HAPTER 5 Programming by contract

We've now seen boolean expressions and conditional statements in some detail. Conditional statements allow us to define a number of alternative courses of action. The specific action taken depends on the runtime value of the condition. Since a condition can be true at some times, false at others, the same conditional statement might result in different actions being performed at different times in the computation.

We've also introduced the notions of precondition, postcondition, and invariant. These are all conditions—assertions that can be true or false. Preconditions, postconditions, and invariants are not formally part of the Java language. They are not constructs recognized by the compiler or understood by the interpreter. However, they play an important role in the design, specification, and verification of systems. We include them in comments, and they directly influence the implementations we develop.

Since they are not part of Java *per se*, the language syntax does not restrict how we express preconditions, postconditions, and invariants. We have considerable flexibility in specifying these conditions, and can be quite informal if we want. On the other hand, we do not have the compiler and interpreter making sure that what we write is not utter nonsense. We use the Java syntax for boolean expressions wherever practical.

In this chapter, we further develop the ideas of precondition and postcondition, and introduce a programming style called *programming by contract*. The point of programming by contract is to clearly delineate in a method's specification the respective responsibilities of client and server. Preconditions and postconditions play a key role in defining these responsibilities.

ectives

After studying this chapter you should understand the following:

- programming by contract, defensive programming, and the difference between the two;
- consequences of a client's lack of adherence to a contract;
- purpose and use of the assert statement.

Also, you should be able to:

- express the responsibilities of client and server as a contract;
- use assert statements to verify a client's preconditions;
- use contracts to reason about a program's behavior.

5.1 Programming by contract

Let's return to an issue raised in the previous chapter: the range of possible value might be returned from an *Explorer*'s `tolerance` query. We have decided that this has a lower bound of 0, and documented this fact in the specification of the method:

```
/**
 * Annoyance (hit points) required to vanquish
 * this Explorer.
 * @ensure     this.tolerance() >= 0
 */
public int tolerance () ...
```

In order to guarantee this, we modified the constructor and the `takeThat` meth that they never assign a negative value to the instance variable `tolerance`, regardle the arguments provided by the client. We need to look at these two cases a little more fully.

In the case of `takeThat`, we protect against the client providing an annoyance t greater than the current tolerance of the *Explorer*:

```
public void takeThat (int annoyance) {
   if (annoyance <= tolerance)
      tolerance = tolerance - annoyance;
   else
      tolerance = 0;
}
```

This does not at all seem unreasonable. We shouldn't expect a client to worry a the tolerance of the *Explorer*. We didn't take the tolerance of the opponent into acc for instance, when we implemented the *Explorer*'s method `poke`:

```
public void poke (Denizen opponent) {
   opponent.takeThat(annoyability);
}
```

There may, in fact, be no method for obtaining the tolerance of a *Denizen*.

In summary, we do not consider it an error for a client to call `takeThat` witl argument greater than `tolerance`. It is simply a possibility that the implementatio `takeThat` must account for.

On the other hand, a call to the constructor with a negative initial value for `tol ance` seems quite different. This is something that should not happen. It is an error

the fault of the client who makes the call. We document our expectation that the client will provide a non-negative value when invoking the constructor as follows:

```
/**
 * Create a new Explorer with the specified name,
 * initial location, annoyability, and tolerance.
 * @require    annoyability >= 0
 *             tolerance >= 0
 */
public Explorer (String name, Room location,
                 int annoyability, int tolerance) ...
```

We introduced this notation in the previous chapter. Recall that conditions labeled "require" are called *preconditions*. They are requirements placed on the *client*. We are stating that *client must make sure* that the arguments provided for `annoyability` and `tolerance` are non-negative. *Postconditions*, labeled "ensure," are requirements on the implementor of the method. Preconditions and postconditions are part of a programming style called *programming by contract*. The basic idea is that use of an object feature (query or command) or constructor is considered to involve a "contract" between the client and server. For an invocation of a feature or constructor to be correct, the client must make sure that the preconditions are satisfied at the time of the call. If the preconditions are satisfied, then the server guarantees that the postconditions will be satisfied when the method completes (*i.e.*, upon "return"). If the preconditions are not satisfied, that is, if the client does not meet his end of the contract, then the server *promises nothing at all.*

> **programming by contract:** a programming style in which the invocation of a method is viewed as a contract between client and server, with each having explicitly stated responsibilities.

Understand that this is not an issue of a user entering bad data. The user interface is responsible for interacting with a user and making sure that bad data doesn't get into the system. What we're concerned with here is one object, a client, invoking a method of another object. If a client invokes a method without the preconditions being satisfied, it is because of a programming error in the client.

The point of this approach is to delineate, clearly and explicitly, responsibilities between the client and the server, and ultimately between the user of a method and the implementor of the method. We want to make sure that any possible error that can arise at run time is detected. But we'd like to do as little explicit error checking as possible. Specifically, we'd like only one test for each possible error condition. While this improves program efficiency and reduces duplication of code, there is a much more important reason for eliminating redundant testing. The most consequential impediment to writing correct, maintainable code is complexity. Adding error checking can make a simple straightforward algorithm unduly convoluted. An approach in which each routine validates all of its arguments—sometimes called *defensive programming*—can result in an excessively high degree of "code pollution." The trick is to make a program reliable but

not so convoluted as to be unmaintainable. In programming by contract, we use prec tions and postconditions to prescribe explicitly who—client or server—is responsib what. Clearly there are many design trade-offs, and we'll talk more about error handl Chapter 15.

