("Hello Junior!"); // single instruction inside 'if'
}
System.out.println("Enjoy your ride");
```

When there is only a single instruction associated with an **if** statement, the braces can be omitted around this instruction, if so desired, as follows:

```
age = keyboard.nextInt();
if (age < 13)
System.out.println("Hello Junior!"); // braces can be omitted around this line
System.out.println("Enjoy your ride");
```

The compiler will always assume that the first line following the **if** test is the instruction being guarded. For clarity, however, we will always use braces around instructions.

3.3.1 Comparison operators

In the example above, the "less than" operator (<) was used to check the value of the `age` variable. This operator is often referred to as a **comparison operator** as it is used to compare two values. Table 3.1 shows all the Java comparison operator symbols.

Table 3.1. The comparison operators of Java

Operator	Meaning
= =	equal to
! =	not equal to
<	less than
>	greater than
> =	greater than or equal to
< =	less than or equal to

Since comparison operators give a **boolean** result of **true** or **false** they are often used in tests such as those we have been discussing. For example, consider a `temperature` variable

being used to record the temperature for a particular day of the week. Assume that a temperature of 18 degrees or above is considered is considered to be a hot day. We could use the "greater than or equal to" operator (>=) to check for this as follows:

```
if (temperature >= 18) // test to check for hot temperature
{
     // this line executed only when the test is true
     System.out.println("Today is a hot day!");
}
```

You can see from table 3.1 that a double equals (= =) is used to check for equality in Java and not the single equals (=), which, as you know, is used for assignment. To use the single equals is a very common error! For example, to check whether an angle is a right angle the following test should be used:

```
if (angle == 90)// note the use of the double equals
{
   System.out.println("This IS a right angle");
}
```

To check if something is *not equal* to a particular value we use the exclamation mark followed by an equals sign (! =). So to test if an angle is *not* a right angle we can have the following:

```
if (angle != 90)
{
   System.out.println("This is NOT a right angle");
}
```

3.3.2 Multiple instructions within an 'if' statement

You have seen how an **`if`** statement guarding a single instruction may or may not be implemented with braces around the instruction. When *more than one* instruction is to be guarded by an **`if`** statement, however, the instructions *must* be placed in braces. As an example, consider once again program 2.3 that calculates the cost of a product.

Reminder of Program 2.3

```
import java.util.*;
public class FindCost3
{
  public static void main(String [] args)
  {
    Scanner keyboard = new Scanner(System.in);
    double price, tax;
    System.out.println("*** Product Price Check ***");
    System.out.print("Enter initial price: ");
```

```
    price = keyboard.nextDouble();
    System.out.print("Enter tax rate: ");
    tax = keyboard.nextDouble();
    price = price * (1 + tax/100);
    System.out.println("Cost after tax = " + price);
  }
}
```

Now assume that a special promotion is in place for those products with an initial price over 100. For such products the company pays half the tax. Program 3.3 makes use of an **`if`** statement to apply this promotion, as well as informing the user that a tax discount has been applied. Take a look at it and then we will discuss it.

Program 3.3

```
import java.util.*;
public class FindCostWithDiscount
{
  public static void main(String[] args )
  {
    double price, tax;
    Scanner keyboard = new Scanner(System.in);
    System.out.println("*** Product Price Check ***");
    System.out.print("Enter initial price: ");
    price = keyboard.nextDouble();
    System.out.print("Enter tax rate: ");
    tax = keyboard.nextDouble();
    // the following 'if' statement allows a selection to take place
    if (price > 100) // test the price to see if a discount applies
    {
      // these two instructions executed only when test is true
      System.out.println("Special Promotion: We pay half your tax!");
      tax = tax * 0.5;
    }
    // the remaining instructions are always executed
    price = price * (1 + tax/100);
    System.out.println("Cost after tax = " + price);
  }
}
```

Now, the user is still always prompted to enter the initial price and tax as before:

```
System.out.print("Enter initial price: ");
price = keyboard.nextDouble();
System.out.print("Enter tax rate: ");
tax = keyboard.nextDouble();
```

The next two instructions are then placed inside an **`if`** statement. This means they may not always be executed:

```
if (price > 100)
{
  System.out.println("Special Promotion: We pay half your tax!");
  tax = tax * 0.5;
}
```

Notice that if the braces were omitted in this case, only the *first* instruction would be taken to be inside the **if** statement – the second statement would not be guarded and so would *always* be executed!

With braces around both instructions, they will be executed only when the test (`price > 100`) returns a **boolean** result of **true**. So, for example, if the user had entered a price of 150 the tax discount would be applied; but if the user entered a price of 50 these instructions would not be executed and a tax discount would not be applied.

Regardless of whether or not the test was **true** and the instructions in the **if** statement executed, the program always continues with the remaining instructions:

```
price = price * (1 + tax/100);
System.out.println("Cost after tax = " + price);
```

Here is a sample program run when the test returns a result of **false** and the discount is not applied:

```
*** Product Price Check ***
Enter initial price: 20
Enter tax rate: 10
Cost after tax = 22.0
```

In this case the program appears to behave in exactly the same way as the original program (program 2.3). Here, however, is a program run when the test returns a result of **true** and a discount does apply:

```
*** Product Price Check ***
Enter initial price: 1000
Enter tax rate: 10
Special Promotion: We pay half your tax!
Cost after tax = 1050.0
```

3.4 The 'if…else' statement

Using the **if** statement in the way that we have done so far has allowed us to build the idea of a choice into our programs. In fact, the **if** statement made one of two choices before continuing with the remaining instructions in the program:

- execute the conditional instructions, or
- do not execute the conditional instructions.

The second option amounts to "do nothing". Rather than do nothing if the test is **false**, an extended version of an **if** statement exists in Java to state an *alternative* course of action. This extended form of selection is the **if**…**else** statement. As the name implies, the instructions to be executed if the test evaluates to **false** are preceded by the Java keyword **else** as follows:

```
if ( /* test goes here */ )
{
    // instruction(s) if test is true go here
}
else
{
    // instruction(s) if test is false go here
}
```

This is often referred to as a **double-branched** selection as there are two alternative groups of instructions, whereas a single **`if`** statement is often referred to as a **single-branched** selection. Program 3.4 illustrates the use of a double-branched selection.

Program 3.4

```
import java.util.*;
public class DisplayResult
{
  public static void main(String[] args)
  {
       int mark;
       Scanner keyboard = new Scanner(System.in);
       System.out.println("What exam mark did you get? ");
       mark = keyboard.nextInt();
       if (mark >= 40)
       {
          // executed when test is true
          System.out.println("Congratulations, you passed");
       }
       else
       {
          // executed when test is false
          System.out.println("I'm sorry, but you failed");
       }
       System.out.println("Good luck with your other exams");
   }
}
```

Program 3.4 checks a student's exam mark and tells the student whether or not he or she has passed (gained a mark greater than or equal to 40), before displaying a good luck message on the screen. Let's examine this program a bit more closely.

Prior to the **`if`**…**`else`** statement the following lines are executed in sequence:

```
int mark;
Scanner keyboard = new Scanner(System.in);
System.out.println("What exam mark did you get? ");
mark = keyboard.nextInt();
```

Then the following condition is tested as part of the **`if`**…**`else`** statement:

```
(mark >= 40)
```

When this test is **`true`** the following line is executed:

```
System.out.print("Congratulations, you passed");
```

When the test is **`false`**, however, the following line is executed *instead*:

```
System.out.println("I'm sorry, but you failed");
```

Finally, whichever path was chosen the program continues by executing the last line:

```
System.out.println("Good luck with your other exams");
```

The **`if`**…**`else`** form of control has allowed us to choose from *two* alternative courses of action. Here is a sample program run:

```
What exam mark did you get?
52
Congratulations, you passed
Good luck with your other exams
```

Here is another sample run where a different course of action is chosen.

