I
n the last issue of Java Magazine, I gave you a quick overview of the BlueJ environment and presented the tools that it offers for teaching programming concepts to beginners. This time, we will discuss a small programming example to see how BlueJ can help us understand some fundamental, but often difficult to grasp, programming concepts.

We will again use a practical programming example, which you can download and then play along with as we go.

The Clock Example
The project we will use to discuss the interaction of objects is a display for a digital clock. The display shows hours and minutes, separated by a colon (see Figure 1). 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, upon closer inspection, that building a 12-hour clock is slightly more difficult, so we will leave that until the end of this article.

Abstraction and Modularization
A first idea for many beginning programmers might be to implement the whole clock display in a single class. That is, after all, what classes are for: they represent things, and a clock is a thing.

However, here we shall approach this problem slightly differently. We will see whether we can identify subcomponents in the problem that we could turn into separate classes. The reason is complexity. As we progress in our programming practice, the programs we build will get more and more complex. Trivial tasks that we encounter as initial exercises 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. We divide the problem into subproblems and divide those again into more subproblems, and so on, until the individual problems are small enough to be easy to deal with. Once we solve one of the subproblems, we do not think about the details of that part anymore, but we treat the solution as a single building block for our next problem. This technique is sometimes referred to as divide and conquer.

Let’s discuss this with an example. Imagine engineers in a car company designing a new car. One engineer might 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 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, and so on. She will think of the engine not as a single entity, but as a complex work of many parts. One of these parts might be a spark plug.

There is an engineer (maybe he works 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 might spend her days thinking about the chemical composition to produce the right materials for making 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 reason why cars are built successfully is that the engineers use modularization and abstraction. They divide the car into independent modules (wheel, engine, gearbox, seat, steering wheel, and so on) 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.

### Modularization and Abstraction in Software

Modularization and abstraction are also used in software development. To maintain an overview in complex programs, we try to identify subcomponents that we can program as independent entities. Then we try to use those subcomponents as if they were simple parts without being concerned about their inner complexities.

In object-oriented programming, these components and subcomponents 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, gearbox, 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. In this article, I will show you how BlueJ can help you build, test, and experiment with these kinds of components.

**Examining the Project**

Open the clock-display project, which you downloaded, in BlueJ. Initially, you will see a very simple class diagram (see Figure 2). We can see that our project consists of two classes named `NumberDisplay` and `ClockDisplay`.

Figure 2
NumberDisplay is a class that represents a single, two-digit display of a number (see Figure 3). We will see that we can use it to build our clock display. In fact, we can use two instances of the number display: one for the hours and one for the minutes. This way, we are breaking down our problem of building a clock display into smaller, easier-to-solve problems.

We first build a class for a two-digit number display, and then we build the clock display by taking two number displays (one for the hours and one for the minutes) and sticking them together. Each of the subproblems is easier to solve than the whole.

The NumberDisplay class represents objects that can store a number value up to a given limit. It can be incremented, and when the value reaches that limit, it rolls around back to zero. We will use one of these objects with a limit of 60 for the minutes and another one with a limit of 24 for the hours.

In BlueJ, we can inspect the object in two different ways: we can read the source code, or we can experiment with the object by invoking its methods. Let’s start with the latter.

Create an instance of the class NumberDisplay by right-clicking the NumberDisplay class in your diagram and selecting the first entry in the menu, which is titled new NumberDisplay(int rollOverLimit) (see Figure 4). By using this menu command, you are invoking the class’s constructor to create an object.

A dialog box will pop up and prompt you to enter a name for the instance and a value for the constructor parameter. Here we can specify the limit at which we want the display to roll over. Let’s leave the instance name as it is suggested, and use 60 for the parameter. Once you click OK, a NumberDisplay object will appear on the object bench (as shown in Figure 2).

We can now experiment with the NumberDisplay by right-clicking the object and trying out its methods (see Figure 5).

We can see that the object has four methods. Try them out by selecting each one of them. You will see that they do the following:

- setValue(int replacementValue) lets you specify a value for this display.
- getValue() returns the current value.
- getDisplayValue() returns the same value (as a String), padded with a leading zero to ensure that the result is always a two-digit number (for example, “04” instead of “4”).
- increment() increments the value by one.

Another way to experiment with the NumberDisplay object is to open the object inspector by using the Inspect function from the object’s menu (see Figure 6).

Here, we can see that the NumberDisplay object holds two fields: one for the current value and one for its limit. Leave the object inspector open, and call some of the object’s methods again. Now you can see the value change and observe the effects of the methods. An interesting thing to try is to call the increment method when the value is just below the limit. Try it out!

