Observer

5

You will now learn a pattern that is similarly easy and intuitive to understand as, for example, the Singleton or the Template Method pattern: the Observer pattern. The idea is that you have an object whose state changes (event source); any number of other objects (observers or listeners) want to be informed about these state changes.

5.1 Introduction

How is information distributed in a targeted and addressee-appropriate manner? Think of a newsletter distribution list. There is a newsletter for every topic that might be of interest. If someone is looking for a new apartment, he signs up for distribution list A; if someone is looking for a new job, he signs up for distribution list B. Everyone gets only the information that interests them, exactly when it is published. This principle, also called `publish/subscribe`, is described by the Observer Pattern.

5.2 A First Realization

The sample project Observer_01 demonstrates a first realization of the pattern. You have someone who wants to announce news, an event source, in the example a job market. This class is able to register objects of the type `JobObserver` as interested parties, or observers; this type is defined by the interface `JobObserver`, which prescribes the method `newOffer()`.

```
public interface JobObserver {
    void newOffer(String job);
}
```

© The Author(s), under exclusive license to Springer Fachmedien Wiesbaden GmbH, part of Springer Nature 2023

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_5

The event source, i.e., the job market, defines three main methods: one to register observers and one to deregister them. Also, a method to save a new job and inform all observers about it. Saving jobs is not actually required in the sample implementation. But if you want to add evaluations about the reported jobs in the job market, for example, you will of course need it.

```
public class JobMarket {
    private final List<JobObserver> observerList =
                                      new ArrayList<>();
    private final List<String> jobList =
                                      new ArrayList<>();

    public void addJob(String job) {
        jobList.add(job);
        observerList.forEach(tempJobObserver -> {
            tempJobObserver.newOffer(job);
        });
    }

    public void addObserver(JobObserver jobObserver) {
        observerList.add(jobObserver);
    }

    public void removeObserver(JobObserver jobObserver) {
        observerList.remove(jobObserver);
    }
}
```

You can see in the code above that the information to all observers runs through the forEach method. And for each tempJobObserver from this list, the `newOffer()` method assigned via the lambda expression is then called. You can of course also use a common for loop at this point:

```
for(JobObserver temJobObserver : observerList)
    tempJobObserver.updateJob(job);
```

It provides the same functionality. The difference is that the for loop is an external iterator (which you write), while the forEach method is an internal iterator (which you only call because it's already built in). forEach also works with sets, queues, and maps, by the way.

The observers define what happens when the method `newOffer()` is called. The parameter of this method is the new job. In the example, the method randomly decides whether the observer applies for the job or not. Below you can see the abbreviated listing for the `Student` class. In addition, you will also find the class `Employee` in the project;

this is to model employees who are looking for a part-time job. They, too, must implement the `JobObserver` interface and thus their own version of the `newOffer()` method. In the example, I only made a small difference: students apply for a new job with 80% probability, employees only with 50% probability. After all, they already have a job.

```
public class Student implements JobObserver {
    @Override
    public void newOffer(String job) {
        var randomnumber = (int) (Math.random() * 10);
        var answer = "Student " + name;
        if (randomnumber <= 8)
            answer = answer + " applies for the job";
        else
            answer = answer + "does not apply";
        System.out.println(answer);
    }
}
```

The test program, which you will find in the project, creates two `JobObservers` and tests which of the two is informed about a new job under which circumstances. The expected console outputs can be found in the comments of the main method of the test driver.

5.3 Extending the Approach

Three things are certain in the life of a programmer: death, taxes, and changes in the requirements of one's software. In this project, too, the requirements will grow.

Tip
To maintain the greatest possible flexibility, always program against interfaces, never against implementations. Please note: In the language of design patterns, the term "interface" is not to be equated exclusively with an interface, and certainly not with interfaces in the sense of data transfer or function calls between systems. When we talk about an interface, we can also mean an abstract class. Consequently, you can "extend" or "implement" an interface, which describes the same process in the patterns language.

This expression has become customary. It would certainly be more appropriate to speak of "assuming a role" instead of "implementing an interface" – the `Student` class could then assume the role of a `JobObserver.`

You have already seen in the previous project Observer_01 the advantage that the job market has no idea what kind of objects it registers. For them it is sufficient that the objects are of the type `JobObserver` – these can be students or employees.

