C H A P T E R 3

Getting Started

3.1 Three little rules 73
3.2 A simple Perl class 80
3.3 Making life easier 89
3.4 The creation and destruction of objects 96
3.5 The CD::Music class, compleat 114
3.6 Summary 117

If you've ever used another object-oriented programming language, or been traumatized by some prior exposure to object orientation, you're probably dreading tackling object orientation in Perl—more syntax, more semantics, more rules, more complexity. On the other hand, if you're entirely new to object orientation, you're likely to be equally nervous about all those unfamiliar concepts, and how you're going to keep them all straight in your head while you learn the specific Perl syntax and semantics.

Relax!

Object-oriented Perl isn't like that at all. To do real, useful, production-strength, object-oriented programming in Perl you only need to learn about one extra function, one straightforward piece of additional syntax, and three very simple rules.[1]

3.1 THREE LITTLE RULES

Let's start with the rules…

[1] The three rules were originally formulated by Larry Wall, and appear in a slightly different form in the perlobj documentation.

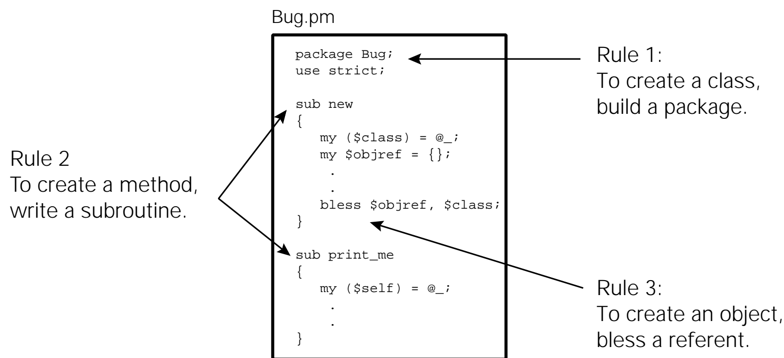

Figure 3.1 Three little rules

3.1.1 Rule 1: To create a class, build a package

Perl packages already have a number of classlike features:

- They collect related code together;
- They distinguish that code from unrelated code;
- They provide a separate namespace within the program, which keeps subroutine names from clashing with those in other packages;
- They have a name, which can be used to identify data and subroutines defined in the package.

In Perl, those features are sufficient to allow a package to act like a class.

Suppose we wanted to build an application to track faults in a system. Here's how to declare a class named Bug in Perl:

```
package Bug;
```

That's it! Of course, such a class isn't very interesting or useful, since it has no attributes or behavior. And that brings us to the second rule…

3.1.2 Rule 2: To create a method, write a subroutine

Methods are just subroutines, associated with a particular class, that exist specifically to operate on objects that are instances of that class.

Happily, in Perl, a subroutine that is declared in a particular package *is* associated with that package. So to write a Perl method, we just write a subroutine within the package acting as our class.

For example, here's how we provide an object method to print our Bug objects:

```
package Bug;

sub print_me
{
   # The code needed to print the Bug goes here
}
```

Again, that's it. The subroutine `print_me` is now associated with the package Bug, so whenever we treat Bug as a class, Perl automatically treats `Bug::print_me` as a method.

Calling the `Bug::print_me` method involves that one extra piece of syntax—an extension to the existing Perl "arrow" notation. If you have a reference to an object of class Bug (we'll see how to get such a reference in a moment), you can access any method of that object by using a `->` symbol, followed by the name of the method.

For example, if the variable `$nextbug` holds a reference to a Bug object, you could call `Bug::print_me` on that object by writing:

```
package main;

# set $nextbug to refer to a Bug object, somehow, and then…

$nextbug->print_me();
```

Calling a method through an arrow should be familiar to any C++ programmers; for the rest of us, it's at least consistent with other Perl usages:

```
$hsh_ref->{"key"};          # Access the hash referred to by $hashref
$arr_ref->[$index];         # Access the array referred to by $arrayref
$sub_ref->(@args);          # Access the sub referred to by $subref
$obj_ref->method(@args);    # Access the object referred to by $objref
```

The only difference with the last case is that the thing referred to by `$objref` has many ways of being accessed, namely, its various methods. So, when we want to access an object, we have to specify which particular way, or method, should be used.

Just to be a little more flexible, Perl doesn't actually require that we hard-code the method name in the call. It is also possible to specify the method name as a scalar variable containing a string matching the name (i.e., a symbolic reference) or as a scalar variable containing a real reference to the subroutine in question. For example, instead of:

```
$nextbug->print_me();
```

we could write:

```
$method_name = "print_me";     # i.e. "symbolic reference" to some &print_me
$nextbug->$method_name();      # Method call via symbolic reference
```

or:

```
$method_ref = \&Bug::print_me;# i.e. reference to &Bug::print_me
$nextbug->$method_ref();       # Method call via hard reference
```

In practice, the method name is almost always hard-coded.

When a method like `Bug::print_me` is called, the argument list that it receives begins with the object reference through which it was called,[2] followed by any arguments that were explicitly given to the method. That means that calling `Bug::print_me("logfile")` is ***not*** the same as calling `$nextbug->print_me("logfile")`. In the first case, `print_me` is treated as a regular subroutine so the argument list passed to `Bug::print_me` is equivalent to:

[2] The object on which the method is called is known as the *invoking object* or, sometimes, the *message target*. It is the reference to this object that is passed as the first argument of any method invoked using the `->` notation.

```
( "logfile" )
```

In the second case, `print_me` is treated as a method so the argument list is equivalent to:

```
( $objref, "logfile" )
```

Having a reference to the object passed as the first parameter is vital, because it means that the method then has access to the object on which it's supposed to operate[3]. Hence you'll find that most methods in Perl start with something equivalent to this:

```
package Bug;

sub print_me
{
    my ($self) = shift;
    # The @_ array now stores the explicit argument list passed to &Bug::print_me
    # The rest of the &print_me method uses the data referred to by $self and
    # the explicit arguments (still in @_)
}
```

or, better still:

```
package Bug;

sub print_me
{
    my ($self, @args) = @_;
    # The @args array now stores the explicit argument list passed to &Bug::print_me
    # The rest of the &print_me method uses the data referred to by $self and
    # the explicit arguments (now in @args)
}
```

This second version is better because it provides a lexically scoped copy of the argument list (`@args`). Remember that the `@_` array is magical in that changing any element of it actually changes the caller's version of the corresponding argument. Copying argument values to a lexical array like `@args` prevents nasty surprises of this kind and improves the internal documentation of the subroutine (especially if a more meaningful name than `@args` is chosen).

The only remaining question is: *how do we create the invoking object in the first place?*

3.1.3 Rule 3: To create an object, bless a referent

Unlike other object-oriented languages, Perl doesn't require that an object be a special kind of recordlike data structure. In fact, you can use *any* existing type of Perl variable—a scalar, an array, a hash—as an object in Perl.[4]

Hence, the issue isn't so much how to *create* the object—you create an object exactly as you would any other Perl variable— but rather how to tell Perl that such an object *belongs* to

---

[3] There are similar automatic features in all object-oriented languages. C++ member functions have a pointer called `this`; Java member functions have a reference called `this`; Smalltalk methods have the `self` pseudo-object; and Python's methods, like Perl's, receive the invoking object as their first argument.

[4] You can also bless other things, such as subroutines, regular expressions, and typeglobs, but we'll leave that for chapter 5.

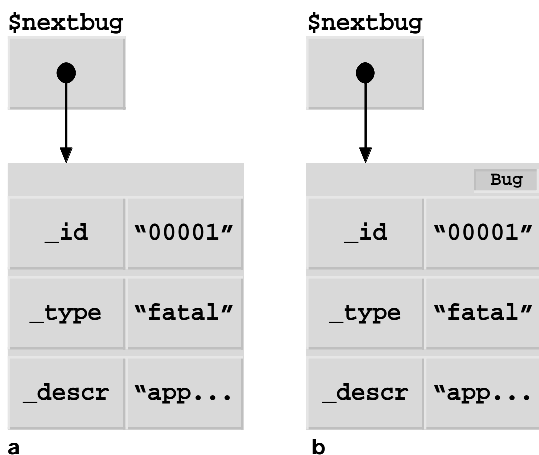

Figure 3.2 What changes when an object is blessed
a Before `bless($nextbug, "Bug")`
b After `bless($nextbug, "Bug")`

a particular class. That brings us to one extra built-in Perl function you need to know. It's called `bless`, and its only job is to mark a variable as belonging to a particular class.

The `bless` function takes two arguments: a reference to the variable to be marked and a string containing the name of the class. It then sets an internal flag on the variable, indicating that it now belongs to the class.[5]

For example, suppose that `$nextbug` actually stores a reference to an anonymous hash:

```
$nextbug =  {
                _id       => "00001",
                _type     => "fatal",
                _descr    => "application does not compile",
            };
```

To turn that anonymous hash into an object of class Bug we write:

```
bless $nextbug, "Bug";
```

And, once again, that's it! The anonymous array referred to by `$nextbug` is now marked as being an object of class Bug. The variable `$nextbug` itself hasn't been altered in any way. We didn't bless the *reference*; we blessed the *referent*. The scalar didn't change—only the nameless hash it refers to has been marked. Figure 3.2 illustrates where the new class membership flag is set.

You can check that the blessing succeeded by applying the built-in `ref` function to `$nextbug`. Normally, when `ref` is applied to a reference, it returns the type of that reference.

[5] Actually, the second argument is optional, and defaults to the name of the current package. However, as we'll see in chapter 6, although omitting the second argument may occasionally be convenient, it's never a good idea. It's better to think of both arguments as being morally required, even if legally they're not.

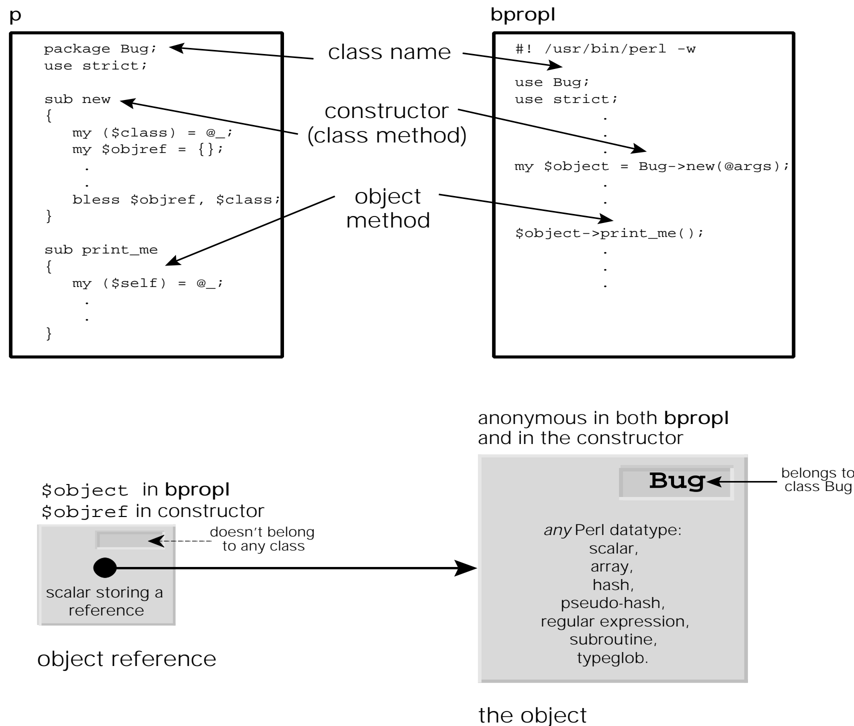

Figure 3.3 Object basics

Hence, before `$nextbug` was blessed, `ref($nextbug)` would have returned the string `'HASH'`.

