5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246

joseph@5sigma.com • http://www.perltraining.com

Object-Oriented Programming in Perl

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Overview

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Object-oriented programming in Perl

- The following sections cover
- Basics—a summary of object-oriented programming features in Perl
- Idioms—how do class and object data work in Perl, and what can be done about it?
- Patterns—see Perl applied to some “Design Patterns”
- By no means a complete exploration of object-oriented Perl, but by the time we’re done, you’ll have a feel for what’s possible

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Basics

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246

joseph@5sigma.com • http://www.perltraining.com

Object-oriented Perl

- Perl is object-oriented and fully “buzzword compliant”
- Classes, methods, inheritance
- Objects, constructors, destructors
- Object-oriented modules
- Pretty much all the other “missing” stuff can be invented if necessary

Classes and methods

- A Perl class is a package
- A method is a function intended to be called via *method call syntax*—in other respects it's a normal function
- There are two kinds of method call syntax
- One is *indirect object*, like filehandle of `print`

```
new Castaway 'last' => 'Gilligan';
shipwreck $gilligan $long, $lat;
```

`Castaway` *is class name;* `new` *is method*
`$gilligan` *is instance;* `shipwreck` *is method*

- The other is the *arrow form*, somewhat like C++

```
Castaway->new('last' => 'Gilligan');
$gilligan->shipwreck($long, $lat);
```

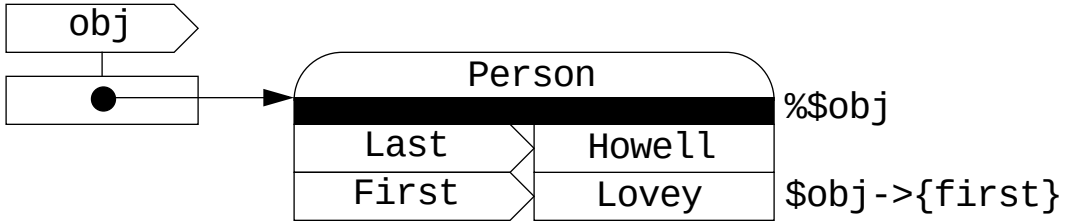
- Both translate into a subroutine call, with the class/instance passed as first parameter

```
&Castaway::new("Castaway", 'last' => 'Gilligan');
Castaway::shipwreck($gilligan, $long, $lat);
```

Objects

- A Perl object is something (generally a hash) that has been blessed
- The `bless` operator takes a reference and blesses *what it points to* into the current class

```
package Person;
$obj = bless { Last => 'Howell', First => 'Lovey' };
```



- The PEGS notation for objects is a rounded corner box containing the class name
- `bless` takes an optional second argument, which is a class name

```
$obj = bless { Last => 'Howell', First => 'Lovey' }, 'Person';
```

- The `ref` operator returns the class name when applied to a reference to an object

Constructors

- A *constructor* in Perl is a method that creates and returns a blessed object

```
package Person;
sub new {
  my $package = shift;                  # the package name comes in as the first parameter
  my %args = @_;                        # remaining args end up in this hash
  my $self = \%args;                    # and then construct a hashref (this will become the object)
  bless $self, $package;                # now cause the object to remember that it came from Person
  $self;                                # and return this new Person
}
package main;
$a = new Person(First => "Lovey", Last => "Howell");
$aa = &Person::new("Person", "First", "Lovey", "Last", "Howell");      # equivalent
```

- There is nothing special about `new`—a constructor can have any name

Objects and methods

- Once we have an instance, we can call instance methods

```
package Person;
sub get_first {
  my $self = shift;                    # first parameter is always the object itself
  $self->{"First"};                    # $self is a hashref, so deref it and fetch the value at this key
}
sub get_last {
  my $self = shift;
  $self->{"Last"};
}
package main;
$first_name = $a->get_first;           # get first name for the object $a
$first_name = &Person::get_first($a);  # equivalent form (mostly)
```

- Note that Perl figured out that `$a` belonged to `Person` to find the method—this is what the `bless` is for during an object's construction

Inheritance

- Perl looks for a package variable called `@ISA`
- `@ISA` contains one or more class names (if more than one, multiple inheritance!)

```
package Castaway;
@ISA = qw(Person);
package main;
$b = new Castaway(Last => 'Howell', First => 'Thurston', Ship => 'Minnow');
```

