Chapter 3 Object interaction

Main concepts discussed in this chapter:

- abstraction
- modularization
- object diagrams
- object creation
- method calls
- debuggers

Java constructs discussed in this chapter:

class types, logic operators (&&, ||), string concatenation, modulo operator (%), object construction (new), method calls (dot notation), *this*

In the previous chapters, we have examined what objects are, and how they are implemented. In particular, we discussed fields, constructors, and methods when we looked at class definitions.

We will now go one step further. To construct interesting applications, it is not enough to build individual working objects. Instead, objects must be combined so that they cooperate to perform a common task. In this chapter we will build a small application from three objects, and arrange for methods to call other methods to achieve their goal.

3.1 The clock example

The project we will use to discuss interaction of objects is a display for a digital clock. The display shows hours and minutes, separated by a colon (Figure 23). For this exercise, we will first build a clock with a European-style 24-hour display. Thus, the display shows the time from 00:00 (midnight) to 23:59 (one minute before midnight). It turns out on closer inspection that building a 12 hour clock is slightly more difficult – we will leave this to the end of this chapter.

Figure 23: A display of a digital clock

3.2 Abstraction and modularization

A first idea might be to implement the whole clock display in a single class. That is, after all, what we have seen so far: how to build classes to do a job.

However, here we will approach this problem slightly differently. We will see whether we can identify sub-components in the problem that we could turn into separate classes. The reason is *complexity*. As we progress in this book, the examples we use and the programs we build will get more and more complex. Trivial tasks like the ticket machine can be solved as a single problem. You can look at the complete task and devise a solution using a single class. For more complex problems, that is too simplistic. As a problem grows larger, it becomes increasingly difficult to keep track of all details at the same time.

The solution we use to deal with the complexity problem is *abstraction*. [concept box: abstraction] We divide the problem into sub-problems, then again into sub-sub-problems, and so on, until the individual problems are small enough to be easy to deal with. Once we solve one of the sub-problems, we do not think about the details of that part anymore, but treat the solution as a single building block for our next problem. This technique is sometimes referred to as *divide-and-conquer*.

Let us discuss this with an example. Imagine engineers in a car company designing a new car. They may think about the parts of the car, such as the shape of the outer body, the size and location of the engine, the number and size of the seats in the passenger area, the exact spacing of the wheels, and so on. Another engineer, on the other hand, whose job it is to design the engine (well, that's a whole team of engineers in reality, but we can simplify a bit here for the sake of the example) thinks of the many parts of an engine: the cylinders, the injection mechanism, the carburetor, the electronics, etc. She will think of the engine not as a single entity, but as a complex work of many parts. One of these parts may be a spark plug.

Then there is an engineer (maybe in a different company) who designs the spark plugs. He will think of the spark plug as a complex artifact of many parts. He might have done complex studies to determine exactly what kind of metal to use for the contacts, or what kind of material and production process to use for the insulation.

The same is true for many other parts. A designer at the highest level will regard a wheel as a single part. Another engineer much further down the chain may spend her days thinking about the chemical composition to produce the right materials to make the tires. For the tire engineer, the tire is a complex thing. The car company will just buy the tire from the tire company, and then view it as a single entity. This is abstraction.

The engineer in the car company *abstracts from* the details of the tire manufacture to be able to concentrate on the details of the construction of, say, the wheel. The designer designing the body shape of the car abstracts from the technical details of the wheels and the engine to concentrate on the design of the body (he will just be interested in the size of the engine and the wheels).

The same is true for every other component. While someone might be concerned with designing the interior passenger space, someone else may work on developing the fabric that will eventually be used to cover the seats.

The point is: if viewed in enough detail, a car consists of so many parts that it is impossible for a single person to know every detail about every part at the same time. If that were necessary, no car could ever be built.

[concept box: modularization] The reason cars are built successfully is that the engineers use *modularization* and abstraction. They divide the car into independent modules (wheel, engine, gear box, seat, steering wheel, etc) and get separate people to work on separate modules independently. When a module is built, they use abstraction. They view that module as a single component that is used to build more complex components.

Modularization and abstraction, thus, complement each other. Modularization is the process of dividing large things (problems) into smaller parts, while abstraction is the ability to ignore details to focus on the bigger picture.

3.3 Abstraction in software

The same principles of modularization and abstraction discussed in the previous section are used in software development. To help us maintain an overview in complex programs, we try to identify sub-components that we can program as independent entities. Then we try to use those sub-components as if they were simple parts without being concerned about their inner complexities.

In object-oriented programming, these components and sub-components are objects. If we were trying to construct a car in software, using an object-oriented language, we would try to do what the car engineers do. Instead of implementing the car in a single, monolithic object, we would first construct separate objects for an engine, gear box, wheel, seat, and so on, and then assemble the car object from those smaller objects.

Identifying what kinds of objects (and with these: classes) you should have in a software system for any given problem is not always easy, and we will have a lot more to say about that later in this book. For now, we will start with a relatively simple example. Now, back to our digital clock.

3.4 Modularization in the clock example

Let us have a closer look at the clock-display example. Using the abstraction concepts we have just described we want to try to find the best way to view this example so that we can write some classes to implement it. One way to look at it is to consider it as consisting of a single display with four-digits (two digits for the hours, two for the minutes). If we now abstract away from that very low-level view, we can see that it could also be viewed as two separate two-digit displays (one pair for the hours and one pair for the minutes). One pair starts at zero, increases by one each hour and rolls back to zero after reaching its limit of twenty-three. The other rolls back to zero after reaching its limit of fifty-nine. The similarity in behavior of these two displays might then lead us to abstract away even further from viewing the hours display and minutes display distinctly. Instead we might think of them as being objects that can display values from 0 up to a given limit. The value can be incremented but if the value reaches the limit, it rolls over back to zero. Now we seem to have reached an appropriate level of abstraction that we can represent as a class: a two-digit display class.

For our clock display, we will first program a class for a two-digit number display (Figure 24), and give it an accessor method to get its value and two mutator methods to set the value and to increment it. Once we have defined this class, we can just create two objects of this class with different limits to construct the whole clock display.

Figure 24: A two-digit number display

3.5 Implementing the clock display

As discussed above, in order to build the clock display, we will first build a two-digit number display. This display needs to store two values. One is the limit to which it can count before rolling over to zero. The other is the current value. We will represent both of these as integer fields in our class (Figure 25).

```
public class NumberDisplay
{
    private int limit;
    private int value;

    Constructor and methods omitted.
}
```

Figure 25: Class for two-digit number display

We will look at the remaining details of this class later. First, let us assume that we can build the class `NumberDisplay`, and think a bit more about the complete clock display. We would build a complete clock display by having an object that has, internally, two number displays (one for the hours and one for the minutes). Each of the number displays would be a field in the clock display (Figure 26). Here, we make use of a detail that we have not mentioned before: classes define types.