Once an object is blessed, `ref` returns the name of its class instead. So, after the blessing, `ref($nextbug)` will return `'Bug'`. Of course the object itself still *is* a hash, but now it's a hash that *belongs* to the Bug class.

The entries of the hash become the attributes of the newly created Bug object. Note that in the above example each key begins with an underscore. This is the Perl convention for indicating that something is internal to a package, or, in this case, to a class. Here it's used to suggest that attributes and methods that are to be treated as not for public use.[6]

Given that we're likely to want to create many such Bug objects, it would be more useful if we had a subroutine that took the necessary information, wrapped it in an anonymous hash, blessed the hash, and gave us back a reference to the resulting object. And, of course, we might as well put such a subroutine in the Bug package itself and call it something that indicates its role. Such a subroutine is called a *constructor* and generally looks like this:

[6] Mind you, it's only a suggestion. Unlike other object-oriented languages, Perl doesn't enforce the encapsulation of attributes. More on that point shortly.

```
package Bug;

sub new
{
    my $class = $_[0];
    my $objref = {
                    _id       => $_[1],
                    _type     => $_[2],
                    _descr    => $_[3],
                 };
    bless $objref, $class;
    return $objref;
}
```

When we call `Bug::new`, we pass the name of the class into which the new object should be blessed (`"Bug"`), followed by the ID, type, and description of the bug. Of course, we can always hard-code the class name into the call to `bless`, but that loses us some important flexibility that we'll need later when we start inheriting from classes in chapter 6.

The `bless` function makes writing constructors like this a little easier by returning the reference passed as its first argument—that is, the reference to whatever it just blessed into objecthood. Since Perl subroutines automatically return the value of their last evaluated statement, that means we can condense the definition of `Bug::new` to:

```
sub Bug::new
{
    bless { _id => $_[1], _type => $_[2], _descr => $_[3] }, $_[0];
}
```

This version has exactly the same effects: slot the data into an anonymous hash, bless the hash into the class specified first argument, and return a reference to the hash.

Regardless of which version we use, whenever we want to create a Bug object, we can just call:

```
$nextbug = Bug::new("Bug", $id, $type, $description);
```

That's a little redundant, since we have to type "Bug" twice. Fortunately, there's another feature of the arrow method-call syntax that solves this problem. If the operand to the left of the arrow is the name of a class—rather than an object reference—the appropriate class method of that class is called. More importantly, if the arrow notation is used, the first argument passed to the method is automatically a string containing the class name. That means that we can rewrite the previous call to `Bug::new` like this:

```
package main;

$nextbug = Bug->new($id, $type, $description);
```

There are other benefits to this notation when your class uses inheritance (chapter 6), so you should always call constructors and other class methods this way.

Apart from encapsulating the messy details of object creation within the class itself, using a class method like this to create objects has another big advantage. If we abide by the convention of only creating new Bug objects by calling `Bug::new`, we're guaranteed that all such objects will always be hashes. Of course, there's nothing to prevent us from manually blessing

arrays, scalars, file handles, and so forth, as Bug objects, but life is *much* easier if we stick to blessing one type of object into each class.

For example, if we can be confident that any Bug object is going to be a blessed hash, we can finally fill in the missing code in the `Bug::print_me` method:

```
package Bug;

sub print_me
{
   my ($self) = @_;
   print "ID: $self->{_id}\n";
   print "$self->{_descr}\n";
   print "(Note: problem is fatal)\n"
      if $self->{_type} eq "fatal";
}
```

3.2 A SIMPLE PERL CLASS

Now, let's take the three rules explained above, plus `bless`, plus the arrow notation, and use them to build a simple, but usable, Perl class. We'll create a class that can be used to store information regarding a particular music CD (its name, the artist, publisher, ISBN, number of tracks, where it's stored in your extensive collection, etc.)

3.2.1 The code

The basic class is defined in listing 3.1. Take a few moments and puzzle through the code, keeping in mind the three rules given above.

Okay, now let's examine the entire class definition to see what's going on and, more importantly, why the class is strucured as it is.

Declaring the class

The first line is a straightforward application of the first rule. We want a new class to store information on music CDs, so we create a package called CD::Music. We could have called it something else, such as Music::CD, but the choice depends on what other related classes we expect to develop later. We might, for example, expect to create other classes for other types of CDs (CD::ROM, CD::WORM, CD::DVD, CD::Shiny::Beer::Mat). The common feature here is the "CD-ishness" of each type of object, and so we make that the more general term in the package name.

In contrast, if we had intended to develop classes for representing other types of music, we might have made Music:: the top level of the package name and had Music::CD, Music::LP, Music::Internet, Music::Of::The::Spheres, and so on.

Providing a constructor

Having successfully named the class, we ask Perl to be strict with us, which is *always* a good idea, no matter what kind of Perl programming we're doing. (See section 3.3.)

Listing 3.1 The CD::Music Class

```
package CD::Music;
use strict;

sub new
{
  my ($class) = @_;
  bless {
            _name       => $_[1],
            _artist     => $_[2],
            _publisher  => $_[3],
            _ISBN       => $_[4],
            _tracks     => $_[5],
            _room       => $_[6],
            _shelf      => $_[7],
            _rating     => $_[8],
        }, $class;
}

sub name       { $_[0]->{_name}       }
sub artist     { $_[0]->{_artist}     }
sub publisher  { $_[0]->{_publisher}  }
sub ISBN       { $_[0]->{_ISBN}       }
sub tracks     { $_[0]->{_tracks}     }

sub location
{
  my ($self, $shelf, $room) = @_;
  $self->{_room}= $roomif $room;
  $self->{_shelf}= $shelfif $shelf;
  return ($self->{_room}, $self->{_shelf});
}

sub rating
{
  my ($self, $rating) = @_;
  $self->{_rating} = $rating if $rating;
  return $self->{_rating};
}
```

We next provide a method for creating CD::Music objects. Note that the overall structure is very similar to the previous `Bug::new` example. We create an anonymous hash, fill in the relevant items, bless the hash, and return a reference to it.

The choice of a hash as the basis of both the Bug class and this one is no coincidence. Hashes are the usual basis for objects in Perl. That's because, unlike a scalar, a hash allows us to store multiple values of various types in the same object. And, unlike an array, a hash allows us to give each of those values a meaningful tag (i.e., its key).

Occasionally, for performance or other special reasons, it may be better to implement objects as something other than hashes. More importantly, Perl 5.005 introduced a new general purpose data type—the pseudo-hash—which is specially designed for implementing objects.

In the next two chapters, we'll look in detail at the pseudo-hash, as well as cases where other data types make more suitable bases for a class. For now, though, we'll stick with hashes.

Accessing an object's read-only data

The CD::Music class declares five methods: `CD::Music::name`, `CD::Music::artist`, `CD::Music::publisher`, `CD::Music::ISBN`, and `CD::Music::tracks`. Each method simply takes the blessed hash reference, looks up the corresponding attribute in the invoking object—that is, the appropriate entry in the hash—and returns its value. Such methods are called *accessors* because their whole purpose is to provide access to attributes

These methods provide a means of *reading* the different entries of the hash, but not of *overwriting* ~them. That restriction makes sense in this example, because the title, artist, publisher, ISBN, and number of tracks on a standard audio CD never change.

You might wonder why we would bother to declare these methods, expecting users of the class to write `$cdref->name()` when they already have a reference to the hash itself—`$cdref`—and can just use the normal Perl arrow syntax for accessing a particular entry: `$cdref->{_name}`.

The reason we don't want to encourage direct access is that those hash entries are collectively implementing the internal data of each object in the class, and one of the cardinal rules of object orientation is this: *Thou shalt not access thy encapsulated data directly, lest thou screwe it up*. You should look back at section 1.2.1 in chapter 1 if you're not sure why this is an important rule to honor.

Of course, unlike most other object-oriented languages, which *enforce* this kind of encapsulation, Perl will quite happily allow you to directly access the hash elements if you choose. But then you're not playing by the rules, and, when Bad Things happen, you'll only have yourself to blame.

If this philosophy of encapsulation by good manners strikes you as unnervingly insecure, take heart. Later in the chapter, and more extensively in chapter 11, we'll explore techniques for ensuring that your encapsulated data is truly unmolestable.

Accessing read-write data

The remaining two methods—`CD::Music::location` and `CD::Music::rating`—are slightly more complex. They still return the value of the appropriate hash entries, but, before that, they check their parameter lists to see if any new values have been specified for those elements.

For example, if `CD::Music::location` is called like so

```
$cdref->location(12)
```

then it:

- Sets the internal data `$cdref->{_shelf}` to 12, then
- Leaves the data in `$cdref->{_room}` unchanged (since no new value was provided), and finally
- Returns a list of the resulting room and shelf numbers.

Such methods are called *mutators* because they can change the internal state of an object.

The two methods can still be called without any argument (just like the other five accessor methods), in which case they just return the current value(s) of the relevant object data.

Catching attempts to change read-only attributes

Of course, because users of the CD::Music class can change the location or ratings information by passing new values to those two methods, they may well expect to do the same with `CD::Music::title`, `CD::Music::artist`, and so forth. This incorrect generalization could lead to subtle logical errors in the program, since those read-only methods will simply ignore any extra parameters they are given.

There are several ways to address this potential source of errors. The most obvious solution is to resort to brute force, and simply kill any program that attempts to call a read-only method with arguments. For example:

```
package CD::Music;
use strict;
use Carp;

sub read_only
{
    croak "Can't change value of read-only attribute " . (caller 1)[3]
        if @_ > 1;
}

sub name       { &read_only; $_[0]->{_name}      }
sub artist     { &read_only; $_[0]->{_artist}    }
sub publisher  { &read_only; $_[0]->{_publisher} }
sub ISBN       { &read_only; $_[0]->{_ISBN}      }
sub tracks     { &read_only; $_[0]->{_tracks}    }
```

Here, each read-only access method calls the subroutine `CD::Music::read_only`, passing its original argument list (by using the old-style call syntax—a leading `&` and no parentheses). The `read_only` subroutine checks for extra arguments and throws an informative exception if it finds any. Of course, there will always be at least one argument to any method, namely the object reference through which the method was originally called.