What change could there be? It could be that students should not only be able to register with the job exchange of the university, but also with a job exchange of the employment agency. In the sample project Observer_02 you will find the interface `JobProvider`, which is implemented by the event sources. For the client, in our case the main method of the test class, it can be useful under certain circumstances not to be bound to a concrete implementation (job exchange or employment agency), but to be bound in general to objects of the type `JobProvider`. It can now exchange the concrete implementations and, for example, post a job with both providers at the same time:

```
var jobMarket = new JobMarket();
var employmentAgency = new EmploymentAgency();
var providers = new JobProvider[] {
    jobMarket, employmentAgency
};

// … abridged

var job = "temp in theater";
System.out.println("\nNew job: " + job);
for (var tempProvider : providers)
    tempProvider.addJob(job);
```

Again, I use the type inference with "var". No matter if it says "`JobMarket jobMarket = new JobMarket();`" or "`JobProvider jobMarket = new JobMarket();`" or "`var jobMarket = new JobMarket();`", it will always create an object of type JobMarket. You can easily check this with a "`System.out.println(jobMarket.getClass());`" interspersed in between your code. As you can see, type inference also works with fields. The field with the two job providers is also created correctly.

A variation of this example is still conceivable: In the previous two examples, you passed a job as a string and passed it to the observers. This procedure is called push method. It is also possible that the event source only informs the observers that new information is available, without passing it immediately. It is then up to the observer to request the complete information from the event source. This procedure is called pull method.

In the example project Observer_03, the students and employees are not looking for jobs, but for apartments. Apartment offers are very extensive objects with their exposés and floor plans, which are only sent on request. The observers first decide randomly whether to request the information at all. Then they decide randomly whether they are interested in the apartment. Please analyze the project on your own.

5.4 Observer in the Java Class Library

In the Java class library, there is the interface `Observer`. This interface defines the role of an observer and prescribes the `update()` method. An object of type `Observable` and an object of general type `Object` are expected as parameters. A link to the event source is passed to the method to enable the observer to request further information from the event source. In addition, the event source can be queried for its data type (method `getClass()`). The parameter of type `Object` contains, for example, a job or an apartment.

The event source is therefore of the type `Observable`. Usually the suffix `-able` indicates an interface, but not here. `Observable` is a class that can be extended. As you are used to from your own implementation, there are methods in the `Observable` class to register and deregister Observers. Unlike your implementation, before you inform Observer, the `setChanged()` method must be called, which sets an internal flag. Only when this flag is set does the `notifyObservers()` method actually communicate the changes to the observers. This procedure can be useful if many changes are made to the database, but the observers are not to be informed until all changes have been completed.

However, the interface `Observer` and the class `Observable` are marked as "deprecated" and should no longer be used if possible. It is not type-safe and also not thread-safe.

For special Observer variants, the class library contains the package `java.util.Flow`, which is designed for thread-safe parallel processing of data and in which you will find the interfaces `Publisher` and `Subscriber`. Together with an `executor` and a `CompletableFuture`, the processing of many individual data packets from a source can be divided among parallel threads, which can then request their respective workload themselves. The code for this is very complex and beyond the scope of this book. However, you will find plenty of comprehensible examples on the Internet.

On the other hand, in the programming of user interfaces, events are triggered by the user to which the application must react: Pressing buttons, clicking with the mouse on buttons or menu items, or even inserting data into lists, which then have to trigger an update of the user interface again. Especially for the latter, there are also special observable interfaces for maps, sets, arrays or lists in the JavaFX library since version 2.0, e.g. javafx.collections.ObservableList.

We'll look at thread-safe observers in the next section, and we'll look at user interfaces after that as well.

