11.10. Preconditions and Postconditions

Programmers spend significant amounts of time maintaining and debugging code. To facilitate these tasks and to improve the overall design, you can specify the expected states before and after a method's execution. These states are called preconditions and postconditions, respectively.

A **precondition** must be true when a method is *invoked*. Preconditions describe constraints on method parameters and any other expectations the method has about the current state of a program just before it begins executing. If the preconditions are not met, then the method's behavior is *undefined*—it may throw an exception, proceed with an illegal value or attempt to recover from the error. You should not expect consistent behavior if the preconditions are not satisfied.

A **postcondition** is true *after the method successfully returns*. Postconditions describe constraints on the return value and any other side effects the method may have. When defining a method, you should document all postconditions so that others know what to expect when they call your method, and you should make certain that your method honors all its postconditions if its preconditions are indeed met.

When their preconditions or postconditions are not met, methods typically throw exceptions. As an example, examine `String` method `charAt`, which has one `int` parameter—an index in the `String`. For a precondition, method `charAt` assumes that `index` is greater than or equal to zero and less than the length of the `String`. If the precondition is met, the postcondition states that the method will return the character at the position in the `String` specified by the parameter `index`. Otherwise, the method throws an `IndexOutOfBoundsException`. We trust that method `charAt` satisfies its postcondition, provided that we meet the precondition. We need not be concerned with the details of how the method actually retrieves the character at the index.

Typically, a method's preconditions and postconditions are described as part of its specification. When designing your own methods, you should state the preconditions and postconditions in a comment before the method declaration.

8.5. Preconditions and Postconditions

A **precondition** is a requirement that the caller of a method must obey. For example, the `deposit` method of the `BankAccount` class has a precondition that the amount to be deposited should not be negative. It is the responsibility of the caller never to call a method if one of its preconditions is violated. If the method is called anyway, it is not responsible for producing a correct result. [Index Term: |method calls|preconditions][Index Term: |preconditions]

NOTE

A precondition is a requirement that the caller of a method must meet.

Therefore, a precondition is an important part of the method, and you must document it. Here we document the precondition that the `amount` parameter must not be negative.

```
/**
   Deposits money into this account.
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
*/
```

Some `javadoc` extensions support a `@precondition` or `@requires` tag, but it is not a part of the standard `javadoc` program. Because the standard `javadoc` tool skips all unknown tags, we simply add the precondition to the method explanation or the appropriate `@param` tag.

Preconditions are typically provided for one of two reasons:

1. To restrict the parameters of a method

2. To require that a method is only called when it is in the appropriate *state*

For example, once a `Scanner` has run out of input, it is no longer legal to call the `next` method. Thus, a precondition for the `next` method is that the `hasNext` method returns `true`.

A method is responsible for operating correctly only when its caller has fulfilled all preconditions. The method is free to do *anything* if a precondition is not fulfilled. What should a method actually do when it is called with inappropriate inputs? For example, what should `account.deposit(-1000)` do? There are two choices.

1. A method can check for the violation and **throw an exception**. Then the method does not return to its caller; instead, control is transferred to an exception handler. If no handler is present, then the program terminates. We will discuss exceptions in Chapter 11.

2. A method can skip the check and work under the assumption that the preconditions are fulfilled. If they aren't, then any data corruption (such as a negative balance) or other failures are the caller's fault.

NOTE

If a method is called in violation of a precondition, the method is not responsible for computing the correct result.

The first approach can be inefficient, particularly if the same check is carried out many times by several methods. The second approach can be dangerous. The *assertion mechanism* was invented to give you the best of both approaches.

An **assertion** is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true. Here is a typical assertion check that tests a precondition:

```
public double deposit (double amount)
{
   assert amount >= 0;
   balance = balance + amount;
}
```

11.11. Assertions

When implementing and debugging a class, it's sometimes useful to state conditions that should be true at a particular point in a method. These conditions, called **assertions**, help ensure a program's validity by catching potential bugs and identifying possible logic errors during development. Preconditions and postconditions are two types of assertions. Preconditions are assertions about its state when a method is invoked, and postconditions are assertions about a program's state after a method finishes.

