Exercise 9.24 Repeat the previous exercise with the **`testMinus`** method. Does it always give the same result?

The experiments above should have alerted you to the fact that not all seems to be right with the **`CalcEngine`** class. It looks like it contains some errors. But what are they, and how can we find them? In the sections that follow, we shall consider a number of different ways in which we can try to locate where errors are occurring in a class.

9.8 Manual walkthroughs

Manual walkthroughs are a relatively underused technique, perhaps because they are a particularly "low-tech" debugging and testing technique. However, do not let this fool you into thinking that they are not useful. A manual walkthrough involves printing copies of the classes you are trying to understand or debug, then getting away from your computer! It is all too easy to spend a lot of time sitting in front of a computer screen not making much progress in trying to deal with a programming problem. Relocating and refocusing your efforts can often free your mind to attack a problem from a completely different direction. We have often found that going off to lunch or cycling home from the office brings enlightenment that has otherwise eluded us through hours of slogging away at the keyboard.

Concept

A **walkthrough** is an activity of working through a segment of code line by line while observing changes of state and other behavior of the application.

A walkthrough involves both reading classes and tracing the flow of control between classes and objects. This helps us understand both how objects interact with one another and how they behave internally. In effect, a walkthrough is a pencil-and-paper simulation of what happens inside the computer when you run a program. In practice, it is best to focus on a narrow portion of an application, such as a single logical grouping of actions or even a single method call.

9.8.1 A high-level walkthrough

We shall illustrate the walkthrough technique with the *calculator-engine* project. You might find it useful to print out copies of the **`CalcEngine`** and **`CalcEngineTester`** classes in order to follow through the steps of this technique.

We shall start by examining the **`testPlus`** method of the **`CalcEngineTester`** class, as it contains a single logical grouping of actions that should help us gain an understanding of how several methods of the **`CalcEngine`** class work together to fulfill the computation role of a calculator. As we work our way through it, we shall often make penciled notes of questions that arise in our minds.

1. For this first stage, we do not want to delve into too much detail. We simply want to look at how the **`testPlus`** method uses an engine object, without exploring the internal details of the engine. From earlier experimentation, it would appear that there are some errors to be found, but we do not know whether the errors are in the tester or the engine. So the first step is to check that the tester appears to be using the engine appropriately.

2. We note that the first statement of **testPlus** assumes that the **engine** field already refers to a valid object:

```
engine.clear();
```

   We can verify that this is the case by checking the tester's constructor. It is a common error for an object's fields not to have been initialized properly, either in their declarations or in a constructor. If we attempt to use a field with no associated object, then a **NullPointerException** is a likely runtime error.

3. The first statement's call to **clear** appears to be an attempt to put the calculator engine into a valid starting state, ready to receive instructions to perform a calculation. This looks like a reasonable thing to do, equivalent to pressing a "reset" or "clear" key on a real calculator. At this stage, we do not look at the engine class to check exactly what the **clear** method does. That can wait until we have achieved a level of confidence that the tester's actions are reasonable. Instead, we simply make a penciled note to check that **clear** puts the engine into a valid starting state as expected.

4. The next statement in **testPlus** is the entry of a digit via the **numberPressed** method:

```
engine.numberPressed(3);
```

   This is reasonable, as the first step in making a calculation is to enter the first operand. Once again, we do not look to see what the engine does with the number. We simply assume that it stores it somewhere for later use in the calculation.

5. The next statement calls **plus**, so we now know that the full value of the left operand is 3. We make a penciled note of this fact on the printout, or make a tick against this assertion in one of the comments of **testPlus**. Similarly, we should note or confirm that the operation being executed is addition. This seems like a trivial thing to do, but it is very easy for a class's comments to get out of step with the code they are supposed to document. So checking the comments at the same time as we read the code can help us avoid being misled by them later.

6. Next, another single digit is entered as the right operand by a further call to **numberPressed**.

7. Completion of the addition is requested by a call to the **equals** method. We make a penciled note that, from the way it has been used in **testPlus**, the **equals** method appears not to return the expected result of the calculation. This is something else that we can check when we look at **CalcEngine**.

8. The final statement of **testPlus** obtains the value that should appear in the calculator's display:

```
return engine.getDisplayValue();
```

   Presumably, this is the result of the addition, but we cannot know that for sure without looking in detail at **CalcEngine**. Once again, we make a note to check that this is indeed the case.

With our examination of **testPlus** completed, we have gained a reasonable degree of confidence that it uses the engine appropriately: that is, simulating a recognizable sequence of key presses to complete a simple calculation. We might remark that the method is not particularly ambitious—both operands are single-digit numbers, and only a single operator is used. However, that is not unusual in test methods, because it is important to test for the most basic functionality before testing more complex combinations. Nevertheless, some more complex tests should be added to the tester at some stage.

Exercise 9.25 Perform a similar walkthrough of your own with the **testMinus** method. Does that raise any further questions in your mind about things you might like to check when looking at **CalcEngine** in detail?

Before looking at the **CalcEngine** class, it is worth walking through the **testAll** method to see how it uses the **testPlus** and **testMinus** methods we have been looking at. From this, we observe the following:

1. The **testAll** method is a straight-line sequence of print statements.
2. It contains one call to each of **testPlus** and **testMinus**, and the values they return are printed out for the user to see. We might note that there is nothing to tell the user what the results should be. This makes it hard for the user to confirm that the results are correct.
3. The final statement boldly states:

   ```
   All tests passed.
   ```

   but the method contains no tests to establish the truth of this assertion! There should be a proper means of establishing both what the result values should be, and whether they have been calculated correctly or not. This is something we should remedy as soon as we have the chance to get back to the source of this class.