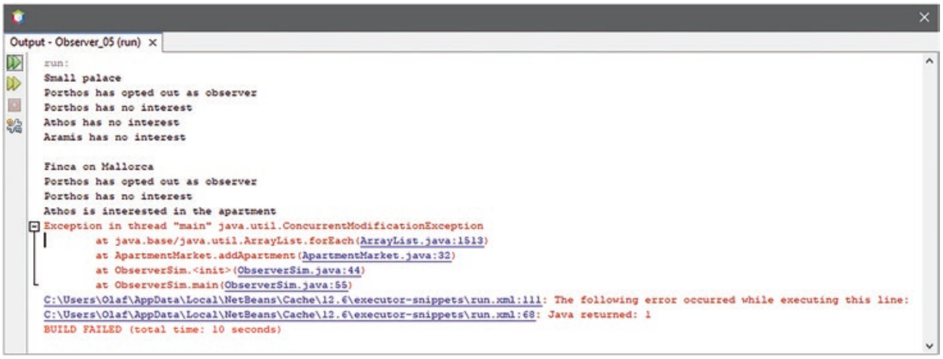
5.5 Concurrent Access

The solution you saw in the Observer_03 project works flawlessly. But if you want to work with it on a concurrent system, you run into problems. Imagine that a thread is handing over a new flat and all observers are informed about it. At the same time, another thread is trying to unsubscribe an observer. There are then two threads that want to access the list of observers at the same time. Open the sample project Observer_04. I have reduced it to the most necessary classes so that you have, for example, only workers and no students. Start the main method of the class ObserverSim. This version starts two threads (ApartmentThread and DeRegisterThread) shortly after each other, both of which can work with the list of observers. The described situation, that two threads want to access the database at the same time, is provoked here deliberately. A concurrentModificationException is thrown after a short time. You may have to start the main method several times.

Another problematic situation can arise when observers are informed about a change and decide to unsubscribe as observers. This situation can be found in the example project Observer_05. In the update method, a random decision is made whether the observer still wants to be informed:

```
@Override
public void updateFlat(ApartmentMarket provider) {
    var random number = (int) (Math.random() * 10);
    if (random number <= 6) {
        var apartment = provider.getDetail();
        if (random number < 5) {
            // … abridged
            provider.removeObserver(this);
        }
    }

    // … abridged
}
```

Also in this case, a `ConcurrentModificationException` is thrown at some point, as Fig. 5.1 shows. The problem here is that the notification comes from the iterator that goes through the list of all observers. If an observer tries to delete itself in the



```
run:
Small palace
Porthos has opted out as observer
Porthos has no interest
Athos has no interest
Aramis has no interest

Finca on Mallorca
Porthos has opted out as observer
Porthos has no interest
Athos is interested in the apartment
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1513)
        at ApartmentMarket.addApartment(ApartmentMarket.java:32)
        at ObserverSim.<init>(ObserverSim.java:44)
        at ObserverSim.main(ObserverSim.java:55)
C:\Users\Olaf\AppData\Local\NetBeans\Cache\12.6\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\Olaf\AppData\Local\NetBeans\Cache\12.6\executor-snippets\run.xml:68: Java returned: 1
BUILD FAILED (total time: 10 seconds)
```

Fig. 5.1 Concurrent access can throw an exception. (Screenshot from NetBeans)

meantime, it causes the method `removeObserver` to manipulate the list, which is being used by the iterator, which may have to notify other observers. And that's where the exception hits.

In the upcoming sections, I will show you how to avoid difficulties from concurrency.

5.5.1 Synchronize Accesses

The example project Observer_06 offers an obvious solution, namely to synchronize accesses to the database.

```
public class ApartmentMarket {
    private final List<ApartmentObserver> observerList =
                                   new ArrayList< >();

    // … abridged

    public synchronized void
                    addApartment(Apartment apartment) {
        // … abridged
    }
    public synchronized void
                addObserver(ApartmentObserver observer) {
        // … abridged
    }

    public synchronized void
              removeObserver(ApartmentObserver observer) {
        // … abridged
    }
}
```

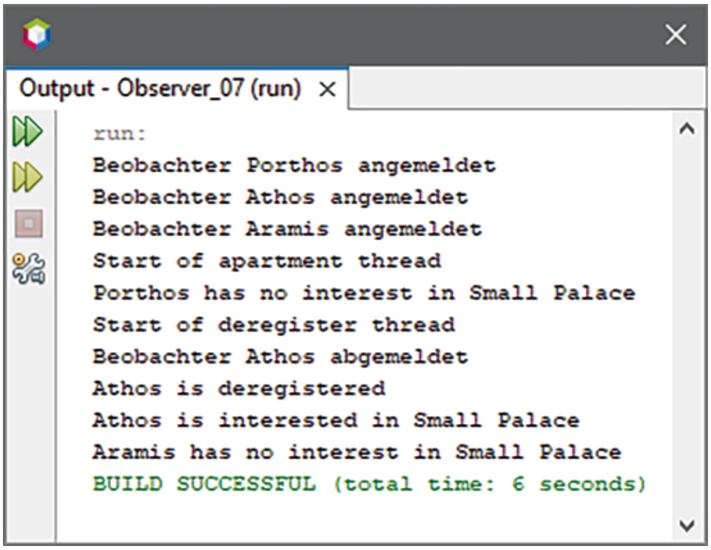
Does that solve all the problems? No! Observers cannot log in or log out as long as information is transmitted to observers. This solves the problem from project Observer_04. Project Observer_05 would still throw an exception. You also need to imagine the following situation: You are in the process of informing all your customers about an important new feature. Since the information is very extensive, it takes probably half an hour until the list of all observers is processed; it would be unpleasant if no new customers could register in this time. So, there must be another solution. By the way, the Observable class I introduced in Sect. 5.4 synchronizes the list of observers in exactly this way.

