to be compiled by clicking the *Compile* button. (You may like to read the 'About compilation' note for more information about what is happening when you compile a class.) Once a class has been compiled, objects can be created again and you can try out your change.

About compilation

When people write computer programs, they typically use a 'higher level' programming language, such as Java. A problem with that is that a computer cannot execute Java source code directly. Java was designed to be reasonably easy to read for humans, not for computers. Computers, internally, work with a binary representation of a machine code, which looks quite different from Java. The problem for us is that it looks so complex that we do not want to write it directly. We prefer to write Java. What can we do about this?

The solution is a program called the *compiler*. The compiler translates the Java code into machine code. We can write Java, run the compiler – which generates the machine code – and the computer can then read the machine code. As a result, every time we change the source code, we must first run the compiler before we can use the class again to create an object. Otherwise the machine code version that the computer needs will not exist.

Exercise 1.13 In the source code of class `Picture`, find the part that actually draws the picture. Change it so that the sun will be blue rather than yellow.

Exercise 1.14 Add a second sun to the picture. To do this, pay attention to the field definitions close to the top of the class. You will find this code:

```
private Square wall;
private Square window;
private Triangle roof;
private Circle sun;
```

You need to add a line here for the second sun. For example:

```
private Circle sun2;
```

Then write the appropriate code for creating the second sun.

Exercise 1.15 *Challenge exercise* (This means that this exercise might not be solved quickly. We do not expect everyone to be able to solve this at the moment. If you do – great. If you don't, then don't worry. Things will become clearer as you read on. Come back to this exercise later.) Add a sunset to the single-sun version of `Picture`. That is, make the sun go down slowly. Remember: The circle has a method `slowMoveVertical` that you can use to do this.

Exercise 1.16 *Challenge exercise* If you added your sunset to the end of the `draw` method (so that the sun goes down automatically when the picture is drawn), change this now. We now want the sunset in a separate method, so that we can call `draw` and see the picture with the sun up, and then call `sunset` (a separate method!) to make the sun go down.

Before attempting these exercises, be sure that you have a good understanding of how ticket machines behave, and how that behavior is implemented through the fields, constructor, and methods of the class.

Exercise 2.39 Modify the constructor of `TicketMachine` so that it no longer has a parameter. Instead, the price of tickets should be fixed at 1000 cents. What effect does this have when you construct ticket machine objects within BlueJ?

Exercise 2.40 Implement a method, `empty`, that simulates the effect of removing all money from the machine. This method should have a `void` return type, and its body should simply set the `total` field to zero. Does this method need to take any parameters? Test your method by creating a machine, inserting some money, printing some tickets, checking the total, and then emptying the machine. Is this method a mutator or an accessor?

Exercise 2.41 Implement a method, `setPrice`, that is able to set the price of tickets to a new value. The new price is passed in as a parameter value to the method. Test your method by creating a machine, showing the price of tickets, changing the price, and then showing the new price. Is this method a mutator?

Exercise 2.42 Give the class two constructors. One should take a single parameter that specifies the price, and the other should take no parameter and set the price to be a default value of your choosing. Test your implementation by creating machines via the two different constructors.

2.10 Reflecting on the design of the ticket machine

In the next few sections, we shall examine the implementation of an improved ticket machine class that attempts to deal with some of the inadequacies of the naïve implementation.

From our study of the internals of the `TicketMachine` class, you should have come to appreciate how inadequate it would be in the real world. It is deficient in several ways:

- It contains no check that the customer has entered enough money to pay for a ticket.
- It does not refund any money if the customer pays too much for a ticket.
- It does not check to ensure that the customer inserts sensible amounts of money: experiment with what happens if a negative amount is entered, for instance.
- It does not check that the ticket price passed to its constructor is sensible.

