8

Object-Oriented Programming

In Chapter 7, "Subroutines and Modules," you learned how to organize your code into subroutines, packages, and modules. In this chapter, you'll find out how to create objects and use the object-oriented features of Perl.

Using a Hash as an Object

Suppose that you want to create a module that handles time. The time module should let you

- Create a time from two integers—say 8:30 out of 8 and 30.
- Add and subtract times (8:30–1:20=7:10).
- Turn the time into a string for printing.

This chapter starts with a function to take two integers and return a time hash. Actually, the function returns a reference to the hash. The function to create a new time looks like this:

```
sub new($$)
{
    my $hours = shift; # Initial hours
    my $minutes = shift; # Initial minutes
    my $self = {};    # The hash we are creating
```

```
    $self->{hours} = $hours;
    $self->{minutes} = $minutes;
    return ($self);
}
```

All this example really does is take the hours and minutes that you get as parameters and stuffs them in a hash. This function returns a reference to the hash you created.

Note that this function does not do any limit or error checking.

Now define a function to add one hash to another:

```
sub add($$)
{
    my $self = shift;    # The hash we wish to add to
    my $other = shift;    # The other hash

    $self->{hours} += $other->{hours};
    $self->{minutes} += $other->{minutes};
    $self->{hours} += int($self->{minutes} / 60);
    $self->{minutes} %= 60;
}
```

Like all functions that manipulate the object (the hash), the first argument to this function is the hash you are manipulating.

This code is straightforward. It simply adds together the two elements of the time and then handles any minutes that might have overflowed.

Again, note that you don't do any range or error checking. You assume that the result is < 24:00.

The `subtract` function works pretty much the same. You can see it in Listing 8.1.

Finally, you need something to turn a time into a string. For that you have the `to_string` function:

```
sub to_string($)
{
    my $self = shift;    # The hash to turn into a string

    return(sprintf("%d:%02d",
    $self->{hours}, $self->{minutes}));
}
```

Listing 8.1 contains the full `time_hash` package.

Listing 8.1 ***time_hash.pm***

```
use strict;
use warnings;
#
# A package to handle time data (implement using a hash)
```

```
#
package time_hash;

sub new($$)
{
    my $hours = shift; # Initial hours
    my $minutes = shift; # Initial minutes
    my $self = {};    # The hash we are creating

    $self->{hours} = $hours;
    $self->{minutes} = $minutes;
    return ($self);
}

sub add($$)
{
    my $self = shift;    # The hash we wish to add to
    my $other = shift;    # The other hash

    $self->{hours} += $other->{hours};
    $self->{minutes} += $other->{minutes};
    $self->{hours} += int($self->{minutes} / 60);
    $self->{minutes} %= 60;
}

sub subtract($$)
{
    my $self = shift;    # The hash we wish to subtract from
    my $other = shift;    # The other hash

    $self->{hours} -= $other->{hours};
    $self->{minutes} -= $other->{minutes};
    while ($self->{minutes} < 0) {
        $self->{minutes} += 60;
        $self->{hours}--;
    }
}

sub to_string($)
{
    my $self = shift;    # The hash to turn into a string

    return(sprintf("%d:%02d",
    $self->{hours}, $self->{minutes}));
}

1;
```

The next step is figuring out how to use the `time hash` package. You can create a new time variable using the function `time_hash::new`. For example:

```
my $time1 = time_hash::new(1,30);
my $time2 = time_hash::new(8,40);
```

You then can add and subtract time, using the functions `add` and `subtract`. For example to add `$time2` to `$time1`, use the statement

```
time_hash::add($time1, $time2);
```

You can print the results with the statement

```
print "Time1 is ", time_hash::to_string($time1), "\n";
```

Listing 8.2 shows a simple test routine.

Listing 8.2 ***test_hash.pl***

```
use strict;
use warnings;

use time_hash;

my $time1 = time_hash::new(1,30);
my $time2 = time_hash::new(8,40);

print "Time1 is ", time_hash::to_string($time1), "\n";
print "Time2 is ", time_hash::to_string($time2), "\n";
time_hash::add($time1, $time2);
print "Time1 is ", time_hash::to_string($time1), "\n";
```

Design Notes

You should note a few things about the design of the package. First, the first parameter for every function, except `new`, is a reference to the object being acted on. This technique is commonly used to do object-oriented programming when you don't have an object-oriented language.

