CHAPTER 11

Encapsulation

Encapsulation is one of the cornerstones of object orientation, but it's the area in which Perl's support for object-oriented programming is weakest. Many would argue that enforced encapsulation is against Perl's philosophy of freedom and flexibility in programming. However, there are situations when too much freedom becomes a trap, and too much flexibility makes it hard to build solid code. Fortunately, Perl's flexibility can be turned against itself to provide a means of building objects that respect the encapsulation imposed by their classes.

11.1 THE PERILS OF TRUST

In practice, the lack of an built-in encapsulation mechanism rarely seems to be a problem in Perl. Most Perl programmers build classes out of standard hashes, and both they and the users of their classes get by happily with the principle of encapsulation by good manners. The lack of formal encapsulation doesn't matter because everybody plays nicely, keeps off the grass, and respects the official interface of objects. Those who don't play by the rules, who directly access a method or attribute that is supposed to be private, get what they deserve—either better performance or a nasty surprise.

The only problem is that this convivial arrangement doesn't scale very well. Leaving your front door open may be fine in a small town, but it's madness in the big city. Likewise, informal mechanisms suitable for a few hundred lines of code written by a single programmer don't work nearly as well when the code is tens of thousands of lines long and developed by a team.

Even if you could trust the entire team to maintain sufficient programming discipline to respect the notional encapsulation of attributes (a dubious proposition), accidents and mistakes happen, especially in rarely used parts of the system that only get used when demonstrating to important clients.

Moreover, deliberate decisions to circumvent the rules (usually taken in the heat of hacking, out of laziness, or for efficiency) are often inadequately documented, leading to problems much later in the development cycle. For example, consider a (notionally) private attribute of an object, which for efficiency reasons is accessed directly in an obscure part of a large system. If the implementation of the object's class changes, that attribute may cease to exist.

In a more static language, this would generate an error message when the external code next attempts to access the now nonexistent attribute. However, Perl's autovivification of hash entries may well resurrect the former attribute when it is next modified, so the now-incorrect access proceeds silently. Bugs such as this can be painfully difficult to diagnose and track down, especially if the original programmer has moved on by the time the problem is discovered.

11.2 ENCAPSULATION VIA CLOSURES

The standard approach to enforcing encapsulation of an object's attributes is to avoid giving the user direct access to the object. At first glance, that may seem impossible; after all, you must have a reference to the object, or you can't call its methods.

The key here is the word *direct*. So long as the reference provided to the user refers to something that has been blessed into the required class, it is possible to call methods through that reference. If we somehow arrange that the blessed something cannot be used to directly access object attributes, then those attributes are safely encapsulated. The trick, of course, is to achieve that encapsulation while still allowing methods to access their own attributes directly.

Curiously, to find that kind of access control for objects, we have to travel briefly in the opposite direction and look at subroutines.

Subroutines provide an obvious form of encapsulation. If you have a subroutine—say for example, the pseudo-random number generator function `rand`—then the *only* way you can access the information it provides is to call it and grab the value it returns. As clients of the `rand` function, we have no way of directly accessing its internals. For all most of us know, it might be implemented like this:

```
{
  my @rand_val = (0.012657, 0.453662, 0.718273);
  sub rand
  {
     push @rand_val, shift @rand_val;  # "rotate" the list
     return ($_[0]||1) * $rand_val[0]; # scale and return first element
  }
}
```

The point is, there's no way to extend `rand`'s pitifully inadequate look-up table. Outside the block in which it's defined, `@rand_val` is out-of-scope, and the only remaining means of accessing it is hidden inside the `rand` function itself.

We have seen the same technique used throughout this book, to encapsulate class attributes—for example, the `$_count` attribute in the CD::Music class. Outside the block in which that lexical variable is defined, the only access to it is via the closures `get_count`, `_incr_count`, etc., that were defined in the same block.

The use of a closure to provide controlled access to an otherwise inaccessible lexical works equally well when applied to the attributes of individual objects. In fact, we could actually *create* the object by blessing the access control subroutine itself.

Listing 11.1 shows another version of the simple Soldier class. Unlike the automatically generated hash-based version in chapter 8, this version of the class is implemented using closures to enforce encapsulation of its attributes.

As usual, the constructor is the most complex part of the class. It creates a lexical hash (`%data`) and initializes it with the appropriate entries from the argument list by assigning one hash slice (`@args{@attrs}`) to another (`@data{@attrs}`).[1]

The constructor then creates a new anonymous subroutine and stores a reference to it in `$accessor`. As the name implies, that subroutine will be used to access the `%data` hash, once the new Soldier object is fully constructed.

The anonymous accessor subroutine takes three arguments: a string indicating what kind of access is required; another string indicating which attribute is to be accessed; and the new attribute value for "set" operations. The accessor subroutine has three courses of action, depending on the arguments it is given.

If the first argument indicates a "get" request, the subroutine simply returns a copy of the requested attribute in the `%data` hash. The subroutine has access to `%data` because that hash was declared in the same lexical scope as the subroutine itself—that is, within the body of the constructor. If the first argument indicates a "set" request, the subroutine checks whether it is the `"rank"` attribute that is being set and, if so, assigns the new value to `$data{"rank"}`. Any other access request—for example, to set the `"name"` attribute—is impolitely rejected.

Finally, once the accessor subroutine is created, it is blessed as the new object, and a reference to it is returned from `Soldier::new`. At that point, the constructor ends, and the lexical variables it created would normally be destroyed. However, the `%data` hash escapes this fate because the anonymous subroutine still refers to it, and so Perl arranges for it to live on, incognito, until the anonymous subroutine itself is no longer accessible.

The result, illustrated in figure 11.1, is that each newly created Soldier object is a blessed subroutine, one which has the only remaining access to the lexical `%data`. It uses that hash as its own private storage area, getting or setting entries in `%data` whenever it is invoked. It's important to realize that, next time `Soldier::new` is invoked, a new—and entirely distinct—lexical hash, also called `%data`, will be created within the constructor. Then a new—and entirely distinct—anonymous subroutine will be created, blessed, and returned. That new and entirely distinct subroutine will subsequently have access to the new `%data` hash.

In this way, repeated calls to `Soldier::new` create a series of distinct hashes, each wrapped up in a personalized, anonymous, encapsulating subroutine. The subroutines are

[1] This method of initialization has much to recommend it: it's concise (just one assignment), declarative (valid attributes are declared in the `@attrs` hash), robust (only attributes specified in `@attrs` can ever be initialized), and easy to maintain (just add the name of any new attribute to `@attrs`).

Listing 11.1 The Soldier class implemented via closures

```
package Soldier;
$VERSION = 1.00;
use strict;

use Carp;

my @attrs = qw(name rank serial_num);

sub new
{
  my ($class, %args) = @_;
  my %data;
  @data{@attrs} = @args{@attrs};
  my $accessor =
    sub
    {
        my ($cmd, $attr, $newval) = @_;
        return $data{$attr}
          if $cmd eq "get";
        return $data{"rank"} = $newval
          if ($cmd eq "set" && $attr eq "rank");
        croak "Cannot $cmd attribute $attr";
    };
  bless $accessor, ref($class)||$class;
}

# These methods provide the only means of accessing object attributes
# (note that only rank can be changed)

sub get_name        { $_[0]->('get','name') }
sub get_rank        { $_[0]->('get','rank') }
sub get_serial_num  { $_[0]->('get','serial_num') }

sub set_rank
{
  my ($self, $newrank) = @_;
  $self->('set','rank',$newrank);
}

1;
```

returned as objects and used to access the corresponding hash in a controlled manner. Instant encapsulation!

Oddly enough, that encapsulation is actually far stronger than is provided by most other object-oriented languages. Not even the members of its own class have direct access to a Soldier object's data. Instead, they too must request access via the encapsulating subroutine.