- Here, `Castaway::new` is not defined, so Perl looks in `@Castaway::ISA` for additional packages
- Ultimately, the method call translates into:

```
$b = &Person::new('Castaway',
                  Last => 'Howell', First => 'Thurston', Ship => 'Minnow');
```

- Note that the package argument supplied to the constructor is `'Castaway'`, even though we are using a `Person` constructor.
- Inheritance also supports the overriding of methods in the traditional manner (see below)

5 Sigma Productions
P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

More about inheritance

- Now we can add additional methods in the child class

```
sub Castaway::get_ship {
  my $self = shift;
  $self->{"Ship"};
}
```

- and mix and match both direct and inherited methods

```
print $b->get_last, ", ", $b->get_first, " lost on ship", $b->get_ship, "\n";
```

- Note that inheritance works only with the method call syntax
- Using normal subroutine invocation syntax forces a simple lookup (or failure) — no `@ISA`

```
$ship = $b->get_ship;                    # succeeds
$ship = &Castaway::get_ship($b);         # also succeeds
$last = $b->get_last;                    # suceeds, calling Person::get_last
$last = &Castaway::get_last($b);         # fails... not defined
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Still more about inheritance

- Methods in derived classes can override those in base classes
- Suppose that, for some bizarre reason, all `Castaway`s should appear to be named Gilligan

```
sub Castaway::get_first { 'Gilligan' }
sub Castaway::get_last { '' }
```

- If you call `$b->get_first` for some `Castaway` object `$b`, you'll get back `'Gilligan'` every time now
- This doesn't affect the way that objects of class `Person` or other classes derived from `Person` behave
- If you don't want the method search to start with the current class, but want to start looking farther up the inheritance chain, you can add the name of the parent class to the method call

```
$last = $b->Person::get_last
```

starts looking for get_last in Person, not Castaway

Multiple inheritance

- Multiple inheritance is as easy as adding more classes to the list in `@ISA`

```
package Actor;
sub get_real { shift->{Real} }
package TVCastaway;
@ISA = qw(Castaway Actor);
package main;
$c = TVCastaway->new(Last => 'Howell', First => 'Thurston',
                     Ship => 'Minnow', Real => 'Jim Backus');
print $c->get_first, "'s real name is ", $c->get_real, "\n";
```

- The `@ISA` tree is searched depth-first, so in this case, both `Castaway` and `Person` are checked for a method named `get_real` before it is found in `Actor`
- Multiple inheritance doesn't involve data in Perl, so it isn't as scary as it is in other languages (fortunately)

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

`isa` and `can`

- There are some methods already built into every class
- `$obj->isa($class)` returns true if `$obj` is or is derived from $class

```
print "an actor!" if $c->isa('Actor');
print "it's a Person" if TVCastaway->isa('Person');
```

also works for classes

- `$obj->can($method)` returns true if `$obj` has a callable method with the name `$method` somewhere in its inheritance hierarchy
- In fact, it returns a code ref to that method

```
$method = $c->can('get_first');
$first = $method->($c) if $method;
```

can `$c get_first`*?*

call the method through the code reference

Destructors

- When an object goes out of scope (or at program termination), Perl will call its destructor if it has one
- A destructor is a method named `DESTROY`