5.5.2 Copying the Database

The sample project Observer_07 shows you a different solution: The list of observers is copied before the notification.

```
public void addApartment(apartment apartment) {
    this.apartment = apartment;
    List<ApartmentObserver> tempList;
    synchronized (this) {
        tempList = List.copyOf(observerList);
    }
    for (var tempObserver : tempList) {
        tempObserver.updateFlat(this);
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            // do nothing
        }
    }
}
```

This blocks the list only for a short moment. When you send the notifications, the copied list is always informed; if an observer logs in or out of the original list at the same time, there is no conflict in access. The disadvantage of this solution is, of course, that the observer list has to be copied more or less effortfully with every update.



```
run:
Beobachter Porthos angemeldet
Beobachter Athos angemeldet
Beobachter Aramis angemeldet
Start of apartment thread
Porthos has no interest in Small Palace
Start of deregister thread
Beobachter Athos abgemeldet
Athos is deregistered
Athos is interested in Small Palace
Aramis has no interest in Small Palace
BUILD SUCCESSFUL (total time: 6 seconds)
```

Fig. 5.2 Observer logs off and is still informed again

At this point, by the way, I use a function that was added in Java 10, along with a few others: List.copyOf(). This is the easiest and fastest way to copy a list. The syntax is very simple, and the same functions exist for the Set and Map classes: Set.copyOf() and Map.copyOf().

If you test the project, you will also notice that an observer who has just logged off will still be informed again – to do this, run the main method a few times (see Fig. 5.2).

5.5.3 Use of a Thread-Safe List

If you do not want to copy by yourself, you can also use a thread-safe `CopyOnWriteArrayList`. The sample project Observer_08 demonstrates the use of this class.

```
public class ApartmentMarket {
    private final List<ApartmentObserver> observerList =
                                new CopyOnWriteArrayList<>();
    // … abridged
}
```

However, working with a `CopyOnWriteArrayList` is comparatively expensive, because whenever a new element is to be inserted or an existing one deleted, the complete database is copied. In this respect, the same criticism applies as in the Observer_07 project. However, there is no trivial solution to this problem, and at least the implementation in the library `java.util.concurrent` is very efficient; you don't have to worry about the exact implementation yourself. In common practice, the best solution is probably to resort to a `CopyOnWriteArrayList`.

5.6 Observer as Listener

You take a different approach when you wrap the event in another class; it's like putting a letter in an envelope and mailing it. The serializable class `EventObject` from the package `java.util` expects a reference to the object that sends the event as a parameter in the constructor. The `getSource()` method returns this source. The `toString method` from the `Object` class is meaningfully overridden. The interface `EventListener` from the same package does not prescribe any methods, but only defines a role.

In the ListenerDemo sample project, there is a `JobEvent` class that extends `EventObject`. It expects the source and a string object that describes the new job as parameters. Of course, you could also provide a flat here instead of the string. The source is passed on to the Super class, the job is stored in its own data field.

```
public final class JobEvent extends EventObject {
    private final String job;

    JobEvent(Object jobProvider, String job) {
        super(jobProvider);
        this.job = job;
    }

    @Override
    public String toString() {
        return job;
    }
}
```

The interface `JobListener` extends the interface `EventListener` and prescribes the method `updateJob()`, which must be passed an object of the type `JobEvent.` Two classes – the students and workers you are familiar with – implement this interface in their own individual ways.

```
public interface JobListener extends EventListener {
    void updateJob(JobEvent jobEvent);
}
```

The `JobAgency` class maintains a list of JobListeners. When a new job is posted, the class generates an event of the type `JobEvent` and transmits it to all JobListeners.