Think of this technique as a form of Pavlovian conditioning for programmers: every time their code actually attempts to assign to a read-only attribute of your class, their program dies. Bad programmer!

As enjoyable as it may be to mess with people's minds in this way, this approach does have a drawback; it imposes an extra cost on each attempt to access a read-only attribute. Moreover, it isn't proactive in preventing users from making this type of mistake; it only trains them not to repeat it after the fact.

Besides, psychology has a much more subtle tool to offer us, in the form of a technique known as *affordances*.[7] Affordances are features of a user interface that make it physically or psy-

[7] The concept of affordances comes from the work of user-interface guru Donald Norman. His landmark book *The Psychology of Everyday Things* (later renamed *The Design of Everyday Things*) is essential reading for anyone who creates interfaces of any kind, including interfaces to classes. See the bibliography for details.

chologically easy to do the right thing. For example, good designers don't put handles on unlatched doors that can only be pushed. Instead, they put a flat plate where the handle would otherwise be. Just about the only thing you can do with a plate is to push on it, so the physical structure of the plate helps you to operate the door correctly. In contrast, if you approach a door with a fixed handle, your natural tendency is to pull, which usually proves to be the right course of action.

Affordances work well in programming too. In this case, we want to make any attempt to change read-only object data psychologically awkward. The best way to do that is to avoid raising the expectation that overwriting this data is even possible.

For instance, we could change the names of the read-only methods to "`get_...`" and separate the two functions of each read-write accessor into distinct "`get...`" and "`set...`" methods:

```
package CD::Music;
use strict;

sub new
{
  # as before
}

sub get_name       { $_[0]->{_name}}
sub get_artist     { $_[0]->{_artist}}
sub get_publisher  { $_[0]->{_publisher}}
sub get_ISBN       { $_[0]->{_ISBN}}
sub get_tracks     { $_[0]->{_tracks}}
sub get_rating     { $_[0]->{_rating}}
sub get_location   { ($_[0]->{_room}, $_[0]->{_shelf}) }

sub set_location
{
    my ($self, $shelf, $room) = @_;
    $self->{_room}     = $room  if $room;
    $self->{_shelf}    = $shelf if $shelf;
}

sub set_rating
{
     my ($self, $rating) = @_;
     $self->{_rating} = $rating if $rating;
}
```

Now, the user of our class has no incentive to pass arguments to the read-only methods, because it doesn't make sense to do so. And, because no `set_name`, `set_artist`, and so on exist, it's obvious that these attributes can't be changed.

Method prototypes

You might be tempted to think that we could have avoided all this psychological manipulation by giving each method a prototype and letting the Perl compiler catch cases where the wrong number of arguments are passed to a method:

```
package CD::Music;

sub name();
sub rating(;$);
sub location(;$$);
# …etc.
```

Unfortunately, this idea doesn't actually work, because Perl doesn't check prototypes when a package subroutine is called as a method, using the `$objref->method(@args)` syntax).

There are good reasons why Perl ignores the prototypes of a method but, as they have to do with inheritance and polymorphism, we'll defer discussion of them until chapter 6.[8] For the moment, it's sufficient to remember not to rely on prototypes to safeguard your methods. Because they won't.

Accessing class data

So far, apart from constructors, we've only looked at the attributes and methods belonging to individual objects of a class. We may also need to implement attributes and methods shared by the class as a whole. These class attributes and class methods are typically provided to access and manipulate information that is not tied to a particular object.

For example, when using the CD::Music class, we might wish at some point to ascertain the total number of CD::Music objects created.[9] So far, that information is a collective property of the class, and so it won't be stored in any particular object.

Instead, we could modify the class as follows:

```
package CD::Music;
use strict;

{
      my $_count = 0;
      sub get_count     { $_count }
      sub _incr_count   { ++$_count }
}
sub new
{
      my ($class,@arg) = @_;
      $class->_incr_count();
      bless     {
                              _name=> $arg[0],
                              _artist=> $arg[1],
                              _publisher=> $arg[2],
                              _ISBN=> $arg[3],
```

---

[8] Oh, all right. The compiler can't check the prototypes because all Perl methods are polymorphic so, in general, it's not possible to know until run time which subroutine will actually be invoked in response to a particular method call.

[9] That's not the necessarily the same thing as the number of `CD::Music` objects *currently* in existence, since some objects may have ceased to exist in the interim. We'll explore that point further in the later section on *Destructors*.

```
                              _tracks=> $arg[4],
                              _room=> $arg[5],
                              _shelf=> $arg[6],
                              _rating=> $arg[7],
               }, $class;
}
```

The extra block just after `use strict` provides a nested lexical scope within the class. Within that scope, we declare a lexical variable (`my $_count`) and initialize it to zero. The `my` means that it is only visible within the scope of the current block (i.e., the nested scope). In object-oriented terms, it's encapsulated within the block and, therefore, within the CD::Music class.

Two methods—`CD::Music::get_count` and `CD::Music::_incr_count`—are also defined in the same block. They have access to this variable, though no other methods or subroutines defined anywhere else, including the other methods of CD::Music, are able to access the variable directly. Access to the methods themselves is not confined to the nested scope. Like all named Perl subroutines, they are not restricted to the scope in which they are defined but are globally accessible.

Normally, when we reach the end of a block, any lexical variables declared within it cease to exist. However, in this case, `$_count` avoids that end-of-scope annihilation because there are still two valid references to it outside the block, namely those within the bodies of `CD::Music::get_count` and `CD::Music::_incr_count`. See the section on *Closures* in chapter 2 if it's not clear why `$_count` goes out of scope, but not out of existence.

In any case, the result is that the CD::Music class now has two extra methods, and those methods provide the only general access to the variable `$_count`. The methods themselves are straightforward: `CD::Music::get_count` can be used to access the current value of the hidden `$_count` variable, while `CD::Music::_incr_count` can be used to increment the same variable. Note that `_incr_count`'s name starts with an underscore, which is the standard Perl convention for indicating that it's intended to be used only within the current package. Although, as usual, Perl in no way enforces that restriction. We'll come back to that point shortly.

The only other change required is to add a single command to the constructor to increment the global count every time a new CD::Music object is created. Now, whenever we need to know how many CD::Music objects have been created, we can call the class method to find out:

```
package main;

# Create and use some CD::Music objects, and then…

print "There have been ", CD::Music->get_count(), " CDs created\n";
```

The more bitterly experienced reader will already be protesting that there is no guarantee that the number returned by `CD::Music::get_count` bears *any* relationship to the actual number of CD::Music objects created, since the `CD::Music::_incr_count` method allows us to manipulate the count to our own nefarious ends:

```
package main;

# Create 100 CD::Music objects, and then…
for $i (1..100)
{
  push @cds, CD::Music->new($data[$i]);
}

# double our productivity!
for (1..100)
{
  CD::Music->_incr_count();
}

print "There have been ", CD::Music->get_count, " CDs created\n";
```

There are two answers to that. The simplest response is that calling a subroutine with a leading underscore that clearly marks it for internal use only just isn't playing by the rules. Programmers who do so thoroughly deserve the grief that inevitably results.

A more pragmatic answer is that it's not difficult to extend the nested scope and use a lexical subroutine reference to remove the dangerous subroutine from public accessibility:

```
package CD::Music;
{
     my $_count = 0;
     sub get_count { $_count }
     my $_incr_count  = sub { ++$_count };

     sub new
     {
          $_incr_count->();
          # etc. as before
     }

     # Other methods that need to adjust the
     # count value, via $_incr_count, go here
}

# Methods that don't need to adjust
# the count value go here
```

In this version, the counter increment subroutine is anonymous and only accessible via a reference stored in the lexical variable `$_incr_count`. That variable is, in turn, only accessible within the block that starts just after the package declaration, so `CD::Music::new` has access to the counter adjustment subroutine, but no code outside the block does. Problem solved.

Other readers might feel that this level of security is un-Perl-like, and possibly bordering on the paranoid, especially when we could get the same effect without all that barbed wire:

```
package CD::Music;
{
     my $_count = 0;
     sub get_count { $_count }

     sub new
     {
          ++$_count;
          # etc. as before
     }

     # Other methods that need to directly
     # access $_count value go here
}

# Methods that don't need to directly
# access $_count go here
```

This simpler solution may be satisfactory in a small application, but, even there, the decision to directly access class attributes is likely to come back and bite you as your code develops. In general, you are far more likely to future-proof your code if you consistently wrap *all* attribute accesses in subroutines. In chapter 6, for example, we will see how things can go horribly wrong with directly accessible class attributes when classes are inherited.

3.2.2 Using the CD::Music class

Once the class is written, we could go about using it like this:

```
package main;

# Create an object storing a CD's details
my $cd = CD::Music->new(   "Canon in D", "Pachelbel",
                           "Boering Mußak GmbH", "1729-67836847-1",
                           1,
                           8,8,
                           5.0);

# What's the CD called?
print $cd->name, "\n";

# Where would we find it?
printf "Room %s, shelf %s\n", $cd->location;

# Move it to room 5, shelf 3
$cd->location(5,3);

# How many CDs in the entire collection?
print CD::Music->get_count, "\n";
```

Just as with ordinary Perl subroutines, if a call to a method doesn't require arguments, we can omit the trailing empty parentheses after the method name. That is, we can write `$cd->name` and `CD::Music->get_count`, rather than `$cd->name()` and `CD::Music->get_count()`. However, unlike regular Perl subroutines, if a method *does* take arguments, you have to put them in parentheses. For example, you can't treat a method as an operator and write something like: `$cd->location 5, 3;`

It's also worth noting that method calls that return lists—for example, `$cd->location`—start with a `$`, not a `@`. This may seem inconsistent at first, but it really isn't. Only arrays and slices must be prefixed with a `@`. Actual lists, such as `(1,2,3)`, and subroutine calls that return lists, such as `caller()`, never have such a prefix.

3.3 MAKING LIFE EASIER

We can make the process of creating and using classes much easier in several ways. Putting them into separate modules is a good start, since it provides an extra level of encapsulation and a great deal more reusability than cutting-and-pasting. Turning on all the debugging features is another obvious way to reduce the unexpected. Finally, there's a useful shortcut that alleviates the repetitive task of setting up accessor methods.

3.3.1 Class modules

Once we have a usable class, the obvious thing to do is to put it into a module, so that its functionality is available to any code that might require it. If you're already familiar with writing modules, then you'll be used to the following procedure, as it's described in chapter 2:

- Create an appropriately named .pm file in an appropriately named subdirectory;
- Put your code in it and make sure the last statement evaluates to true;
- Arrange to import the module's interface—typically one or more subroutines—into the package that's going to use it.