We can completely specify the behavior of the constructor by adding postcondi like this:

```
/**
 * Create a new Explorer with the specified name,
 * initial location, annoyability, and tolerance.
 * @require    annoyability >= 0
 *             tolerance >= 0
 * @ensure     this.name().equals(name)
 *             this.location().equals(location)
 *             this.annoyability() == annoyability
 *             this.tolerance() == tolerance
 */
public Explorer (String name, Room location,
                 int annoyability, int tolerance)
```

The postconditions precisely and concisely describe the relationship betwee parameters of the constructor and the properties of the newly created object. The no means that if the newly created object is queried with the method `name`, for exam will return the string provided as the first argument of the constructor. That is, in the dition

```
this.name().equals(name)
```

`this.name()` refers to the value that the newly created object (`this`) will return the method `name` is invoked. The identifier `name` on the right refers to the first para of the constructor.

Granted, the ensure clause is a bit redundant here. It essentially repeats what is s the descriptive sentence beginning "Create...," and could be omitted without a subst loss in clarity or precision.

We have now completely specified the behavior of the constructor for the client remember the format we use for writing precondition and postconditions is a conve intended to convey information to the human reader: it is not part of the programmin guage. The client must make sure that the preconditions are satisfied *before* the meth invoked, in which case the implementor guarantees that the postconditions will be fied *upon completion* of the method.

5.1.1 Verifying preconditions: the *assert* statement

We have placed preconditions, `annoyability >= 0` and `tolerance >= 0`, o constructor. If the client invokes the constructor with non-negative `annoyabilit` `tolerance` arguments, we are committed to produce a new, well-formed *Exp*

object. If the client invokes the method with either argument negative, *we promise nothing at all.* This is important to realize. If the client does not adhere to its end of the contract, the implementation is not committed to any particular action.

In fact, if the client does not adhere to the contract, the program is erroneous and the behavior of an erroneous program is, by definition, unpredictable.

But of course *something* must happen. What we do depends to some degree on how much we trust our client. It may be that we are absolutely convinced that the constructor will never be called with a negative second argument—perhaps we're writing the client code ourselves—in which case we need do nothing. But systems change, and initial values are often computed in complex (and error-prone) ways. We can easily imagine, for instance, employing a nontrivial function that determines a random initial value for `tolerance` or `annoyability`.

The real problem with not verifying the values of the constructor arguments is that we can end up violating a class invariant. Suppose we simply leave the constructor as we originally wrote it:

```
public Explorer (String name, Room location,
                 int annoyability, int tolerance) {
   this.name = name;
   this.location = location;
   this.annoyability = annoyability;
   this.tolerance = tolerance;
}
```

If the client violates the precondition, there is no specific requirement on the constructor *with regard to the contract.* However, we have an *internal implementation requirement* in the form of the invariant condition on the component `tolerance`:

```
private int tolerance;      // current tolerance
                            // invariant:
                            // tolerance >= 0
```

Executing the constructor with the precondition violated will result in an *Explorer* object being created that does not satisfy the invariant. For this reason, rather than for the requirements of the contract, the original implementation is not particularly satisfactory.

We can, of course, return to the implementation suggested in the previous chapter, in which the argument value provided for `tolerance` is explicitly checked by the constructor:

```
public Explorer (String name, Room location,
                 int annoyability, int tolerance) {
   ...
   if (tolerance >= 0)
      this.tolerance = tolerance;
   else
      this.tolerance = 0;
}
```

But this is not entirely satisfactory either, since it treats an error condition (viol of the precondition) as a normal, expected, occurrence, and introduces the explicit cl ing we're trying to avoid.

What would we like to happen if the client violates the precondition? Perhap most we could hope for is that the interpreter or runtime system would recognize tha precondition was violated and generate an informative runtime error. Ideally, we'd lik to happen without having to write anything but the precondition. (Remember a major of this approach is to avoid cluttering our code with excessive error checks.) That is, like the interpreter to automatically check that preconditions are satisfied whene method is invoked. If the preconditions are met, the method executes normally. If no computation is interrupted and the user informed of the error condition. Error reporti this kind is particularly useful while we're developing, testing, and debugging a syste

Although Java will not automatically verify preconditions for us, the language vides an *assert statement*[1] that can be useful. The assert statement allows us to disting error handling from normal cases and get the error handling out of the way of the rithm. The statement has two formats. The simpler form is

```
assert booleanExpression ;
```

The boolean expression is evaluated, and if it is true, the statement has no effect. If false, the statement raises an error condition—an *exception*. This will stop execution o program and display some information about the cause of the exception. We can use *a* statements to verify preconditions in the constructor as follows:

```
public Explorer (String name, Room location,
                 int annoyability, int tolerance)
   assert annoyability >= 0;
   assert tolerance >= 0;

   this.name = name;
   this.location = location;
   this.annoyability = annoyability;
   this.tolerance = tolerance;
}
```

If a client invokes the constructor with a negative tolerance argument, the program terminate and we'll get a message that looks something like this:

```
Exception in thread "main" java.lang.AssertionError
   at mazeGame.Explorer.<init>(Explorer.java:16)
   at mazeGame.ExplorerTest.runTest(TestExplorer.java:
   at mazeGame.TestExplorer.main(TestExplorer.java:7)
```

(Exactly what the message looks like depends on the system we're running and where error occurred.)

