Object Interaction

Java version

Objectives

To understand interactions between multiple objects in an application.

Main concepts discussed in this chapter

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

Resources

Classes needed for this lab - *chapter03.jar.*

To do

Installing and launching the lab session

Create a specific directory for this lab session (just like in the earlier lab sessions). Download and unarchive *chapter03.jar* into the directory.

The clock example

We will build a digital clock which can display the time like

Abstraction and modularization

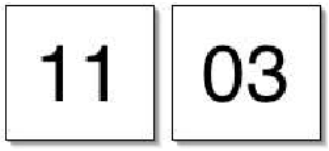
It would be possible to create simple applications such as this by using only a single class. However, as the projects become more complex, doing everything within one class becomes practically impossible.

Abstraction in software

Modularization in the clock example

There are (at least) two different ways to consider the clock display above:

- As one four-digit display.
- As two separate two-digit displays where:
    - the right-hand display represents minutes;
    - the left-hand display represents hours.



We will take the latter approach since we can create a single class and use it to display both minutes and hours.

Implementing the clock display

A single class is needed to display either the minutes or the hours. It needs to contain information about the current value and the modulo (the limit at which the numbers roll over).

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

    Constructor and methods omitted.
}
```

The clock display now needs to use two **`NumberDisplay`** objects

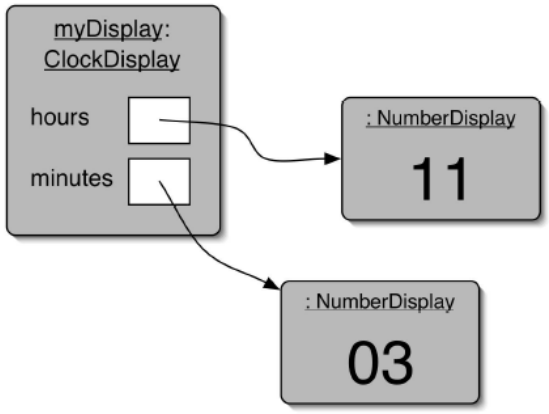
```
class ClockDisplay {
    private final NumberDisplay hours;
    private final NumberDisplay minutes;