While assertions can be stated as comments to guide you during program development, Java includes two versions of the `assert` statement for validating assertions programatically. The `assert` statement evaluates a `boolean` expression and, if `false`, throws an `AssertionError` (a subclass of `Error`). The first form of the `assert` statement is

```
assert expression;
```

which throws an `AssertionError` if *expression* is `false`. The second form is

```
assert expression1 : expression2;
```

which evaluates *expression1* and throws an `AssertionError` with *expression2* as the error message if *expression1* is `false`.

You can use assertions to implement preconditions and postconditions programmatically or to verify any other intermediate states that help you ensure that your code is working correctly. Figure 11.7 demonstrates the `assert` statement. Line 11 prompts the user to enter a number between 0 and 10, then line 12 reads the number. Line 15 determines whether the user entered a number within the valid range.

If the number is out of range, the `assert` statement reports an error; otherwise, the program proceeds normally.

Fig. 11.7. Checking with `assert` that a value is within range.

```
 1   // Fig. 11.7: AssertTest.java
 2   // Checking with assert that a value is within range
 3   import java.util.Scanner;
 4
 5   public class AssertTest
 6   {
 7      public static void main( String[] args )
 8      {
 9         Scanner input = new Scanner( System.in );
10
11         System.out.print( "Enter a number between 0 and 10: " );
12         int number = input.nextInt();
13
14         // assert that the value is >= 0 and <= 10
15         assert ( number >= 0 && number  <= 10 ) : "bad number: " + number;
16
17         System.out.printf( "You entered %d\n", number );
18      } // end main
19   } // end class AssertTest
```

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

You use assertions primarily for debugging and identifying logic errors in an application. You must explicitly enable assertions when executing a program, because they reduce performance and are unnecessary for the program's user. To do so, use the `java` command's `-ea` command-line option, as in

```
java -ea AssertTest
```

Users should not encounter any `AssertionError`s through normal execution of a properly written program. Such errors should only indicate bugs in the implementation. As a result, you should never catch an `AssertionError`. Rather, you should allow the program to terminate when the error occurs, so you can see the error message, then locate and fix the source of the problem. Since application users can choose not to enable assertions at runtime, you should not use `assert` to indicate runtime problems in production code—use the exception mechanism for this purpose.

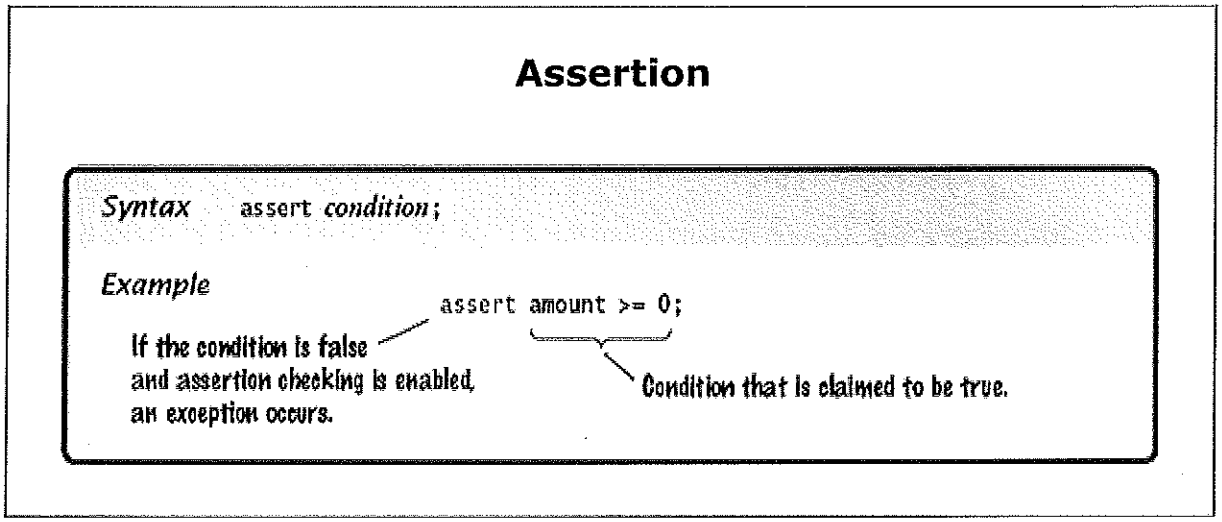
NOTE

An assertion is a logical condition in a program that you believe to be true.