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and methods omitted.
}
```

Figure 26: The ClockDisplay class containing two NumberDisplays

When we discussed fields in [Chapter 2], we said that the word "private" in the field declaration is followed by a type and a name for the field. Here, we use the class

`NumberDisplay` as the type for the fields named `hours` and `minutes`. This shows that class names can be used as types.

[concept box: classes define types] The type of a field specifies what kind of values can be stored in the field. If the type is a class, the field can hold objects of that class.

3.6 Class diagrams versus object diagrams

The structure described in the previous section (one `ClockDisplay` object holding two `NumberDisplay` objects) can be visualized in an *object diagram* as shown in Figure 27a. In this diagram, you see that we are dealing with three objects. Figure 27b shows the *class diagram* for the same situation.

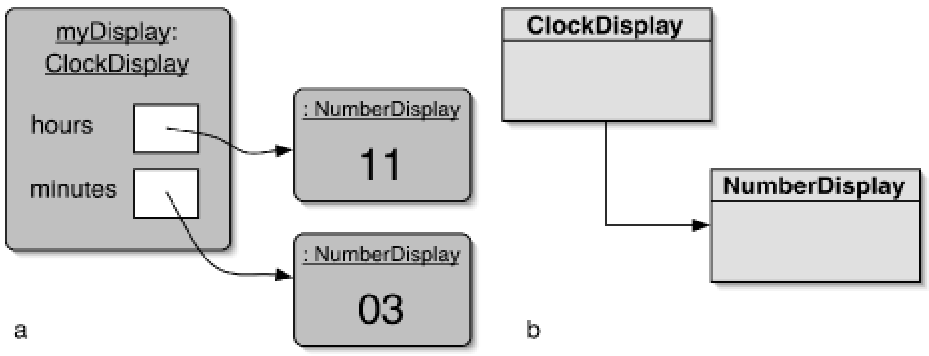


Figure 27: Object diagram and class diagram for the ClockDisplay

Note that the class diagram shows only two classes, while the object diagram shows three objects. This has to do with the fact that we can create multiple objects from the same class. Here, we create two `NumberDisplay` objects from the `NumberDisplay` class.

[concept box: class diagram] These two diagrams offer different views of the same application. The class diagram shows the *static* view. It depicts what we have at the time of writing the program. We have two classes, and the arrow indicates that the class `ClockDisplay` makes use of the class `NumberDisplay` (that is, `NumberDisplay` is mentioned in the source code of `ClockDisplay`). We also say that `ClockDisplay` *depends on* `NumberDisplay`.

[concept box: object diagram] To start the program, we will create an object of class `ClockDisplay`. We will program the clock display so that it automatically creates two `NumberDisplay` objects for itself. Thus, the object diagram shows the situation at *runtime* (when the application is running). This is also called the *dynamic view*.

[concept box: object reference] The object diagram also shows another important detail: When a variable stores an object, the object is not stored in the variable directly, but rather an *object reference* is stored in the variable. In the diagram, the variable is shown as a white box, and the object reference is shown as an arrow. The object referred to is stored outside the referring object, and the *object reference* links the two.

It is very important to understand these two different diagrams and different views. BlueJ only displays the static view. You see the class diagram in its main window. In order to plan and understand Java programs, you need to be able to construct object diagrams on paper or in your head. When we think about what our program will do, we will think about the object structures it creates, and how these objects interact. Being able to visualize the object structures is essential.

Exercise:

3-1 Think again about the lab-classes project that we discussed in [Chapter 1] and [Chapter 2]. Imagine we create a `LabClass` object and three `Student` objects. We then enroll all three students in that lab. Try to draw a class diagram and an object diagram for that situation. Identify and explain the differences between them.

3.7 Primitive types and object types

[concept box: primitive types] Java knows two very different kinds of type: *primitive types* and *object types*. Primitive types are all predefined in the Java language. They include `int` and `boolean`. A complete list of primitive types is given in [Appendix B]. Object types are those defined by classes. Some classes are defined by the standard Java system (such as `String`), others are those classes we write ourselves. Both can be used as types, but there are situations in which they behave differently. One difference is how values are stored. As we could see from our diagrams, primitive values are stored directly in a variable (we have written the value directly into the variable box, for example in [Chapter 2, Figure 7]). Objects, on the other hand, are not stored directly in the variable, but instead a reference to the object is stored (drawn as an arrow in our diagrams, Figure 27).

We will see other differences between primitive types and object types later.

```
/**
 * The NumberDisplay class represents a digital number display
that
 * can hold values from zero to a given limit. The limit can be
 * specified when creating the display. The values range from
zero
 * (inclusive) to limit-1. If used, for example, for the seconds
 * on a digital clock, the limit would be 60, resulting in
display
 * values from 0 to 59. When incremented, the display
automatically
 * rolls over to zero when reaching the limit.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2001.05.26
 */
public class NumberDisplay
{
    private int limit;
    private int value;

    /**
```

```
         * Constructor for objects of class Display
         */
        public NumberDisplay(int rollOverLimit)
        {
            limit = rollOverLimit;
            value = 0;
        }

        /**
         * Return the current value.
         */
        public int getValue()
        {
            return value;
        }

        /**
         * Set the value of the display to the new specified value.
    If
         * the new value is less than zero or over the limit, do
         * nothing.
         */
        public void setValue(int replacementValue)
        {
            if((replacementValue >= 0) && (replacementValue <
    limit))
                value = replacementValue;
        }

        /**
         * Return the display value (that is, the current value as a
         * two-digit String. If the value is less than ten, it will
    be
         * padded with a leading zero).
         */
        public String getDisplayValue()
        {
            if(value < 10)
                return "0" + value;
            else
                return "" + value;
        }

        /**
         * Increment the display value by one, rolling over to zero
    if
         * the limit is reached.
         */
        public void increment()
        {
            value = (value + 1) % limit;
        }
    }
```

Figure 28: Implementation of the NumberDisplay class

3.8 The ClockDisplay source code

Before we start to analyze the source code, it will help if you have a look at the example yourself.

Exercise:

3-2 Start BlueJ, open the *clock-display* example and experiment with it. To use it, create a **ClockDisplay** object, then open an inspector window for this object. With the inspector open, call the object's methods. Watch the **displayString** field in the inspector. Read the project comment (by double-clicking the text note icon on the main screen) to get more information.

3.8.1 Class NumberDisplay

We will now analyze a complete implementation of this task. The project *clock-display* in the examples attached to this book contains the solution. First, we will look at the implementation of the class **NumberDisplay**. Figure 28 shows the complete source code. Overall, this class is fairly straightforward. It has the two fields discussed above (section 3.5), one constructor and four methods (**getValue**, **setValue**, **getDisplayValue**, and **increment**).

The constructor receives the roll-over limit as a parameter. If, for example, twenty-four is passed in as the roll-over limit, the display will roll over to zero at that value. Thus, the range for the display value would be zero to twenty-three. This allows us to use this class for both hour and minute displays. For the hour display, we will create a **NumberDisplay** with limit twenty-four, for the minute display we will create one with limit sixty.

The constructor then stores the roll-over limit in a field and sets the current value of the display to zero.

Logic operators

Logic operators operate on boolean values (true or false) and produce a new boolean value as a result. The three most important logical operators are *and*, *or* and *not*. They are written in Java as:

&&	(and)
||	(or)
!	(not)

The expression

    a && b

is true if a and b both are true, and false in all other cases. The expression

    a || b

is true if either a or b or both are true, and false if they are both false. The expression

    !a

is true if a is false, and false if a is true.

Next follows a simple accessor method for the current display value (**getValue**). This allows other objects to read the current value of the display.

The following mutator method **setValue** is more interesting. It reads:

```
public void setValue(int replacementValue)
{
    if((replacementValue >= 0) && (replacementValue < limit))
        value = replacementValue;
}
```

Here, we pass the new value for the display as a parameter into the method. However, before we assign the value, we have to check whether the value is legal. The legal range for the value, as discussed above, is zero to one below the limit. We use an if statement to check that the value is legal before we assign it. The symbol '**&&**' is a logical “and” operator. It causes the condition in the if statement to be true if both the conditions on either side of the ‘`&&`’ symbol are true. See the “Logic Operators” sidebar for details. [Appendix D] shows a complete table of logic operators in Java.

Exercises:

3-3 What happens when the `setValue` method is called with an illegal value? Is this a good solution? Can you think of a better solution?

3-4 What would happen if you replace the ‘>=’ operator in the test with ‘>’, so that it reads

```
if((replacementValue > 0) && (replacementValue < limit))
```

3-5 What would happen if you replace the ‘`&&`’ operator in the test with ‘||’, so that it reads

```
if((replacementValue >= 0) || (replacementValue < limit))
```

The next method, `getDisplayValue`, also returns the display’s value, but in a different format. The reason is that we want to display the value as a two-digit string. That is, if the current time is 3:05am, we want the display to read `03:05`, and not `3:5`. To enable us to do this easily, we have implemented the `getDisplayValue` method. This method returns the current value as a string, and it adds a leading zero if the value is less than ten. Here is the relevant section of the code:

```
if(value < 10)
    return "0" + value;
else
    return "" + value;
```

Note that the zero (“0”) is written in double quotes. Thus, we have written the *string* 0, not the *integer number* 0. Then, the expression:

```
"0" + value
```

‘adds’ a string and an integer (since the type of `value` is integer). Thus, the plus operator represents string concatenation again, as seen in Section [2.8]. Before continuing, we will now look at string concatenation a little more closely.

3.8.2 String concatenation

The plus operator (+) has different meanings, depending on the type of its operands. If both operands are numbers, it represents addition, as we would expect. Thus:

```
42 + 12
```

adds those two numbers and the result is 54. However, if the operands are strings, then the meaning of the plus sign is string concatenation, and the result is a single string that consists of both operands stuck together. For example, the result of the expression:

```
"Java" + "with BlueJ"
```

is the single string:

```
"Javawith BlueJ"
```

Note that the system does not automatically add a space between the strings. If you want a space, you have to include it yourself within one of the strings.

If one of the operands of a plus operation is a string, and the other is not, then the other operator is automatically converted to a string, and then a string concatenation is performed. Thus:

```
"answer: " + 42
```

results in the string

```
"answer: 42"
```

This works for all types. Whatever type is 'added' to a string is automatically converted to a string and then concatenated.

Back to our code in the `getDisplayValue` method. If `value` contains 3, for example, then the statement:

```
return "0" + value;
```

will return the string "03". In the case where the value is greater than 9, we have used a little trick:

```
return "" + value;
```

Here, we concatenate `value` with an empty string. The result is that the value will be converted to a string, and no other characters will be prefixed to it. We are using the plus operator for the sole purpose of forcing a conversion of the integer value to a value of type `String`.

Exercises:

3-6 Does the `getDisplayValue` method work correctly in all circumstances? What assumptions are made within it? What happens if you create a number display with limit 800, for instance?

3-7 Is there any difference in the result of writing

```
return value + "";
```

rather than

```
return "" + value;
```

in the `getDisplayValue` method?

3.8.3 The modulo operator

The last method in the `NumberDisplay` class increments the display value by one. It takes care that the value resets to zero when the limit is reached:

```
public void increment()
{
    value = (value + 1) % limit;
}
```

This method uses the *modulo* operator (%). The modulo operator calculates the remainder of an integer division. For example, the result of the division

```
27 / 4
```

can be expressed in integer numbers as

```
result = 6, remainder = 3
```

The modulo operator returns just the remainder of such a division. Thus, the result of the expression `(27 % 4)` would be `3`.

Exercises:

3-8 Explain the modulo operator. You may need to consult more resources (online Java language resources, other Java books, etc.) to find out the details.

3-9 What is the result of the expression `(8 % 3)`?

3-10 What are all possible results of the expression `(n % 5)`, where `n` is an integer variable?

3-11 What are all possible results of the expression `(n % m)`, where `n` and `m` are integer variables?

3-12 Explain in detail how the increment method works.

3-13 Rewrite the increment method without the modulo operator, using an if statement. Which solution is better?

3-14 Using the *clock-display* project in BlueJ, test the `NumberDisplay` class by creating a few `NumberDisplay` objects and calling their methods.

3.8.4 Class ClockDisplay

Now that we have seen how we can build a class that defines a two-digit number display, we will look in more detail at the `ClockDisplay` class – the class that will create two number displays to create a full time display. Figure 29 shows the complete source code of the `ClockDisplay` class.

```
/**
 * The ClockDisplay class implements a digital clock display for a
 * European-style 24 hour clock. The clock shows hours and minutes.
 * The range of the clock is 00:00 (midnight) to 23:59 (one minute
 * before midnight).
 *
 * The clock display receives "ticks" (via the timeTick method) every
 * minute and reacts by incrementing the display. This is done in the
 * usual clock fashion: the hour increments when the minutes roll
 * over to zero.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2001.05.26
 */
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at the time specified by the
     * parameters.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every minute - it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
        }
        updateDisplay();
    }

    /**
     * Set the time of the display to the specified hour and
     * minute.
     */
    public void setTime(int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        updateDisplay();
    }

    /**
     * Return the current time of this display in the format HH:MM.
     */
    public String getTime()
    {
        return displayString;
    }

    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
        displayString = hours.getDisplayValue() + ":" +
                        minutes.getDisplayValue();
    }
}
```

Figure 29: Implementation of the ClockDisplay class

As with the `NumberDisplay` class, we will briefly discuss all fields, constructors, and methods.

In this project, we use the field `displayString` to simulate the actual display device of the clock (as you could see in exercise 3-2). Were this software to run in a real clock, we would present the output on the real clock display instead. So this string serves as our software simulation for the clock's output device.

To achieve this, we use one string field and a method:

```
public class ClockDisplay
{
    private String displayString;

    Other fields and methods omitted.