Hence the `get_name`, `get_rank`, and `get_serial_num` accessors each take the reference to a blessed subroutine through which they are invoked (i.e., `$_[0]`) and call that

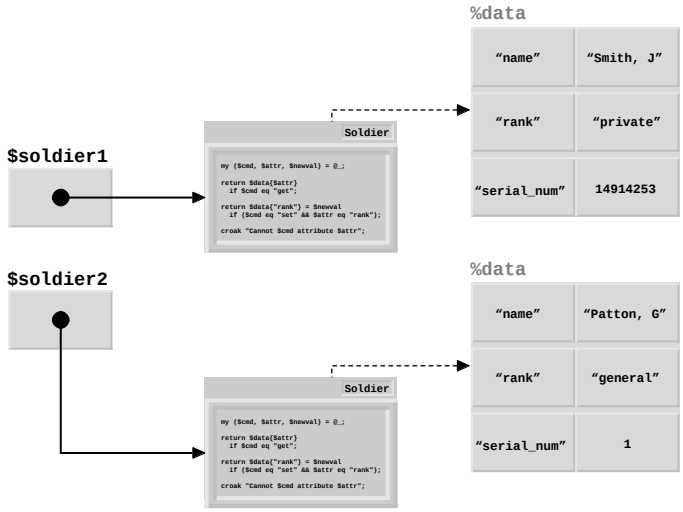


Figure 11.1 Structure of closure-based Soldier objects

subroutine, passing it an argument list requesting retrieval of the appropriate attribute value. Likewise, the `set_rank` method invokes the subroutine, asking it to update its encapsulated `$data{"rank"}` attribute with the specified new value. There is no point in providing a `set_name` or a `set_serial_num` method, since the definition of the encapsulating subroutine makes it impossible to set these attributes.

11.2.1 A variation for the paranoid

In a sense, the accessor methods of class Soldier exist only as conveniences. Instead of writing:

```
$soldier->set_rank("Colonel");
print $soldier->get_serial_num();
```

we could take advantage of the fact that we have a reference to the accessor subroutine (i.e., in `$soldier`), and call that subroutine directly:

```
$soldier->("set","rank","Colonel");
print $soldier->("get","serial_num");
```

Well, we *could* do that, but it's probably not a good idea. In fact, it would probably be better if we couldn't do it at all.

This level of paranoia may appear to have no purpose, except to "satisfy certain fastidious concerns of programming police and related puritans."[2] However, there are good reasons for

[2] ..as suggested by Tom Christiansen in the perltoot man page. Of course, since the Puritans helped to found the most powerful nation in history, and the police exist to protect the personal rights and liberties of a free citizenry, it may be that Tom is actually in favor of absolute encapsulation and is praising it with faint damns.

preventing client code from accessing an object in any way except through their defined methods.

The most obvious reason is that, as maintainers of the Soldier class's code, we may later need to change the interface to the anonymous accessor subroutine or even dispense with subroutine-based objects entirely. Either of these changes will invalidate any client code that calls the accessor subroutine directly, which will result in hundreds of irate users contacting us to ask why their client code no longer works.

Furthermore, as we have already seen in chapter 5, if a class like Soldier is ever to be inherited (Soldier::Foot, Soldier::Paratrooper, Soldier::Is::A::Marine::Sir::HOO::AH, etc.), it may be vital that client code accesses Soldier attributes only via defined methods. If attributes are accessed in any other way—either directly (for example, `$soldier->{"name"}`), or indirectly (for example, `$soldier->("get","name")`)—then that client code is no longer treating the object polymorphically, and it may not work correctly when given an object of a derived class.

Fortunately, it's easy to ensure that the accessor subroutine that implements a Soldier object can only be accessed from the class's defined methods. We simply modify the Soldier constructor as follows:

```
sub new
{
  my ($class, %args) = @_;
  my %data;
  @data{@attrs} = @args{@attrs};
  my $accessor =
    sub
    {
      my ($cmd, $attr, $newval) = @_;
      croak "Invalid direct access. Use the ${cmd}_$attr method instead"
        unless caller()->isa("Soldier");
      return $data{$attr}
        if $cmd eq "get";
      return $data{"rank"} = $newval
        if ($cmd eq "set" && $attr eq "rank");
      croak "Cannot $cmd attribute $attr";
    };
  bless $accessor, ref($class)||$class;
}
```

In this version, when the anonymous accessor subroutine is called, it checks to see that it was called by a method belonging to a class that *is-a* Soldier. If not, it immediately throws an exception. This means that methods like `Soldier::get_name` can invoke the accessor method directly, but subroutines outside the Soldier class hierarchy cannot. We could make the access rules even stricter:

```
croak "Invalid direct access. Use the ${cmd}_$attr method instead"
  unless caller() eq "Soldier";
```

and limit access to methods of the Soldier class itself, in which case derived classes would also have to use those methods to access their own (inherited) attributes.

Ultimately, of course, nothing will stop the determined programmer from circumventing the proper interface of the Soldier class:

```
my $soldier = Soldier->new(name=>"Alexander", rank=>"General");

package Soldier;                # Step back into the Soldier package and…
print $soldier->("get","name"); # …call the accessor directly!
```

But this technique does serve to effectively catch accidental breaches of the interface, and thereby minimize nasty surprises.

11.3 ENCAPSULATION VIA SCALARS

A less well-known approach to encapsulation uses scalar-based objects to implement a technique known as the *flyweight pattern*. In the flyweight pattern, objects don't carry around their own information, so that information can't be accessed directly via the object. Instead, flyweight objects merely serve as an index into a shared table of values, stored within the class itself. For example, an object may be an integer that indexes into a table of values stored as a class attribute.

Flyweight objects are most frequently used in object-oriented languages that pass objects around by value because flyweight objects remain extremely small (no matter how much data they contain). Hence, they are cheap to pass around. Because Perl objects are invariably accessed via references, this advantage is not significant.

However, the flyweight pattern still has something to offer in Perl, because it provides a simple mechanism for preventing direct access to object attributes, thereby enforcing encapsulation. As a bonus, it also provides a means of easily keeping track of every object in a class, something closure-based encapsulation doesn't provide.

11.3.1 Name, rank, and serial number

Listing 11.2 shows a flyweight implementation of the Soldier class. The entire class is contained in a pair of curly braces to ensure that any lexical variable declared within their scope is not directly accessible outside that scope. Not surprisingly, the first thing the class does is declare some lexical variables.

Listing 11.2 The Soldier class implemented via scalars

```
package Soldier;
$VERSION = 2.00;
use strict;

{
  # Table storing references to hashes containing object data
  my @_soldiers;

  # Allowable attributes and their default values
  my %_fields = (name=>'???', rank=>'???', serial_num=>-1);

  # Constructor adds object data to table and blesses a scalar
```

```
  # storing the index of that data

  sub new
  {
    my ($class, %args) = @_;
    my $dataref = {%_fields};
    foreach my $field ( keys %_fields )
    {
        $dataref->{$field} = $args{$field}
          if defined $args{$field};
    }
    push @_soldiers, $dataref;
    my $object = $#_soldiers;
    bless \$object, $class;
  }
  # These methods provide the only means of accessing object attributes
  # (note that only rank can be changed)

  sub get_name        { return $_soldiers[${$_[0]}]->{name} }
  sub get_rank        { return $_soldiers[${$_[0]}]->{rank} }
  sub get_serial_num  { return $_soldiers[${$_[0]}]->{serial_num} }

  sub set_rank
  {
   my ($indexref, $newrank) = @_;
   $_soldiers[$$indexref]->{rank} = $newrank
  }
  # This class method provides an iterator over every object

  my $_cursor = -1;
  sub each
  {
   my $nextindex = ++$_cursor;
   if ($nextindex < @_soldiers)
   {
     return bless \$nextindex, ref($_[0])||$_[0];
   }
   else
   {
     $_cursor = -1;
     return undef;
   }
  }
}
```

The lexical array `@_soldiers` is used to store the data for each object. That data is directly accessible to the methods declared within the surrounding curly braces, but nowhere else. It is this restriction that eventually provides the desired encapsulation of object data.