---

1. The *assert statement* is available in Java version 1.4 and later.

The second form of the assert statement is

```
assert booleanExpression : expression ;
```

As before, the boolean expression is evaluated, and if it is true, the statement has no effect. If it is false, the second expression is evaluated and the result incorporated into the exception message. For instance, we might write

```
assert tolerance >= 0 :
   "precondition: tolerance (" + tolerance + ") >= 0";
```

Here the expression produces a *String* that includes the argument value of `tolerance`. (Remember, + is concatenation in this expression.) If a client invokes the constructor with a negative tolerance argument, the message will look something like this:

```
Exception in thread "main" java.lang.AssertionError:
precondition: tolerance (-10) >= 0
   at mazeGame.Explorer.<init>(Explorer.java:16)
   at mazeGame.ExplorerTest.runTest(TestExplorer.java:16)
   at mazeGame.TestExplorer.main(TestExplorer.java:7)
```

We tend to use assert statements sparingly in our examples. This is not to indicate a style to be copied, but simply to reduce distractions in the examples.

Assertions are disabled by default

There is a difficulty with assert statements that leads some programmers to avoid them for precondition testing. When the program is run, the interpreter, unless told to do otherwise, simply ignores them. Assertions are explicitly enabled with command line switches as explained in Appendix i.

Because of the possibility that a program might be run without precondition testing, some programmers prefer to test preconditions explicitly with if statements. But as we have said, an if statement implies an ordinary, expected case that must be handled by the program. A precondition failure, on the other hand, is an error and occurs only in an incorrect program.

Assertions and postconditions

Of course, assert statements can also be used to verify postconditions. For several reasons, though, verifying postconditions is less common than verifying preconditions. First, as we shall see, a postcondition often says something about the state of the object after the method is completed *in terms of* the state of the object when the method is called. Verifying the postcondition in such a case requires saving the state of the object at the start of the method.

Second, writing an assert statement to verify a precondition essentially says "I'm not willing to trust each client to do its job, and I'm not going to complicate my code by doing the client's job." Writing an assert statement to verify a postcondition says "I don't trust myself to write correct code, but I do trust myself to write and test a correct postcondition." We are much more likely to adopt the former attitude than the latter.

Open the file `CombinationLock.java`, located in `locks`, in `ch5`, in `nhText`. file contains a definition of the class *CombinationLock* as presented in the previous ter.

Note that the package name is `ch5.locks`. We'll incorporate the chapter numl a package name from now on, so that classes with the same name but in different cha don't become confused.

1. Select *Preferences...* from the *Edit* pull-down menu. A *Preferences* window appear.
2. Choose the category *Miscellaneous* from the *Preferences* window. Make sur *Enable Assert Statement Execution* option is activated. If necessary, click the c box, then press *Apply* and *OK*.
3. In the *Interactions* pane, create a *CombinationLock* with an invalid combination

```
> import ch5.locks.*;
> CombinationLock myLock = new CombinationLock(100(
```

   and note the result.
4. Modify the assert statement, adding an expression:

```
assert 0 <= combination && combination <= 999 :
   "bad combination: " + combination;
```

   Note the condition is separated from the following expression by a colon. Save a recompile.
5. In the *Interactions* pane, again create a *CombinationLock* with an invalid combin and note the result.

5.2 Further examples

Named constants in preconditions and postconditions

Recall that in the class *TrafficSignal*, we defined three named constants, `GREEN`, `YEL` and `RED`, to represent the different lights. The method `light` was specified in Sectio as

> **`public int`** `light ()`
>
> The current light that is on. Returns `TrafficSignal.GREEN`, `TrafficSignal.YELLOW`, or `TrafficSignal.RED`.

We can express this a little more clearly with a postcondition:

> **`public int`** `light ()`
>
> The current light that is on.

ensure:

```
this.light() == TrafficSignal.GREEN ||
this.light() == TrafficSignal.YELLOW ||
this.light() == TrafficSignal.RED
```

Named constants should be used in preconditions and postconditions rather than literals whenever possible. Furthermore, though we occasionally resort to informal English when writing preconditions and postconditions, standard Java syntax is preferable. There is no chance for ambiguities creeping into our statements if we stick to the formal language notation.

Specifying a command

Next, let's take another look at the *Explorer* method `takeThat`, which we have specified as:

`public void` `takeThat (int annoyance)`
Receive a poke of the specified number of hit points.

We've decided that we won't require `annoyance` be bounded by our *Explorer*'s current tolerance. But we might want to put a lower limit of 0 on the value of this argument. (Unless we decide that some pokes can *increase* our *Explorer*'s tolerance: pokes with a magic wand?)

`public void` `takeThat (int annoyance)`
Receive a poke of the specified number of hit points.

require:

```
annoyance >= 0
```

It is important to realize that if we don't put this precondition on `annoyance`, the method *must be prepared to accept a negative argument*. Furthermore, the documentation must explain exactly what a negative argument means.

We might consider making the precondition `annoyance > 0`. After all, what's the point of a 0 valued poke? But unlike a negative valued poke, a 0 valued poke has a reasonable meaning. It's a poke that does no damage. We can understand it without additional explanation. In general, our methods will be simpler and easier to use if we don't arbitrarily exclude reasonable argument values, even if we don't expect some of these values to occur ordinarily.