To set up a module containing object-oriented code, the first two steps are exactly the same: put the code implementing the class into a suitable file and add a `1;` after it. Normally, we'd then set up a list of subroutines names to be exported, either by using the Exporter module or writing our own `import` subroutine. The question is: *what subroutines should we export from an object-oriented module?*

And the answer is, *absolutely none!*

The entire point of building a class is to encapsulate attributes and methods within the namespace of that class to ensure that they're accessed in a controlled manner. Exporting a class's attributes would compromise that encapsulation, so there's no reason to export any variables from an object-oriented module.

Exporting the methods of a class usually doesn't make much sense either, since methods are always supposed to be called through an invoking object, or through the package itself. In either case, Perl will automatically look for the method in the namespace of the class package, not the namespace to which subroutines are exported.

For the moment, just remember that you don't need to export anything from an object-oriented module (but see chapter 14 for some interesting exceptions).

3.3.2 use strict and the -w flag

Using `use strict` and the `-w` flag in serious code should be second nature. Perl's range of diagnostics is exceptionally comprehensive, and the compiler is remarkably adept in identifying even the most arcane of semantic mistakes. Even when it guesses wrong, the error

messages it generates will still tell you that something is amiss. By turning on those facilities, you will save yourself hours of time puzzling over the unexpected behavior of your code.

In object-oriented code, `use strict` will pick up nasty little traps such as this:[10]

```
package CD::Music;

# WARNING: BAD CODE AHEAD…

sub set_location
{
     my ($self, @loc) = @_;
     $self{_room} ||= $loc->[0];
     $self{_shelf}||= $loc->[1];
     return;
}
```

Occasionally the `-w` flag will nag about things that you know are okay in your particular code. Rather than switching off all warnings, you can temporarily switch off warnings by localizing the `$^W` variable:

```
# This code may generate a warning if "more" is unavailble,
# but it's okay to ignore it…
sub print_paged
{
   my ($self) = @_;
   local $^W;                                  # Locally reset warning switch
   local STDOUT;
   open STDOUT, "|more" or open STDOUT, ">-"; # Might generate warning
   $self->print_me;
}
# Warnings are active again from this point in the execution
```

Be careful, however, since the localized `$^W` variable propagates into any subroutine called from `CD::Music::print_paged` (for example, into `CD::Music::print_me`). This could mask other problems elsewhere in your code.

Some people are also reluctant to give up the syntactic liberties that `use strict` denies them. Indeed, the documentation on the `use strict` pragma suggests that, in its full glory, it is too stringent for casual programming. But, whatever casual programming may be, it is almost never object-oriented programming, so it's a wise move always to include a `use strict` at the top of any object-oriented code you create.

Remember, though, that a `use strict` pragma only affects code that *follows* it in the same scope, so put it near the top of your module. You should also be aware that, although `use strict` respects block scopes, it ignores package boundaries. So if you give one package a `use strict`, it may also apply to any packages that appear later in the file.

[10] If it's not immediately obvious what's wrong in the code, you *definitely* need to `use strict`! The problem is that `$self{_room}` is accessing an entry in the package variable `%CD::Music::self`, not the one in the hash referred to by `my $self`. Likewise `$loc->[0]` is attempting to access the first element of the array referred to by the scalar package variable `$CD::Music::loc`, not the first element of `my @loc`.

3.3.3 Automating data member access

Previously, we saw how to create accessor methods to provide access to an object's data in a controlled manner. For example, class CD::Music defined the following read-only data accessors:

```
package CD::Music;

sub get_name       { $_[0]->{_name}      }
sub get_artist     { $_[0]->{_artist}    }
sub get_publisher  { $_[0]->{_publisher} }
sub get_ISBN       { $_[0]->{_isbn}      }
sub get_tracks     { $_[0]->{_tracks}    }
sub get_rating     { $_[0]->{_rating}    }
sub get_location   { ($_[0]->{_room}, $_[0]->{_shelf}) }
```

Even such simple accessors quickly become tedious to write, especially if there are many of them. Apart from the tedium, it's easy to be mesmerized into making a mistake, as we just did with the `get_ISBN` method above. (It should access `$_[0]->{_ISBN}`, not `$_[0]->{_isbn}`.) Mistakes such as this can be hard to track since the compiler gives no warning of them, even (alas!) under `use strict` and `-w`.

Sometimes a better solution is to provide a single catchall method that can be called in response to any attempt to call an accessor. Packages already provide the ability to define such a catchall by defining a subroutine called `AUTOLOAD`. Since a Perl class is just a package with delusions of grandeur, it should come as no surprise that we can use `AUTOLOAD` as a catchall for methods as well.

For example, we can replace the series of "`get_`..." methods with the following:

```
package CD::Music;
use strict;
use vars '$AUTOLOAD';                         # keep 'use strict' happy

sub AUTOLOAD
{
  my ($self) = @_;
  $AUTOLOAD =~ /.*::get(_\w+)/                # extract attribute name
    or croak "No such method: $AUTOLOAD";
  exists $self->{$1}                          # locate attribute
    or croak "No such attribute: $1";
  return $self->{$1}                          # return attribute
}

sub get_location { ($_[0]->{_room}, $_[0]->{_shelf}) }

# But don't define the other get_… methods
```

Now, whenever Perl fails to find a method for an object of class CD::Music, the `CD::Music::AUTOLOAD` method is invoked instead. The `AUTOLOAD` method itself is simple. It's invoked just like the methods it replaces; that is, with a reference to an object passed as its first argument. The name of the method actually requested is provided in the `$AUTOLOAD` package variable.

Therefore, if the original method call was: `$cdref->get_artist()`, then the catchall method `CD::Music::AUTOLOAD` is called with one argument—the object reference stored in `$cdref`—and the package variable `$CD::Music::AUTOLOAD` contains the string `"CD::Music::get_artist"`.

The CD::Music class's `AUTOLOAD` first uses a regular expression to locate and extract (as `$1`) the name of the actual object attribute being requested. It checks that the requested attribute is in fact present in the object and then it returns the corresponding value.

If that extract-and-lookup process fails for any reason—either because the method name didn't have a `get_` prefix, or because the corresponding entry didn't exist in the object hash—`CD::Music::AUTOLOAD` gives up and throws an appropriate exception.

We still need to provide an explicit definition for `get_location` since it doesn't fit into the common structural pattern that `AUTOLOAD` simulates. Since the `AUTOLOAD` method for a class is only called if normal method lookup fails, the explicit version of `get_location` is found and called before `AUTOLOAD` is considered.

Problems still arise with the above version of `AUTOLOAD` if the CD::Music class also uses hash entries to implement nonpublic attributes of an object. For example, if an entry with the key `'_read_count'` is used to track how often each object has been read-accessed, then the previous `AUTOLOAD` allows that internal data to be accessed via a call to the accessor `$cdref->get__read_count()`. We can provide better control by making `AUTOLOAD` a little smarter:

```
package CD::Music;
use strict;
use vars '$AUTOLOAD';  # Keep 'use strict' happy

{
  my %_attrs =
     (  _name        => undef,
        _artist      => undef,
        _publisher   => undef,
        _ISBN        => undef,
        _tracks      => undef,
        _rating      => undef,
        _room        => undef,
        _shelf       => undef,
     );

  sub _accessible { exists $_attrs{$_[1]} }
}
sub AUTOLOAD
{
  my ($self) = @_;
  $AUTOLOAD =~ /.*::get(_\w+)/
     or croak "No such method: $AUTOLOAD";
  $self->_accessible($1)
       or croak "No such attribute: $1";
  $self->{_read_count}++;
  return $self->{$1};
}
```

In this version, AUTOLOAD checks the requested attribute name against a predefined list of publicly accessible attributes, rather than simply checking for existence in the object. The keys of the encapsulated hash %_attrs enumerate the attributes to which AUTOLOAD is allowed to provide access. Notice that we use the nested scope trick again to encapsulate the internal data and provide controlled access to it via a method.

In the above example, the values of the %_attrs hash convey no useful information. But they could. For instance, we can arrange for AUTOLOAD to handle the "set_…" methods of the class as well. The values of %_attrs can then be used to indicate whether a particular attribute is read-only or writable as well. That requires further modifications to the above code, as shown in listing 3.2.

The tests are a little more "Perlified," but the only significant difference in this version is that the class method CD::Music::_accessible now checks whether the specified attribute is accessible in the required mode (i.e., 'read' or 'write').

The use of an encapsulated hash to specify the valid attributes of a class and other related information is a technique commonly used in object-oriented Perl. We'll see variations on this approach at the end of this chapter and in chapter 4.

Reducing the cost of autoloading

The convenience of having accessor methods conjured up for us whenever they're needed comes at a price. In order to determine that autoloading is required, Perl must first attempt to locate a suitable method in the current class and fail to do so, invoking the AUTOLOAD instead. That's more expensive than just finding the method and calling it. As we'll see in chapter 6, if the class also inherits from another class, the search for the correct method becomes even more expensive, as does locating the appropriate AUTOLOAD method.

Even when the AUTOLOAD method is eventually invoked, it's less efficient than a hard-coded method would be. In the CD::Music class, for example, it has to identify the method with one or two pattern matches, determine whether the method is callable (with another method call to accessible), and, finally, simulate the method itself. By comparison, a hard-coded method could simply do its job immediately without any identification or verification phases.

Worst of all, the CD::Music class never learns from the experience of resorting to autoloading. The next time the same method is called, AUTLOAD will be forced to go through the same expensive lookup–identify–verify sequence all over again.

Fortunately, because Perl provides direct run-time access to a package's symbol table, it's easy to extend an AUTOLOAD method so that all those extra costs are incurred only the first time that AUTOLOAD is required to implement a particular method. In other words, with surprisingly little extra effort, we can arrange for AUTOLOAD to teach its class a new method whenever one is needed:

```
sub CD::Music::AUTOLOAD
{
   no strict "refs";
   my ($self, $newval) = @_;

   # Was it a get_… method?
```

Listing 3.2 A smart AUTOLOAD method for the CD::Music class

```
package CD::Music;
use strict;
use vars '$AUTOLOAD';    # Keep 'use strict' happy

# constructor and destructor, as before…
# and then…

{
  my %_attrs =
     (  _name      => 'read',
        _artist    => 'read',
        _publisher => 'read',
        _ISBN      => 'read',
        _tracks    => 'read',
        _rating    => 'read/write,
        _room      => 'read/write',
        _shelf     => 'read/write',
     );

  sub _accessible
  {
     my ($self, $attr, $mode) = @_;
     $_attrs{$attr} =~ /$mode/
  }
}
sub AUTOLOAD
{
  my ($self, $newval) = @_;

  # Was it a get_… method?
  $AUTOLOAD =~ /.*::get(_\w+)/
     and $self->_accessible($1,'read')
     and return $self->{$1};

  # Was it a set_… method?
  $AUTOLOAD =~ /.*::set(_\w+)/
     and $self->_accessible($1,'write')
     and do { $self->{$1} = $newval; return }

  # Must have been a mistake then…
  croak "No such method: $AUTOLOAD";
```

```
    if ($AUTOLOAD =~ /.*::get(_\w+)/ && $self->_accessible($1,'read'))
    {
        my $attr_name = $1;
        *{$AUTOLOAD} = sub { return $_[0]->{$attr_name} };
        return $self->{$attr_name}
    }

    # Was it a set_… method?
    if ($AUTOLOAD =~ /.*::set(_\w+)/ && $self->_accessible($1,'write'))
    {
        my $attr_name = $1;
        *{$AUTOLOAD} = sub { $_[0]->{$attr_name} = $_[1]; return };
        $self->{$1} = $newval;
        return
    }

    # Must have been a mistake then…
    croak "No such method: $AUTOLOAD";
}
```

Note how similar this version is to the one shown in figure 3.2. The difference is that here, when `AUTOLOAD` determines that a valid `get_…` or `set_…` accessor has been called, it *creates* an optimized version of that accessor (as an anonymous subroutine) and then installs that accessor in the appropriate symbol table.[11]

The anonymous subroutine that `AUTOLOAD` creates is a closure, so it remembers the value of the lexical `$attr_name` variable even after that variable goes out of scope. That way, each subroutine generated by `AUTOLOAD` is specific to whichever attribute is required for the get or set operation that `AUTOLOAD` is currently handling.