The lexical hash `%_fields` performs the dual function of recording (in its keys) the names of valid attributes of a Soldier object and storing (in its values) the default values for those attributes.

The constructor begins like most others we've seen so far, by creating an anonymous hash and initializing it with the default attribute values for the class. It loops over the valid fields of the class, overwriting those default values with any corresponding argument that was passed to the constructor.

At this point, a typical constructor blesses and returns the reference in `$objref`, making the anonymous hash into the new object. Instead, `Soldier::new` pushes the hash onto the end of the encapsulated `@_soldiers` array and blesses a scalar storing the index of that newly added array element.

Thus a constructor call such as

```
my $grunt = Soldier->new(name       => "Smith, J.",
                         rank       => "private",
                         serial_num => 149162536);
```

leaves `$grunt` with a reference to a scalar—that is, to the index of the data—rather than a reference to a hash—that is, to the data itself). Figure 11.2 illustrates the process.

Theoretically, the effect is the same. Since we have the index and know which array it refers to, we can still find the actual data. In practice, however, there's an important difference. Outside the curly braces surrounding the class, the `@_soldiers` array is inaccessible so, even though we have the index for the object's data, we can't access that data directly.

11.3.2 Controlled access

Instead, it's up to the accessor methods of the class to provide the required access. Since they are all defined within the encapsulating curly braces, they *do* have access to `@_soldiers`. So, the accessor methods can dereference the blessed index (**`${$_[0]}`**), index into the array to get a reference to the appropriate hash data (**`$_soldiers[`**`${$_[0]}`**`]`**), and then access the correct field of that data using the arrow notation (`$_soldiers[${$_[0]}]`**`->{name}`**).

The implementation shown in listing 11.2 doesn't provide write accessors for a Soldier's name or serial number. The lack of write access provides real data security since, without the accessors, there is no way of modifying these attributes once they are set.

Even imposing a new method on the class

```
package main;
use Soldiers;

my $general = Soldier->new( name       => "Caesar, G.J.",
                            rank       => "Prodictator",
                            serial_num => "MMXLVIII");

# Oops, that serial number was out by one.
# Strange, there's no method to change it.
# Oh well, let's just add one ourselves…
sub Soldiers::set_serial_num { $_soldiers[${$_[0]}]->{serial_num} = $_[1] }
```

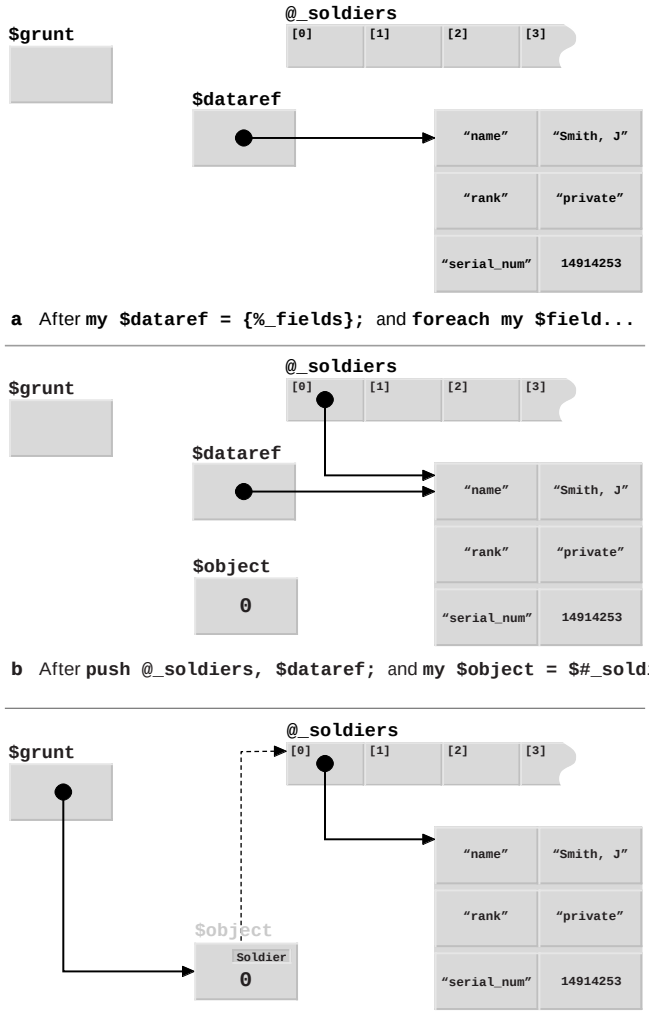


Figure 11.2 Construction of a Soldier object

```
# …and use it…

$general->set_serial_num("MMXLIX");
```

will not circumvent encapsulation. Although the new method *is* in the class's namespace (and hence, callable through its objects), it *isn't* in the lexical scope of the original encapsulating curly braces, so it doesn't have access to the lexical `@_soldiers` array.

It's worth noting that Perl visits a satisfying form of Instant Justice on the author of this code. Since the code doesn't `use strict`, Perl concludes that the `@_soldiers` array being modified in `Soldier::set_rank` is the package variable `@main::_soldiers`. Thus, the code executes without complaint, yet mysteriously fails to update any soldier's serial number, leading to happy hours of fruitless debugging.

11.3.3 Roll call

The other advantage of a scalar-based object representation like this is that the class itself has direct and continuing access to the data of every object blessed into it. That makes it easy to provide class methods to iterate that data.

The Soldier class demonstrates this by providing an iterator method (`Soldier::each`), which steps through the indices of the `@_soldiers` array, returning a blessed version of each index (i.e., a Soldier object). The method can be used like this:

```
while (my $soldier = Soldier->each)
{
  printf "name: %s\nrank: %s\n s/n: %d\n\n",
    $soldier->get_name(),
    $soldier->get_rank(),
    $soldier->get_serial_num();
}
```

By the way, as elegant as it might look, don't be tempted to write

```
while (my $soldier = each Soldier) {…}
```

hoping that this is one of the few places where the indirect object syntax will work. It isn't. Instead, Perl will assume you wanted to use the built-in `each` function to iterate the package hash `%Soldier`, and just forgot the `"%"` prefix. Once again, `use strict` will prevent Perl from helping you cut your own throat.

11.3.4 A question of identity

It's instructive to contemplate what the `Soldier::each` method is actually doing, every time it returns an object. Consider the following code:

```
use Soldier;

my $soldier_ref1 = Soldier->new(name=>"Temuchin", rank=>"Khan");
my $soldier_ref2 = Soldier->each;
```

Assuming that this is the entire program, then the objects referred to by `$soldier_ref1` and `$soldier_ref2` should be the same. And, in almost every important sense, they *are*. They have the same name, rank, and serial number, and any changes made via `$soldier_ref1->set_rank()` will be reflected in subsequent calls to `$soldier_ref2->get_rank()`. However, the two references themselves do not compare equal, because the two blessed scalar objects to which they refer are distinct (though they have the same value).

Each time it needs to return a given index of `@_soldiers`, `Soldier::each` creates an entirely new blessed scalar object containing that index. Because that object has the same value as the original object for the given data, it is logically equivalent to the original, and

only distinguishable by its distinct address. Such objects are sometimes called *proxies*, since they act in place of the original object.