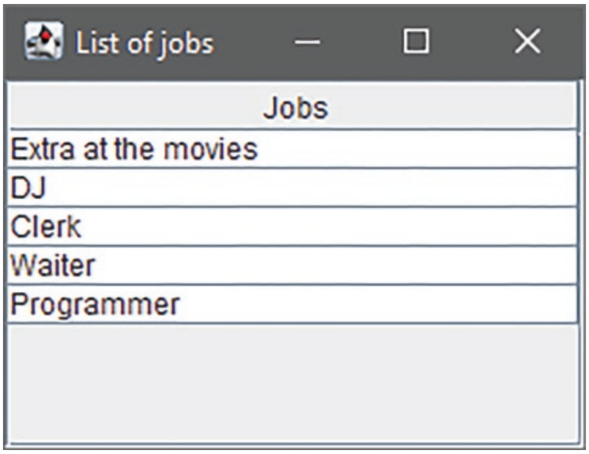
```
public class Employment Office {
    private final List<JobListener> listener =
                           new CopyOnWriteArrayList<>();

    public void addJob(String newJob) {
        JobEvent jobEvent = new JobEvent(this, newJob);
        listener.forEach((tempListener) -> {
            tempListener.updateJob(jobEvent);
        });
    }

    // … abridged
}
```

Let's look at how this variant can be used practically in the next section.

5.7 Listener in GUI Programming

If you look at the `EventObject` class in the API documentation, you will see that many event classes that are needed in GUI programming are derived from it: `ListDataEvent`, `ListSelectionEvent`, `AWTEvent`, and quite a few more. One event, the `TableModelEvent`, I would like to introduce now. In the last chapter I briefly showed how a `Jlist` queries its data from the database. The interaction of a `Jtable` and its database is similar: You have a database that you need to describe to the `Jtable` so that it is able to display the data. The bridge between the two is the `TableModel` interface. The `Jtable` expects an object of type `TableModel`, which you pass to it using the `setModel()` method. With this object, the `Jtable` registers itself as a listener, which is why you must implement the methods `addTableModelListener()` and `removeTableModel-Listener()`. You use `getColumnCount()` to return how many columns to display. The `getColumnName()` method returns the heading of a column. Since you can return any data type, `getColumnClass()` tells you what class an object in a particular column is based on. The `getRowCount()` method is equivalent to the `ListModel`'s `get-Size()` method; it returns the number of rows in a column. A cell in a table can always be edited; if you want to prevent this, the `isCellEditable()` method must return a `false value`. The methods `getValueAt()` and `setValueAt()` describe on the one hand which value is contained in a certain cell, on the other hand how the database should react when a cell has been edited. Each cell is uniquely defined by its column and row. In the MVC_1 sample project, the set jobs are displayed in a table (Fig. 5.3).

To generate this display, there are two classes, a display and a database. The database describes the data and the column header in a way that is understandable for the `Jtable` class. In addition, there are two methods that define what happens when a new job is posted. The `addJob()` method is passed a string containing the new job; it adds the job to the job list and calls the private `fireStateChanged()` method. This method generates a `TableModelEvent` and passes it to the listeners, which then update their display.


List of jobs

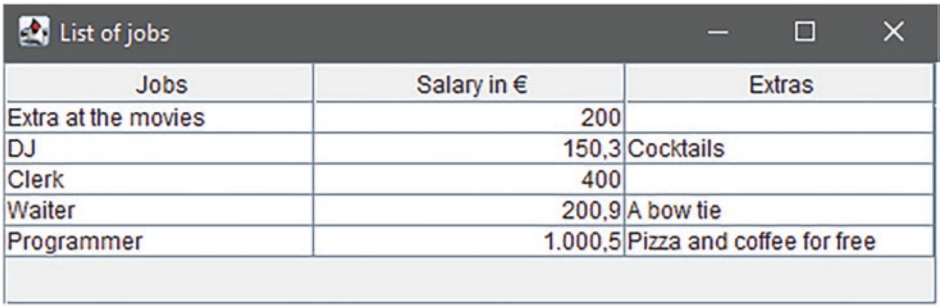
Jobs
Extra at the movies
DJ
Clerk
Waiter
Programmer

Fig. 5.3 Display of jobs in a table – Project MVC_1

```
public class Database implements TableModel {
    private final List<TableModelListener> listener =
                               new ArrayList<>();

    private final List<String> jobList =
                               new ArrayList<>();

    private final String[] headers =
                           new String[] {"Jobs"};

    public void addJob(String job) {
        jobList.add(job);
        this.fireStateChanged();
    }

    private void fireStateChanged() {
        TableModelEvent event =
                        new TableModelEvent(this);
        listener.forEach((tempListener) -> {
            tempListener.tableChanged(event);
        });
    }

    // … abridged
}
```