    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
        Method implementation omitted.
    }
}
```

Whenever we want the display of the clock to change, we will call the internal method `updateDisplay`. In our simulation, this method will change the display string (we will examine the source code to do this below). In a real clock, this method would also exist – there it would change the real clock display.

Apart from the display string, the `ClockDisplay` class has only two more fields: `hours` and `minutes`. Each of these fields can hold an object of type `NumberDisplay`. The logical value of the clock's display (the current time) is stored in these `NumberDisplay` objects. Figure 30 shows an object diagram of this application when the current time is 15:23.

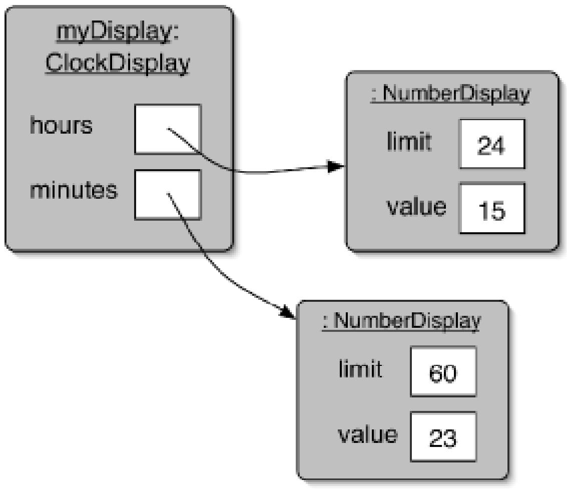


Figure 30: Object diagram of the clock display

3.9 Objects creating objects

The first question we have to ask ourselves is: where do these three objects come from? When we want to use a clock display, we might create a `ClockDisplay` object. We then assume that our clock display has hours and minutes. So by simply creating a clock display, we expect that we have implicitly created two number displays for the hours and minutes.

[concept box: object creation] As writers of the `ClockDisplay` class, we have to make this happen. We simply write code in the constructor of the `ClockDisplay` that creates and stores two `NumberDisplay` objects. Since the constructor is automatically executed when a `ClockDisplay` object is created, the `NumberDisplay` objects will automatically be created at the same time. Here is the code of the `ClockDisplay` constructor that makes this work.

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Remaining fields omitted.

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    Methods omitted.
}
```

Each of the first two lines in the constructor creates a new `NumberDisplay` object and assigns it to a variable. The syntax of an operation to create a new object is

new *ClassName* (*parameter-list*)

The **new** operation does two things:

1. It creates a new object of the named class (here: **NumberDisplay**).
2. It executes the constructor of that class.

If the constructor of the class is defined to have parameters, then the actual parameters must be supplied in the **new** statement. For instance, the constructor of class **NumberDisplay** was defined to expect one integer parameter:

formal parameter

```
public NumberDisplay(int rollOverLimit)
```

Thus, the **new** operation for the **NumberDisplay** class, which calls this constructor, must provide one actual parameter of type **int** to match the defined constructor header:

actual parameter

```
new NumberDisplay(24);
```

This is the same as for methods, discussed in Section [2.4]. With this constructor, we have achieved what we wanted: If someone now creates a **ClockDisplay** object, the **ClockDisplay** constructor will automatically execute and create two **NumberDisplay** objects. Then the clock display is ready to go.

Exercise:

3-15 Create a **ClockDisplay** object by selecting the following constructor:

```
new ClockDisplay()
```

Call its **getTime** method to find out the initial time the clock has been set to. Can you work out why it starts at that particular time?

3-16 How many times would you need to call the tick method on a newly created **ClockDisplay** object to make its time reach 01:00? How else could you make it display that time?

3.10 Multiple constructors

You might have noticed when you created a **ClockDisplay** object that the popup menu offered you two ways to do that:

```
new ClockDisplay()
new ClockDisplay(hour, minute)
```

This is because the **ClockDisplay** class contains two constructors. What they provide are alternative ways of initializing a **ClockDisplay** object. If the constructor with no arguments is used, then the starting time displayed on the clock will be 00:00. If, on the other hand, you want to have a different starting time, you can set that up by using the second constructor. It is common for class definitions to contain alternative versions of constructors or methods that provide various ways of achieving a particular task via their distinctive sets of parameters. This is known as *overloading* a constructor or method. [concept box: overloading]

Exercise:

3-17 Look at the second constructor in `ClockDisplay`'s source code. Explain what it does and how it does it.

3-18 Identify the similarities and differences between the two constructors. Why is there no call to `updateDisplay` in the second constructor, for instance?

3.11 Method calls

3.11.1 Internal method calls

The last line of the first `ClockDisplay` constructor consists of the statement

```
updateDisplay();
```

This statement is a *method call*. As we have seen above, the `ClockDisplay` class has a method with the following signature:

```
private void updateDisplay()
```

[concept box: internal method call] The method call above invokes this method. Since this method is in the same class as the call of the method, we also call it an *internal method call*. Internal method calls have the syntax

methodName (*parameter-list*);

In our example, the method does not have any parameters, so the parameter list is empty. This is signified by the set of parentheses with nothing between them.

When a method call is encountered, the matching method is executed, and then execution returns to the method call and continues at the next statement after the call. For a method signature to match the method call, both the name and the parameter list of the method must match. Here, both parameter lists are empty, so they match. This need to match against both method name and parameter lists is important because there may be more than one method of the same name in a class – if that method is overloaded.

In our example, the purpose of this method call is to update the display string. After the two number displays have been created, the display string is set to show the time indicated by the number display objects. The implementation of the `updateDisplay` method will be discussed below.

3.11.2 External method calls

Now let us examine the next method: `timeTick`. The definition is:

```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {  // it just rolled over!
        hours.increment();
        }
    }
```

Were this display connected to a real clock, this method would be called once every sixty seconds by the electronic timer of the clock. For now, we just call it ourselves to test the display.

When the **`timeTick`** method is called, it first executes the statement

```
minutes.increment();
```

[concept box: external method call] This statement calls the **`increment`** method of the **`minutes`** object. Thus, when one of the methods of the **`ClockDisplay`** object is called, it in turn calls a method of another object to do part of the task. A method call to a method of another object is referred to as an *external method call*. The syntax of an external method call is

object . *methodName* (*parameter-list*)

This syntax is known as *dot notation*. It consists of an object name, a dot, the method name, and parameters for the call. It is particularly important to appreciate that we use the name of an *object* here and not the name of a class. We use the field **`minutes`** rather than **`NumberDisplay`**.