11.3.5 A variation for the truly paranoid

Although it's useful that two physically distinct objects can be logically identical, there's a down-side: that duality also means that any Soldier object can be converted to any other Soldier object, *even if there is no preexisting reference to that other object in the current scope.*

For example, consider the following code

```
use Soldier;
OUTER:
{
  INNER:
  {
     my $commander = Soldier->new(name=>"Smythe, Sir X.A.StJ.",
                                  rank=>"Field Marshall");
  }
  my $private1 = Soldier->new( name=>"Smith, J.",
                               rank=>"Private");

  $$private = 0;                 # Guess the right index
  bless $private, Soldier;       # and become…
  print $private->get_rank();    # …Field Marshall!
}
```

Even though the outer scope has lost access to the `$commander` object before the `$private` object is even created, `$private` can still steal `$commander`'s identity by changing (and reblessing) the index stored in its object.

If this kind of referential "bed-swapping" is unacceptable, or if it is important that all references to the same Soldier object always compare equal, then a slightly more sophisticated approach, such as that shown in listing 11.3, is required.

Listing 11.3 A more secure version of Soldier class, implemented via scalars

```
package Soldier;
$VERSION = 3.00;
use strict;

{
  # Hash table storing references to hashes containing object data
  my %_soldiers;

  # Allowable attributes and their default values
  my %_fields = (name=>'???', rank=>'???', serial_num=>-1);

  # Constructor adds object data to hash table and blesses a scalar
  # storing the key of that data

  sub new
  {
```

```
    my ($class, %args) = @_;

    # Build the data for the object…
    my $dataref = {%_fields};
    foreach my $field ( keys %_fields )
    {
       $dataref->{$field} = $args{$field}
          if defined $args{$field};
    }

    # Build a unique unguessable key…
    $dataref->{_key} = rand
       until $dataref->{_key} && !exists $_soldiers{$dataref->{_key}};

    # Insert the data into the table and return the key…
    $_soldiers{$dataref->{_key}} = $dataref;
    bless \$dataref->{_key}, $class;
  }

  # These methods provide the only means of accessing object attributes
  # (note that only rank can be changed)

  sub get_name{ return $_soldiers{${$_[0]}}->{name} }
  sub get_rank{ return $_soldiers{${$_[0]}}->{rank} }
  sub get_serial_num{ return $_soldiers{${$_[0]}}->{serial_num} }

  sub set_rank
  {
    my ($keyref, $newrank) = @_;
    $_soldiers{$$keyref}->{rank} = $newrank
  }

  # This class method provides an iterator over every object

  sub each
  {
    my $nextkey = each %_soldiers;
    return \$_soldiers{$nextkey}->{_key} if defined $nextkey;
    return undef;
  }
}
```

Version 2.00 of the Soldier class makes it much harder to locate the data for a particular object by guessing its location in the internal `@_soldiers` array. Instead of the array, with its orderly and predictable sequence of indices, this version uses a hash table (`%_soldiers`), and chooses hash keys that are much harder to guess.

The keys are generated by a call to the built-in `rand` function, which produces floating point numbers in the range zero to one. When these numbers are stringified to produce hash keys, they are typically rendered to 15 decimal digits, all of which are independently random (assuming double precision on a 32-bit architecture). Hence, the odds of guessing a particular key are one in a quadrillion.

The code itself does not even trust these odds and uses a `while` loop to guarantee that a given key is never reused for separately constructed objects.

Each key is stored as an entry in the hash of data belonging to its object. The tricky bit is that this scalar entry is then blessed to *become* the object itself. Trying to visualize and understand the relationships between keys, data, and objects in this version will almost certainly give you a headache, mainly because—in a complete reversal of normal object-oriented physics—objects are now stored *inside* their own data! Figure 11.3 illustrates how the constructor call:

```
my $grunt = Soldier->new(name       => "Smith, J.",
                         rank       => "private",
                         serial_num => 149162536);
```

would be handled. You may find it helpful to compare the sequence illustrated, with the corresponding sequence in figure 11.2.

The benefit of this fascinating arrangement is that it's now *very* unlikely that any piece of code will be able to guess the key of an otherwise inaccessible Soldier object (and thereby assume its identity).

Moreover, since the original objects are actually stored as one of their own atrributes, it's possible for `Soldiers::each` to return a reference to the original objects, rather than having to manufacturing a proxy. This guarantees that the object references returned by `Soldier::new` and `Soldier::each` always compare equal for a given object.

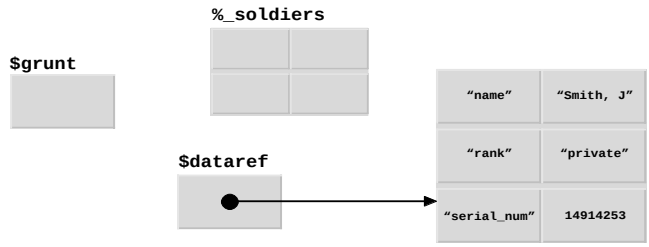
In fact, `Soldiers::each` is considerably simplified in this version, since all it needs to do is to use the built-in `each` function to iterate through the entries of `%_soldiers` hash and extract a reference to the original object from each entry.

11.4 ENCAPSULATION VIA TIES

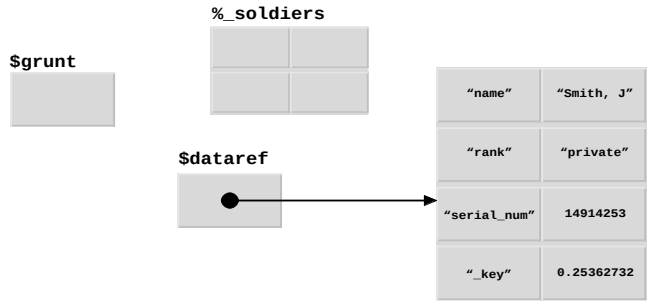
Other object-oriented languages support varying degrees of encapsulation. For example, C++ and Java programmers can declare object and class data members as public, protected, or private, to restrict access to them to certain well-defined scopes. Likewise, attributes of Eiffel classes can be declared with an export list that specifies the classes that can access them.

The closest Perl comes to an explicit encapsulation feature is the behavior of the `fields.pm` and `base.pm` modules (as described in chapters 4 and 6). Pseudo-hash fields whose name starts with an underscore are not imported by a call to `use base`, and thus, to some extent, they mimic private attributes. Unfortunately, all this really means is that derived class objects don't have a `%FIELDS` entry for underscored fields inherited from a base class. Such fields can still be accessed anywhere.

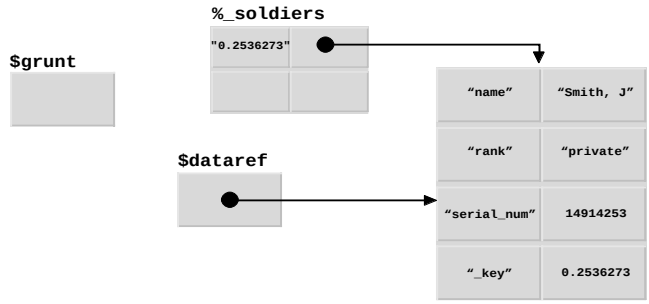
So far we have seen two clever techniques for encapsulating the attributes of a class: within a closure and via a scalar implementing the flyweight pattern. Both techniques effectively provided a bottleneck that controls access to object attributes. These techniques work well, but they are all-or-nothing propositions. Every attribute is encapsulated, even from methods of the same class. Moreover, both techniques are moderately complicated to understand and code, particularly by beginners—who are most likely to need the safety net of explicit encapsulation.