At this stage, we should not be distracted by the final point into making changes that do not directly address the errors we are looking for. If we make those sorts of changes, we could easily end up masking the errors. One of the crucial requirements for successful debugging is to be able to trigger the error you are looking for easily and reproducibly. When that is the case, it is much easier to assess the effect of an attempted correction.

Having checked over the test class, we are in a position to examine the source of the **CalcEngine** class. We can do so armed with a reasonable sequence of method calls to explore from the walkthrough of the **testPlus** method, as well as with a set of questions thrown up by it.

9.8.2 Checking state with a walkthrough

A **CalcEngine** object is quite different in style from its tester. This is because the engine is a completely passive object. It initiates no activity of its own, but simply responds to external method calls. This is typical of the server style of behavior. Server objects often rely heavily on their state to determine how they should respond to method calls. This is particularly true of the calculator engine. So an important part of conducting the walkthrough is to be sure that we always have an accurate representation of its state. One way to do this on paper is by making up a table of an object's fields and their values (Figure 9.7). A new line can be entered to keep a running log of the values following each method call.

This technique makes it quite easy to check back if something appears to go wrong. It is also possible to compare the states after two calls to the same method.

1. As we start the walkthrough of **CalcEngine**, we document the initial state of the engine, as in the first row of values in Figure 9.7. All of its fields are initialized in the constructor. As we observed when walking through the tester, object initialization is important, and we make a note here to check that the default initialization is sufficient—particularly as the default value of **previousOperator** would appear not to represent a meaningful operator. Furthermore, this makes us think about whether it really is meaningful to have a *previous* operator before the first real operator in a calculation. In noting these questions, we do not necessarily have to try to discover the answers right away, but they provide prompts as we discover more about the class.

2. The next step is to see how a call to **clear** changes the engine's state. As shown in the second data row of Figure 9.7, the state remains unchanged at this point because **displayValue** is already set to 0. But we note another question here: Why is the value of only one of the fields set by this method? If this method is supposed to implement a form of reset, why not clear all of the fields?

3. Next, a call to **numberPressed** with an actual parameter of 3 is investigated. The method multiplies an existing value of **displayValue** by 10 and then adds in the new digit. This correctly models the effect of appending a new digit onto the right-hand end of an existing number. It relies on **displayValue** having a sensible initial value of 0 when the first digit of a new number is entered, and our investigation of the **clear** method gives us confidence that this will be the case. So this method looks all right.

4. Continuing to follow the order of calls in the **testPlus** method, we next look at **plus**. Its first statement calls the **applyPreviousOperator** method. Here we have to decide whether to continue ignoring nested method calls or whether to break off and see what it does. Taking a quick look at the **applyPreviousOperator** method, we can see that

Figure 9.7
Informal tabulation of an object's state

Method called	displayValue	leftOperand	previousOperator
initial state	0	0	' '
clear	0	0	' '
numberPressed(3)	3	0	' '

it is fairly short. Furthermore, it is clearly going to alter the state of the engine, and we shall not be able to continue documenting the state changes unless we follow it up. So we would certainly decide to follow the nested call. It is important to remember where we came from, so we mark the listing just inside the **`plus`** method before following through the **`applyPreviousOperator`** method. If following a nested method call is likely to lead to further nested calls, we need to use something more than a simple mark to help us find our way back to the caller. In that case, it is better to mark the call points with ascending numerical values, reusing previous values as calls return.

5 The **`applyPreviousOperator`** method gives us some insights into how the **`previousOperator`** field is used. It also appears to answer one of our earlier questions: whether having a space as the initial value for the previous operator was satisfactory. The method explicitly checks to see whether **`previousOperator`** contains either a + or a − before applying it. So another value will not result in an incorrect operation being applied. By the end of this method, the value of **`leftOperand`** will have been changed, so we note its new value in the state table.

6 Returning to the **`plus`** method, the remaining two fields have their values set, so the next row of the state table contains the following values:

```
plus  0  3  '+'
```

The walkthrough of the engine can be continued in a similar fashion, by documenting the state changes, gaining insights into its behavior, and raising questions along the way. The following exercises should help you complete the walkthrough.

Exercise 9.26 Complete the state table based on the following subsequent calls found in the **`testPlus`** method:

```
numberPressed(4);
equals();
getDisplayValue();
```

Exercise 9.27 When walking through the **`equals`** method, did you feel the same reassurances that we felt in **`applyPreviousOperator`** about the default value of **`previousOperator`**?

Exercise 9.28 Walkthrough a call to **`clear`** immediately following the call to **`getDisplayValue`** at the end of your state table, and record the new state. Is the engine in the same state as it was at the previous call to **`clear`**? If not, what impact do you think this could have on any subsequent calculations?

Exercise 9.29 In the light of your walkthrough, what changes do you think should be made to the **`CalcEngine`** class? Make those changes to a paper version of the class, and then try the walkthrough all over again. You should not need to walk through the **`CalcEngineTester`** class, just repeat the actions of its **`testAll`** method.