What kind of postcondition can we write to describe to the client the effect of the method? We want to indicate that the *Explorer*'s tolerance will decrease: that the value returned by the query `tolerance()` after execution of `takeThat` will be less than the value returned by this query before execution of `takeThat`. To do this, we must be able to refer to the state of the *Explorer* when the method is invoked as well as to the state of the *Explorer* when the method completes. It's common for a command postcondition to describe the state of the object after the command is executed in terms of the object's state when the command is invoked.

When writing a postcondition, we use "`this`" to refer to the state of the object time the method completes, and "`old`" to refer to the state of the object *at the tin method was invoked.* The notation "`old`" is not a Java construct; it is a convention u a comment specifying a precondition.

We might start by writing something like this:

ensure:

```
this.tolerance() ==
   old.tolerance() - annoyance
```

As noted, `this.tolerance()` refers to the tolerance property of the object whe method completes, and `old.tolerance()` refers to the tolerance property whe method begins. The condition says that the value returned by `tolerance` afte method is complete will be the `tolerance` value when the method was called minu argument value.

But this postcondition is not correct: if `annoyance` is greater than tolerance, t ance will end up 0, and not `old.tolerance() - annoyance`. We can correc problem by writing:

ensure:

```
this.tolerance() ==
   max (old.tolerance() - annoyance, 0)
```

This says that after the method competes, tolerance will be the larger of 0, and the sta tolerance minus the argument.

The postcondition is now correct but almost certainly too strong, since it promise client exactly how the method is implemented. Suppose we decide later in the deve ment that the *Explorer* should be able to do something to lessen the effect of a poke– on a parka, for instance. Not only would the postcondition need to be changed, but sin client's correctness depends on our promises, any client that invoked the method w need to be reexamined as well. We don't want a client to depend on irrelevant impleme tion details.

The following would probably be an adequate postcondition:

ensure:

```
this.tolerance() <= old.tolerance()
```

It simply promises that the *Explorer*'s tolerance, after executing the method, will be greater than it was when the method was called.

The definition of `takeThat` can be written as follows:

```
/**
 * Receive a poke of the specified number of
 * hit points.
 * @require     annoyance >= 0
 * @ensure      this.tolerance() <= old.tolerance()
 */
public void takeThat (int annoyance) {
```

```
        if (annoyance <= tolerance)
            tolerance = tolerance - annoyance;
        else
            tolerance = 0;
    }
```

Note that if the method is invoked with a negative `annoyance`, `tolerance` is increased and the postconditions not satisfied. But if the client violates the preconditions, the sever is under no obligation to comply with the postconditions.

Assigning responsibilities

As a final example, let's look at the nim game class *Pile* from Section 3.2. Remember that a *Pile* instance modeled a pile of sticks from which players in turn removed 1, 2, or 3 sticks. The command `remove` is used to remove sticks:

`public void remove (int number)`
Reduce the number of sticks by the specified amount.

There are at least three "what if" questions that come to mind when we read this specification:

- what if `number` is negative? Is this legal? If so, what does this mean?
- what if `number` is greater than the number of sticks remaining the pile?
- what if `number` is not 1, 2, or 3?

Each of these questions must be answered to complete the specification of the method. In each case, we can write a precondition that excludes the case, and put the responsibility on the client. Or we can handle the case in the method.

For instance, we can write a precondition requiring `number` to be non-negative. Then it's the client's responsibility to make sure that the argument is not negative and it's a programming error if the method is ever invoked with a negative argument. Or we can decide that the method will accept a negative argument, in which case we must document clearly what this means. But we must choose one or the other. If we don't exclude negative arguments, a client can legitimately invoke the method with a negative argument and expect something reasonable to happen.

What should we do in each case? First, it does not seem meaningful for a client to remove a negative number of sticks. (Maybe there should be a method for adding sticks to the pile, but this shouldn't happen in a method named `remove`.) A negative argument would certainly be due to a program bug, and is not something we want to handle quietly. We'll exclude this possibility with a precondition:

require:
`number >= 0`

(We might also wonder whether an argument of 0 should be allowed. But "remove 0 sticks" has a clear meaning, so we'll accept a 0 argument.)

It's not so obvious how to handle the second case, where number is greater th number of sticks remaining in the pile. We could say that if a client attempts to r more sticks than remain in the pile, all the sticks are removed. But attempting to r more sticks than there are in the pile seems likely to be a client error. So we'll exclu case too, though handling it in the method could also be justified:

require:

```
number >= 0
number <= this.sticks()
```

Finally, what if number is not 1, 2, or 3? The number of sticks than can lega removed by a player is determined by the rules of the game. Knowing the rules game really doesn't seem like it should be the responsibility of the *Pile*. So we restrict the argument further.

Adding a postcondition, the complete specification reads as follows:

```
public void remove (int number)
```

Reduce the number of sticks by the specified amount.

require:

```
number >= 0
number <= this.sticks()
```

ensure:

```
this.sticks() == old.sticks() - number
```

Note that in the precondition, the expression this.sticks() refers to the nu of sticks in the pile *when the method is invoked.* In the postcondition, the expre this.sticks() refers to the number of sticks *when the method completes.*

DrJava: preconditions and postconditions

Open the file Toy.java, located in folder exercise, in ch5, in nhText. The does not model anything, but simply serves the exercise.