    Constructor and methods omitted.
}
```

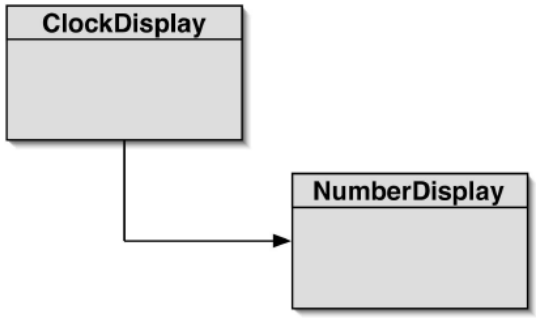
The first field refers to the object which stores the hours, the second field to the object which stores the minutes. Both fields are of type **`NumberDisplay`**.

Class diagrams versus object diagrams

The following *object diagram* shows the above structure schematically.



The object called **`myDisplay`** of type **`ClockDisplay`** contains two fields, **`hours`** and **`minutes`**, both of type **`NumberDisplay`**. The structure can also be shown by the following *class diagram*



which says that the **`ClockDisplay`** class uses the **`NumberDisplay`** class.
The above are different ways of viewing the structure of the application.

- The class diagram shows a static view when writing the code - the **`ClockDisplay`** class depends on the **`NumberDisplay`** class
- The object diagram shows a dynamic view when running the code - two objects of type **`NumberDisplay`** are created as fields of an object of type **`ClockDisplay`**.

Exercises

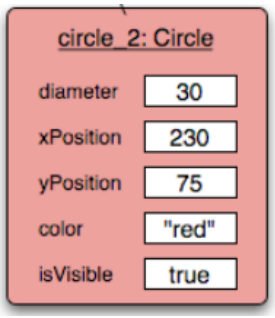
1. In the *labclasses* package from the first lab session, imagine that there is one **`LabClass`** object and three **`Student`** objects enrolled in that lab.
2. Draw object and class diagrams for the situation.
3. Identify and explain the differences between them.
    - At what time(s) can a class diagram change? How is it changed?
    - At what time(s) can an object diagram change? How is it changed?
4. Write a definition of a field named **`tutor`** that can hold a reference to an object of type **`Instructor`**.

Primitive types and object types

Java has two different categories of types

- primitive, eg, **`int`**, **`float`**, **`boolean`**, etc.
- object, eg, **`String`**, **`ClockDisplay`**, etc.

Primitive types are stored directly in a field, shown in an object diagram as follows



Object type fields store a reference to where the actual object is stored, as shown by the arrows in the object diagram for the **`ClockDisplay`** on the previous page.
For completeness, Java's eight primitive types are given in the following table.

Type name	Description	Example
	Integer numbers	
`byte`	8-bit numbers, $-2^7 \le b < 2^7$	24, -2
`short`	16-bit numbers, $-2^{15} \le s < 2^{15}$	137, -119
`int`	32-bit numbers, $-2^{31} \le i < 2^{31}$	5409, -2005
`long`	64-bit numbers, $-2^{63} \le l < 2^{63}$	423266353L, 55l
	Real numbers	
`float`	single-precision, 1.4e-45 ≤ positive f ≤ 3.4028235e38	43.889f, 43.889F
`double`	double-precision, 4.9e-324 ≤ positive d ≤ 1.79769313486223157e308	45.63, 2.4e5
	Other	
`char`	single character (16 bit)	'm', '?', '\u00F6'
`boolean`	boolean value (true or false)	true, false

The `ClockDisplay` source code

Exercise

5. Open the *clockdisplay* package and create a **`clockdisplay.Main`** class to run everything. Create a **`ClockDisplay`** object; invoke the object's methods and observe the results.

Class `NumberDisplay`

Here is the code for the **`NumberDisplay`** class

```
package clockdisplay;

/**
 * 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 Kolling and David J. Barnes
```

```
 * @version 2016.02.29
 */
class NumberDisplay {
    private final int limit;
    private int value;

    /**
     * Constructor for objects of class NumberDisplay.
     * Set the limit at which the display rolls over.
     */
    NumberDisplay(int rollOverLimit) {
        limit = rollOverLimit;
        value = 0;
    }

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

    /**
     * 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).
     */
    String getDisplayValue() {
        if (value < 10) {
            return "0" + value;
        } else {
            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.
     */
    void setValue(int replacementValue) {
        if ((replacementValue >= 0) && (replacementValue < limit)) {
            value = replacementValue;
        }
    }

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

The constructor sets the roll-over limit for the number value, otherwise known as the *modulo*, ie, the value is between 0 and limit - 1. The setter method **`setValue`** checks that the argument is in the proper range before assigning a new value. The **`&&`** operator is Java's *logical and* which returns the boolean **`true`** if and only if both the operands are true. The logical operators are the following

Math	and	or	exclusive or	not
Java	**`&&`**	**`||`**	**`^`**	**`!`**

Exercises

6. What happens if **`setValue`** is called with an illegal value? Is this a good solution? Can you think of a better solution?
7. What if you replaced the **`>=`** relation in **`setValue`** with **`>`**?
8. What if you replaced the **`&&`** operator in **`setValue`** with **`||`**?
9. Which of the following expressions return **`true`**?
    a. **`! (4 < 5)`**
    b. **`! false`**
    c. **`(2 > 2) || ((4 == 4) && (1 < 0))`**
    d. **`(2 > 2) || (4 == 4) && (1 < 0)`**
    e. **`(34 != 33) && ! false`**
10. Write an expression using boolean variables **`a`** and **`b`** that evaluates to **`true`** when either **`a`** and **`b`** are both **`true`** or both **`false`**.
11. Write an expression using boolean variables **`a`** and **`b`** that evaluates to **`true`** when only one of **`a`** and **`b`** is **`true`**, and which is **`false`** when **`a`** and **`b`** are both **`true`** or both **`false`**. (This is called *exclusive or*).
12. Consider the expression (**`a && b`**). Write an equivalent expression without using the **`&&`** operator.