By installing the anonymous subroutine in the package's symbol table in response to a method call, we have effectively created a new method of the same name within the class. Next time that method is called, the look-up mechanism will find an entry for it in the symbol table and immediately call the corresponding subroutine. `AUTOLOAD` will no longer be required to handle calls to that particular method, which will now be executed much more quickly.

3.3.4 Documenting a class

Having written the code, the task of building a class is approximately half done. If the class is to be anything more than a one-off, throw-away convenience, it needs to be documented.

Perl makes documenting code particularly easy. You can embed documentation written in the POD markup language right in your module, even interspersing it through the code if you wish. The perlpod documentation that comes with Perl explains how to document your code. This section provides a guide on *what* to document.

When documenting a class, you need to provide users with at least the following information:

---

[11] …by assigning it to the typeglob `*{$AUTOLOAD}`. Since `$AUTOLOAD` holds the full name of the required method, it can be used as a symbolic reference into the symbol table. See the section on *Symbolic references* in chapter 2.

- The name and purpose of the class.
- The version of the class which the documentation documents.
- A brief synopsis of how the class is used.
- A more extensive description of how the class is used. This should include specific documentation on how to create objects of the class, what methods those objects provide, what class methods are available, and any special features or limitations of the class.
- A complete list of diagnostics that the class is likely to generate (whether they be exceptions thrown, special values returned, or warning messages generated), plus a description of likely error conditions that the class will not be able to diagnose itself.
- Any environment variables or files that can—or must—be used.
- Any other modules that the class relies on, and how to obtain them if they're not available on the CPAN.
- A list of any known bugs, with suggested workarounds.
- Cross-references to any other relevant documentation.
- A copyright notice.
- The name and contact details of the author or authors.

Listing 3.3 provides a POD skeleton of suitable documentation for a class.

3.4 THE CREATION AND DESTRUCTION OF OBJECTS

The object-oriented features of Perl have been around long enough for many conventions and *idioms* to have evolved. We've already discussed a number of those that relate to methods and attributes. In this section, we'll look at a few conventions that users of your Perl classes will expect you to observe in regard to the creation and removal of objects.

However, as with many aspects of Perl programming, these matters are customs, not graven in stone. You are free to ignore any or all of them, though that may get your code talked about.[12]

3.4.1 Constructors

By convention, each Perl class provides a class method that can be called to produce new objects of the class. That method is called a constructor (as it is in C^{++} or Java) and, just as in the examples above, it is usually called `new`. Of course, it's perfectly legitimate to call your constructor `create`, `make`, `conjure_forth_from_the_Eternal_Void_I_adjure_thee`, or anything else that's appropriate for your application, but `new` has the three distinct advantages of being short, accurate, and predictable.

Some object-oriented programmers prefer to completely separate the process of object creation from the process of initialization, and so provide two methods: `new` to create the object, and `init` to set up its internal data. This type of fastidiousness makes sense in other languages where the process of construction may fail, often with fatal and hard-to-detect consequences, but it's usually misplaced and excessively paranoid in Perl. Besides, such behavior has the overwhelming disadvantage of making it inevitable that someone will create an ob-

[12] …or worse still, ignored.

Listing 3.3 Class documentation template

```
=head1 NAME

Full::Class::Name - One line summary of purpose of class

=head1 VERSION

This document refers to version N.NN of Full::Class::Name,
released MMMM DD, YYYY.

=head1 SYNOPSIS

  # Short examples of Perl code that illustrate the use of the class

=head1 DESCRIPTION

=head2 Overview

=head2 Constructor and initialization

=head2 Class and object methods

=head2 Any other information that's important

=head1 ENVIRONMENT

List of environment variables and other O/S related information
on which the class relies

=head1 DIAGNOSTICS

=over 4

=item "error message that may appear"

Explanation of error message

=item "another error message that may appear"

Explanation of another error message

etc…

=back

=head1 BUGS

Description of known bugs (and any work-arounds).
Usually also includes an invitation to send the author bug reports.

=head1 FILES

List of any files or other Perl modules needed by the class and a brief
explanation why.

=head1 SEE ALSO

Cross-references to any other relevant documentation.

=head1 AUTHOR(S)

Name(s)
(email address(s))

=head1  COPYRIGHT

Copyright (c) YYYY(s), Author(s). All Rights Reserved.
This module is free software. It may be used, redistributed
and/or modified under the same terms as Perl itself.
```

ject and fail to initialize it. Better to put all your object initialization in a single constructor method.

That's not to say that you shouldn't separate the two processes *within* the constructor. If your initialization sequence is even moderately complex, you should consider putting it in a separate method, like so:

```
package CD::Music;
use strict;

sub new
{
     my $self = {};
     bless $self, shift;
     $self->_incr_count();
     $self->_init(@_);
     return $self;
}

{
     my @_init_mems =
         qw( _name _artist _publisher _ISBN _tracks _room _shelf _rating );

     sub _init
     {
         my ($self,@args) = @_;
         my %inits;
         @inits{@_init_mems} = @args;
         %$self = %inits;
     }
}
```

In this version, `CD::Music::new` performs only the actions associated with object creation: incrementing the global object count, creating the anonymous hash that implements the object, blessing that hash into the class. Then the `CD::Music::_init` method is called to populate the hash with appropriate values before `CD::Music::new` returns a reference to the new object. Note that `_init` is underscored to indicate that it's nonpublic. As always, nothing enforces this distinction except the goodwill of any client code.

The way in which `CD::Music::_init` goes about populating the hash is interesting. It first creates a temporary hash (`%inits`) and immediately generates a "slice" of it (`@inits{@_init_mems}`). This slice is then assigned the various arguments `_init` was passed. Finally, the now-initialized temporary hash is assigned back to the blessed object being initialized.

This approach offers the advantage that adding another data member to the class is now simply a matter of adding another element to the secret `@_init_mems` array—and remembering to pass the correct argument list.

Another way to call a constructor

Perl provides a second syntax for calling a constructor, or any other method belonging to a class. It's known as the indirect object syntax, and it's already familiar to you. We'll discuss it here, and then you should tear out this page and eat it, so that you'll never be tempted to use the syntax. You'll see why shortly.

The general forms of the syntax are:

```
methodname OBJECTREF ARGLIST
methodname CLASSNAME ARGLIST
methodname BLOCK ARGLIST
```

In other words, it's exactly like the standard print-to-a-file handle syntax:

```
print STDERR "arg", "u", "mentl", "ist";
```

So what does that have to do with constructors? Well, the same indirect syntax that we use for printing is available to call any method. So you can call a constructor like this

```
my $cd = new CD::Music (   "Toccata and Fugue", "J.S.Bach",
                           "Classic Records", "1456-432443424-2",
                           6, 2,7, 9.5);
```

Many programmers prefer this indirect object syntax, at least for constructor calls, since it's less densely punctuated and more reminiscent of constructor invocation in several other object-oriented languages.

The indirect object syntax does, however, suffer from the same type of ambiguity problems that sometime befuddles `print`. Provided Perl has already seen the class name before it reaches the indirect object call, there's no problem as long as you always use a bareword class name or a scalar variable as the *CLASSNAME* element:

```
use CD::Music;

$CDM = 'CD::Music';

my $cd1 = new CD::Music (@data);          # okay
my $cd2 = new $CDM (@data);               # okay too
```

Things don't go so well if you use a function call in that position:

```
my $cd3 = new get_classname() (@data);    #Compilation error!
```

So what, (you're thinking) *when am I* ever *going to do something arcane like that?* Well, you'd be surprised how easy it is. Suppose you were trying to follow good software engineering practice and factor out a widely used explicit string into a predefined constant:

```
package main;
use constant CLASS => "CD::Music";

# and later…

new CLASS (@data);
```

Oops! The constant.pm module works by defining a tiny subroutine called `main::CLASS`, like this:

```
sub main::CLASS() { return "CD::Music" }
```

Now you *do* have a function call in the *CLASSNAME* slot, and the compiler gets confused. In fact, because it's looking for a parameter list straight after the function name, the compiler thinks that you're trying to call `main::CLASS` with the argument list `(@data)`, and use the result as the argument list to a normal subroutine (`main::new`). In other words, what was intended to be a CD::Music constructor call is parsed as if it were

```
main::new( main::CLASS(@data) );
```

Of course, since the empty prototype for `main::CLASS` forbids it to take any arguments, and the main package probably doesn't have a `new` subroutine, the compiler rejects the entire expression (with an obscure error message complaining that **unquoted string new may clash with future reserved word...**).

You *can* use a predeclared pseudo-constant like `CLASS`, but, to do so, you have to use the third form the indirect object syntax and put the function call in its own block:

```
new {CLASS} (@data);
```

This doesn't seem all that much clearer than an explicit method call (`CLASS->new(@data)`). It's probably safer to stick with the direct call syntax, which doesn't have any special cases.

An even better reason not to use the indirect object syntax is that another inherent ambiguity exists when a method is called without arguments. For example, consider a call intended for the `CD::Music::get_name` method:

```
package main;

print(get_name $cd);
```

Now, because the indirect object notation can be used with any method, that should be identical to

```
package main;

print($cd->get_name);
```

And it usually is. The only problem occurs if `main` happens to have its own subroutine called `name`. In that case, the compiler assumes you are calling `main::get_name` *without* parentheses around its arguments, as if you'd meant

```
package main;

print( main::get_name($cd) );
```

Even if you use a bareword, as is commonly the case when calling a constructor, things can go horribly wrong if the class you want is declared too late in the program. For example

```
package SeenFirst;
sub new { print "called &SeenFirst::new('$_[0]')\n" }

package main;
sub new { print "called &main::new('$_[0]')\n" }

SeenFirst->new();
new SeenFirst;

SeenLater->new();
new SeenLater;

package SeenLater;
sub new { print "called &SeenLater::new('$_[0]')\n" }
```

The four (supposed) constructor calls actually produce the following output:

called &SeenFirst::new('SeenFirst')
called &SeenFirst::new('SeenFirst')
called &SeenLater::new('SeenLater')
called &main::new('SeenLater)

because, at the point where it executes the statement `new SeenLater`, Perl doesn't yet know that `SeenLater` is a class name. Consequently, Perl doesn't realize that the call is supposed to be to an indirect object method, rather than a regular subroutine.