How is the data displayed? The display class builds a `JFrame` and passes an instance of the DataBase class to the `JTable` as a table model. Furthermore, it provides the method `addJob()`, which receives a job and passes it to the database.

```
public class Display {
    private final JFrame mainFrame =
                           new JFrame("Joblist");

    private final database jobModel = new database();

    Display() {
        var pnlDisplay = new JPanel();
        pnlDisplay.setLayout(new BorderLayout());
        var tblJobs = new JTable();
        tblJobs.setModel(jobModel);
        pnlDisplay.add(new JScrollPane(tblJobs),
                               BorderLayout.CENTER);
        mainFrame.getContentPane().add(pnlDisplay);
        mainFrame.setSize(500, 500);
```


List of jobs

Jobs	Salary in €	Extras
Extra at the movies	200	
DJ	150,3	Cocktails
Clerk	400	
Waiter	200,9	A bow tie
Programmer	1.000,5	Pizza and coffee for free

Fig. 5.4 Screenshot “JTable_Demo”

```
        mainFrame.setLocationRelativeTo(null);
mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setVisible(true);
    }

    void addJob(String job) {
        jobModel.addJob(job);
    }
}
```

The `main` method of the `Display` class adds a new job to the display instance every second. I refrain from printing, please analyze the sample code for yourself.

Excursus: Working with Table Models
A table is only really a table if it also contains multiple columns. A table with one column is a list and you have seen that in the last chapter. In the sample project JTable_Demo, the salary offered is displayed next to the job. Since good employees are not so easy to find anymore, employers in certain industries still have to come up with an extra bonus to find applicants. The figure Fig. 5.4 shows the program in action. Two things have changed: First, there are two more columns: the salary and the extra bonus. Furthermore, the salary is displayed right-justified, the other two columns left-justified.

The most important change in the project is that a job is now no longer just a string, but a separate data type `Job`. The description of the job, the salary and an extra bonus are stored in it. I have overloaded the `Job` constructor. Jobs that are in demand don’t need an extra bonus.

```
public class Job {
    final String description;
    final double salary;
    final String extra;

    // … abridged
```

```
        Job(String description, double salary, String extra) {
            this.description = description;
            this.salary = salary;
            this.extra = extra;
        }
    }
```

The `main` method of the start class again creates some jobs and passes them to the database:

```
    public static void main(String[] args) {
        Job[] jobs = new Job[] {
            new Job("Extra at the movies", 200),
            new Job("DJ", 150.30, "Cocktails"),
            new Job("Clerk", 400),
            new Job("Waiter", 200.90, "A bow tie"),
            new Job("Programmer", 1000.50, "Pizza and coffee for free")
        };
        // … abridged
    }
```

The last change concerns the data model. The array with the column headers has become larger. If you want to output the value of a specific cell, first have the job in this row returned. Then query the relevant value depending on the column.

```
    public class Database implements TableModel {
        private final List<TableModelListener> listener =
                                       new LinkedList<>();
        private final List<Job> jobList = new ArrayList<>();
        private final String[] headers = new String[] {
             "Jobs", "Salary in €", "Extra"
        };

        @Override
        public Object getValueAt(int rowIndex,
                                           int columnIndex) {
            Job job = jobList.get(rowIndex);
            return switch (columnIndex) {
                case 0 -> job.description;
                case 1 -> job.salary;
                case 2 -> job.extra == null ? "" : job.extra;
                default -> job.description;
            }
        }

        // … abridged
    }
```

Another important change relates to the specification of the data type of a column. The second column (index = 1) is of type `double`. All other columns are strings. By default, a `double value` is displayed by `JTable` right-justified, strings left-justified.

```
@Override
public Class<?> getColumnClass(int columnIndex) {
    return switch (columnIndex) {
        case 1 -> Double.class;
        default -> String.class;
    };
}
```

As you can see, the effort to describe the database increases with the number of columns. However, it is not difficult to implement a `TableModel`.

In the two program excerpts above you can see another Java novelty compared to Java 8: Switch Expressions. They have been available as a preview feature since Java 12, were supplemented by the `yield` keyword in Java 13 (which I will show you in a later chapter), and have been included in the Java standard since Java 14.