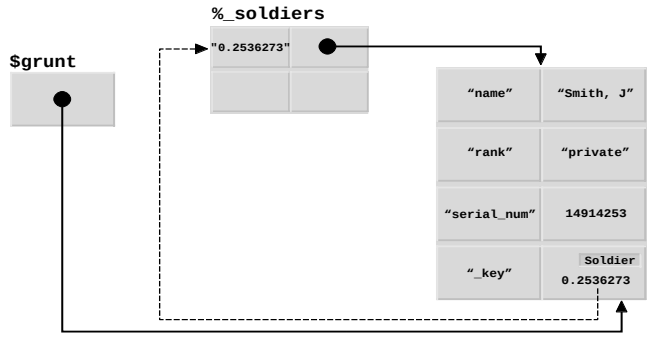
a After `my $dataref = {%_fields};` and `foreach my $field...`



b After `$dataref->{_key} = rand while...`



c After `$_soldiers{$dataref->{_key}} = $dataref;`



d After `bless \$dataref->{_key}, $class;` and constructor returns

Figure 11.3 Construction of a paranoid Soldier object

What we really need is a mechanism that builds objects using a simple hash-like data type and yet is able to specify different levels of accessibility to individual attributes stored in that hash.

11.4.1 A limited-access hash

The Tie::SecureHash module, which is available from the CPAN, provides a flexible means of restricting access to individual attributes of a hash. The module mimics a normal hash, via the standard `tie` mechanism described in chapter 9, but allows keys to be fully qualified as if they were independent package variables. Using these qualifiers, the module restricts attribute accessibility to specific namespaces.

A Tie::SecureHash object—let's call it a securehash—is created in one of two ways: either by tie-ing an existing hash to the Tie::SecureHash module:

```
my %securehash;
tie %securehash, Tie::SecureHash;
```

or by calling the constructor `Tie::SecureHash::new`:

```
my $securehash_ref = Tie::SecureHash->new();
```

The value returned by `Tie::SecureHash::new` is a reference to an anonymous securehash, which has also been blessed into the Tie::SecureHash class. (See chapter 9 for an explanation of how a single package can be used both to tie and bless an object.)

Generally speaking, the resulting securehash acts like a regular Perl hash. You can:

- Access individual entries using normal hash access syntaxes: `$securehash{$key}` or `$securehash_ref->{$key}`;
- Confirm the existence of specific entries: `exists $securehash_ref->{$key}`,
- Obtain a list of the keys and values that it currently contains: `keys %securehash`, `values %{$securehash_ref}`;
- Iterate through the entire hash: `each %securehash`.

The Tie::SecureHash module also provides object methods corresponding to most of these features—such as `$securehash_ref->values()`, `$securehash_ref->each()`, `$securehash_ref->exists($key)`, and so forth—which may be invoked on securehashes that were created with `Tie::SecureHash::new`.

The following subsections look at each aspect of using a securehash, concentrating on how it differs from a regular hash.

11.4.2 Constructing a securehash

Although you can create a securehash by explicitly tie-ing an existing hash to the Tie::SecureHash package, it's an ungainly way of producing one.[3] Since securehashes are designed to be used as the basis of regular Perl classes, the `Tie::SecureHash::new` method provides a convenient way of obtaining a reference to a "pre-blessed" securehash.

[3] It also makes optimization using the "fast" option difficult (see section 11.4.8).

`Tie::SecureHash::new` takes a single optional argument that specifies the class you want the new securehash blessed into. So, instead of writing a constructor like this:

```
sub new
{
  my ($class, %args) = @_;
  my %hash;
  tie %hash, Tie::SecureHash;
  my $self = bless \%hash, $class;

  # initialization here

  return $self;
}
```

you can just write

```
sub new
{
  my ($class, %args) = @_;
  my $self = Tie::SecureHash->new($class);

  # initialization here

  return $self;
}
```

If `Tie::SecureHash::new` is called without the optional argument, it blesses the securehash into class Tie::SecureHash itself.

11.4.3 Declaring securehash entries

Securehashes differ from regular Perl hashes in several respects. Perhaps the most important difference is that securehash entries are not autovivifying. In fact, specific entries cannot be accessed at all until they have been declared to exist.

A securehash entry is declared by referring to it through a *qualified key*. A qualified key is a string consisting of one or more characters except `':'`, preceded by a standard Perl package qualifier. For example, the following are all qualified keys, suitable for specifying entries in a securehash:

```
'MyClass::key'         # key: 'key',       qualifier: 'MyClass::'
'MyClass::a key'       # key: 'a key',     qualifier: 'MyClass::'
'CD::Music::_tracks'   # key: '_tracks',   qualifier: 'CD::Music::'
'Railroad::_tracks'    # key: '_tracks',   qualifier: 'Railroad::'
'PerlGuru::__password' # key: '__password',qualifier: 'PerlGuru::'
'main::mainkey'        # key: 'mainkey',   qualifier: 'main::'
'::mainkey'            # key: 'mainkey',   qualifier: 'main::' (implicitly)
```

Each qualifier indicates the package that owns the key. Hence, the first two keys above are owned by class MyClass and the last two by the main package. Qualified keys with the same key but different qualifiers—for example, `'Railroad::_tracks'` and `'CD::Music::_tracks'`—are treated as being distinct, even if they label two entries in the same securehash.

Hence, the qualifiers are just like the classname prefixes used in chapter 6[4] to prevent derived class attributes from clobbering those of the same name inherited from the base class. Indeed, as we'll see shortly, qualifiers serve exactly the same purpose in securehashes.

To create an entry in a securehash, it must first be referred to by its fully qualified name. This would typically happen in a class's constructor:

```
package File;

sub new
{
   my ($class) = @_;
   my $self = Tie::SecureHash->new($class);

   $self->{"File::name"}     = $_[1];
   $self->{"File::_type"}    = $_[2];
   $self->{"File::__handle"} = $_[3];

   return $self;
}
```

The class whose name is used as a qualifier to declare an entry is thereafter considered to be the owner of that entry. Owner classes have special access privileges to their attributes, as described in the next section. Because of that special relationship, an entry can only be declared within the namespace of its owner's package. In other words, the qualifier for any entry declaration must be the name of the current package, as in the example above.

Once the entries have been declared, they can subsequently be accessed (subject to the constraints explained in the next section) either by their fully qualified key or their actual key, so long as it's unambiguous. For example:

```
sub File::dump
{
   my ($self) = @_;

   print "Dumping file $self->{name}\n";       # Just use key
   print "(of type $self->{File::_type}):\n";  # Use qualified key

   while (my $nextline = readline(*{$self->{__handle}}))      # Just use key
   {
      print " > $nextline";
   }
}
```

11.4.4 Accessing securehash entries

The use of underscores in some of the key names shown above is not an accident. Securehashes are aware of the usual Perl conventions about leading underscores. More importantly, they *enforce* those conventions.

[4] …in the subsection *Naming attributes of derived classes*…

If the unqualified key of a securehash begins with a single underscore, access to the entry for that key is restricted to its owner class and any classes derived from the owner. If the unqualified key of a securehash begins with two or more underscores, access to the entry for that key is restricted to its owner class alone. Entries with unqualified keys that don't begin with an underscore are accessible everywhere. In other words, in C++/Java parlance:

- No leading underscore indicates a public attribute.
- One leading underscore signifies a protected attribute.
- Two or more leading underscores mark an attribute as private.

For example, in the `File::new` constructor shown above, the attribute with the key `"name"` is universally accessible, the attribute with the key `"_type"` is accessible within class File or any class derived from it, and the attribute with the key `"__handle"` is only accessible within class File itself.

This arrangement is similar to the usual Perl conventions regarding the labeling of attributes although the distinction between protected and private is rarely made in Perl classes. The difference here is that Tie::SecureHash polices the intended accessibilities at run time. Whenever a piece of code attempts to access a securehash entry, the securehash checks whether the key indicates that the entry is legally accessible to that code. If it isn't, the securehash throws an exception. For example:

```
package ASCII_File;
@ISA = qw( File );
use strict;

use IO::File;
sub open
{
   my ($self) = @_;
   $self->{__handle} = IO::File->new($self->{name});
}
```

throws an exception stating: `Private key '__handle' of tied SecureHash is inaccessible from package ASCII_File`. The leading underscores of the key `'__handle'` indicate that it's a private attribute, and the qualifier indicates that it's private to the package File.