The **`timeTick`** method then has an if statement to check whether the hours should also be incremented. As part of the condition in the if statement, it calls another method of the minutes object: **`getValue`**. This method returns the current value of the minutes. If that value is zero, then we know that the display just rolled over, and we should increment the hours. That is exactly what the code does.

If the value of the minutes is not zero, then we're done. We don't have to change the hours in that case. Thus, the *if* statement does not need an *else* part.

We should now also be able to understand the remaining three methods of the **`ClockDisplay`** class (see Figure 29). The method **`setTime`** takes two parameters, the hour and the minute, and sets the clock to the specified time. Looking at the method body, we can see that it does so by calling the **`setValue`** methods of both number displays, the one for the hours and the one for the minutes. Then it calls **`updateDisplay`** to update the display string accordingly, just as the constructor does.

The **`getTime`** method is trivial – it just returns the current display string. Since we always keep the display string up to date, this is all there is to do.

Finally, the **`updateDisplay`** method is responsible for updating the display string so that the string correctly reflects the time as represented by the two number display objects. It is called every time the time of the clock changes. It works by calling the **`getDisplayValue`** methods of each of the **`NumberDisplay`** objects. These methods return the value of each separate number display. It then uses string concatenation to concatenate these two values with a colon in the middle to a single string.

3.11.3 Summary of the clock display

It is worth looking for a minute at the way this example uses abstraction to divide the problem into smaller parts. Looking at the source code of the class **`ClockDisplay`**, you will notice that we just create a **`NumberDisplay`** object without being particularly interested in what that object looks like internally. We can then just call methods (**`increment`**, **`getValue`**) of that object to make it work for us. At this level, we just assume that **`increment`** will correctly increment the display's value, without being concerned with how it does it.

In real-world projects, these different classes are often written by different people. You might have already noticed that all these two people have to agree on is what method signatures the class should have and what they should do. Then one person can concentrate on implementing the methods, while the other person can just use them.

The set of methods an object makes available to other objects is called its *interface*. We will discuss interfaces in much more detail later in this book.

Exercises:

3-19 *Challenge exercise*: Change the clock from a 24-hour clock to a 12-hour clock. Be careful: this is not as easy as it might at first seem. In a 12 hour clock, the hours after midnight and after noon are not shown as 00:30, but as 12:30. Thus, the minute display shows values from 0 to 59, while the hour display shows values from 1 to 12!

3-20 There are (at least) two ways that you can make a 12-hour clock. One possibility is to just store hour values from 1 to 12. On the other hand, you can just leave the clock to work internally as a 24-hour clock, and just change the display string of the clock display to show **`4:23`** or **`4.23pm`** when the internal value is **`16:23`**. Implement both versions. Which option is easier? Which is better? Why?

3.12 Another example of object interaction

We will now examine the same concepts with a different example, using different tools. We are still concerned with understanding how objects create other objects, and how objects call each other's methods. In the first half of this chapter, we have used the most fundamental technique to analyze a given program: code reading. The ability to read and understand source code is one of the most essential for a software developer, and we will need to apply it in every project we work on. However, sometimes it is beneficial to use additional tools in order to help us gain a deeper understanding about how a program executes. One tool we will now look at is a *debugger*.

[concept box: debugger] A debugger is a program that lets programmers execute an application one step at a time. It typically provides functions to stop and start a program at selected points in the source code, and to examine the values of variables.

Debuggers vary widely in complexity. Those for professional developers have a large number of functions useful for sophisticated examination of many facets of an application. BlueJ has a built-in debugger that is much simpler. We can use it to stop our program, step through it one line of code at a time, and examine the values of our variables. Despite the debugger's apparent lack of sophistication, this is enough to give us a great deal of information.

Before we start experimenting with the debugger we will take a look at the example we will use for debugging: a simulation of an email system.

The name "debugger"
Errors in computer programs are commonly known as "bugs". Thus, programs that help in the removal of errors are known as "debuggers".

It is not entirely clear where the term "bug" comes from. There is a famous case of what is known as "The first computer bug" - a real bug (a moth, in fact) that was found inside the Mark II computer by Grace Murray Hopper, an early computing pioneer, in 1945. A log book still exists in the National Museum of American History of the Smithsonian Institute that shows an entry with this moth taped into the book and the remark "first actual case of a bug being found". The wording, however, suggests that the term "bug" had been in use before this real one caused trouble in the Mark II.

To find out more, do a web search for "first computer bug" – you will even find pictures of the moth!

3.12.1 The mail system example

We start by investigating the functionality of the *mail-system* project. At this stage it is not important to read the source, but mainly to execute the existing project to get an understanding of what it does.

Exercise

3-21 Open the *mail-system* project, which you can find in the book's support material. Create a **`MailServer`** object. Create two **`MailClient`** objects. When doing this, you need to supply the **`MailServer`** instance, which you just created, as a parameter. You also need to specify a user name for the mail client. (A mail client is a program to read and write email. Every instance you create represents an email program for a different user.)

Experiment with the **`MailClient`** objects. They can be used to send messages from one mail client to another (using the **`sendMessage`** method) and to receive messages (using the **`getNextMailItem`** or **`printNextMailItem`** methods).

Examining the mail system project we see that:

It has three classes: **`MailServer`**, **`MailClient`** and **`MailItem`**.

One mail server object must be created that is used by all mail clients. It handles the exchange of messages.

Several mail client objects can be created. Every mail client has an associated user name.

Messages can be sent from one mail client to another via a method in the mail client class.

Messages can be received by a mail client from the server one at a time, using a method in the mail client.

The `MailItem` class is never instantiated explicitly by the user. It is used internally in the mail clients and server to store and exchange messages.

Exercise

3-22 Draw an object diagram of the situation you have after creating a mail server and three mail clients. Object diagrams were discussed in section 3.6.

The three classes have different degrees of complexity. `MailItem` is fairly trivial. We will discuss only one small detail, and leave the rest up to the reader to investigate. `MailServer` is quite complex at this stage – is makes use of concepts discussed only much later in this book. We will not investigate that class in detail here. Instead, we just trust that it does its job – another example of how abstraction is used to hide detail that we don't need to be aware of.

The `MailClient` class is the most interesting, and we will examine it in some detail.

3.12.2 The this keyword

The only section we will discuss from the `MailItem` class is the constructor. It uses a Java construct that we have not encountered before. The source code is shown in Figure 31.

```
public class MailItem
{
    // The sender of the item.
    private String from;
    // The intended recipient.
    private String to;
    // The text of the message.
    private String message;

    /**
     * Create a mail item from sender to the given recipient,
     * containing the given message.
     * @param from    The sender of this item.
     * @param to      The intended recipient of this item.
     * @param message The text of the message to be sent.
     */
    public MailItem(String from, String to, String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }

    Methods omitted.
}
```

Figure 31: Fields and constructor of the MailItem class

The new Java feature in this code fragment is the use if `this` keyword:

```
this.from = from;
```

The whole line is an assignment statement. It assigns the value from the right hand side (`from`) to the variable on the left (`this.from`).