All in all, the seductive intuitiveness of the indirect object syntax probably isn't worth either the burden of remembering when it's safe to use or the pain of tracking down obscure bugs like these when it isn't. Stick with the arrow syntax for all methods, including constructors and class methods.

Constructor argument lists

One of the most pleasant features of object-oriented programming is that method calls don't generally involve passing a long list of arguments. That's because most of the data a method needs is typically already stored in the object on which the method is called.

Constructors are frequently an exception to this rule, because they exist to convey data to an object, rather than extract an object's data or internally manipulate it. For example, the constructor for the CD::Music class takes eight arguments, whereas no other method of that class takes more than two, and the majority take none.

Such argument-laden methods are painful to call because it's easy to get the argument order wrong or forget an argument. An alternative and safer way to pass data to a constructor is to pass the argument list as though it were a hash (see *Named arguments* in chapter 2).

For example we could rewrite `CD::Music::new` as follows:

```
package CD::Music;
use strict;

sub new
{
     my ($class, %arg) = @_;
     $class->_incr_count();
     bless {
                _name       => $arg{name},
                _artist     => $arg{artist},
                _publisher  => $arg{publisher},
                _ISBN       => $arg{ISBN},
                _tracks     => $arg{tracks},
                _room       => $arg{room},
                _shelf      => $arg{shelf},
                _rating     => $arg{rating},
           }, $class;
}
```

Now, the creation of a new CD::Music object is self-documenting, and the data can be specified in any convenient order:

```
my $cd = CD::Music->new(   name      => "Piano Concerto 20",
                           artist    => "Mozart",
                           rating    => 10,
                           room      => 5,
                           shelf     => 1,
                           publisher => "Salieri Intl.",
                           ISBN      => "1426-43235624-2",
                           );
```

The use of named arguments is certainly not a universally applied convention, and is occasionally a topic of minor philosophical debate, but the greater verbosity of named arguments more than pays for itself every time code has to be maintained. As a rule of thumb, if your constructor takes more than two or three arguments, it will be far easier to use if those arguments are named.

Constructor default values

Did you notice in the previous example that we neglected to specify how many tracks there were? Of course, even with named parameters, if an argument is accidentally omitted when a constructor is called, the corresponding internal data will be undefined. That may be a reasonable default value, especially if it's a conscious choice, or it may be better to provide explicit defaults for any missing value:

```
package CD::Music;
use strict;

sub new
{
     my ($class, %arg) = @_;
     $class->_incr_count();
     bless{
               _name       => $arg{name}       || croak("missing name"),
               _artist     => $arg{artist}     || "???",
               _publisher  => $arg{publisher}  || "???",
               _ISBN       => $arg{ISBN}       || "???",
               _tracks     => $arg{tracks}     || "???",
               _room       => $arg{room}       || "uncataloged",
               _shelf      => $arg{shelf}      || "",
               _rating     => $arg{rating}     || ask_rating($arg{name}),
          }, $class;
}
sub ask_rating { print "What is your rating for $_[0]? "; scalar <> }
```

The defaults specified in this way may be explicit values, or a particular action (such as throwing an exception, or prompting for missing data).

The use of the `||` operator—a common Perl idiom—means that no argument can have the value `0`, since the left-hand side of the operation would then be false, so the default on the right-hand side would used. If valid values of `0` and other false values like `"0"`, or `""` are required, the above code would have to be uglified with the ternary operator instead:

```
bless{
        _name=> defined($arg{name}) ? $arg{name} : croak("missing name"),
        _artist=> defined($arg{artist}) ? $arg{artist} : "???",

        # etc.

     }, $class;
```

Constructors as object duplicators

We can get even more sophisticated. For example, we can arrange that, whenever the constructor is called as an object method (with no arguments), the values for the newly created object are taken from the existing object through which the constructor was called. In other words, the constructor can also act like a copy operation.

Note that a constructor call is required. Simple object-to-object assignment won't do the trick for several reasons:

- A simple assignment of object references (`$objref1 = $objref2`) doesn't copy the referent. Both variables will end up pointing to the same object.
- A simple assignment of the underlying objects (`%$objref1 = %$objref2`) won't assign the blessing of the original object to the new one, nor will it adjust a class attribute that keeps count of objects.
- Even if we *could* assign the objects with their blessings, *and* have the count correctly updated, if any of the original object's attributes themselves contained references, we'd have the same problem all over again, only at the next level down.

Thus, Perl's shallow copy semantics frustrates our desire to use simple assignment as a copying mechanism.

The way the constructor was previously set up, to copy an object we'd first have to determine its type (using the built-in `ref` function). We could then create an object of the same type by calling `new` via the resulting class name. Finally, we could initialize the newly created object with a simple hash-to-hash assignment. Like this:

```
%{$objref2 = ref($objref1)->new()} = %{$objref1};
```

Provided the object doesn't contain any nested references, this works quite well. But it's not obvious to code, nor easy to understand once coded. Laziness is a cardinal virtue in Perl, so a custom has developed that when users of a class call a class's constructor as an object method, the defaults that the constructor uses are taken from the original object. This means that the copy operation can be accomplished just by writing

```
$objref2 = $objref1->new();
```

To implement that behavior we need one extra tweak in the CD::Music constructor:

```
package CD::Music;

 {
     my $_class_defaults =
     {  _name        => "???",
        _artist      => "???",
```

```
        _publisher  => "???",
        _ISBN       => "???",
        _tracks     => "???",
        _room       => "uncataloged",
        _shelf      => "",
        _rating     => -1,
    };

    sub _class_defaults     { $_class_defaults }
    sub _class_default_keys { map { tr/_//d; $_ } keys %$_class_defaults}
}
sub new
{
    my ($caller, %arg) = @_;
    my $class = ref($caller);
    my $defaults = $class ? $caller : $caller->_class_defaults();
    $class ||= $caller;
    $class->_incr_count();
    my $self = bless {}, $class;
    foreach my $attrname ( $class->_class_default_keys )
    {
        if (exists $arg{$attrname})
           { $self->{"_$attrname"} = $arg{$attrname} }
        else
           { $self->{"_$attrname"} = $defaults->{"_$attrname"} }
    }
    return $self;
}
```

In this version, `CD::Music::new` first determines if `$caller` is an object, in which case `ref($caller)` returns a class name. In that case, the default values for the initialization should come from that object, so `$defaults` is made to refer to the object, `$caller`, itself. If `ref($caller)` returns an empty string instead, then `$caller` itself must have stored a class name. That is, the constructor must have been called as a class method: `CD::Music->new(@data)`. In that case, the default values are taken from the class itself—`$caller->_class_defaults()`—and `$class` is reassigned the value of `$caller` (i.e., the class name).

After that point, we are guaranteed that `$class` stores the class name into which the object is to be blessed, and `$defaults` stores a reference to a hash with entries suitable for use as default values. That hash may be another object, or it may be the hash referred to by `$_class_defaults`, but we no longer care which.

Having blessed an empty hash into the class, all that is required is to initialize that object by stepping through each valid internal datum—conveniently specified by the keys of the class defaults. For each key, we strip the leading underscore to generate the external name, `$attrname`, that would have been used to label the corresponding argument to `CD::Music::new`. We assign either the argument passed to the constructor, `$arg{$attrname}`, or, if no suitable argument was passed, the default value from the hash referred to by `$caller`, `$caller->{"_$attrname"}`.

With a constructor like this, we can now copy an existing object:

```
my $cdref2 = $cdref1->new();
```

Or we can copy and modify an object in one step:

```
my $cdref2 = $cdref1->new(  name    => "Also Sprach Zarathustra",
                            artist  => "Strauss");
```

Or just use the standard defaults

```
my $cdref2 = ref($cdref1)->new();
```

This last constructor call *doesn't* use the values in the object referred to be `$cdref1`, because `ref($cdref1)` is the name of the object's class, not a reference to the object itself.

A separate clone method

While many experts view the overloading of constructors as object copiers to be a natural extension of their functionality, others consider the technique unintuitive, too subtle, and more likely to produce obscure and hard-to-maintain code. Whether or not that's the case, it certainly *is* true that the code for the constructors themselves is considerably more complicated.

The alternative is to provide a completely separate method for duplicating objects. Such a method is typically called `clone` or `copy` and would be implemented like this:

```
package CD::Music;

# constructor as in earlier versions

sub clone
{
      my ($self) = @_;
      my $class = ref($self);
      $class->_incr_count();
      bless { %{$self} }, $class;
}
```

The new `clone` method reproduces the essential behavior of the CD::Music constructor—incrementing the object count, then blessing a hash as the new object—but requires much less initialization code, since it can simply copy the contents of the existing object, `%{$self}`, on the assumption that it's already correctly structured.

However, this approach is not always sufficient, particularly if some of a class's attributes are implemented as references. In such cases we need to copy each reference attribute from the `$self` object separately. For example, if the `"_artist"` and `"_ISBN"` attributes were actually references to objects of the classes Artist and ISBN (each with its own `clone` method), then we would have to implement the `CD::Music::clone` method like this:

```
sub clone
{
      my ($self) = @_;
      my $class = ref($self);
      $class->_incr_count();
      my $newobj = bless { %{$self} }, $class;
      $newobj->{_artist} = $self->{_artist}->clone();
      $newobj->{_ISBN}   = $self->{_ISBN}->clone();
      return $newobj;
}
```

Whether you decide to provide object cloning facilities implicitly (as part of a standard constructor), or explicitly (as a separate method), depends on the nature of the application you're building, and—more importantly—on the expectations and conventions of those who may use your code. If you have a choice, it's probably better to code a separate `clone` method. Apart from keeping your individual methods simpler and less prone to bugs, the method's name will force client code to be more clearly self-documenting.

3.4.2 Destructors

Most object-oriented languages provide the ability to specify methods that are called automatically when an object ceases to exist. Such methods are usually called *destructors* and are used to undo any side-effects caused by the previous existence of an object, including:

- Deallocating related memory (although, in Perl, that's almost never necessary since reference counting usually takes care of it for you);
- Closing file or directory handles stored in the object;
- Closing pipes to other processes;
- Closing databases used by the object;
- Updating classwide information;
- Anything else that the object should do before it ceases to exist (such as logging the fact of its own demise, or storing its data away to provide persistence, etc.).