Note that in the following line in the method **`getDisplayValue`**

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

it is the string **`"0"`** and not the number **`0`** that is being "added" with **`value`**. In fact, this has the effect of first converting **`value`** to a string and then concatenating the two strings, resulting in another string.

String concatenation

Java's **`+`** operator is overloaded with two different uses.

- When both operands are numeric, it is just ordinary addition
- When at least one operand is a string, it represents string concatenation, ie, joining the strings together. In this case, if the other operand is not a string, it is first automagically converted to a string.
- Note that this is the case in the following line

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

    the empty string is concatenated to the string value, producing a string.

Exercises

13. Does **`getDisplayValue`** always work correctly? What assumptions are made within it? What happens if you create a number display with limit 800 for instance?
14. Is there any difference between the result of writing

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

rather than

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

in the **getDisplayValue** method?

The modulo operator

The increment method

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

uses the arithmetic modulo operator, **%**.

Exercises

15. Explain the modulo operator.
16. What is the result of the expression (**8 % 3**) ?
17.
18. What are all possible results of the expression (**n % 5**) where **n** is of type **int**?
19. What are all possible results of the expression (**n % m**) where **n** and **m** are integer variables?
20. Explain in detail how the **increment** method works.
21. Rewrite the **increment** method without using the modulo operator.
22. Test the **NumberDisplay** class in the *clockdisplay* package by creating a **NumberDisplay** object and calling its methods.

Class ClockDisplay

Here is the code for the **ClockDisplay** class

```
package clockdisplay;

/**
 * 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 Kolling and David J. Barnes
 * @version 2016.02.29
 */
class ClockDisplay {
    private final NumberDisplay hours;
    private final NumberDisplay minutes;
    private String displayString; // simulates the actual display
```

```
/**
 * Constructor for ClockDisplay objects. This constructor
 * creates a new clock set at 00:00.
 */
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.
 */
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.
 */
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.
 */
private 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.
 */
private String getTime() {
    return displayString;
}

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

```
                + minutes.getDisplayValue();
    }
}
```

Objects creating objects

According to the object diagram, we need two objects of type **NumberDisplay** inside an object of type **ClockDisplay**. Where do they come from? When a **ClockDisplay** object is created, its constructor is executed, and in the constructor the lines

```
hours = new NumberDisplay(24);
minutes = new NumberDisplay(60);
```

each create a new object and assign its reference to a field. The syntax

```
new <name of class>(<arguments to constructor>)
```

creates an object of the appropriate type, runs its constructor with the given arguments, and returns a reference to the newly-created object. Argument-passing to the constructor works exactly the same as for methods.

Exercises

23. Create a **ClockDisplay** object by selecting the following constructor

    ```
    new ClockDisplay();
    ```

    Call its **getTime** method to find the initial time the clock is set to.
24. How many times would you need to call the **timeTick** method on a newly created **ClockDisplay** object to make its time reach 01:00? How else could you make it display that time?
25.
26. Write the signature of a constructor that matches the following object creation instruction

    ```
    new Editor("readme.txt", -1);
    ```

27. Write Java statements that define a variable named window of type **Rectangle** and then create a rectangle object and assign it to that variable. The rectangle constructor has two **int** parameters.

Multiple constructors

The **ClockDisplay** class has two constructors.

Exercises

28. Look at the second constructor in **ClockDisplay**. Explain what it does and how it does it.
29. Identify the similarities and differences between the two constructors. Why is there no call to **updateDisplay** in the second constructor?

Method calls

Internal method calls

An expression of the form

```
<method name>(<method arguments>)
```

is a call to a method defined inside the same class as the expression. This is an internal method call. The method arguments must match the list of arguments in a method declaration of the same name as in the expression. A class may have more than one method with the same name, but they must have different arguments. In this case the method name is *overloaded*.

External method calls

Consider the **timeTick** method

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

The first statement calls the method **increment** of the **minutes** object. This is an external method call since it calls a method of another object, and the general syntax