The reason for using this construct is that we have a situation which is known as *name overloading* – the same name being used for two different entities. The class contains three fields, named `from`, `to`, and `message`. The constructor has three parameters, also named `from`, `to`, and `message`!

So while we are executing the constructor, how many variables exist? The answer is six – three fields and three parameters. It is important to understand that the fields and the parameters are separate variables that exist independently of each other, even though they share similar names. A parameter and a field sharing a name is not really a problem in Java.

The problem we do have, though, is how to reference the six variables so as to be able to distinguish between the two sets. If we simply use the variable name "from" in the constructor (for example in a statement `System.out.println(from);`) – which variable will be used, the parameter or the field?

The Java specification answers this question. It specifies that always the definition from the closest enclosing block will be used. Since the `from` parameter is defined in the constructor, and the `from` field is defined in the class, the parameter will be used. Its definition is "closer" to the statement that uses it.

Now all we need is a mechanism to access a field when there is a more closely defined variable with the same name. That is what the `this` keyword is used for. The expression `this` refers to the current object. Writing `this.from` refers to the `from` field in the current object. Thus, this construct gives us a means to refer to the field instead of the parameter with the same name. Now we can read the assignment statement again:

```
this.from = from;
```

This statement, as we can see now, has the following effect:

field named "`from`*" = parameter named "*`from`*";*

In other words: it assigns the value from the parameter to the field with the same name. This is, of course, exactly what we need to do to properly initialize the object.

One last question remains: why are we doing this at all? The whole problem could easily be avoided just by giving the fields and the parameters different names. The reason is readability of source code.

Sometimes there is one name that perfectly describes the use of a variable. It fits so well, that we do not want to invent a different name for it. We want to use it for the parameter, where it serves as a hint to the caller indicating what needs to be passed, and we want to use it for the field, where it is useful as a reminder for the implementer of the class, indicating what the field is used for. If one name perfectly describes the use, it is reasonable to use it for both and to go through the trouble of using the **this** keyword in the assignment to resolve the name conflict.

3.13 Using a debugger

The most interesting class in the mail system example is the mail client. We will now investigate it in some more detail by using a debugger. The mail client has three methods: **getNextMailItem**, **printNextMailItem**, and **sendMessage**. We will first investigate the **printNextMailItem** method.

Before we start with the debugger, set up a scenario we can use to investigate (exercise 3-23).

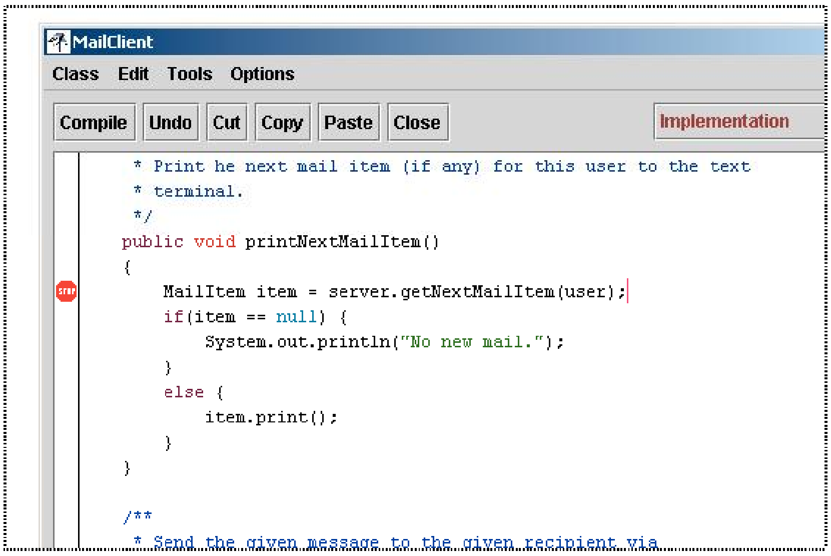
Exercise

3-23 Set up a scenario for investigation: Create a mail server, then create two mail clients for the users "Sophie" and "Juan" (you should name the instances "sophie" and "juan" as well, so that you can better distinguish them on the object bench). Then use Sophie's **sendMessage** method to send a message to Juan. Do not yet read the message.

After the setup in exercise 3-23, we have a situation where one mail item is stored on the server for Juan, waiting to be picked up. We have seen that the **printNextMailItem** method picks up this mail item and prints it to the terminal. Now we want to investigate exactly how this works.

3.13.2 Setting breakpoints

To start our investigation, we set a breakpoint (exercise 3-24). A breakpoint is a flag attached to a line of source code that will stop the execution of a method at that point when it is reached. It is represented in the BlueJ editor as a small stop sign (Figure 32).



```
 * Print he next mail item (if any) for this user to the text
 * terminal.
 */
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    }
    else {
        item.print();
    }
}

/**
 * Send the given message to the given recipient via
```

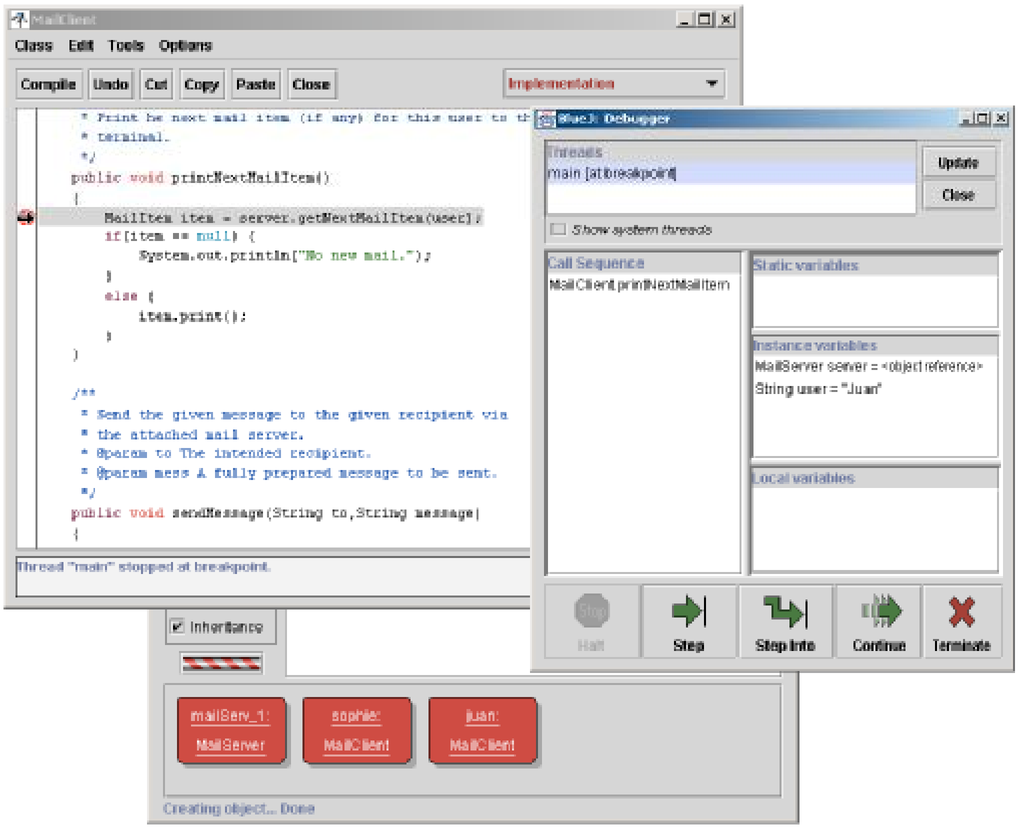
Figure 32: A breakpoint in the BlueJ editor