```
What exam mark did you get?
35
I'm sorry, but you failed
Good luck with your other exams
```

3.5 Logical operators

As we've already pointed out, the test in an **`if`** statement is an expression that produces a **`boolean`** result of **`true`** or **`false`**. Often it is necessary to join two or more tests together to create a single more complicated test.

As an example, consider a program that checks the temperature in a laboratory. Assume that, for the experiments in the laboratory to be successful, the temperature must remain between 5 and 12 degrees Celsius. An **`if`** statement might be required as follows:

```
if (/* test to check if temperature is safe */)
{
    System.out.println ("TEMPERATURE IS SAFE!");
}
else
{
    System.out.println("UNSAFE: RAISE ALARM!!");
}
```

The test should check if the temperature is safe. This involves combining two tests together:

1 check that the temperature is greater than or equal to 5 (`temperature >= 5`)
2 check that the temperature is less than or equal to 12 (`temperature <= 12`)

Both of these tests need to evaluate to **`true`** in order for the temperature to be safe. When we require two tests to be `true` we use the following symbol to join the two tests:

```
&&
```

This symbol is read as "AND". So the correct test is:

```
if (temperature >= 5 && temperature <= 12)
```

Now, if the temperature were below 5 the first test (`temperature >= 5`) would evaluate to **`false`** giving a final result of **`false`**; the **`if`** statement would be skipped and the **`else`** statement would be executed:

```
UNSAFE: RAISE ALARM!!
```

If the temperature were greater than 12 the second part of the test (`temperature <= 12`) would evaluate to **`false`** also giving an overall result of **`false`** and again the **`if`** statement would be skipped and the **`else`** statement would be executed.

However, when the temperature is between 5 and 12 *both* tests would evaluate to **`true`** and the final result would be **`true`** as required; the **`if`** statement would then be executed instead:

```
TEMPERATURE IS SAFE!
```

Notice that the two tests must be *completely* specified as each needs to return a **`boolean`** value of **`true`** or **`false`**. It would be wrong to try something like the following:

```
// wrong! second test does not mention 'temperature'!
if (temperature >= 5 && <= 12)
```

This is wrong as the second test (`<= 12`) is not a legal **`boolean`** expression. Symbols that join tests together to form longer tests are known as **logical operators**. Table 3.2 lists the Java counterparts to the three common logical operators.

Table 3.2 The logical operators of Java	
Logical operator	**Java counterpart**
AND	&&
OR	||
NOT	!

Both the AND and OR operators join two tests together to give a final result. While the AND operator requires both tests to be **`true`** to give a result of **`true`**, the OR operator requires only that *at least one* of the tests be **`true`** to give a result of **`true`**. The NOT operator flips a value of **`true`** to **`false`** and a value of **`false`** to **`true`**. Table 3.3 gives some examples of the use of these logical operators:

Table 3.3 Logical operators: some examples		
Expression	**Result**	**Explanation**
`10>5 && 10>7`	**`true`**	Both tests are true
`10>5 && 10>20`	**`false`**	The second test is false
`10>15 && 10>20`	**`false`**	Both tests are false
`10>5 || 10>7`	**`true`**	At least one test is true (in this case both tests are true)
`10>5 || 10>20`	**`true`**	At least one test is true (in this case just one test is true)
`10>15 || 10>20`	**`false`**	Both tests are false
`! (10 > 5)`	**`false`**	Original test is true
`! (10 > 15)`	**`true`**	Original test is false

As an example of the use of the NOT operator (!), let us return to the temperature example. We said that we were going to assume that a temperature of greater than 18 degrees was going to be considered a hot day. To check that the day is not a hot day we could use the NOT operator as follows:

```
if (!(temperature > 18) ) // test to check if temperature is not hot
{
    System.out.println("Today is not a hot day!");
}
```

Of course, if a temperature is not greater than 18 degrees then it must be less than or equal to 18 degrees. So, another way to check the test above would be as follows:

```
if (temperature <= 18) // this test also checks if temperature is not hot
{
    System.out.println("Today is not a hot day!");
}
```

3.6 Nested 'if…else' statements

Instructions within **`if`** and **`if`**…**`else`** statements can themselves be *any* legal Java commands. In particular they could contain other **`if`** or **`if`**…**`else`** statements. This form of control is referred to as **nesting**. Nesting allows multiple choices to be processed.

As an example, consider program 3.5 below, which asks a student to enter his or her tutorial group (A, B, or C) and then displays on the screen the time of the software lab.

Program 3.5

```
import java.util.*;
public class Timetable
{
  public static void main(String[] args)
  {
     char group; // to store the tutorial group
     Scanner keyboard = new Scanner(System.in);
     System.out.println("***Lab Times***"); // display header
     System.out.println("Enter your group (A,B,C)");
     group = keyboard.next().charAt(0);
     // check tutorial group and display appropriate time
     if (group == 'A')
     {
       System.out.print("10.00 a.m"); // lab time for group A
     }
     else
     {
       if (group == 'B')
       {
         System.out.print("1.00 p.m"); // lab time for group B
       }
       else
       {
         if (group == 'C')
         {
           System.out.print("11.00 a.m"); // lab time for group C
         }
         else
         {
           System.out.print("No such group"); // invalid group
         }
       }
     }
  }
}
```

As you can see, nesting can result in code with many braces that can become difficult to read, even with our careful use if tabs. Such code can be made easier to read by not including the braces associated with all the **`else`** branches.