In Perl, you can set up a destructor for a class by defining an object method called `DESTROY`. The method is automatically called on an object just before that object's memory is reclaimed. That happens either as soon as the program loses its last reference to the object—that is, when the object's reference count reaches zero—or when the interpreter thread in which the object was created shuts down. Typically, the destructor is called when the last variable holding a reference to the object goes out of scope or has another value assigned to it.

For example, we could provide a destructor for the CD::Music class like this:

```
package CD::Music;

sub DESTROY
{
   my ($self) = @_;
   print "<<here lies the noble '", $self->name(), "'>>\n";
}
```

Now, every time an object of class CD::Music is about to cease to exist, that object will automatically have its `DESTROY` method called, which will print an epitaph for the object. For example, the following script

```
package main;
use CD::Music;

open CDDATA, "CD.dat" or die "Couldn't find CD data";
while (<CDDATA>)
{
     my @data = split ',', $_;
     my $cd = CD::Music->new(@data);
```

```
    print "Title: ", $cd->title, "\n";
}
print "(end of list)\n";
```

prints out something like the following

```
Title: Canon in D
<<here lies the noble 'Canon in D' >>
Title: Toccata and Fugue
<<here lies the noble 'Toccata and Fugue' >>
Title: Concerto in D
<<here lies the noble 'Concerto in D' >>
 Title: The Four Seasons'
<<here lies the noble 'The Four Seasons' >>
 (end of list)
```

That's because, at the end of each iteration of the `while` loop, the variable `$cd` goes out of scope, taking with it the only reference to the CD::Music object created earlier in the same loop. That object's reference count immediately becomes zero, and, because it was blessed, the corresponding `DESTROY` method, `CD::Music::DESTROY`, is automatically called on the object, printing out the "here lies…" message.

Of course, in a real program you want your destructor to bury your CDs, not to praise them. Rather than printing a valedictory, we could do some useful work in the destructor and, for example, remedy the bug in the class atrribute that keeps track of the number of CD::Music objects.

The problem is that the CD count keeps merrily incrementing every time `CD::Music::new` is called, but the count is never decremented, even when CD::Music objects cease to exist. Technically, it's a count of how many objects have ever existed, not how many currently exist.

Up to this point we've had no way of ensuring that the shared count is decreased whenever an object vanishes, but now it's easy:

```
package CD::Music;
use strict;

{
   my $_count = 0;

   sub get_count { $_count }
   sub _incr_count { ++$_count }
   sub _decr_count { --$_count }
}

sub new
{
   my $class = ref($_[0]) || $_[0];
   $class->_incr_count();
   # etc. as before
}
```

```
sub DESTROY
{
  my ($self) = @_;
  $self->_decr_count();
}
```

We are now guaranteed that the collective object count is correctly updated each time a CD::Music object is destroyed.

Destructors and circular data structures

Apart from manipulating global attributes like the object count, destructors are rarely needed in object-oriented Perl. By and large, Perl cleans up after itself so effectively that there's usually nothing left for a destructor to do. There is one situation, however, where a destructor *is* required to help clean up after objects: reclaiming circular data structures.

Let's consider a class that represents a network of some kind (i.e., a set of nodes connected by one-way links). Such a class might be needed for email routing, or traffic monitoring software, or in a LAN configuration program, or the finite state machine of a parser, or to implement a neural net. The Network class would probably look something like this:

```
package Network;
use strict;

sub new
{
  my ($class) = @_;
  bless { _nodes => [] }, $class;
}

sub node
{
  my ($self, $index) = @_;
  return $self->{_nodes}->[$index];
}

sub add_node
{
  my ($self) = @_;
  push @{$self->{_nodes}}, Node->new();
}
```

Notice that it makes use of another class, Node. The Node class looks like this:

```
package Node;
use strict;

{
  my $_nodecount=0;
  sub _nextid { return ++$_nodecount }
}
```

```
sub new
{
  my ($class) = @_;
  bless { _id => _nextid(), _outlinks => [] }, $class;
}
sub add_link_to
{
  my ($self, $target) = @_;
  push @{$self->{_outlinks}}, Link->new($target)
}
```

The Node class in turn relies on another class, Link, whose objects represent a single unidirectional link to another Node. The Link class looks like this:

```
package Link;
use strict;

{
  my $_linkcount=0;
  sub _nextid { return ++$_linkcount }
}

sub new
{
  my ($class) = @_;
  bless {  _id  => _nextid(),
           _to  => $_[1],
        }, $class;
}
```

Therefore, a Network consists of zero or more Node objects, each of which has an ID number and a list of references to zero or more outward-going Link objects. Each Link in turn has an ID number and a reference to the Node at which it terminates. Links act as connectors between Nodes, Nodes act as end-points of Links, and the Network object acts as a container for the lot. Figure 3.4(a) illustrates these relationships for a simple three-node network implemented by the following code:

```
use Network;

my $network = Network->new();

foreach (0..2) { $network->add_node(); }

$network->node(0)->add_link_to($network->node(1));
$network->node(0)->add_link_to($network->node(2));
$network->node(1)->add_link_to($network->node(2));
$network->node(2)->add_link_to($network->node(1));
```

This kind of interaction among several classes illustrates one of the most important features of object-oriented programming: the construction of classes out of the interactions of other simpler classes. Unfortunately, in this case, it's in those interactions that the problem arises.

Suppose that three-node network were created in some nested lexical scope, say, in a subroutine:

```
use Network;

sub analyse_network
{
   my $network = Network->new();

   foreach (0..2) { $network->add_node(); }

   $network->node(0)->add_link_to($network->node(1));
   $network->node(0)->add_link_to($network->node(2));
   $network->node(1)->add_link_to($network->node(2));
   $network->node(2)->add_link_to($network->node(1));

   # Do analysis here
}
```

Everything works fine, until we reach the end of `analyse_network`, and the lexical variable `$network` goes out of scope. At that point, the Network object being referred to by `$network` has no other references to it, so it also ceases to exist. That, in turn, means that each of the Node objects in the Network object's node list has its reference count decremented. We might assume that those counts go to zero and the Nodes are also removed, but that isn't the case, as figure 3.4 illustrates.

The first Node object in the list, `$network->node(1)`, does indeed end up with a zero reference count and is correctly reclaimed. That decrements the reference counts of the first two Link objects, causing them to disappear as well. Each of those links has a reference to one of the remaining Node objects, so their reference counts also decrement. Those counts only reduce to 1, because each of the two remaining Link objects still contains a reference to a Node object. At that point, the cascade of destructor calls ceases, since all the remaining objects have nonzero reference counts.

There's the problem. From the point-of-view of the rest of the program, the two remaining Nodes and their interconnecting Links are inaccessible, since the rest of the program has no reference to them. And the memory they occupy will never be reclaimed because their reference counts are nonzero.

The stubborn Nodes and Links form a chain, in which each object stores a reference to some other, which stores a reference to some other, which stores a reference to some other, which stores a reference back to the first. Such chains of references are self-sustaining, because even if no other reference exists to any of the objects, every one of them is referred to at least once. Consequently, their reference counts can never be zero and they can never be reclaimed.[13]

To avoid this leakage of memory, we need to be able to break the sequence of mutual references before losing access to the offending Nodes and Links; in other words, before `$network` ceases to exist. We can ensure such a break occurs by providing the Network class with a destructor that explicitly removes the Links between Nodes at the appropriate time:

```
package Network;

# as before
```

[13] At least, not until the end of the current interpreter thread.

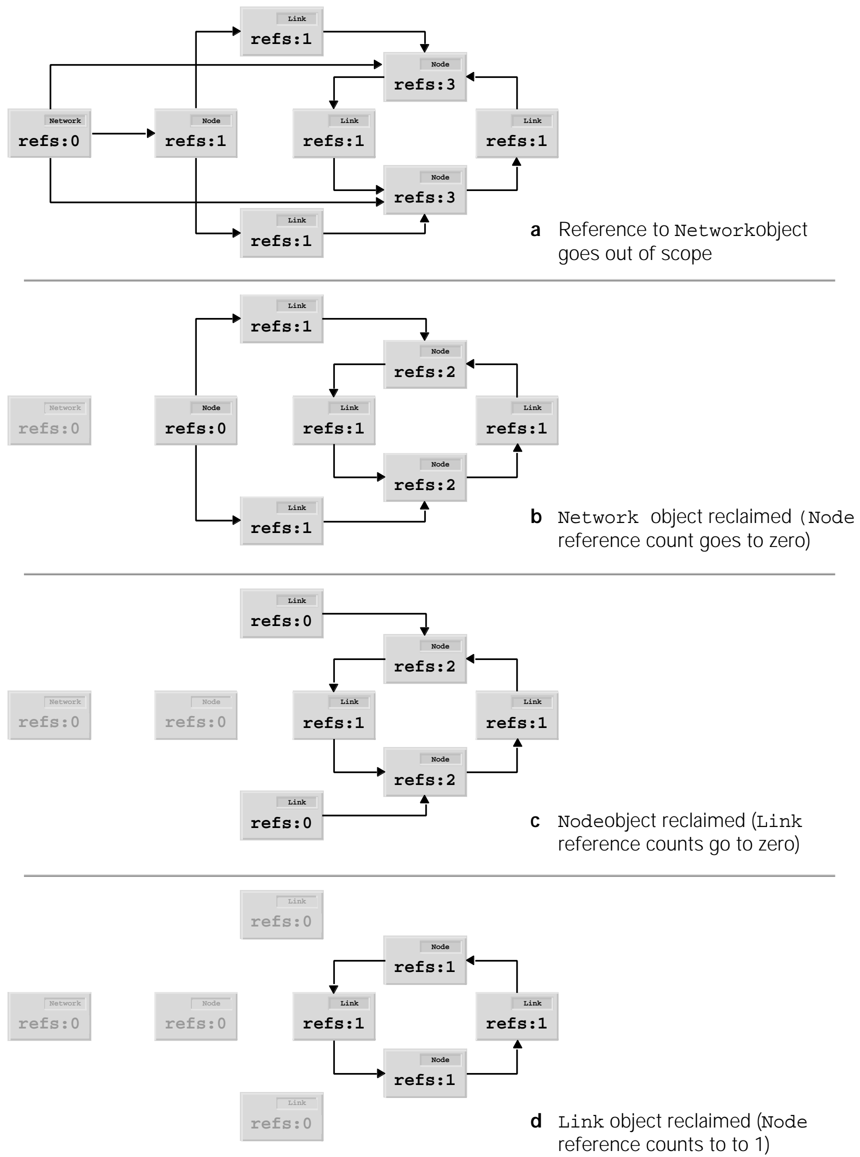

Figure 3.4 Leaking network caused by circular references

```
sub DESTROY
{
   my ($self) = @_;
   foreach my $node ( @{$self->{_nodes}} )
   {
      $node->delete_links();
   }
}
package Node;

# as before

sub delete_links
{
   my ($self) = @_;
   delete $self->{_outlinks};
}
```

The presence of this destructor solves the problem of reference chains, because, whenever a Network object is about to cease to exist, its `DESTROY` method is called. In turn, that `DESTROY` method calls the `delete_links` method for each Node object in its list. `Node::delete_links` eliminates all references to any Link object, which sends those objects' reference counts to zero and causes them to be collected.