Similarly, an access attempt such as:

```
package main;

my $file = File->new();
print $file->{_type};
```

dies with the message: `Protected key '_type' of tied SecureHash is inaccessible from package main`, since the leading underscore indicates that the `"_type"` entry is accessible only within the hierarchy derived from the class in which the entry was created (i.e., File), and therefore not accessible from the main package.[5]

[5] …unless, of course, `@main::ISA = ('File')`. But that's unlikely.

For additional security, private attributes have a further access restriction. They can only be accessed within the source file in which they were originally declared. That catches most accidental abuses of encapsulation[6] such as:

```
use File;                   # Import class File (i.e., from some other file)

package File;               # Reopen the class…

sub reset_handle            # …and rummage around inside
{
  my ($self, $newval) = @_;
  $self->{__handle} = $newval;
}
```

The leading underscores of the attribute name clearly indicate it is intended to be private and should therefore be left alone. If class File had been implemented using a regular Perl hash, messing about with such an attribute would merely constitute a dangerous breach of Perl etiquette. However, because `$self` is a securehash, it is, instead, a fatal error.

11.4.5 Iterating a securehash

The issue of entry accessibility extends to iterations. The built-in functions `each`, `keys`, and `values`, when applied to a securehash, respect the accessibility constraints of its entries.

This means that the `each` iterator only returns those entries accessible at the point where the iteration occurs. In other words, if `$file` contains a reference to a File object, then the number of entries printed out by a loop such as:

```
while ( ($key, $value) = each %{$file} )
{
  print "$key => $value\n"
}
```

depends on where the loop is executed. The four possibilities are:

- The loop executes in package File in the source file in which `File::new` was originally declared. In this case, all three entries—that is, those with the keys `"name"`, `"_type"`, and `"__handle"`—are accessible, and so all three are returned by `each`.
- The loop executes within the logical bounds of package File, but in the physical bounds of some other source file. In this case, two entries (for `"name"`, and `"_type"`) are returned. The `"__handle"` entry is be skipped because it's only accessible in the file in which it was originally declared.
- The loop executes a package derived from File. Once again, two entries (`"name"` and `"_type"`) are returned. This time the `"__handle"` entry is skipped because it's only accessible from the class in which it was originally declared.
- The loop executes in a package not derived from File. Only the public entry (`"name"`) is returned. The `"__handle"` entry is skipped because it is only accessible from the class in

[6] Though determined encapsulation abusers can always resort to the `#line` directive and effectively wish themselves into any source file they choose.

which it was originally declared. The `"_type"` entry would be skipped because it is accessible only from the hierarchy of the class in which it was originally declared.

Thus, for a securehash, `each` (and likewise `keys` and `values`) only iterates through the currently accessible entries, and silently skips the rest. It's also worth noting that `each` and `keys` both return fully qualified keys, which can be used to access the iterated entry unambiguously (see the next section).

11.4.6 Ambiguous keys in a securehash

The ability to access securehash entries by unqualified keys is an important convenience. It can also be a useful programming technique when using inheritance, since, as we'll see in a moment, it allows us to create polymorphic attributes. But it also creates problems under some circumstances.

The convenience aspect is obvious. Requiring that securehash keys always be fully qualified would go against the cardinal virtue of Laziness. Who would bother to use a securehash if they always had to write `$self->{CD::Music::__rating}`, instead of `$self->{__rating}`? In most cases, the securehash contains only a single matching unqualified key, so it is redundant to require it to be qualified.

However, the use of inheritance can bring complications. It should be possible to derive one class from another without worrying about conflicts with inherited attributes. But, as we saw in chapter 6, when using a standard hash as the basis of an object, it's all too easy to set up name collisions between a class's attributes and those of an ancestral class.

Let's recreate the collection class with the settable flag from chapter 6, using securehashes instead:

```
package Settable;
use Tie::SecureHash;

sub new
{
  my ($class, $set) = @_;
  my $self = Tie::SecureHash->new($class);
  $self->{Settable::_set} = $set;           # Is the Settable object set?
  return $self;
}

sub set
{
  my ($self) = @_;
  $self->{_set} = 1;                        # Access Settable_set
}

package Collection;
@ISA = qw( Settable );

sub new
{
  my ($class, %items) = @_;
```

```
  my $self = $class->SUPER::new();
  $self->{Collection::_set} = { %items };    # Set of items in collection
  return $self;
}

sub list
{
  my ($self) = @_;
  print keys %{$self->{_set}};               # Collection::_set or
                                             # Settable_set?
}
```

We would probably expect that `Collection::list` would be smart enough to work out that accesses to the key `"_set"` in the Collection class should refer to `"Collection::_set"`, rather than `"Settable::_set"`. And, indeed, Tie::SecureHash resolves such ambiguous cases in exactly that way. The key whose owner is the least distance away up the inheritance hierarchy is the one selected. Hence, unlike those stored in a normal hash, object attributes stored in a securehash can behave polymorphically. Just like methods, attributes declared in derived classes can supersede those of the same name that were inherited from a base class, instead of colliding with them.

The concept of an attribute being the least distance away up the hierarchy means "…with respect to the class that *owns the current method*," not "…with respect to the *actual class* of the object." Otherwise, there would be problems if a Collection object called the inherited `Settable::set` method. In that case, there are still two `"_set"` keys in the securehash, but we want the appropriate one for the Settable portion of the Collection object. If the securehash looked at the type of the object (Collection), rather than the location of the method (in Settable), it would guess wrongly.

That's not to say that a securehash can always correctly interpret an unqualified key. Take the following example:

```
package Chemical;

sub new
{
  my ($class, $chem_name) = @_;
  my $self = Tie::SecureHash->new($class);
  $self->{Chemical::name} = $chem_name;
  return $self;
}

package Medicine;
@ISA = qw( Chemical );

sub new
{
  my ($class, $product_name, $chemical_name) = @_;
  my $self = Chemical->new($class, $chemical_name);
  $self->{Medicine::name} = $product_name;
  return $self;
}
```

Within any methods of the Chemical class, the unqualified public key `"name"` is always resolved to `"Chemical::name"`. Likewise, within Medicine's methods, the same key is unambiguously resolved to `"Medicine::name"`. But suppose we attempt to use the unqualified key from the main package? That is:

```
package main;
my $nostrum = Medicine->new("Didroxyfen", "dihydrogen oxide");
print $nostrum->{name};
```

Since we're not attempting to access either `"name"` entry from the namespace of its owner, there's no way to decide which entry was intended. Tie::SecureHash sidesteps the issue by immediately throwing an exception that explains the difficulty.

Of course, there's no problem if we remember to fully qualify any access to a public attribute of a securehash outside its class hierarchy. An even better solution is not to use public attributes in the first place!

Unfortunately, even if we virtuously avoid declaring public keys, ambiguity can still arise within a class hierarchy. The problem lies, as usual, with multiple inheritance. If a class inherits protected attributes with the same unqualified key from two ancestral classes, any subsequent unqualified attempt to access one of those attributes is inherently ambiguous. Listing 11.4 shows a particularly nasty case.

The reference to `$self->{_handle}` in `IO::okay` is inherently ambiguous. There are two matching keys—`"Reader::_handle"` and `"Writer::_handle"`—in the securehash referred to by `$self`, and each of their owners is equally close to the IO class in the inheritance tree, since both are owned by an immediate parent of the current class.