In this method, the programmer expects that the quantity `amount` can never be negative. When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, *and assertion checking is enabled*, then the program terminates with an `AssertionError`.

However, if assertion checking is disabled, then the assertion is never checked, and the program runs at full speed. By default, assertion checking is disabled when you execute a program. To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MainClass
```



You can also use the shortcut `-ea` instead of `-enableassertions`. You definitely want to turn assertion checking on during program development and testing.

You don't have to use assertions for checking preconditions—throwing an exception is another reasonable option. But assertions have one advantage: You can turn them off after you have tested your program, so that it runs at maximum speed. That way, you never have to feel bad about putting lots of assertions into your code. You can also use assertions for checking conditions other than preconditions.

Many beginning programmers think that it isn't "nice" to abort the program when a precondition is violated. Why not simply return to the caller instead?

```
public void deposit(double amount)
{
   if (amount < 0)
      return; // Not recommended
   balance = balance + amount;
}
```

That is legal—after all, a method can do anything if its preconditions are violated. But it is not as good as an assertion check. If the program calling the `deposit` method has a few bugs that cause it to pass a negative amount as an input value, then the version that generates an assertion failure will make the bugs very obvious during test-ing—it is hard to ignore when the program aborts. The quiet version, on the other hand, will not alert you, and you may not notice that it performs some wrong calculations as a consequence. Think of assertions as the "tough love" approach to precondition checking.

When a method is called in accordance with its preconditions, then the method promises to do its job correctly. A different kind of promise that the method makes is called a **postcondition**. There are two kinds of postconditions:

1. The return value is computed correctly.
2. The object is in a certain state after the method call is completed.

NOTE

If a method has been called in accordance with its preconditions, then it must ensure that its postconditions are valid.

Here is a postcondition that makes a statement about the object state after the `deposit` method is called.

```
/**
   Deposits money into this account.
   (Postcondition: getBalance() >= 0)
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
*/
```

As long as the precondition is fulfilled, this method guarantees that the balance after the deposit is not negative.

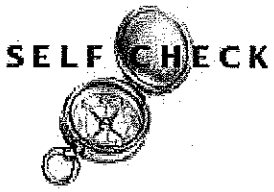
Some `javadoc` extensions support a `@postcondition` or `@ensures` tag. However, just as with preconditions, we simply add postconditions to the method explanation or the `@return` tag, because the standard `javadoc` program skips all tags that it doesn't know.

Some programmers feel that they must specify a postcondition for every method. When you use `javadoc`, however, you already specify a part of the postcondition in the `@return` tag, and you shouldn't repeat it in a postcondition.

```
// This postcondition statement is overly repetitive.
/**
   Returns the current balance of this account.
   @return the account balance
   (Postcondition: The return value equals the account balance.)
*/
```

Note that we formulate pre- and postconditions only in terms of the *interface* of the class. Thus, we state the precondition of the `withdraw` method as `amount <= getBalance()`, not `amount <= balance`. After all, the caller, which needs to check the precondition, has access only to the public interface, not the private implementation.

Preconditions and postconditions are often compared to *contracts*. In real life, contracts spell out the obligations of the contracting parties. For example, a car dealer may promise you a car in good working order, and you promise in turn to pay a certain amount of money. If either party breaks the promise, then the other is not bound by the terms of the contract. In the same fashion, pre- and postconditions are contractual terms between a method and its caller. The method promises to fulfill the postcondition for all inputs that fulfill the precondition. The caller promises never to call the method with illegal inputs. If the caller fulfills its promise and gets a wrong answer, it can take the method to "programmer's court". If the caller doesn't fulfill its promise and something terrible happens as a consequence, it has no recourse.



10. Why might you want to add a precondition to a method