```
package Person;
sub DESTROY {
  my $self = shift;
  print "destroying '$self->{First} $self->{Last}'\n";
}
```

- Destructors are useful for printing informational messages, cleaning up temporary files, releasing system resources (lockfiles, semaphores, etc.)
- Destructors can be inherited (called as `$deadobject->DESTROY`)
- Cleaning up the object may invoke additional destructors during the destruction

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246

joseph@5sigma.com • http://www.perltraining.com

Idioms

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Object-oriented programming idioms in Perl

- Perl has a very sparse object-oriented framework
- You might think this is good, or bad, or both
- It's good because the missing features are what add complexity (sometimes inordinate amounts of it) to other programming languages
- For example, lack of access control—public, protected, private—makes Perl class design much simpler
- Of course C++ design is simpler, too, if you make everything public
- It's bad because those features are valuable—presumably they weren't built into C++ and other languages just for the heck of it
- Certain idioms, like private class data, are very easy to implement in clear, idiomatic Perl
- Others, like private object data, are impossible to implement as you would expect, but not too difficult to emulate

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

A contract or a cage

- One of the things that makes Perl a great language for prototyping (and some will argue, regular development too) is that it doesn't have much in the way of access control
- In general, code in one part of a Perl program can access code and data in another
- There are exceptions, like `my` variables
- C++ has many access control features—`public`/`private`/`protected` members and inheritance, `const` types, for example
- Access control makes things complicated, thus C++ also has `mutable` members
- As a developer, do you want a complicated environment where you can define tight boundaries for client software, or do you want a simpler environment where you put trust in your clients
- As a client, do you want a rigid interface that you can't violate (a cage), or a less substantial interface that you shouldn't violate (a contract)?
- Defining a rigid interface requires attention to detail
- A rigid interface is typically large and difficult to change

Class data

- Class variables belong to a class, not particular objects—they are “shared” within a class
- “Class” = “Package” ... hmm
- If you don’t care about access control, just use a package variable
- Suppose we want to keep track of the number of `Person` objects created so far

```
package Person;
sub new {                                    # constructor for class Person
  my $class = shift;
  my $self = { @_ };
  $count++;                                  # $Person::count contains the count
  bless $self, $class;                       # return a new Person
}
```

- Within package `Person`, `$count` contains the number of times the `Person` constructor has been called
- We can also access this from outside `Person` as `$Person::count`

Private class data

- What if you want to prevent access to `$count` from outside `Person`?
- Use a `my` variable

```
{                                          put all Person methods inside these braces
  package Person;
    my $count;                             $count is visible only within the braces
    sub new {
      my $class = shift;
      my $self = { @_ };
      $count++;
      bless $self, $class;
    }
    sub get_count { $count }                ignore the class argument
 }
```

- Now there is no way to access `$count` directly from outside its scope (the braces)

Initializing private class variables

- If you write class `Person` as a module (probably a good idea in the long run), you don't need the braces—the file provides a scope for the `my` variables
- To initialize the `my` variable to something other than its default of `undef`, use a `BEGIN` block

```
BEGIN {                                    code in here is executed at compile time
   package Person;
   my $count = 1000;                       initialize $count to 1000 before new is called
   sub new { ...                                                  rest same as above
}
```

- Methods inside the `BEGIN` block have access to `$count` just as before, but now it is initialized to `1000` at compile time
- If `Person` is a module, its code is automatically in a `BEGIN` block if you `use` it

Inheriting object data

- Perl doesn't have built-in object data inheritance
- Generally not a problem
- Perl objects represented as hashes do not have predeclared members or sizes, so there is no need for a "class" structure to be declared
- You can't inherit a structure that doesn't exist!
- Also, Perl constructors are "thin" and inheritable
- A single constructor can serve the needs of a whole family of derived classes
- As long as you are using something like the generic, inheritable constructor you don't have to worry about data inheritance

```
sub new { my $pkg = shift; bless { $_ }, $pkg; }
```

- This constructor doesn't presuppose anything about the number or kind of arguments
- Nor does it initialize anything in a class-specific way

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Inheriting object data, without initialization

- Here's an example

```
package Graphic;
sub new { my $pkg = shift; bless { @_ }, $pkg; }

package Text;
@ISA = qw(Graphic);

package main;
my $g = Graphic->new(pen => 2, color => 'blue');
my $t = Text->new(color => 'blue', font => 'courier');
```

- `Text` inherits `Graphic`'s constructor
- This is fine—the constructor can create objects for any derived class
- —Because the constructor presumes nothing about the layout of the object's data

Inheriting object data, with initialization

- Suppose we alter `Graphic`'s constructor to provide default values for some members

```
package Graphic;
sub new {
  my $pkg = shift;
  bless { pen => 1, color => 'black', @_ }, $pkg;
}
```

- We may want to do the same thing for `Text`, perhaps define a default font of `'Times'`
- `Text` now needs its own constructor, but it needs to call `Graphic`'s to pick up its defaults too
- Here is a *wrong* way to do it