In this case, Tie::SecureHash has two options:

- Implement a resolution process similar to the dispatch process for methods (that is, resolve the unqualified key to the one owned by the left-most depth-first ancestor);
- Simply flag an ambiguity.

Since the situation really is ambiguous, securehashes choose the second alternative and throw an exception listing the accessible qualified keys that made the unqualified key ambiguous. Once again, the problem disappears if we say exactly what we mean and use a fully qualified key instead:

```
sub okay
{
   my ($self) = @_;
   return !$self->{Writer::_handle}->error();
}
```

It's important to note that Tie::SecureHash only ever considers accessible keys when determining whether an unqualified key is ambiguous.[7] That means, for example, that even though entries for both keys `"Reader::_handle"` and `"Writer::_handle"` may be present in the securehash object, the `Reader::next` method can unambiguously resolve an

[7] In contrast, for example, to C++, where an unqualified member access under multiple inheritance will be flagged as ambiguous even if only one of the possible targets is actually accessible at that point.

Listing 11.4 Key ambiguity within a derived class

```
package Reader;

sub init
{
   my ($self, $source) = @_;
   $self->{Reader::_handle}= new IO::File("<$source");
   $self->{Reader::__lastread}= undef;
}

sub next
{
   my ($self) = @_;
   $self->{__lastread} = $self->{_handle}->readline(); # "Reader::_handle"
}

package Writer;

sub init
{
   my ($self, $destination) = @_;
   $self->{Writer::_handle} = new IO::File(">$destination");
}

package IO;
@ISA = qw( Reader Writer );

sub new
{
   my ($class, $source, $destination) = @_;
   my $self = Tie::SecureHandle->new($class);
   $self->Reader::init($source);
   $self->Writer::init($destination);
   $self->{IO::__mode} = "read";
   return $self;
}

sub okay
{
   my ($self) = @_;
   return !$self->{_handle}->error();                    # Which "_handle"???
}
```

unqualified access to `$self->{_handle}` since only the `"Reader::_handle"` entry is accessible from package Reader.

11.4.7 Debugging a securehash

Because two or more keys in a secure hash can have the same unqualified name, and because the accessibility rules for keys are moderately complex, the behavior of securehashes blessed

into complex inheritance hierarchies can be difficult to debug in some cases. Moreover, since securehashes strictly enforce encapsulation in a most un-Perl-like manner, they can reveal unsuspected problems in a class design. Hence, it's important to be able to debug securehashes effectively.

The Tie::SecureHash module provides a method (`debug`) that may be called to dump the contents of a securehash to `STDERR`. The method can be called on any securehash—regardless of the class into which it's been blessed—with an explicit method call. For example, if we were debugging the `IO::okay` method discussed in the previous section, we might modify it:

```
sub okay
{
   my ($self) = @_;
   $self->Tie::SecureHash::debug();
   return !$self->{_handle}->error();
}
```

Alternatively, the class using a Tie::SecureHash can inherit from it as well, to make `Tie::SecureHash::debug` directly available through its objects:

```
package IO;
@ISA = qw( Reader Writer Tie::SecureHash );

sub okay
{
   my ($self) = @_;
   $self->debug();
   return !$self->{Writer::_handle}->error();
}
```

Either way, when `IO::okay` is called, the `debug` method will print:

```
In subroutine 'IO::okay' called from package 'IO':

   Writer::
      (?) '_handle'  =>  'IO::File=GLOB(0x10028ba0)'
               >>> Ambiguous unless fully qualified. Could be:
               >>>  Reader::_handle
               >>>  Writer::_handle

   IO::
      (+) '__mode'  =>  'read'

   Reader::
      (?) '_handle'  =>  undef
               >>> Ambiguous unless fully qualified. Could be:
               >>>  Reader::_handle
               >>>  Writer::_handle

      (–) '__lastread'  =>  undef
               >>> Private entry of Reader::
               >>> is inaccessable from IO.
```

In other words, `Tie::SecureHash::debug` reports the current location details[8] and the key and value of each entry of the securehash, categorized by owner. More importantly, it reports the accessibility of each entry at the point where it was called. Entries preceded by a "(+)" are accessible, entries preceded by a "(–)" are not, and entries preceded by a "(?)" are accessible but ambiguous unless the key is fully qualified.

11.4.8 "Fast" securehashes

Securehashes provide an easy means of controlling the accessibility of object attributes on a per-attribute basis. Unfortunately, that ease and flexibility comes at a cost.

As explained in chapter 9, accessing the entries of tied hashes is often five to ten times slower than for untied hashes. Add to that the cost of the tests that a securehash has to perform before it can grant access to an entry, and the cost blows out to between ten and twenty times as much as for an untied hash. That makes the use of securehashes impractical in most production code.

With this problem in mind, the Tie::SecureHash module provides a way to have your cake (properly encapsulated attributes…) and eat it too (…accessed at untied hash speeds). The trick lies in observing that the actual enforcement of access restrictions is only required when a piece of code attempts to violate those restrictions. In other words, if no one ever breaks the law, you don't need any actual police to enforce it.

The solution is to develop the application using Tie::SecureHash to enforce proper encapsulation, then optimize the final code by converting every securehash to a regular hash (which provides no enforcement). As long as the development code has been fully tested, the enforcement code provided by the securehashes is no longer required.

To convert from securehashes to regular hashes, it's not necessary to change any of the code that *accesses* a securehash, only the code that *creates* it. That's because a securehash's interface mimics that of a regular hash,[9] so code that accesses one will access the other just as well. It's a form of polymorphism: keeping the interface the same means the client code doesn't have to worry about the implementation at all.

Of course, in the typical large application in which you might want to use securehashes, hunting for every situation where a securehash is created and replacing that securehash with a regular hash can be time-consuming and error-prone. If we have to locate every call to `Tie::SecureHash::new` and every use of `tie %somehash, Tie::SecureHash,` we're likely to miss at least one.

To reduce that burden, the Tie::Securehash module provides a special "fast" mode, in which a call to `Tie::SecureHash::new` returns a reference to an ordinary hash, rather than to a securehash. Hence, in fast mode, we don't have to replace any call to `Tie::SecureHash::new`, since it correctly adjusts its behavior automatically. Of course, that doesn't solve the problem of any raw `tie %somehash, Tie::SecureHash`, but that's just another reason to use `Tie::SecureHash::new` instead.

---

[8] Access violations often occur because methods are not actually called from the expected package (or file), or they're not defined in the class in which they're assumed to be.

[9] Well, almost. See section 11.4.9 for the single exception.

Fast mode is activated by importing the entire module with an extra argument:

```
use Tie::SecureHash "fast";
```

Converting to fast mode: an example

Developing securehash-based classes that can later be converted to fast mode requires three phases of coding. First, we create the code using securehashes:

```
package Color;
use Tie::SecureHash;

sub new
{
  my $self = Tie::SecureHash->new($_[0]);
  $self->{red}             = $_[1];
  $self->{Colour::green}   = $_[2];
  $self->{Component::blue} = $_[3];
  $self->{Color::__bright} = 0.299*$_[1] + 0.587*$_[2] + 0.114*$_[3];
  return $self;
}

package main;

my $color = Color->new(128,255,255);
print $color->{Color::__bright};
```

Then, we debug the code to eliminate the error messages that the securehashes will have produced in response to access violations:

```
package Color;
use Tie::SecureHash;

sub new
{
  my $self = Tie::SecureHash->new($_[0]);
  $self->{Color::red}      = $_[1];  # Add missing owner name
  $self->{Color::green}    = $_[2];  # Correct wrongly spelt owner name
  $self->{Color::blue}     = $_[3];  # Replace wrong owner name
  $self->{Color::__bright} = 0.299*$_[1] + 0.587*$_[2] + 0.114*$_[3];
  return $self;
}

# add accessor for private attribute
sub brightness { return $_[0]->{Color::__bright} }

package main;

my $color = Color->new(128,255,255);
print $color->brightness;  # use accessor instead of private attribute
```

Finally, we optimize the entire code, converting every securehash to a regular hash by activating fast mode:

```
package Color;
use Tie::SecureHash "fast"; # switch to "fast" mode

sub new
{
  my $self = Tie::SecureHash->new($_[0]);
  $self->{Color::red}      = $_[1];
  $self->{Color::green}    = $_[2];
  $self->{Color::blue}     = $_[3];
  $self->{Color::__bright} = 0.299*$_[1] + 0.587*$_[2] + 0.114*$_[3];
  return $self;
}

sub brightness { return $_[0]->{Color::__bright} }

package main;

my $color = Color->new(128,255,255);
print $color->brightness;
```

Apart from that one extra argument to `use Tie::Securehash`, the debugged source code doesn't change in any way. But `Tie::SecureHash::new` now returns a reference to a regular hash and, although the code works exactly as before, access to attributes has been greatly accelerated.