You can set a breakpoint by opening the BlueJ editor, selecting the appropriate line (in our case the first line of the **`printNextMailItem`** method) and selecting "Set Breakpoint" from the menu. You can also just click into the area next to the line of code where the breakpoint symbol appears to set or remove breakpoints. Note that the class has to have been compiled to do this, and that recompiling will remove all breakpoints.

Exercise

3-24 Open the editor for the **`MailClient`** class and set a breakpoint at the first line of the **`printNextMailItem`** method, as shown in Figure 32.

Once you have set the breakpoint, invoke the **`printNextMailItem`** method on Juan's mail client. The editor window for the **`MailClient`** class and a debugger window will pop up (Figure 33).



```
     * Print he next mail item (if any) for this user to th
     * terminal.
     */
    public void printNextMailItem()
    {
        MailItem item = server.getNextMailItem(user);
        if(item == null) {
            System.out.println("No new mail.");
        }
        else {
            item.print();
        }
    }

    /**
     * Send the given message to the given recipient via
     * the attached mail server.
     * @param to The intended recipient.
     * @param mess A fully prepared message to be sent.
     */
    public void sendMessage(String to,String message)
    {
```

Thread "main" stopped at breakpoint.

Threads: main [at breakpoint]

Call Sequence: MailClient.printNextMailItem

Instance variables:
MailServer server = <object reference>
String user = "Juan"

Creating object... Done

Figure 33: The debugger window, execution stopped at a breakpoint.

Along the bottom of the debugger window are some control buttons. They can be used to continue or interrupt the execution of the program. (For a more detailed explanation of the debugger controls, see [Appendix X]).

On the right hand side of the debugger window are three areas for variable display, titled *static variables*, *instance variables*, and *local variables*. We will ignore the static variable area for now. We will discuss static variables later, and this class does not have any.

We see that this objects has two instance variables (or fields), **`server`** and **`user`**, and we can see the current values. The **`user`** variable stores the string "Juan" and the server variable stores a reference to another object. The object reference is what we have drawn as an arrow in the object diagrams.

Note that there is no local variable yet. This is because execution stops *before* the line with the breakpoint is executed. Since the line with the breakpoint contains the declaration of the only local variable, and that line has not yet been executed, no local variable exists at the moment.

The debugger not only allows us to interrupt the execution of the program and inspect the variable, it also lets us step forward slowly.

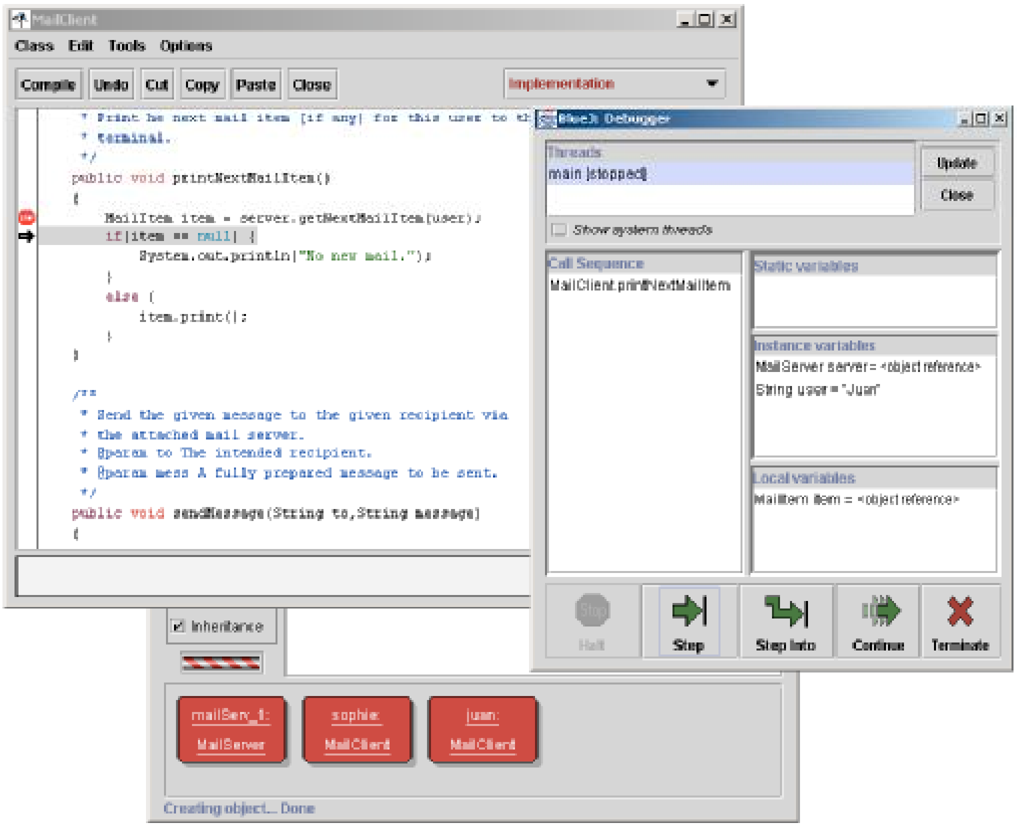
3.13.3 Single stepping

When stopped at a breakpoint, clicking the "Step" button executes a single line of code and then stops again.

Exercise

3-25 Step one line forward in the execution of the **`printNextMailItem`** method by clicking the *Step* button.