```
<object name>.<method name>(<method arguments)>
```

is called *dot notation*. Note that the method is invoked on an object name and not a class name.

Exercise

30. Given a variable

    ```
    Printer p1;
    ```

    which currently holds a **Printer** object, and two methods inside the **Printer** class with the headers

    ```
    void print(String filename, boolean doubleSided)
    int getStatus(int delay)
    ```

    write two possible calls to each of these methods.

Summary of the clock display

The design of the **ClockDisplay** class is a good illustration of modularization. In using the **NumberDisplay** class, we treat it as a black box, ie, we know what it does but we ignore how it does it. For example, we know that for an object of type **NumberDisplay**, we can invoke its **increment** and **getValue** methods without knowing anything about how they work internally. All we need to know is what arguments they take and what values they return. What happens inside the methods is not our problem - it is the

responsability of whoever developed those methods.

Exercises

31. Change the clock from a 24-hour clock to a 12-hour clock. Note that in a 12-hour clock, 27 minutes past midnight or past noon is shown as 12:27 and not as 00:27! Thus the minutes display shows values from 0 to 59, while the hour display shows values from 1 to 12.
32. There are (at least) two ways in which you can make a 12-hour clock.
    - One possibility is to just store hour values from 1 to 12.
    - Or you can leave the clock internally to work as a 24-hour clock, but 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?

Another example of object interaction

So far we have been analyzing code by developing the skill of code-reading. You'll be spending much of your professional life reading and trying to understand other people's code. It is often harder to read code than to write code.

The mail system example

You'll need the code in the archive file *chapter03_mailsystem.jar*.

Exercise

33. Open the *mailsystem* package. The idea of this project is to simulate users sending mail to each other. A user uses a mail client to send mail items to a server, for delivery to another user's mail client. Create a **`MailServer`** object and two **`MailClient`**s. You need to pass the **`MailServer`** object as a parameter to the **`MailClient`**'s constructor.
    Experiment with the **`MailClient`** objects. Send messages between the **`MailClient`** objects (using the **`sendMailItem`** method, and use the **`getNextMailItem`** and **`printNextMailItem`** methods to receive mail).
34. Draw an object diagram of the situation you have after creating a mail server and three mail clients.

The `this` keyword

Consider the constructor for the **`MailItem`** class

```
MailItem(String from, String to, String message) {
    this.from = from;
    this.to = to;
    this.message = message;
}
```

The syntax

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

is an assignment from the constructor argument **`from`** to the object field **`this.from`**. Why not have just written

```
from = from;
```

instead? Because this would have been an assignment from the constructor argument **`from`** to the constructor

argument **`from`**, in other words from a variable to itself! The keyword **`this`** refers to the *current object*, ie, the object whose code is being executed.
We could of course have called the constructor argument something else, but that might make the code less clear and easy to understand.

Summary

Concept summary

abstraction

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

modularization

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.

classes define types

A class name can be used as the type of a variable. Variables that have a class as their type can store objects of that class.

class diagram

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.

object 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 references

Variables of object types store references to objects.

primitive type

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.

object creation

Objects can create other objects using the **`new`** operator.

overloading

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.

internal method call

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

method call.

external method call

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

Additional Exercises

Exercises

54. Add a subject line for an email to mail items in the **`mailsystem`** package. Make sure printing messages also prints the subject line. Modify the mail client accordingly.
55. Given the following class (only shown in fragments here)

```
class Screen {
    Screen(int xRes, int yRes) {...}
    int numberOfPixels() {...}
    void clear(boolean invert) {...}
}
```

    write some lines of Java code that create a **`Screen`** object, and then call its **`clear`** method if (and only if) its number of pixels is greater than 2 million
56. Describe required changes for the **`ClockDisplay`** class to display hours, minutes, and seconds.
57. Implement the above changes.