```
package Text; @ISA = qw(Graphic);
sub new {
  my $pkg = shift;
  Graphic->new(font => 'Times', size => 12, @_);
}
```

what class does this return?

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Inheriting object data, with initialization (cont'd)

- If we call `Graphic`'s constructor directly, it blesses the data into `Graphic`, not `Text`
- We could re-bless it in `Text::new`, or call the constructor as a subroutine, but there's a better way
- Use the `SUPER::` pseudo-class

```
package Text;
@ISA = qw(Graphic);
sub new {
  my $pkg = shift;
  $pkg->SUPER::new(font => 'Times', size => 12, @_);
}
```

- `SUPER::new` refers to the new method next up the inheritance hierarchy—`Graphic::new` in this case
- Using `$pkg` on the left of the arrow gets the right class (`Text`, or perhaps a subclass of `Text`) passed into `Graphic::new`
- The result is blessed into `$pkg`, and you have just a ... super new object

Controlling object data with %FIELDS

- On the one hand, the generic, inheritable constructor is easy to remember, write, and use
- On the other hand, it does nothing to ensure that objects aren't created with completely bogus contents
- We'll come back to this later, but for now, let's look at one way to limit what can be put in an object
- (It's not going to be pretty)
- Recent version of Perl have a special package variable called %FIELDS
- If present, %FIELDS defines a mapping of field names to array indices
  `BEGIN { %FIELDS = qw(pen 1 color 2) }`
- If the %FIELDS hash exists you can use hash syntax on some array references, and the hash operations are translated into array accesses
- Works for variables declared via the "typed my" syntax
  `my Graphic $typed_g = [];` *now* `$g->{pen}` *means* `$g->[$FIELDS{pen}]`, *or* `$g->[1]`
- You can also use a pseudo-hash ...

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Controlling object data with %FIELDS (cont'd)

- A pseudo-hash is a special kind of array reference

```
my $pseudo_g = [ {pen => 1, color => 2}, 2, 'blue' ];      pseudo-hash
my $g = { pen => 2, color => 'blue' };                     equivalent hash
```

- The first element in the array is a hash ref that maps field names to array indices (like %FIELDS)
- In fact, it makes sense to put a reference to %FIELDS there

```
my $pseudo_g = [ \%FIELDS, 2, 'blue' ];
```

- Why is %FIELDS useful?
- The mapping from field name to array index is hardwired
- Attempting to access an unspecified field yields a fatal error

```
$g->{beauty} = 'moderate';                    no error
$typed_g->{beauty} = 'high';                  fatal compile-time error
$pseudo_g->{beauty} = 'higher';               fatal run-time error
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Controlling object data with %FIELDS (cont'd)

- The easy way to set up %FIELDS is with the fields pragma

```
package Graphic;
use fields qw(pen color);
```

more or less like the BEGIN *block above*

- Now, this might be useful if we could only figure out how to make it work with objects
- To do that, we need to create a constructor that returns an *array* ref, not a hash ref

```
sub new { my $pkg = shift; bless [], $pkg }
```

- Now we can create and use objects with the typed my syntax

```
my Graphic $g = new Graphic;
$g->{pen} = 2;
$g->{beauty} = 'fair';
```

fatal compile-time error

- But this doesn't work without typed my

```
my $g = new Graphic;
$g->{pen} = 2;
```

error: Can't coerce array into hash

- Nor does this overly simple-minded constructor let us pass in initial values

Controlling object data with %FIELDS (cont'd)

- It begins to get hairy! We have to write the constructor so that it returns a pseudo-hash

```
sub new {
  no strict 'refs';                                  # turn off the security cameras
  my $pkg = shift;
  my $self = bless [ \%{"$pkg\::FIELDS"} ], $pkg;    # can't use typed my—inheritable
  @$self{keys %$_} = values %$_ for { @_ };          # fancy slice initialization
  $self;
}
```

- Goodness.
- Making this inheritable is what makes it a little ugly
- We can't just say %FIELDS because we may need to use %FIELDS from a derived class (coming soon), thus we use the symbolic reference to get at $pkg::FIELDS
- Copying the arguments in @_ into the pseudo-hash after it has been created gives us checking
- We can add defaults, too—{ %defaults, @_ }

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

%FIELDS and inheritance

- Derived classes need their own %FIELDS, which should be a superset of that in the base class
- The base pragma makes this straightforward

```
package Text;
use base qw(Graphic);
```