Second, the name `new` was chosen to be the constructor for the object. There's nothing magical about the name `new`, except that every C++ programmer associates `new` with construction of an object.

Finally, in all the functions, the variable `$self` is used to refer to the object. Again, you could have used any name; by convention, most Perl programmers use `$self` to refer to the current object. (C++ uses the built-in keyword `this` to do the same thing.)

Basic Perl Objects

The `time_hash` package has all the ingredients that make up an object. Data and functions act on that data. In Perl, these are called *methods*; in C++, they are called *member functions*.

The only problem is that you must manually specify the class variable every time you call a method. It would be nice if you could do that automatically.

The hash knows all about the data contained in it. The only thing it doesn't know is what methods can be used on that data, and you need that information if you are about to write object-oriented code. Perl solves this problem through the use of the `bless` function.

This function tells Perl to associate the subroutines (methods) in the current package or, in other words, the package where this function was called with the current hash. Modify your `new` function and put the following statement at the end of it:

```
return (bless $self);
```

Now object `$self` (a hash created by the new function in the `time_hash` package) is associated with this package.

Now to create a new `time_hash` variable, use the following statement:

```
my $time1 = time_hash::new(1, 30);
```

Perl looks for the method `new` in the package `time_hash` and calls it. The `new` method returns an object, also known as a *blessed hash*.

Here's how to call the `add` function:

```
$time1->add($time2);
```

This adds `$time2` to `$time1`. Notice that you didn't have to specify the package name. Because `$time1` is a blessed hash, Perl knows that it comes from the `time_hash` package and therefore knows how to find the `time_hash::add function`.

Perl does one other thing in this case; it adds the hash reference (`$time1`) to the beginning of the parameter list. So, although you call `add` with one parameter (`$time2`), the `add` function receives two parameters (`$time1`, `$time2`).

The `add` function begins with

```
sub add($$)
{
    my $self = shift;    # The hash we wish to add to
    my $other = shift;    # The other hash
```

The first thing that the function `add` does is take the reference to the hash off the parameter list and assign it to `$self`. The variable `$self` is the equivalent of the C++ keyword `this`. It is a pointer to data in the class.

However, in C++ you can omit references to `this` when you access member functions. You can't do this in Perl; the `$self` is required. For example:

```
$self->{minutes}
```

Although there's nothing magical about using the variable name `$self`, it has become the standard name used by most Perl programmers.

Because of the way Perl does object-oriented programming, all the member functions, except `new`, look the same as in Listing 8.3.

However, your use of this package is much simpler. You can see the new test program in Listing 8.3.

Listing 8.3 ***test_obj.pl***

```
use strict;
use warnings;

use time_hash;

my $time1 = time_hash::new(1,30);
my $time2 = time_hash::new(8,40);

print "Time1 is ", $time1->to_string(), "\n";
print "Time2 is ", $time2->to_string(), "\n";
$time1->add($time2);
print "Time1 is ", $time1->to_string(), "\n";
```

Polymorphism

One nice thing about C++ classes is the capability to create a derived class. For example, you can define a base class animal and then create derived classes for pigs, horses, and dogs.

In C++, a derived class might look like this:

```
class dog: public animal {
    // ....
```

You can do the same thing in Perl through the use of the `@ISA` array. In a derived class, or package, the derived class contains a list of base packages. For example:

```
package dog;
@ISA = qw(animal);
```

When Perl looks for a method (C++ member function), it first looks in the package being called. It then searches through the packages specified by the `@ISA` array.

The search is done depth first, so if you have the following packages:

```
package computer;
@ISA = qw(motherboard case);

package motherboard;
@ISA = qw(cpu memory);

package case;
@ISA = qw(power_supply disks);
```

the search order for the methods in the package computer is

- computer
  - motherboard
    - cpu
    - memory
  - case
    - power_supply
    - disks

Note that the searching occurs only if you have a variable of the type `computer`. For example, suppose that `start_disk` is a method in the package `disks`. Then you can use the following code:

```
my $server = computer->new();
$server->start_disk();
```

It will not work if you call a method directly:

```
computer::start_disk($server);   # Fails
```

Information Hiding

In C++, the protection keywords `public`, `private`, and `protected` tell the user who may access the member functions and member variables. In Perl, everything is public. No access protection exists at all.