The result of executing the first line of the **`printNextMailItem`** method is shown in Figure 34. We can see that execution has moved on by one line (a small black arrow next to the line of source code indicates the current position) and the local variable list in the debugger window indicates that a local variable **`item`** has been created, and an object has been assigned to it.



```
     * Print he next mail item (if any) for this user to th
     * terminal.
     */
    public void printNextMailItem()
    {
        MailItem item = server.getNextMailItem(user);
        if(item == null) {
            System.out.println("No new mail.");
        }
        else {
            item.print();
        }
    }

    /**
     * Send the given message to the given recipient via
     * the attached mail server.
     * @param to The intended recipient.
     * @param mess A fully prepared message to be sent.
     */
    public void sendMessage(String to,String message)
    {
```

Threads: main (stopped)

Call Sequence: MailClient.printNextMailItem

Instance variables:
MailServer server = <object reference>
String user = "Juan"

Local variables:
MailItem item = <object reference>

Figure 34: Stopped again after a single step

Exercise

3-26 Predict which line will be marked as the next line to execute after the next step. Then execute another single step and check your prediction. Were you right or wrong? Explain what happened and why.

We can now repeatedly use the Step button to step to the end of the method. This allows us to see the path the execution takes. This is especially interesting in conditional statements: we can clearly see which branch of an if statement is executed, and use this to see whether it matches our expectations.

Exercise

3-27 Call the same method (`printNextMailItem`) again. Step through the method again, as before. What do you observe? Explain why this is.

3.13.4 Stepping into methods

When stepping through the `printNextMailItem` method, we have seen two method calls to objects of our own classes. The line

```
    MailItem item = server.getNextMailItem(user);
```

includes a call to the `getNextMailItem` method of the `server` object. Checking the instance variable declarations, we can see that the `server` object is declared of class `MailServer`.

The line

```
    item.print();
```

calls the `print` method of the `item` object. We can see in the first line of the `printNextMailItem` method that `item` is declared to be of class `MailItem`.

Using the step command in the debugger we have used abstraction: we have viewed the `print` method of the `item` class as a single instruction, and we could observe that its effect is to print out the details (sender, addressee, and message) of the mail item.

If we are interested in more detail, we can look further into the process, and see the `print` method itself execute step by step. This is done by using the *Step Into* command in the debugger, instead of the *Step* command. *Step Into* will step into the method being called, and stop at the first line inside that method.

Exercise

3-28 Set up the same test situation as we did before. That is: send a message from Sophie to Juan. Then invoke the `printNextMailItem` message of Juan's mail client again. Step forward as before. This time, when you reach the line

```
item.print()
```

use the *Step Into* command instead of the *Step* command. Make sure you can see the text terminal window as you step forward. What do you observe? Explain what you see.

3.14 Method calling revisited

In the experiments in section 3.13 we have seen another example of object interaction similar to one we had seen before: objects calling methods of other objects. In the `printNextMailItem` method, the `MailClient` object made a call to a `MailServer` object to retrieve the next mail item. This method (`getNextMailItem`) returned a value – an object of type `MailItem`. Then there was a call to the `print` method of the mail item. Using abstraction, we can view the `print` method as a single command. Or, if we are interested in more detail, we can go to a lower level of abstraction and look inside the `print` method.

In a similar style, we can use the debugger to observe one object creating another one. The `sendMessage` method in the `MailClient` class shows a good example. In this method, a `MailItem` object is created in the first line of code:

```
    MailItem mess = new MailItem(user, to, message);
```

The idea here is that the mail item is used to encapsulate a mail message. The mail item contains information about the sender, the addressee, and the message itself.

When sending a message, a mail client creates a mail item with all this information, and then stores this mail item on the mail server. There it can later be picked up by the mail client of the addressee.

In the line of code above we see the `new` keyword being used to create the new object, and we see the parameters being passed to the constructor. (Remember: constructing an object does two things – the object is being created and the constructor is executed.) Calling the constructor works in a very similar fashion to calling methods. This can be observed by using the *Step Into* command at the line where the object is being constructed.

Exercises

3-29 Set a breakpoint in the first line of the `sendMessage` method in the `MailClient` class. Then invoke this message. Use the *Step Into* function to step into the constructor of the mail item. In the debugger display for the `MailItem` object, you can see the instance variables and local variables that have the same names, as discussed in section 3.12.2. Step further to see the instance variables get initialized.

3-30 Use a combination of code reading, execution of methods, breakpoints, and single stepping to familiarize yourself with the `MailItem` and `MailClient` classes. Note that we have not discussed enough for you to understand the implementation of the `MailServer` class yet, so you can ignore this for now. (You can, of course, look at it if you feel adventurous, but don't be surprised if you find it slightly baffling…) Explain in writing how the `MailClient` and `MailItem` classes interact. Draw object diagrams as part of your explanations.

3.15 Summary

In this chapter, we have discussed how a problem can be divided into sub-problems. We can try to identify subcomponents in those objects that we want to model, and we can implement subcomponents as independent classes. Doing so helps in reducing the complexity of implementing larger applications, since it enables us to implement, test and maintain individual classes separately.

We have seen how this results in structures of objects working together to solve a common task. Objects can create other objects, and they can invoke each other's methods. Understanding these object interactions is essential in planning, implementing, and debugging applications.

We can use pen-and-paper diagrams, code reading and debuggers to investigate how an application executes or to track down bugs.

Terms introduced in this chapter

abstraction, modularization, divide and conquer, class diagram, object diagram, object reference, overloading internal method call, external method call, dot notation, debugger, breakpoint.

Concept summary

Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem. [abstraction]

Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways. [modularization]

Classes define types. A class name can be used as the type for a variable. Variables that have a class as their type can store objects of that class. [classes define types]

The class diagram shows the classes of an application and the relationships between them. It gives information about the source code. It presents the static view of a program. [class diagram]

The object diagram shows the objects and their relationships at one moment in time during the execution of an application. It gives information about objects at runtime. It presents the dynamic view of a program. [object diagram]

Variables of class types store references to objects. [object references]

The primitive types in Java are the non-object types. Types such as **`int`**, **`boolean`**, **`char`**, **`double`**, and **`long`** are the most common primitive types. Primitive types have no methods. [primitive type]

Objects can create other objects using the *new* operator. [object creation]

A class may contain more than one constructor, or more than one method of the same name, as long as each has a distinctive set of parameter types. [overloading]

Methods can call other methods of the same class as part of their implementation. This is called an internal method call. [internal method call]

Methods can call methods of other objects using dot notation. This is called an external method call. [external method call]

A debugger is a software tool that helps in examining how an application executes. It can be used to find bugs. [debugger]

Exercises:

3-31 Use the debugger to investigate the *clock-display* project. Set breakpoints in the **`ClockDisplay`** constructor and each of the methods and then single step through them. Does it behave as you expected? Did this give you new insights? If so, what were they?

3-32 Use the debugger to investigate the **`insertMoney`** method of the *better-ticket-machine* project from [Chapter 2]. Conduct tests that cause both branches of the if statement to be executed.

3-33 Add a subject line for an email to mail items in the *mail-system* project. Make sure printing messages also prints the subject line. Modify the mail client accordingly.