1. In the *Interactions* pane create an instance of the class:

```
> import ch5.exercise.*;
> Toy t = new Toy();
```

2. Since each of the functions twice and increment take integer argument return integer values, they can be composed. That is we can write the expression

```
> t.next(t.twice(0))
> t.twice(t.next(0))
```

3. Try the expressions above. Find other values for which the composition is lega evaluate them.
4. Now try other input values for which the composition is not quite legal, such as:

```
> t.twice(t.next(5))
> t.next(t.twice(3))
```

As you can see, the expressions are evaluated, but `twice`'s and `next`'s contracts are broken in the process.

5. To avoid illegal compositions, add an assert statement to method `twice`.

```
assert 0 <= number && number <= 10;
```

Make sure that the preference *Enable Assert Statement Execution* has been activated. (See page 240.) Save and compile.

6. In the *Interactions* pane, create a *Toy* instance as above, and key

```
t.twice(t.next(5))
```

Read the error message produced.

7. Add the appropriate assert statement to `next`.
Save, compile, and evaluate `t.next(t.twice(3))`.

Preconditions and postconditions: a summary of use

Preconditions

Preconditions must be satisfied by the client when invoking the method. There are several situations in which preconditions are used.

- Most often preconditions constrain the possible values that the client can provide as a method or constructor argument. This is the case in the *Explorer* constructor and `takeThat` command considered above. The precondition for `takeThat`, for example, requires that the client provide a non-negative argument. Remember that if an argument is not constrained by a precondition, the method must be prepared to accept *any value of the specified type*. Furthermore, the method must explicitly document its actions for all possible arguments.
- Occasionally, preconditions constrain the order in which methods can be invoked or require that the server be in a certain state before a given method can be invoked. For instance, it might be necessary that a door be unlocked before it can be opened, or that an automobile be started before it can be moved. Modeling a vending machine, enough money must be entered before an item can be retrieved. In a word processor, text must be cut before a paste can be requested.

  In such situations the server will often explicitly handle the "wrong state" case, rather than requiring the client to verify that the server state is correct. For instance, suppose a door must be unlocked before it can be opened. The design choices are to *require* that the door be unlocked before an open operation can be attempted—client's responsibility, specified by a precondition; or to let the server handle an attempt to open an unlocked door by responding in an appropriate way—for instance, by simply not opening the door. In the first case, it is a program error for client code to attempt an open operation without verifying that the door is unlocked. In the second case, it is not.

If a precondition requires the server to be in some particular state, it must be po for a client to verify the correctness of the server's state. For instance, if the open ope requires that the door be unlocked, it must be possible for a client to determine th door is unlocked.

Query postconditions

When an object responds to a query, it does not change state. It simply provides a va the client. Thus query postconditions inevitably say something about the value retu We sometimes use the term "result" to refer to the value returned by the query instance, we might specify the *Counter* method `currentCount` in either of the fo ing ways:

```
public int currentCount ()
        Current count; the number of items counted.
        ensure:
            this.currentCount() >= 0

public int currentCount ()
        Current count; the number of items counted.
        ensure:
            result >= 0
```

Command postconditions

Commands result in a change of state. Thus command postconditions typically des the new state of the object, its state after execution of the command. The new state is compared to the previous state, the state of the object when the command was invc For this reason, it is convenient to have a notational convention for referring to the sta the object when the command is invoked. We use "`old`" for this purpose, as illustı above with the *Explorer* method `takeThat`.

Constructor postconditions

Not surprisingly, constructor postconditions typically describe the initial state of the ne created object. This is the case with the *Explorer* constructor given above.

Preconditions and postconditions are part of the specification

It is important to remember that preconditions and postconditions for public methods part of an object's specification. As such, they should *never* mention private impleme tion components. The following specification of the *Counter* method `reset`, for insta is incorrect. The *Counter* instance variable `count` is part of the implementation: it is part of the object's specification, and is meaningless to the client.

```
public void reset ()
```
Reset the count to 0.

ensure:

```
count == 0
```
This is not correct! count is private.

The method `currentCount`, however, is part of the public specification of the class. The following is the proper way to express the postcondition:

```
public void reset ()
```
Reset the count to 0.

ensure:

```
this.currentCount() == 0
```

Furthermore, the server must provide an adequately rich set of features so that a client can verify preconditions in a straightforward manner. In defining the *Date* constructor in the previous chapter, for instance, we required that the day, month, and year arguments represented a legal calendar date. This involves a nontrivial calculation, and it would be inappropriate for every *Date* client to include an algorithm to do this. Thus we added the method `isLegalDate` to the class.

Implied preconditions and postconditions

There are two situations in which preconditions and/or postconditions are generally assumed and not stated explicitly. The first two arguments of the *Explorer* constructor, for example, reference a *String* and a *Room*. Recall from Section 1.5 that there is a special null value, denoted by the literal **`null`**, that doesn't reference any object.

We assume that an argument value cannot be null unless the specification explicitly allows it. We also assume that the value returned by a query cannot be null unless the specification explicitly allows this possibility. Thus the *Explorer*'s constructor precondition is equivalent to

```
* @require    name != null
*             location != null
*             annoyability >= 0
*             tolerance >= 0
```

We rarely write these non-null requirements explicitly in the specification.

Second, the type of a numeric property implies some expected "range of reasonable use." For example, the query `currentCount` of a *Counter* is specified as

```
public int currentCount ()
```
The number of items counted.

ensure:

```
this.currentCount() >= 0
```

The return type (**`int`**) determines an upper bound on the returned value, namely 2,147,483,647. Thus the counter can only be legitimately incremented if its value is less than this limit:

public void incrementCount ()

Increment the count by 1.

require:

this.currentCount() < 2147483647

The underlying assumption, however, is that instances of this class will only be in situations where the count will not exceed the range of the type **int**. Thus we rarely write a precondition like the one shown above. (What happens if the count is i mented past 2147483647? See Exercise 5.9.)