> **Common Mistake: Exporting the Methods**
> When you design an ordinary package, you must put all the public functions and variables in the `@EXPORT` array. Otherwise, they will not be readily accessible to the caller when strict syntax checking is turned on.

continues

> However, when you are doing object-oriented programming, the methods (functions) are addressed through the object. For example:
>
> ```
> $time->to_string();
> ```
>
> Because `$time` has been blessed, Perl knows what methods are associated with this object. In this case, the methods do not need to be exported.
>
> If they are, it pollutes the namespace unnecessarily, so don't export methods.

Operator Overloading

Perl enables you to perform operator overloading. This enables you to define a function that's called when two `time` objects are added such as the following:

```
my $time3 = $time1 + $time2;
```

This section defines a new package called `time_over` that uses operator overloading to allow you to add, subtract, and print times.

To define a function for adding two time classes, you need to use the `overload` package:

```
use overload (
    '+' => \&add,
    '-' => \&subtract,
    '"' => \&to_string
);
```

Now when Perl sees you add two `time` objects, it calls this function to perform the addition. Listing 8.4 shows the subroutine to add two `time` objects.

Listing 8.4 ***add* Function**

```
# Incomplete function, see below for the full version
sub add($$)
{
    my $time1 = shift;  # The first time for adding
    my $time2 = shift;  # The second time for adding
    my $result = {};    # Result of the addition

    $result->{hours} = $time1->{hours} + $time2->{hours};
    $result->{minutes} = $time1->{minutes} + $time2->{minutes};
    $result->{hours} += int($result->{minutes} / 60);
    $result->{minutes} %= 60;
    return (bless $result);
}
```

But what happens when you add a `time` object to a number? For example:

```
my $time1 = $time1 + 10;
```

The answer is that Perl calls your `add` function. Unfortunately, your function can't handle this situation.

In C++, you would solve this problem by overloading the `add` function. Thus, you would create two functions:

```
time &add(const time &t1, const time &t2);
time &add(const time &t1, const int t2);
```

Perl doesn't let you overload functions. Instead, all `add` calls, no matter what the type of the variable, are funneled through this one function.

So the function needs to be smart enough to detect when the second parameter is an object and when it's a scalar. To do this, use the `ref` function. If `$var` is a reference, the `ref` function returns the name of the object being referenced. If `$var` is not a reference, the `ref` function returns `undef`.

So to check whether you have a number, use the following statement:

```
    if (not ref($time2))
```

If you find that you have a number, change it to an object:

```
    if (not ref($time2)) {
        $time2 = new time_over($time2);
    }
```

Listing 8.5 shows the full `add` function.

Listing 8.5 **Full *add* Function**

```
sub add($$)
{
    my $time1 = shift;    # The first time for adding
    my $time2 = shift;    # The second time for adding
    my $result = {};    # Result of the addition

    if (not ref($time2)) {
    $time2 = new time_over($time2);
    }

    $result->{hours} = $time1->{hours} + $time2->{hours};
    $result->{minutes} = $time1->{minutes} + $time2->{minutes};
    $result->{hours} += int($result->{minutes} / 60);
    $result->{minutes} %= 60;
    return (bless $result);
}
```

So you've taken care of the following case:

```
$time1 = $time2 + 10;
```

but what happens when you have

```
$time1 = 10 + $time2;
```

The answer is that Perl switches the operators so that `$time2` is first and then calls the `add` function.

Perl also does a number of other little things behind your back. If you use the += operator and don't explicitly specify a function for it, Perl turns the += into an `add`. It also does the same for the ++ operator. So by defining the code for one operator, you can perform all sorts of explicit and implied additions:

```
$time1 = $time2 + $time3;
$time1 += 5;
$time1 += $time2;
$time1++;
```

The `subtract` function looks pretty much like `add`, but there's a problem. When the first argument is a number and the second argument is an object, Perl switches the argument. This is not a problem with addition, but in subtraction, it causes strange answers.

Fortunately, Perl supplies an extra argument that tells you when the operands have to be switched. This flag is true if Perl has flipped the arguments for you.

To use this in the `subtract` function, you must first grab the argument:

```
sub subtract($$$)
{
    my $time1 = shift;    # The hash we wish to subtract from
    my $time2 = shift;    # The other hash
    my $flag = shift;    # The manipulation flag
```

Then after all the reference stuff is handled, check to see whether the arguments need to be flipped:

```
    if ($flag) {
        ($time1, $time2) = ($time2, $time1);
    }
```

Finally, overload the operator double quote (`"`). This operator is used to turn the object into a string. This is useful for printing. For example:

```
print "The time $time\n";
```

Listing 8.6 shows the full overloaded time package. Listing 8.7 presents the code used to test it.