Take another close look at the two examples in bold. Several things stand out at first glance:

1. `Switch` is now available as an expression, in both cases now in the `return statement of` a method. But they can just as well be on the right side of an assignment or a comparison.
2. The colon in the switch statement is now replaced by an arrow (as in lambda expressions). Behind it in the two examples above is then exactly what would also be on the right side of an assignment or in a comparison.
3. There is no `break` (and also no `return`). Unlike the switch statement, the processing in the switch expression ends at the end of the respective expression for the selected case. The processing does not "fall through".
4. The default case – in the first code snippet – is identical to one of the normal cases, but must still be in its own case distinction.

The switch statement still exists. But with the newly added switch expressions, suitable code can be formulated much more elegantly and readably. It is also possible to process several statements in one case, which are then combined with curly brackets. To define the return value of the expression in this case, the `yield command` (instead of a `return` or a `break`) is used.

For the case distinction of `enum values` the compiler checks whether all values are either checked or alternatively a default case is available. Conversely, this means that the default case can be omitted when explicitly checking all cases. For all other types in the case distinction, however, a default case must be specified.

The `AbstractTableModel` class overrides the methods that are amenable to standard handling: registering and unregistering listeners, for example. By default, it also provides that cells may not be edited. You can (but don't have to) override these methods. This is the same principle as the Template Method pattern, where the `AbstractListModel` class did some of the work for you. In the directory for this chapter, you'll also find the AbstractTableModel_Demo sample project; here, the database is derived from an `AbstractTableModel.` There is no need to discuss the code – in this version, the database only needs the methods required to describe the data to be displayed. All other methods are inherited from the superclass.

Why have I described the `TableModel` here? In very few Java introductions I find explanations of the interface `TableModel`. Most of them describe the class `DefaultTableModel`. Maybe that's the reason why the `TableModel` leads a rather shadowy existence in practice. In practice, you will much more often find table models that rely on the `DefaultTableModel.` It is better documented and – at least at first glance – easier to use. In fact, however, the class is not without its problems. To name just one criticism, it uses vectors for the internal representation of the data – and that costs performance unnecessarily. I am therefore promoting the idea of developing your own table models on the basis of `TableModel,` and hope to be able to interest you a little in this.

5.8 The Model-View-Controller Pattern

The MVC_1 project leads to a discussion about the MVC pattern. MVC stands for model, view, and controller. Each of these entities has clearly defined tasks: The Model contains the data, the View is responsible for displaying it on the screen, and the Controller mediates between the two. The user – in the example the `main method` - always accesses the controller. If the data basis changes, it informs the logged-on listeners, the view units, of the change. The view then queries the data from the data basis and thus brings itself back into a consistent state. All components, for example `JList` and `JTree,` work according to this principle.

What is gained with this solution? You can register any number of observers – views – with the data model. Both data model and view are independent of each other and can be exchanged.

Where can the MVC model be verified in the MVC_1 project? The database, i.e. the model, is the `TableModelObject` with its job list. The `Display` class uses an instance of the `JTable` class to display the data on the screen. One might now be tempted to define the class `JTable` as a view. However, this assumption is wrong. But why?

The database expects an object of the type `TableModelListener` as listener. If the class `JTable` were the view, it would have to implement the interface `TableModelListener`. However, you can see from the API documentation that `JTable` does not implement this interface at all. So where is the listener? When you pass a table model to a `JTable object` with `setModel()`, it connects the database to an instance of the inner class `AccessibleJTable`. This inner class implements the `TableModelListener` interface and defines the `tableChanged()` method that you called in the database. The `tableChanged()` method queries the `TableModelEvent` for the scope of the change and tells the `JTable` superclass to redraw itself. So, the View in terms of MVC is this inner class `AccessibleJTable`; whereas the class `JTable` is the controller. Since the view accesses data and methods of the `JTable` class that encloses it, the boundaries of controller and view become blurred; you combine them into the delegate. In practice, Java programmers also tend to speak of model-delegate instead of model-view-controller. In Java introductions, the classes `JTable`, `JButton`, and so on, like to be represented as views. This is fine to explain the principle behind it. However, when looked at closely, this view is not correct.

Background Information
In the directory of this chapter, you will also find the sample project MVC_2. Here, the classes `Student` and `Employee` reappear, which you still know from further above. Both implement the `JobListener` interface, which extends the `TableModelListener` interface. So Student and Worker take on the role of the View and must override the `tableChanged()` method. I won't go into any more detail about the code; it's not difficult to understand.