⁂ 5.4 Enumeration classes

In defining the classes *TrafficSignal* and *PlayingCard* in Chapter 2, we used named stants to define what are essentially types with only a few values. For instance, ther three possible values for a *TrafficSignal* light: green, yellow, and red. To represent values, we used integers named TrafficSignal.GREEN, TraficSignal.Y LOW, and TrafficSignal.RED. Similarly, there are four possible values for a *Pla Card* suit, and thirteen values for a *PlayingCard* rank. In each case, we used intege represent these values.

One problem with this approach is that there is no way for the compiler to ensur a client uses appropriate integers for these values. There is nothing to prevent a client writing, for example,

```
new PlayingCard(27,-4)
```

Of course, we can check at runtime with an assert statement:

```
public PlayingCard (int suit, int rank) {
   assert suit == SPADE || suit == HEART ||
      suit == DIAMOND || suit == CLUB;
   ...
```

But compile time checking, when we can achieve it, is much more effective than ru checking.

Another problem is that there is no way to prevent the client from making use c actual integers used to represent values. For instance, a client might check to see if a is red by doing

```
if (card.suit() == 2 || card.suit() == 3) ...
```

even though the fact that 2 and 3 are used to represent diamonds and hearts is not p the class specification.

An alternative approach is to use an *enumeration*. An enumeration is a class hav small number of fixed, named, instances. We can define an enumeration classes fc class *PlayingCard* as follows.

```
public class PlayingCard {

   public enum Suit {clubs, diamonds, hearts, spades}
   public enum Rank {two, three, four, five, six,
      seven, eight, nine, ten, jack, queen, king, ace}
   ...
```

The format of the definition includes the keyword **enum**, followed by the name of the class, followed by a list of class instances.

These definitions define classes *PlayingCard.Suit* and *PlayingCard.Rank*. *PlayingCard.Suit* comprises four objects, named `clubs`, `diamonds`, `hearts`, and `spades`. *PlayingCard.Rank* comprises thirteen objects named `two`, `three`, *etc.* These objects are essentially "named constants." `PlayingCard.Suit.clubs`, for example, is a named constant in much the same way as `PlayingCard.CLUB` in our original definition of *PlayingCard*. However, `PlayingCard.Suit.clubs` denotes a *PlayingCard.Suit* object, while `PlayingCard.CLUB` denotes an **int**.

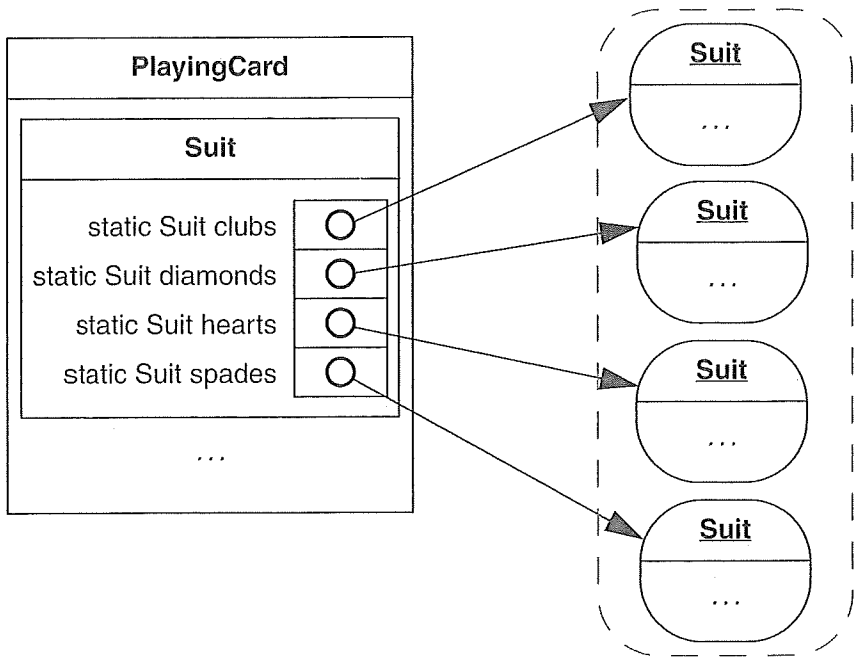

Figure 5.1 **The objects of the class *PlayingCard.Suit*.**

We can now define the *PlayingCard* constructor and queries in terms of the classes *Suit* and *Rank* as shown in Listing 5.1. A client is required to provide *PlayingCard.Suit* and *PlayingCard.Rank* arguments when invoking the constructor:

```
new PlayingCard(
   PlayingCard.Suit.clubs, PlayingCard.Rank.ace);
```

Furthermore, since the enumeration classes and instance identifiers are static, they can be imported into a compilation unit with a static import statement. (See Section 3.7.3.) If the client compilation unit includes

```
import static PlayingCard.*;
```

Listing 5.1 **The class *PlayingCard***

```
public class PlayingCard {

    public enum Suit {clubs, diamonds, hearts, spades}
    public enum Rank {two, three, four, five, six, seven,
       eight, nine, ten, jack, queen, king, ace}

    private Suit suit;
    private Rank rank;

    public PlayingCard (Suit suit, Rank rank) {
       this.suit = suit;
       this.rank = rank;
    }

    public Suit suit () {
       return suit;
    }

    public Rank rank () {
       return rank;
    }

    public String toString () {
       return rank + " of " + suit;
    }
}
```

the constructor can be invoked as

```
new PlayingCard(Suit.clubs, Rank.ace);
```