After a thorough delinking, each node in the original Network object is referred to only by the Network object itself. When that object finally ceases to exist, the reference counts of the individual Nodes go to zero, and they are cleaned up as well. Figure 3.5 illustrates the steps in the new clean-up sequence initiated by the Network destructor.

Destructors and autoloading

There's one gotcha with destructors—or, more accurately, *without* them—when we're using an `AUTOLOAD` method. `AUTOLOAD` is invoked whenever an undefined method is called for the invoking object. The problem is that whenever includes whenever an object's destructor is called.

Normally, when an object goes out of scope, Perl looks for a suitable destructor and, if it doesn't find one, simply continues with the rest of the program. However, if the object's class has an `AUTOLOAD` method and the search for a `DESTROY` method fails, `AUTOLOAD` will be called instead.

So, at very least, if you intend to provide a class with an `AUTOLOAD` but not a `DESTROY`, you need to make sure that the `AUTOLOAD` can handle a destructor call—as well as anything else it's supposed to cope with. For example, if we weren't intending to provide `CD::Music::DESTROY`, we'd need to modify the `CD::Music::AUTOLOAD` method in Figure 3.5 like this:

```
sub CD::Music::AUTOLOAD
{
   my ($self, $newval) = @_;

   # Was it a destructor call?
```

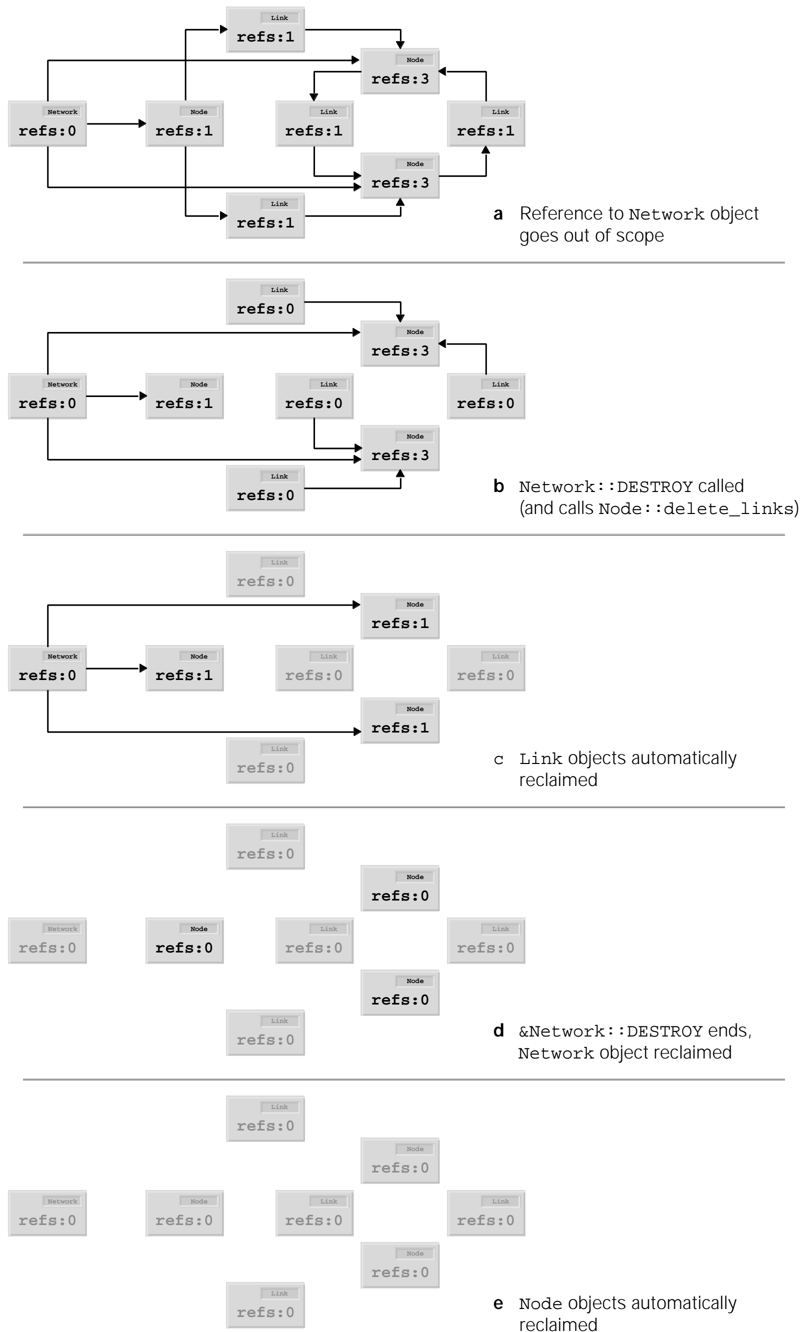

Figure 3.5 Network leakage overcome by implementing destructor

```
    return if $AUTOLOAD =~ /::DESTROY$/;

    # Was it a get_… method?
    $AUTOLOAD =~ /.*::get(_\w+)/
        and $self->_accessible($1,'read')
        and return $self->{$1};

    # Was it a set_… method?
    $AUTOLOAD =~ /.*::set(_\w+)/
        and $self->_accessible($1,'write')
        and do { $self->{$1} = $newval; return };

    # Must have been a mistake then…
    croak "No such method: $AUTOLOAD";
}
```

Even though `CD::Music::AUTOLOAD` can now detect and ignore destructor calls, it is still better to provide a trivial destructor instead:

```
sub CD::Music::DESTROY
{
    # THIS SPACE DELIBERATELY LEFT BLANK
}
```

Calls to `AUTOLOAD` are relatively expensive, and we'd prefer not to incur that extra cost every single time a CD::Music object ceases to exist.

3.5 THE CD::MUSIC CLASS, COMPLEAT

To round out this first taste to object-oriented Perl, Listing 3.4 shows an updated version of the CD::Music class, making use of the techniques and automations described in previous sections of this chapter. In this final version, the default attribute values (from *Constructor default values*) and the access specifiers (from *Catching attempts to change read-only attributes*) are consolidated into a single hash (`%_attr_data`).

Listing 3.4 The completed CD::Music class

```
package CD::Music;
$VERSION = 1.00;
use strict;
use vars qw( $AUTOLOAD );    # Keep 'use strict' happy
use Carp;

{
# Encapsulated class data

    my %_attr_data = #    DEFAULT          ACCESSIBILITY
        (  _name        => [ '???',          'read'],
           _artist      => [ '???',          'read'],
           _publisher   => [ '???',          'read'],
           _ISBN        => [ '???',          'read'],
           _tracks      => [ '???',          'read'],
```

```
        _rating     => [ -1,             'read/write'],
        _room       => [ 'uncataloged', 'read/write'],
        _shelf      => [ "",             'read/write'],
    );

  my $_count = 0;

# Class methods, to operate on encapsulated class data

  # Is a specified object attribute accessible in a given mode
  sub _accessible
  {
    my ($self, $attr, $mode) = @_;
    $_attr_data{$attr}[1] =~ /$mode/
  }

  # Classwide default value for a specified object attribute
  sub _default_for
  {
    my ($self, $attr) = @_;
    $_attr_data{$attr}[2];
  }

  # List of names of all specified object attributes
  sub _standard_keys
  {
    keys %_attr_data;
  }

  # Retrieve object count
  sub get_count
  {
    $_count;
  }

  # Private count increment/decrement methods
  sub _incr_count { ++$_count }
  sub _decr_count { --$_count }

}

# Constructor may be called as a class method
# (in which case it uses the class's default values),
# or an object method
# (in which case it gets defaults from the existing object)

sub new
{
  my ($caller, %arg) = @_;
  my $caller_is_obj = ref($caller);
  my $class = $caller_is_obj || $caller;
  my $self = bless {}, $class;
  foreach my $attrname ( $self->_standard_keys() )
  {
```

```
        my ($argname) = ($attrname =~ /^_(.*)/);
        if (exists $arg{$argname})
           { $self->{$attrname} = $arg{$argname} }
        elsif ($caller_is_obj)
           { $self->{$attrname} = $caller->{$attrname} }
        else
           { $self->{$attrname} = $self->_default_for($attrname) }
    }
    $self->_incr_count();
    return $self;
}

# Destructor adjusts class count
sub DESTROY
{
    $_[0]->_decr_count();
}

# get or set room&shelf together

sub get_location    { ($_[0]->get_room(), $_[0]->get_shelf()) }

sub set_location
{
      my ($self, $room, $shelf) = @_;
      $self->set_room($room) if $room;
      $self->set_shelf($shelf) if $shelf;
      return;
}

# Implement other get_… and set_… methods (create as necessary)

sub AUTOLOAD
{
      no strict "refs";
      my ($self, $newval) = @_;

      # Was it a get_… method?
      if ($AUTOLOAD =~ /.*::get(_\w+)/ && $self->_accessible($1,'read'))
      {
         my $attr_name = $1;
         *{$AUTOLOAD} = sub { return $_[0]->{$attr_name} };
         return $self->{$attr_name}
      }

      # Was it a set_… method?
      if ($AUTOLOAD =~ /.*::set(_\w+)/ && $self->_accessible($1,'write'))
      {
         my $attr_name = $1;
         *{$AUTOLOAD} = sub { $_[0]->{$attr_name} = $_[1]; return };
         $self->{$1} = $newval;
         return
      }
```

```
        # Must have been a mistake then…
        croak "No such method: $AUTOLOAD";
}

1;  # Ensure that the module can be successfully use'd
```

3.6 SUMMARY

- A Perl class is just a package containing subroutines that implement methods.
- Any of Perl's standard data types—hash, array, scalar, typeglob, subroutine, regex—may be used as an object.
- To convert something to an object, we use the `bless` function to associate the thing with the appropriate package. Always use the two-argument form of `bless`.
- A constructor is just a method that blesses an object, initializes it, and returns a reference to it.
- A destructor is a special method called `DESTROY` that is automatically called when an object is about to be garbage-collected. Destructors are rarely needed in Perl, except when circular data structures are involved.
- Accessors are methods that provide read or write access to an attribute. It is much safer to encapsulate attributes in accessors than to access them directly.
- To create class attributes, use a lexical variable declared in some nested block within the package. Define accessors for the variable within the same nested block.
- An `AUTOLOAD` method can provide a "catchall" or generic method for a class. Alternatively, `AUTOLOAD` may create and install a suitable method at run time. When using autoloaded methods, it's a good idea to provide a destructor, even if it doesn't do anything.