- We can add some more fields

```
use fields qw(font size);
```

- We can inherit the existing constructor, or, if need be, define a new one and invoke the base constructor from it

```
%defaults = (font => 'Times', size => 12);
sub new {
  my $pkg = shift;
  $pkg->SUPER::new(%defaults, @_);
}
```

More thoughts about controlling object data

- Certainly there are other ways to control object data
- You can have your constructors take positional arguments, not named arguments, and do some checking on them

```
package Graphic;
my %colors = map { $_ => 1 } qw(white black red green blue);
sub new { # pen, color
  my $pkg = shift;
  die "Graphic constructor takes two arguments" unless @_ == 2;
  my ($pen, $color) = @_;
  die "invalid color '$color'" unless $colors{$color};
  bless { pen => $pen, color => $color }, $pkg;
}
package main;
my $g = new Graphic(5, 'purple');
```

dies with invalid color

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Still more thoughts about object data

- But is it worth it?
- We've been looking at the "cage" view—what about the "contract" view?
- You can just write simple constructors and document the way they are supposed to be used
- If they're called incorrectly, run-time errors will likely result
- But hey—Perl is so loosely typed that run-time errors are what *most other techniques give you, too*
- It is not possible to move all object construction and access checks to run time
- `%FIELDS` and other techniques do some compile-time checks, but mostly build run-time assertions into your code
- You might be better off just adding assertions of your own here and there
- The simpler and more direct your infrastructure is, the easier it is to change as your program and requirements evolve
- Heavy-duty infrastructures require heavy-duty maintenance

Another take on inherited object data

- So far, we have assumed that at least some users of object data will be getting at it directly, through the object hash

```
$obj->{field} = $some_value;
```

- Although the contents of the object hash might not be considered fit for public consumption, your design might consider it appropriate for member functions, at least, to manipulate the hash directly
- This is certainly the simplest approach
- And there's always something to be said for simplicity
- However, if your design makes extensive use of inheritance, and in particular must deal with inherited data in a fairly formal or rigid architecture, you may want something different
- Although Perl doesn't have facilities for data inheritance, it does handle method inheritance well
- Many object-oriented architectures funnel access to all object data through "get" and "set" methods
- Expressing data inheritance in terms of methods works well in Perl

Accessor methods

- You don’t have to be sophisticated when coding methods to get and set object data—also called accessor methods

```
sub get_pen { shift->{pen} }
sub set_pen { my $self = shift; $self->{pen} = shift }
my $g = Graphic->new;
$g->set_pen(2);
```

- A common refinement is to use the same method to get and set

```
sub pen { my $self = shift; @_ ? ($self->{pen} = shift) : $self->{pen} }
my $g = Graphic->new;
$g->pen(2);
print "g->pen is ", $g->pen, "\n";
```

- All these methods are inheritable
- Of course, this way you need to code a separate method for each piece of object data

Systematically creating accessor methods

- Let's suppose that you have a list of the names of your member data
- Instead of coding each of the "accessor" functions individually, let's produce them all at once in a loop

```
my @fields = qw(pen color);
for my $field (@fields) {
  no strict 'refs';
  *$field = sub {
    my $self = shift;
    @_ ? ($self->{$field} = shift) : $self->{$field};
  }
}
```

`$field` ***has*** *to be a* `my` *variable*
`*$field` *on the next line is a symbolic ref*
create anonymous subroutine and install it
body of the subroutine basically the same as before

- This loops through the names of member data (`pen` and `color` in this case) and for each one creates an anonymous subroutine that can get and set the appropriate piece of data
- Each subroutine is then installed into the symbol table by assigning it to a glob

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Accessor methods and inheritance

- You can use `keys %FIELDS` as the source of your member names
  ```
  for my $field (keys %FIELDS) { ...
  ```
- However, each `%FIELDS` has to contain the names of inherited members as well as the ones new to the current class
- Is this a problem? Not necessarily
- You won't be inheriting accessor methods from parent classes — you'll be replicating them
- You could add another level of indirection to allow you to create only "new" accessor methods
  ```
  package Text;
  use constant FIELDS => qw(font size);
  use base qw(Graphic); use fields FIELDS;
  ...
  for my $field (FIELDS) { ...
  ```
- You'll probably want to find a way to reuse the accessor-generating code, too