11.4.9 "Strict" securehashes

This develop-with-restrictions-then-run-without-them approach works well provided we accept two limitations: always use `Tie::SecureHash::new` to create securehashes, and never use unqualified keys to access them.

The need to use `Tie::SecureHash::new` was explained above. If the `Color::new` constructor had been implemented like this

```
sub new
{
  my %securehash;
  tie %securehash, Tie::SecureHash;
  my $self = bless \%securehash, $_[0];
  $self->{Color::red}      = $_[1];
  $self->{Color::green}    = $_[2];
  $self->{Color::blue}     = $_[3];
  $self->{Color::__bright} = 0.299*$_[1] + 0.587*$_[2] + 0.114*$_[3];
  return $self;
}
```

then, even with fast mode activated, the constructor still ties the object to the Tie::SecureHash class. The resulting code works, but slowly. So, the source code has to be manually changed when moving to fast mode—by replacing the `tie` statement. In other words, `Tie::Secure-`

`Hash::new` knows about fast mode and can adjust for it, but the built-in `tie` function doesn't and can't.[10]

The second restriction is a more significant problem. One of the useful features of a securehash is that, once an entry has been declared with its full qualifier, you can thereafter refer to it without the qualifier and expect the securehash to get it right in all unambiguous cases. However, if we're replacing the securehash with a regular hash, that "do what I mean" intelligence disappears. Since a regular hash doesn't recognize an unqualified key as being the same as a fully qualified key, this can lead to subtle bugs when the securehashes are removed. For example, if we code `Color::brightness` like so:

```
sub brightness { return $_[0]->{__bright} }
```

it works perfectly as long as Color objects are implemented as securehashes, but silently breaks as soon as the securehashes are replaced by regular hashes in fast mode.

That's because, although the constructor stores the brightness value under the key `"Color::__bright"`, the `brightness` method looks it up under the key `"__bright"`. Since a regular hash considers these two keys to be completely unrelated, it won't redirect the access request to the `"Color::__bright"` entry. Instead, it autovivifies an entry for the key `"__bright"` and returns that new entry's `undef` value, which would probably then be automatically converted to zero. Oops!

These two restrictions are not particularly onerous, but they can be difficult to apply consistently in a large application.[11] To make conversion to fast mode easier, Tie::SecureHash offers another mode called "strict." Like fast mode, this mode can be invoked by importing the module with the appropriate argument:

```
use Tie::SecureHash "strict";
```

In strict mode, securehashes control access in their normal way, except that they also produce warnings whenever a hash is explicitly tied to Tie::SecureHash and whenever an unqualified key is used to access a securehash. Thus, code that uses securehashes and runs without warnings in strict mode is guaranteed to behave identically in fast mode.

11.4.10 The formal access rules

The access rules for a securehash are designed to provide secure encapsulation with minimal inconvenience and maximal intuitiveness—so that keys need only be qualified when they are created and where they would be ambiguous. However, to produce this appearance of transparency, the formal access rules are quite complicated. The following subsections list them explicitly. Unless you're planning to use the module immediately, you may like to skip this bit for now.

---

[10] Actually, the way Tie::SecureHash is set up, any attempt to tie a securehash while in fast mode causes a warning to be generated. That doesn't make converting such code back to regular hashes any easier, but at least it tells you where the problems are.

[11] …or to retrofit to an existing one when the decision to use fast mode is made only after the code is complete.

All entries

- No entry for an unqualified key is autovivifying. Each entry must be declared before it is used. Qualified keys *do* autovivify their entry, so an entry may be declared as part of its initial use.
- The key of each entry must be explicitly qualified (in the form "`<owner>::<key>`") when an entry is declared.
- An entry is owned by the package whose name was used as the explicit qualifier in its declaration.
- Entries must be declared by code that's within the namespace of their owner's package and file.
- An unqualified key is always interpreted as referring to the key owned by the current package, if such a key exists, no matter how many other accessible matching keys the hash may also contain.
- Otherwise, accesses through an unqualified key throw an exception if the number of *accessible* matching keys in the securehash is not 1 (either `...key does not exist...` if the number is zero, or `...key is ambiguous...` if it is greater than 1).
- A fully qualified key is never ambiguous, though it may be nonexistent, or inaccessible from a particular namespace.

Public entries

- Public accessibility of entries is indicated by their unqualified key beginning with a character other than an underscore.
- Public entries may be subsequently accessed from any package in any source file.
- A public entry's key is ambiguous if it isn't explicitly qualified, *and* no matching key is owned by the current package, *and* two or more matching unqualified keys are owned by any other packages.

Protected entries

- Protected accessibility of entries is indicated by their unqualified key beginning with a single underscore.
- Protected entries may subsequently be accessed from any package (P) in any source file, provided that at the point of access, P is, or inherits from, the owner package (*Owner*). That is, a protected entry is accessible in any package P, where `P->isa("Owner")` is true.
- Protected keys declared to be owned by a given package will hide entries with the same unqualified key inherited from parent classes of that package. Any inherited entry hidden in this way is inaccessible from the namespace of the derived class, unless accessed via a qualified key.
- A protected key is ambiguous if it's not explicitly qualified, *and* no matching key is owned by the current package, *and* two or more accessible matching keys are owned by two or more other packages, *and* those other packages are inherited by the current package through two distinct entries in its inheritance hierarchy.

Private entries

- Private accessibility of entries is indicated by their unqualified key beginning with two or more underscores.
- Private entries can be accessed only from within the namespace of their owner package, and only from the source file in which they were originally declared.
- Unqualified private keys are never ambiguous. Because private entries are only ever accessible from a single class, there can be at most only one accessible matching private key.

11.5 WHERE TO FIND OUT MORE

Tom Christiansen's perltoot tutorial has an excellent description of the use of closures to enforce encapsulation. Closures themselves are discussed in the perlref, perlsub, and perlfaq7 documentation and in chapter 4 of *Advanced Perl Programming*.

The flyweight pattern is discussed at great length in *Design Patterns* (although in the context of C++, not Perl).

The Tie::SecureHash is available from the CPAN in the directory http://www.perl.com/CPAN/authors/id/DCONWAY/.

11.6 SUMMARY

- Techniques for enforcing encapsulation of Perl objects rely on hiding the interface of the datatype that is implementing each object, typically by taking advantage of the limited scope of lexical variables.
- One approach is to use a closure as an object. The closure itself provides restricted access to out-of-scope lexicals, which in turn store attribute values.
- Alternatively, a scalar object can be used to hold an index into a lexical table that stores the actual objects. This approach is known as the *flyweight pattern*.
- The Tie::SecureHash module simulates a regular hash, but provides three levels of enforced encapsulation on individual entries.
- Hashes tied to Tie::SecureHash are slower than regular hashes, so the module is best used in development and then removed (using the fast option) in production code.