If the client compilation unit includes

```
import static PlayingCard.Suit.*;
import static PlayingCard.Rank.*;
```

the constructor can be invoked as

```
new PlayingCard(clubs, ace);
```

The method toString

There are several methods predefined for enumeration instances. The method `toStri` is defined to return the name of the object as a *String*. For instance,

```
PlayingCard.Suit.clubs.toString() ⇒ "clubs"
```

This can be helpful in testing and debugging. With our original *PlayingCard* class, `card.suit()` returns an **`int`**. The statement

```
System.out.println("suit: " + card.suit());
```

produces something like

```
suit: 1
```

Using enumerations, this statement produces a more readable

```
suit: clubs
```

The method compareTo

The elements of an enumeration are ordered, with the ordering established by the order in which they are declared. The method `compareTo` can be used to determine the relative order of two objects.

`public int` `compareTo (`*`EnumClass`* `obj)`

Compare this enum constant with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Enum constants are comparable only to other enum constants of the same enum class. The natural order implemented by this method is the order in which the constants are declared.

For instance, if `card` is a *PlayingCard*,

```
card.suit().compareTo(PlayingCard.Suit.diamonds)
```

returns a negative value if `card.suit()` is `clubs`, zero if it is `diamonds`, and a positive value if it is `hearts` or `spades`. Similarly, the ranks of `card1` and `card2` can be compared by invoking

```
card1.rank().compareTo(card2.rank())
```

and seeing if the result is negative, zero, or positive.

It is also possible to define additional features for an enumeration. See Supplement d for details.

Summary

In this chapter, we introduced a programming style called programming by contract. The basic idea is to make explicit the respective responsibilities of client and server in a method invocation. To this end, the invocation of a server method by a client is viewed as involving a contract between the client and the server. The server promises to perform the action specified by the method and to ensure that the method's postconditions are satisfied, but only if the client meets the preconditions. Preconditions are the client's responsibility;

postconditions are the server's. If the client fails to meet the preconditions, the cont void: the server is not obligated to behave in any specific way.

Using this approach, it is a programming error for a client to invoke a method w satisfying the method's preconditions. We talk more about errors in Chapter 15 versely, if the client satisfies the preconditions, the server must accomplish the act specified.

Preconditions can be verified using Java's assert statement. If the boolean expr in the assert statement is true, the statement has no effect. If it is false, an error exc occurs and the program terminates.

Preconditions most often constrain the values a client can provide as argument conditions for a query generally say something about the value returned. Postcon for a command generally describe the state of the object after the command is com in terms of the state before the command was begun.

SELF-STUDY QUESTIONS

5.1 What is "programming by contract"? For what language constructs are contracts de How is the contract between client and server specified?

5.2 How does programming by contract help manage complexity and improve efficien software system?

5.3 Compare how the system deals with bad data entered by the user and illegal argu passed to a method.

5.4 Indicate whether each of the following statements is true or false.

a. In programming by contract, the server is always required to meet specified pos ditions.

b. In programming by contract, the server should verify client-provided argument correct them if possible.

c. Java's runtime system checks the validity of preconditions and postconditions.

5.5 Consider the class *JetCalibrator* partially given below. What this class models important.

```
1.  public class JetCalibrator {

2.      /**
3.       * @ensure      -5 <= this.jetSetting() &&
4.       *              this.jetSetting() <= 5
5.       */
6.      public int jetSetting () ...

7.      /**
8.       * @require    -3 <= offSet && offSet <= +3
9.       */
10.     public void adjust (int offSet) ...
```

```
11.      /**
12.       * @ensure     this.jetSetting() >= 0
13.       */
14.      public void normalize () ...

15.      private int jet;     // invariant:
16.                           // -5 <= jet && jet <= +5
    }
```

Which of the following statements are true?

a. Ensuring that the condition of line 8 holds is the responsibility of the client.

b. In a correct program, the method `adjust` will never be invoked with an argument of 4.

c. The implementation of the method `adjust` must check the value of `offSet` in case the user enters a value that is out of range.

d. The condition of line 12 implies that the server will never execute the method `normalize` when the property `jetSetting` is negative.

e. The condition of line 16 implies that the value of the instance variable `jet` will never be 6.

5.6 Let `c`, `i`, and `j` be variables defined as follows:

```
JetCalibrator c = new JetCalibrator(...);
int i;
int j;
```

where *JetCalibrator* is the class sketched in Exercise 5.5. Which of the lettered statements are true after the following sequence is executed?

```
i = c.jetSetting();
c.normalize();
j = c.jetSetting();
```

a. `i` and `j` are guaranteed to have the same value.

b. `i` can be -5.

c. `j` can be -5.

5.7 Given the variables of Exercise 5.6, suppose the following statements are executed:

```
i = 4;
c.adjust(i);
j = c.jetSetting();
```

Which one of the following are true?

a. `j` will be 4.

b. It is not possible to tell what will happen.