Another, more traditional way to inspect a class is to read its source code. Open the source code for the NumberDisplay class (by double-clicking the class icon) and read through it. You will see how each of the four methods was implemented.

Examining ClockDisplay

Through experimentation with the NumberDisplay object, we can get a feeling for the behavior and functionality of that object. Now we will look at how the ClockDisplay class uses two NumberDisplay objects to create a clock display.

Let’s start again by creating an...
object and experimenting with its methods:

- Create an object of class `ClockDisplay` by right-clicking the class and selecting its constructor. You will see that there are two constructors—one with parameters and one without. Try both.
- Right-click the `ClockDisplay` object on the object bench and try out its methods. You will see that there are three: `getTime`, `setTime`, and `timeTick`.
- Open the object inspector for the `ClockDisplay` object (see Figure 7) by choosing Inspect from the menu, and observe its fields while you call its methods. If you do this, you will observe a number of things:
  - The `ClockDisplay` object has two fields of type `NumberDisplay`, named `hours` and `minutes`, for storing its time.
  - The `displayString` field in the `ClockDisplay` object shows the current time of the clock. We use this in our project to simulate the display of our clock. In a real clock, this is the time that would be shown on its display.
  - The `timeTick` method does not seem to do anything.

### Inspecting the Source Code

Let’s start by inspecting the `ClockDisplay` source code. Open its editor by double-clicking the `ClockDisplay` class icon.

Looking through the code, we can observe the following interesting facts. Three fields are declared, two of type `NumberDisplay` and one of type `String`:

```java
private NumberDisplay hours;
private NumberDisplay minutes;
private String displayString;
```

The first two hold the `NumberDisplay` objects for the hours and minutes, and the last represents the current display of the clock. The class’s constructor initializes the three fields:

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

Examine the source code of the other methods to familiarize yourself with how they work. You will see that their implementation is not very complex. You will also see that the method `timeTick` is empty, explaining why it does not yet work. We will now implement this method.

The purpose of the `timeTick` method is to advance the clock by one minute. We can achieve this by adding the following line of code to its body:

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

This is almost all that’s required, except for one thing: we have to deal with the case where the hours are also incremented (for example, the time advances from 03:59 to 04:00). We know that this should happen when the minutes roll over to zero, so we can add the following:

```
if(minutes.getValue() == 0) {
    hours.increment();
}
```

And after doing this increment, we also have to update the display value:

```
updateDisplay();
```

This is the whole implementation. After adding this to your class, your `timeTick` method should look like the one shown in Listing 1.

### Do It Yourself

Now try it out yourself. Type the code as we have shown it here, and compile the class (click the Compile button either in the editor window or in the main window). If you get any errors, carefully inspect your code and fix them.

Now create a `ClockDisplay` object again, open its object inspector, and experiment with its methods. In particular, try the newly implemented `timeTick` method. You will see that it should work. Try it also at a point where the hour changes and make sure that works as well.

### And Then Do More . . .

I have shown you how to examine an existing project with BlueJ and how to make a small improvement. As a programming exercise, you can now try several further improvements, for example:

- Display a warning when a user...
tries to set the time to an invalid value.
- Add seconds to the clock display.
- Modify the clock to show a U.S.-style 12-hour display (04:23 p.m.) instead of the European-style 24-hour version (16:23).
Each of these tasks makes a nice little programming exercise for beginners who know the basics and want to practice their programming skills.

Conclusion
Today, I have presented a small programming example that started with examining a given body of code, progressed to gaining an understanding of existing functionality, and finished by extending the project with new functionality. We can clearly see now how BlueJ helps in the learning of fundamental concepts.

For teachers and learners, several aspects of the BlueJ environment are crucial to the process of learning:
- We can look into objects using the object inspector.
- We can easily test and experiment using the object bench.
- We can concentrate on the object structure and logic of the program first without the distraction of a GUI or a main method. This teaches good object-oriented principles before tackling tedious detail. (Of course, GUIs and main methods can be added as well.) If you are a teacher, or someone who wants to teach someone object-oriented programming, you probably get the idea. BlueJ offers a unique toolset that is much better suited for learning the fundamentals of programming than big, professional environments, such as NetBeans or Eclipse. It has just the right kind and right amount of tool support needed for programmers in their first year, and developers can then graduate from BlueJ to the professional environments once they have developed a good mental model of object orientation.

So far, I have just shown the basics of the BlueJ tools. In the next episode, we will look at specialized support for testing.

%f  "Download all listings in this issue as text"

\*
\*
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();
}