If we could remedy these problems, then we would have a much more functional piece of software that might serve as the basis for operating a real-world ticket machine. In order to see that we can improve the existing version, open the *better-ticket-machine* project. As before, this project contains a single class – `TicketMachine`. Before looking at the internal details of the class, experiment with it by creating some instances and see whether you notice any differences in behavior between this version and the previous naïve version. One specific difference is that the new version has one additional method, `refundBalance`. Later in this chapter we shall use this method to introduce an additional feature of Java, so take a look at what happens when you call it.

2.14 Fields, parameters, and local variables

With the introduction of `amountToRefund` in the `refundBalance` method, we have now seen three different kinds of variable: fields, formal parameters, and local variables. It is important to understand the similarities and differences between these three kinds. Here is a summary of their features:

- All three kinds of variable are able to store a value that is appropriate to their defined type. For instance, a defined type of `int` allows a variable to store an integer value.
- Fields are defined outside constructors and methods.
- Fields are used to store data that persist throughout the life of an object. As such, they maintain the current state of an object. They have a lifetime that lasts as long as their object lasts.
- Fields have class scope: their accessibility extends throughout the whole class, and so they can be used within any of the constructors or methods of the class in which they are defined.
- As long as they are defined as `private`, fields cannot be accessed from anywhere outside their defining class.
- Formal parameters and local variables persist only for the period that a constructor or method executes. Their lifetime is only as long as a single call, so their values are lost between calls. As such, they act as temporary rather than permanent storage locations.
- Formal parameters are defined in the header of a constructor or method. They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call.
- Formal parameters have a scope that is limited to their defining constructor or method.
- Local variables are defined inside the body of a constructor or method. They can be initialized and used only within the body of their defining constructor or method. Local variables must be initialized before they are used in an expression – they are not given a default value.
- Local variables have a scope that is limited to the block in which they are defined. They are not accessible from anywhere outside that block.

Exercise 2.55 Add a new method, `emptyMachine`, that is designed to simulate emptying the machine of money. It should both return the value in `total` and reset `total` to be zero.

Exercise 2.56 Is `emptyMachine` an accessor, a mutator, or both?

Exercise 2.57 Rewrite the `printTicket` method so that it declares a local variable, `amountLeftToPay`. This should then be initialized to contain the difference between `price` and `balance`. Rewrite the test in the conditional statement to check the value of `amountLeftToPay`. If its value is less than or equal to zero, a ticket should be printed, otherwise an error message should be printed stating the amount still required. Test your version to ensure that it behaves in exactly the same way as the original version.

Exercise 2.58 *Challenge exercise* Suppose we wished a single `TicketMachine` object to be able to issue tickets with different prices. For instance, users might press a button on the physical machine to select a particular ticket price. What further methods and/or fields would need to be added to `TicketMachine` to allow this kind of functionality? Do you think that many of the existing methods would need to be changed as well?

Save the *better-ticket-machine* project under a new name and implement your changes to the new project.

2.15 Summary of the better ticket machine

In developing a more sophisticated version of the `TicketMachine` class, we have been able to address the major inadequacies of the naïve version. In doing so, we have introduced two new language constructs: the conditional statement and local variables.

- A conditional statement gives us a means to perform a test and then, on the basis of the result of that test, perform one or other of two distinct actions.
- Local variables allow us to calculate and store temporary values within a constructor or method. They contribute to the behavior that their defining method implements, but their values are lost once that constructor or method finishes its execution.

You can find more details of conditional statements and the form that their tests can take in Appendix D.

2.16 Self-review exercises

This chapter has covered a lot of new ground and we have introduced a lot of new concepts. We will be building on these in future chapters, so it is important that you are comfortable with them. Try the following pencil-and-paper exercises as a way of checking that you are becoming used to the terminology that we have introduced in this chapter. Don't be put off by the fact that we suggest you do these on paper rather than within BlueJ. It will be good practice to try things out without a compiler.

Exercise 2.59 List the name and return type of this method:

```
public String getCode()
{
    return code;
}
```