```
if (group == 'A')
{
     System.out.print("10.00 a.m");
}
else if (group == 'B')
```

```
{
    System.out.print("1.00 p.m");
}
else if(group == 'C')
{
    System.out.print("11.00 a.m");
}
else
{
    System.out.print("No such group");
}
```

This program is a little bit different from the ones before because it includes some basic **error checking**. That is, it does not *assume* that the user of this program will always type the *expected* values. If the wrong group (not A, B or C) is entered, an error message is displayed saying "*No such group*".

```
// valid groups checked above
else // if this 'else' is reached, group entered must be invalid
{
    System.out.print("No such group"); // error message
}
```

Error checking like this is a good habit to get into.

This use of nested selections is okay up to a point, but when the number of options becomes large the program can again look very untidy. Fortunately, this type of selection can also be implemented in Java with another form of control: a **`switch`** statement.

3.7 The 'switch' statement

Program 3.6 behaves in exactly the same way as program 3.5 but using a **`switch`** instead of a series of nested **`if`**…**`else`** statements allows a neater implementation. Take a look at it and then we'll discuss it.

Program 3.6

```
import java.util.*;
public class TimetableWithSwitch
{
  public static void main(String[] args)
  {
    char group;
    Scanner keyboard = new Scanner(System.in);
    System.out.println("***Lab Times***");
    System.out.println("Enter your group (A,B,C)");
    group = keyboard.next().charAt(0);
    switch(group) // beginning of switch
    {
      case 'A': System.out.print("10.00 a.m ");
                break;
      case 'B': System.out.print("1.00 p.m ");
                break;
      case 'C': System.out.print("11.00 a.m ");
                break;
      default: System.out.print("No such group");
```

```
    } // end of switch
  }
}
```

As you can see, this looks a lot neater. The **switch** statement works in exactly the same way as a set of nested **if** statements, but is more compact and readable. A **switch** statement may be used when

- only one variable is being checked in each condition (in this case every condition involves checking the variable `group`);
- the check involves specific values of that variable (e.g. 'A', 'B') and not ranges (for example >=40).

As can be seen from the example above, the keyword **case** is used to precede a possible value of the variable that is being checked. There may be many **case** statements in a single **switch** statement. The general form of a **switch** statement in Java is given as follows:

```
switch(someVariable)
{
    case value1: // instructions(s) to be executed
                 break;
    case value2: // instructions(s) to be executed
                 break;
    // more values to be tested can be added
    default: // instruction(s) for default case
}
```

where

- `someVariable` is the name of the variable being tested. This variable is usually of type **int** or **char** but may also be of type **long**, **byte**, or **short**.
- `value1`, `value2,` etc. are the possible values of that variable.
- **break** is a command that forces the program to skip the rest of the **switch** statement.
- **default** is an optional (last) case that can be thought of as an "otherwise" statement. It allows you to code instructions that deal with the possibility of none of the cases above being **true**.

The **break** statement is important because it means that once a matching case is found, the program can skip the rest of the cases below. If it is not added, not only will the instructions associated with the matching case be executed, but also all the instructions associated with all the cases below it. Notice that the last set of instructions does not need a **break** statement as there are no other cases to skip.

3.7.1 Grouping `case` statements

There will be instances when a particular group of instructions is associated with more than one **case** option. As an example, consider program 3.7 again. Let's assume that both groups A and C have a lab at 10.00 a.m. The following **switch** statement would process this without grouping case 'A' and 'C' together:

```
// groups A and C have labs at the same time
switch(group)
{
     case 'A': System.out.print("10.00 a.m ");
               break;
     case 'B': System.out.print("1.00 p.m ");
               break;
     case 'C': System.out.print("10.00 a.m ");
               break;
     default: System.out.print("No such group");
}
```

While this will work, both **`case`** 'A' and **`case`** 'C' have the same instruction associated with them:

```
System.out.print("10.00 a.m ");
```

Rather than repeating this instruction, the two **`case`** statements can be combined into one as follows:

```
// groups A and C have been processed together
switch(group)
{
     case 'A': case 'C': System.out.print("10.00 a.m ");
                         break;
     case 'B': System.out.print("1.00 p.m ");
               break;
     default: System.out.print("No such group");
}
```

In the example above a time of 10.00 a.m will be displayed when the group is either 'A' or 'C'. The example above combined two **`case`** statements, but there is no limit to how many such statements can be combined.

3.7.2 Removing `break` statements

In the examples above we have always used a **`break`** statement to avoid executing the code associated with more than one **`case`** statement. There may be situations where it is *not* appropriate to use a **`break`** statement and we *do* wish to execute the code associated with more than one case statement.

For example, let us assume that spies working for a secret agency are allocated different levels of security clearance, the lowest being 1 and the highest being 3. A spy with the highest clearance level of 3 can access all the secrets, whereas a spy with a clearance of level of 1 can see only secrets that have the lowest level of security. An administrator needs to be able to view the collection of secrets that a spy with a particular clearance level can see. We can implement this scenario by way of a **`switch`** statement in program 3.7 below. Take a look at it and then we will discuss it.

Program 3.7

```
import java.util.*;
public class SecretAgents
{
  public static void main(String[] args)
  {
    int security;
    Scanner keyboard = new Scanner(System.in);
    System.out.println("***Secret Agents***");
    System.out.println("Enter security level (1,2,3)");
    security = keyboard.nextInt();
    switch(security) // check level of security
    {
      case 3: System.out.println("The code to access the safe is 007."); // level 3 security
      case 2: System.out.println("Jim Kitt is really a double agent."); // level 2 security
      case 1: System.out.println("Martinis in the hotel bar may be poisoned."); // level 1 security
              break; // necessary to avoid error message below
      default: System.out.println("No such security level.");
    }
  }
}
```

You can see that there is just a single **break** statement at the end of **case** 1.

```
case 3: System.out.println("The code to access the safe is 007.");
case 2: System.out.println("Jim Kitt is really a double agent.");
case 1: System.out.println("Martinis in the hotel bar may be poisoned.");
        break; // the only break statement
```

If the user entered a security level of 3 for example, the `println` instruction associated with this case would be executed:

```
case 3: System.out.println("The code to access the safe is 007.");
```

However, as there is no **break** statement at the end of this instruction, the instruction associated with the **case** below is then also executed:

```
System.out.println("Jim Kitt is really a double agent.");
```

We have still not reached a **break** statement so the instruction associated with the next **case** statement is then executed:

```
System.out.println("Martinis in the hotel bar may be poisoned.");
break; // the only break statement
```

Here we do reach a **break** statement so the **switch** terminates. Here is a sample test run:

```
***Secret Agents***
Enter security level (1,2,3)
3
```

```
The code to access the safe is 007.
Jim Kitt is really a double agent.
Martinis in the hotel bar may be poisoned.
```

Because the security level entered is 3 *all* secrets can be revealed. Here is another sample test run when security level 2 is entered:

```
***Secret Agents***
Enter security level (1,2,3)
2
Jim Kitt is really a double agent.
Martinis in the hotel bar may be poisoned.
```

Because the security level is 2 the first secret is not revealed.

The last **`break`** statement is necessary in program 3.7 as we wish to avoid the final error message if a valid security level (1, 2 or 3) is entered. The error message is only displayed if an invalid security level is entered:

```
***Secret Agents***
Enter security level (1,2,3)
8
No such security level.
```

Self-test questions

1 Explain the difference between *sequence* and *selection*.

2 When would it be appropriate to use

- an **`if`** statement?
- an **`if`**…**`else`** statement?
- a **`switch`** statement?

3 Consider the following Java program, which is intended to display the cost of a cinema ticket. Part of the code has been replaced by a comment:

```
import java.util.*;
public class SelectionQ3
{
  public static void main(String[] args)
  {
    double price = 10.00;
    int age;
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter your age: ");
    age = keyboard.nextInt();
      // code to reduce ticket price for children goes here
   System.out.println("Ticket price = " + price);
  }
}
```

Replace the comment so that children under the age of 14 get half price tickets.

4 Consider the following program:

```
import java.util.*;
public class SelectionQ4
{
  public static void main(String[] args)
  {
    int x;
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a number: ");
    x = keyboard.nextInt();
    if (x > 10)
    {
      System.out.println("Green");
      System.out.println("Blue");
    }
    System.out.println("Red");
  }
}
```

What would be the output from this program if

a) the user entered 10 when prompted?

b) the user entered 20 when prompted?

c) the braces used in the **`if`** statement are removed, and the user enters 10 when prompted?

d) the braces used in the **`if`** statement are removed, and the user enters 20 when prompted?

5 Consider the following program:

```
import java.util.*;
public class SelectionQ5
{
  public static void main(String[] args)
  {
    int x;
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a number: ");
    x = keyboard.nextInt();
    if (x > 10)
    {
      System.out.println("Green");
    }
    else
    {
      System.out.println("Blue");
    }
    System.out.println("Red");
  }
}
```

What would be the output from this program if

a) the user entered 10 when prompted?

b) the user entered 20 when prompted?

6 Consider the following program:

```
import java.util.*;
public class SelectionQ6
{
  public static void main(String[] args)
  {
    int x;
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a number: ");
    x = keyboard.nextInt();
    switch (x)
    {
      case 1: case 2: System.out.println("Green"); break;
      case 3: case 4: case 5: System.out.println("Blue"); break;
      default: System.out.println("numbers 1-5 only");
    }
    System.out.println("Red");
  }
}
```

What would be the output from this program if

a) the user entered 1 when prompted?
b) the user entered 2 when prompted?
c) the user entered 3 when prompted?
d) the user entered 10 when prompted?
d) the **`break`** statements were removed from the **`switch`** statement and the user entered 3 when prompted?
e) the **`default`** were removed from the **`switch`** statement and the user entered 10 when prompted?

Programming exercises

1 Design and implement a program that asks the user to enter two numbers and then displays the message “NUMBERS ARE EQUAL”, if the two numbers are equal and “NUMBERS ARE NOT EQUAL”, if they are not equal.

Hint : Don't forget to use the double equals (==) to test for equality.

2 Adapt the program developed in the question above so that as well as checking if the two numbers are equal, the program will also display “FIRST NUMBER BIGGER” if the first number is bigger than the second number and display “SECOND NUMBER BIGGER” if the second number is bigger than the first.

3 Design and implement a program that asks the user to enter two numbers and then guess at the sum of those two numbers. If the user guesses correctly a congratulatory message is displayed, otherwise a commiseration message is displayed along with the correct answer.

4 Implement program 3.4 which processed an exam mark and then adapt the program so that marks of 70 or above are awarded a distinction rather than a pass.

5 Write a program to take an order for a computer system. The basic system costs 375.99. The user then has to choose from a 38 cm screen (costing 75.99) or a 43 cm screen (costing 99.99).The following extras are optional.

Item	**Price**
DVD/CD Writer	65.99
Printer	125.00

The program should allow the user to select from these extras and then display the final cost of the order.

6 Consider a bank that offers four different types of account ('A', 'B', 'C' and 'X'). The following table illustrates the annual rate of interest offered for each type of account.

Account	**Annual rate of interest**
A	1.5%
B	2%
C	1.5%
X	5%

Design and implement a program that allows the user to enter an amount of money and a type of bank account, before displaying the amount of money that can be earned in one year as interest on that money for the given type of bank account. You should use the **`switch`** statement when implementing this program.

Hint: be careful to consider the case of the letters representing the bank accounts. You might want to restrict this to, say, just upper case. Or you could enhance your program by allowing the user to enter either lower case or upper case letters.

7 Consider the bank accounts discussed in exercise 6 again. Now assume that each type of bank account is associated with a minimum balance as given in the table below:

Account	**Minimum balance**
A	250
B	1000
C	250
X	5000

Adapt the **`switch`** statement of the program in exercise 6 above so that the interest is applied only if the amount of money entered satisfies the minimum balance requirement for the given account. If the amount of money is below the minimum balance for the given account an error message should be displayed.