Listing 8.6 ***time_over.pm***

```
use strict;
use warnings;
package time_over;

sub add($$);
sub subtract($$$);
sub to_string($$);

use overload (
    '+' => \&add,
    '-' => \&subtract,
    '""' => \&to_string
);

sub new($;$)
{
    my $class = shift;   # The class name
    my $hours = shift;   # Initial hours
    my $minutes = shift; # Initial minutes
    my $self = {};       # The hash we are creating

    if (not defined($minutes)) {
    $minutes = 0;
    }

    $self->{hours} = $hours;
    $self->{minutes} = $minutes;
    return (bless $self, $class);
}

sub add($$)
{
    my $time1 = shift;   # The first time for adding
    my $time2 = shift;   # The second time for adding
    my $result = {};     # Result of the addition

    if (not ref($time2)) {
        $time2 = new time_over($time2);
    }

    $result->{hours} = $time1->{hours} + $time2->{hours};
    $result->{minutes} = $time1->{minutes} + $time2->{minutes};
    $result->{hours} += int($result->{minutes} / 60);
    $result->{minutes} %= 60;
    return (bless $result);
}

sub subtract($$$)
{
```

continues

Listing 8.6 **Continued**

```
    my $time1 = shift;   # The hash we wish to subtract from
    my $time2 = shift;   # The other hash
    my $flag = shift;    # The manipulation flag
    my $result = {};     # The result we are computing

    if (not ref($time2)) {
        $time2 = new time_over($time2);
    }
    if ($flag) {
        ($time1, $time2) = ($time2, $time1);
    }
    $result->{hours} = $time1->{hours} - $time2->{hours};
    $result->{minutes} = $time1->{minutes} - $time2->{minutes};
    while ($result->{minutes} < 0) {
        $result->{minutes} += 60;
        $result->{hours}--;
    }
    return (bless ($result));
}

sub to_string($$)
{
    my $time = shift;     # The hash to turn into a string

    return(sprintf("%d:%02d",
    $time->{hours}, $time->{minutes}));
}

1;
```

Listing 8.7 ***test_over.pl***

```
use strict;
use warnings;

use time_over;

my $time1 = time_over->new(1,30);
my $time2 = time_over->new(8,40);

my $time3 = $time1 + $time2;
print "$time1 + $time2 = $time3\n";

my $time4 = $time1 + 10;
print "$time1 + 10 = $time4\n";

my $time5 = 10 - $time2;
print "10 - $time2 = $time5\n";
```

```
$time5 = $time2 - 1;
print "$time2 - 1 = $time5\n";

my $time6 = time_over->new(8,40);
my $time7 = $time6;
$time6 += 2;
print "8:40 += 2 => $time6\n";

$time6++;
print "10:40++ => $time6\n";
```

Summary

With Perl's object-oriented features and operator overloading, it's easy to create simple-to-use, reusable code. Proper organization and a good design are key to making an efficient and effective program.

In the next few chapters, you'll see how to apply the techniques provided here to use Perl to solve some real-world problems. These include CGI programming to handle web forms, using the Tk package to build a GUI, and even mixing C and Perl programs together through the use of inline C functions.

Exercises

1. Write a persistent list object. This object acts just like an array except that it stores its values in a file and reloads when the program is run again. In other words, the values of the array persist even after the program is terminated.
2. Write a package to handle Roman numerals.
3. Write classes to support a workflow tool. The base class should provide
   - A title for this step in the workflow
   - A flag indicating whether the work has been done
   - A virtual method called `do_it`, which actually does the work

   There are two different types of steps: manual, which is done by an operator, and automatic, which is done by a system command. Devise derived classes to handle both these cases.
4. Write a complex number class.
5. Devise a base class called `person` and derived classes for men and women with different methods. Try to avoid triggering your organization policy on sexual harassment in the process.

Resources

Online Documentation

- **`perldoc perlobj`**—Basic information on creating Perl objects.
- **`perldoc perlboot`**, **`perldoc perltoot`**, **`perltootc`**—A three-part tutorial on the object-oriented features of Perl.
- **`perldoc perlbot`**—Object-oriented tricks and tips.

Modules

- **`overload`**—Module to handle operator overloading.