5.8 Write an assert statement to verify the precondition of the method `takeThat`, sh page 242.

5.9 The following are specifications for constructors and methods in a *Counter* class. C their completeness.

(a)
```
public Counter (int a, int b)
```
Create a new *Counter.*

(b)
```
public void increment ()
```
Increment this *Counter.*

(c)
```
public void reset ()
```
Reset this *Counter* to the starting value.

ensure:
```
this.count() == Counter.STARTING_VALUE
```

5.10 The class *CombinationLock* defined in Section 4.5 includes a method `close`, spec

```
public void close ()
```
Lock this *CombinationLock.*

and an instance variable `isOpen`, defined as

```
private boolean isOpen;   // the lock is unlocked
```

Here are three implementations of the method `close`:

(a)
```
public void close () {
    isOpen = false;
}
```

(b)
```
public void close () {
    assert isOpen;
    isOpen = false;
}
```

(c)
```
public void close () {
    isOpen = !isOpen;
}
```

What preconditions. if any, should be added to the method to make each of these ir mentations correct?

EXERCISES

5.1 Add preconditions and/or postconditions as appropriate to the balls and strikes class of Exercise 4.12.

5.2 Add appropriate assert statements to the *Pile* method `remove`, as specified on page 244. Write and run a test in which the precondition is violated and note the error message produced.

5.3 Add preconditions and postconditions to the *Rectangle* class specified in Listing 3.4. Add assert statements to the implementation. Modify the `RectangleDisplay` program so that it attempts to create an illegal *Rectangle*. Run the program and observe the error message generates.

5.4 Add preconditions and postconditions to the class *PlayingCard*, specified in Listing 2.5.

5.5 Write assert statements that explicitly verify that the name and location arguments of the *Explorer* constructor are not null.

5.6 Write assert statements to verify the preconditions for the *Date* constructor, as defined on page 202.

5.7 Suppose that for the *Employee* method `pay` of Exercise 4.7, `hours` and `rate` are parameters rather than instance variables. That is, suppose the method is specified as

```
public double pay (int hours, double rate)
```

Write a complete specification, including reasonable preconditions and postconditions, for this method.

5.8 Assume that the method `dayOfWeek` takes a day of the year and year as arguments, and returns the day of the week. That is, `dayOfWeek` is specified

```
public int dayOfWeek (int day, int year)
```

and `dayOfWeek(51,1999)` will tell us that the 51st day of 1999 was a Saturday.

Assume that the class *Date* has named constants defined for each day of the week:

```
public static final int MONDAY = ...
...
```

Write a complete specification, including reasonable preconditions and postconditions, for this method.

5.9 Write, compile, and run a simple program with the following `main`. Explain the results.

```
public static void main (String[] argv) {
   int i = 2147483647;
   i = i + 1;
   System.out.println("i = " + i);
}
```

5.10 Suppose we want to build a maze game in which *Denizens*, when poking an *Explorer*, sometimes magically increase the *Explorer*'s tolerance. We represent a magic tolerance-giving hit by furnishing a negative argument to the `takeThat` method. Furthermore, we allow an *Explorer* to have deficit tolerance, also represented by a negative value. An *Explorer* with deficit tolerance can be revived only by a tolerance-giving poke. Can we reuse the class *Explorer* as it exists in this new game? Explain your answer.

5.11 Can an assert statement be used to verify query postconditions? Why do you think pr ditions are verified far more often than postconditions?

Can an assert statement generally be used to verify command postconditions? Why or not?

* 5.12 Given the definition of *PlayingCard* in Listing 5.1, implement a method

```
boolean higherThan (PlayingCard c1, PlayingCard c2)
```

that returns true if `c1` is higher than `c2`, where cards are first compared by rank (an a higher than a king, a king higher than a queen, and so on), and cards of equal rank ar compared by suit (a spade is higher than a heart which is higher than a diamond whic higher than a club).

ANSWERS TO SELF-STUDY QUESTIONS

5.1 Programming by contract is a programming style in which the invocation of a meth viewed as a contract between client and server, with each having explicitly stated resp bilities. Contracts are defined for methods and constructors. Contracts are specified b doc comment preceding the method or constructor. In particular, preconditions (req detail responsibilities of the client, postconditions (ensure) detail responsibilities o server.

5.2 By localizing responsibilities, the number of methods that must contain code to verif validity of data is minimized. The result is reduced complexity and improved efficien

5.3 It is the responsibility of the user interface subsystem to interact with the user and pre bad data from being entered. Sometimes this is easy, for instance, if the user enters a r when a data is expected. Sometimes this is impossible, for instance, if the user keys rather than "8" when entering the price for a can of beans. A client invoking a ser method with an illegal argument, however, is the result of a program bug. We can hope that the program will terminate with an error message.

5.4 All three statements are false.

5.5 *(a) true* *(b) true* *(c) false* *(d) false* *(e) true*

5.6 *(a) false* *(b) true* *(c) false*

5.7 *(b)* is true.

5.8
```
assert annoyance >= 0;
```

5.9 *(a)* The specification adds nothing. What are the parameters? What is the initial valu the *Counter*?

(b) Increment by how much? One would guess one, but it would be better to be explic No preconditions imply that the method can be invoked at any time.

(c) Specifications are satisfactory. No preconditions imply that the method can be invo at any time.

5.10 *(a)* No preconditions are needed.

(b) and *(c)* both need `this.isOpen()` as a precondition. *(b)* will fail if this precondition is not satisfied, and *(c)* will leave open a lock that is already closed. (This is probably not a reasonable precondition. There is no obvious reason for requiring that the client to make sure that the lock is open before attempting to close it.)