**Exercise 9.30** Try a walkthrough of the following sequence of calls on your corrected version of the engine:

```
clear();
numberPressed(9);
plus();
numberPressed(1);
minus();
numberPressed(4);
equals();
```

What should the result be? Does the engine appear to behave correctly and leave the correct answer in **`displayValue`**?

### 9.8.3 Verbal walkthroughs

Another way in which the walkthrough technique can be used to find errors in a program is to try explaining to another person what a class or method does. This works in two completely different ways:

- The person you explain the code to might spot the error for you.
- You will often find that the simple act of trying to put into words what a piece of code should do is enough to trigger in your own mind an understanding of why it does not.

This latter effect is so common that it can often be worth explaining a piece of code to someone who is completely unfamiliar with it—not in anticipation that *they* will find the error, but that *you* will!

## 9.9 Print statements

Probably the most common technique used to understand and debug programs—even amongst experienced programmers—is to annotate methods temporarily with print statements. Print statements are popular because they exist in most languages, they are available to everyone, and they are very easy to add with any editor. No additional software or language features are required to make use of them. As a program runs, these additional print statements will typically provide a user with information such as:

- which methods have been called
- the values of parameters
- the order in which methods have been called
- the values of local variables and fields at strategic points

Code 9.5 shows an example of how the **`numberPressed`** method might look with print statements added. Such information is particularly helpful in providing a picture of the way in which the state of an object changes as mutators are called. To help support this, it is often worth including a debugging method that prints out the current values of all the fields of an object. Code 9.6 shows such a **`reportState`** method for the **`CalcEngine`** class.

**Code 9.5**
A method with debugging print statements added

```java
/**
 * A number button was pressed.
 * @param number The single digit.
 */
public void numberPressed(int number)
{
    System.out.println("numberPressed called with: " + number);

    displayValue = displayValue * 10 + number;

    System.out.println("displayValue is: " + displayValue +
                       "at end of numberPressed");
}
```

**Code 9.6**
A state-reporting method

```java
/**
 * Print the values of this object's fields.
 * @param where Where this state occurs.
 */
public void reportState(String where)
{
    System.out.println("displayValue: " + displayValue +
                       " leftOperand: " + leftOperand +
                       " previousOperator: " + previousOperator +
                       " at " + where);
}
```

If each method of **CalcEngine** contained a print statement at its entry point and a call to **reportState** at its end, Figure 9.8 shows the output that might result from a call to the tester's **testPlus** method. (This was generated from a version of the calculator engine that can be found in the *calculator-engine-print* project.) Such output allows us to build up a picture of how control flows between different methods. For instance, we can see from the order in which the state values are reported that a call to **plus** contains a nested call to **applyPreviousOperator**.

Print statements can be very effective in helping us understand programs or locate errors, but there are a number of disadvantages:

- It is not usually practical to add print statements to every method in a class. So they are only fully effective if the right methods have been annotated.
- Adding too many print statements can lead to information overload. A large amount of output can make it difficult to identify what you need to see. Print statements inside loops are a particular source of this problem.
- Once their purpose has been served, it can be tedious to remove them.
- There is also the chance that, having removed them, they will be needed again later. It can be very frustrating to have to put them back in again!

**Figure 9.8**
Debugging output from a call to **`testPlus`**

```
clear called
displayValue: 0 leftOperand: 0 previousOperator: at end of clear
numberPressed called with: 3
displayValue: 3 leftOperand: 0 previousOperator: at end of number...
plus called
applyPreviousOperator called
displayValue: 3 leftOperand: 3 previousOperator: at end of apply...
displayValue: 0 leftOperand: 3 previousOperator: + at end of plus
numberPressed called with: 4
displayValue: 4 leftOperand: 3 previousOperator: + at end of number...
equals called
displayValue: 7 leftOperand: 0 previousOperator: + at end of equals
```

**Exercise 9.31** Open the *calculator-engine-print* project and complete the addition of print statements to each method and the constructor.

**Exercise 9.32** Create a **`CalcEngineTester`** in the project and run the **`testAll`** method. Does the output that results help you identify where the problem lies?

**Exercise 9.33** Do you feel that the amount of output produced by the fully annotated **`CalcEngine`** class is too little, too much, or about right? If you feel that it is too little or too much, either add further print statements or remove some until you feel that you have the right level of detail.

**Exercise 9.34** What are the respective advantages and disadvantages of using manual walkthroughs or print statements for debugging? Discuss.

## 9.9.1 Turning debugging information on or off

If a class is still under development when print statements are added, we often do not want to see the output every time the class is used. It is best if we can find a way to turn the printing on or off as required. The most common way to achieve this is to add an extra **`boolean`** debugging field to the class, then make printing dependent upon the value of the field. Code 9.7 illustrates this idea.

**Code 9.7**
Controlling whether debugging information is printed or not

```
/**
 * A number button was pressed.
 * @param number The single digit.
 */
public void numberPressed(int number)
{
    if(debugging) {
        System.out.println("numberPressed called with: " + number);
    }
```

**Code 9.7 continued**
Controlling whether debugging information is printed or not

```
    displayValue = displayValue * 10 + number;

    if(debugging) {
        reportState();
    }
}
```

A more economical variation on this theme is to replace the direct calls to print statements with calls to a specialized printing method added to the class.[2] The printing method would print only if the **debugging** field is **true**. Therefore, calls to the printing method would not need to be guarded by an if statement. Code 9.8 illustrates this approach. Note that this version assumes that **reportState** either tests the **debugging** field itself or calls the new **printDebugging** method.

**Code 9.8**
A method for selectively printing debugging information

```
/**
 * A number button was pressed.
 * @param number The single digit.
 */
public void numberPressed(int number)
{
    printDebugging("numberPressed called with: " + number);

    displayValue = displayValue * 10 + number;
    reportState();
}
```

```
/**
 * Only print the debugging information if debugging is true.
 * @param info The debugging information.
 */
public void printDebugging(String info)
{
    if(debugging) {
        System.out.println(info);
    }
}
```

As you can see from these experiments, it takes some practice to find the best level of detail to print out to be useful. In practice, print statements are often added to one or a small number of methods at a time, when we have a rough idea in what area of our program an error might be hiding.

[2] In fact, we could move this method to a specialized debugging class, but we shall keep things simple in this discussion.