Most books on design patterns – including the GoF – describe a statistic (the model) in the Observer Pattern, which is displayed once as a pie chart and then as a bar chart (view). In my example, the database is the job board, which is displayed by two different views. However, the principle is the same.

5.9 Observer – The UML Diagram

In Fig. 5.5 I show you an example of the UML diagram for the sample project Observer_08.

5.10 Summary

Go through the chapter again in key words:

- The Observer Pattern is always used when the status of an object changes and any number of other objects are to be informed of this.

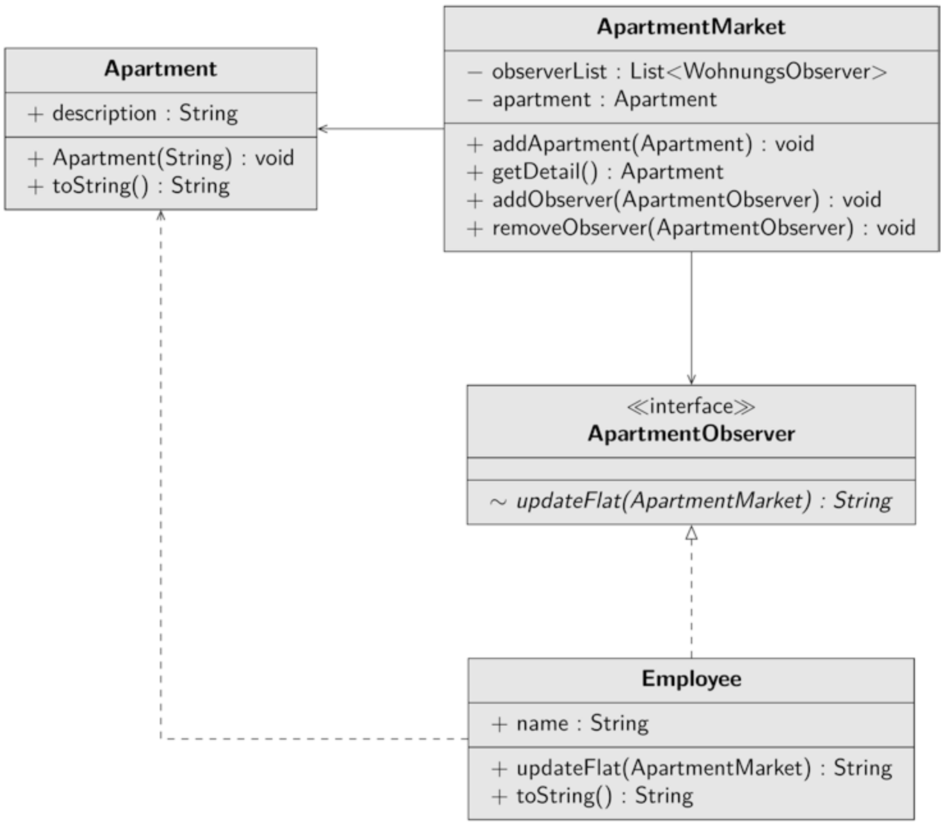

Apartment
+ description : String
+ Apartment(String) : void
+ toString() : String
ApartmentMarket
− observerList : List<WohnungsObserver>
− apartment : Apartment
+ addApartment(Apartment) : void
+ getDetail() : Apartment
+ addObserver(ApartmentObserver) : void
+ removeObserver(ApartmentObserver) : void
≪interface≫
ApartmentObserver
∼ updateFlat(ApartmentMarket) : String
Employee
+ name : String
+ updateFlat(ApartmentMarket) : String
+ toString() : String

Fig. 5.5 UML diagram of the Observer Pattern. (Sample project Observer_08)

- There is an event source and observers.
- Observers must be interchangeable.
- Two threads cannot access the list of observers at the same time.
- It is useful to copy the list or use a `CopyOnWriteArrayList` before informing the observers.
- The class library of Java (still) contains the class `Observable` and the interface `Observer`, whose use is not unproblematic.
- The GUI programming is based on the MVC pattern.
- MVC separates the responsibilities into model, view and controller.
- The model contains the database.
- The view displays the data.
- The controller mediates between the model and the view.
- In Java, the boundaries between view and controller are fluid, which is why we speak of a delegate.

5.11 Description of Purpose

The Gang of Four describes the purpose of the “Observer” pattern as follows:

> Define a 1-to-n dependency between objects so that changing the state of one object causes all dependent objects to be notified and automatically updated.