Class data, again

- We can also create accessor methods for class data
- This is good, because you can't really inherit class data without using accessor methods
- We can take the simple approach

```
package Graphic;
my $min_pen_size = .1;
my $max_pen_size = 10;
...
sub min_pen_size {
  shift;                                          ignore class argument
  @_ ? ($min_pen_size = shift) : $min_pen_size;
}
print "min_pen_size is ", Graphic->min_pen_size, "\n";
```

- We can inherit this method, allowing subclasses to access the contents of `$min_pen_size` just as readily as the base class

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Creating class data accessor methods

- If you put class data into a single hash (not a bad idea in general), you can generate class data accessor methods much the same as for member data

```
package Graphic;
my %Graphic = (min_pen_size => .1, max_pen_size => 10);
for my $field (keys %Graphic) {
  no strict 'refs';
  *$field = sub {
    my $pkg = shift;  # just ignore
    @_ ? ($Graphic{$field} = shift) : $Graphic{$field};
  };
 }
```

- `%Graphic` is a hash containing all the class data—the "Eponymous Meta-Object" if you read `perltootc`
- One way or another, you will have to resort to methods to inherit class data

Thoughts on class data

- The (new) `perltootc` man page describes techniques like these that you can use to deal with class data, and deals with it in more detail and generality
- If you have a small class hierarchy with a few pieces of class data, don't pull out the heavy machinery—just code plain old accessor methods and be done with it
- For more complex projects, you will probably want more help dealing with your data
- (Perhaps someone will put some of the `perltootc` constructs into an easy-to-use module)

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Private object data

- Hashes make good objects, but aren't helpful at all when it comes to hiding object data from the outside world
- You can readily see what's in any hash-based object

```
my $obj = new SuperSecretObject;
print "guts are ", map("$_=$obj->{$_} ", keys %$obj), "\n";
```

- And, of course, you can change it just as readily

```
$obj->{precious_secret_value} = "hello, world!";
```

- If you believe in contracts, you'll trust your class's clients to leave the innards of your objects alone
- Or you can try to hide your object's contents
- Hiding object data is awkward, but possible ... sort of

Private object data

- Obviously you can't hide object data if it's in a hash—you have to hide it elsewhere
- One way to do this is to keep an auxiliary hash in the class
- For each object, there's an entry in the hash that points to that object's private data
- Suppose that we want to maintain some private data for each `Graphic` object created
- We will use a hash called `PRIVATE`, whose keys are stringified object references, and whose values are references to the corresponding private data (in this example, hash refs)
- The hash will be class data—visible to the class methods but not to the rest of the program
- A stringified reference is one that has been converted to string form—looks like `Graphic=HASH(0xc2064)`
- A stringified reference is guaranteed to be unique as long as the associated object remains in existence, thus it makes an acceptable hash key

Private object data

- Here's how we might implement it

```
package Graphic;
my %PRIVATE;
sub new {
  my $pkg = shift;
  my $self = bless { @_ }, $pkg;
  $PRIVATE{$self}{born} = time;
  $self;
}
sub get_time {
  my $self = shift;
  $PRIVATE{$self}{born};
}
```

$PRIVATE{'Graphic=HASH(0xc2064)'}{born}

using `$self` *as a string again*

- The contents of `%PRIVATE` are completely inaccessible outside its scope, and thus the private object data is too

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Private object data (cont'd)

- We should also add a destructor to remove an object's entry from the hash upon destruction
  ```
  sub DESTROY { delete $PRIVATE{shift} }
  ```
- Of course, you can just nevermind all this
- —Put all your data in the object's hash, document what is supposed to be known, and let your users suffer appropriately if they tinker with its innards
- Along these lines, you will sometimes see "private" members prefixed with _
  ```
  $g->{font} = 'Times';
  $g->{_born} += 10;
  ```
- Again, the usual tradeoffs
- Simple projects should employ simple coding practices
- More elaborate projects will require more elaborate practices
- There is always more than one way to do it

Non-hash objects

- Almost always, objects are hashes—almost always, this is the right thing
- However, objects can be any Perl data type that you can create a reference to

```
$array_obj = bless [@data], $class;
$scalar_obj = bless do { \(my $scalar = $val) }, $class;
```

$array_obj is a blessed array ref

- That's any type

```
sub Code::new {
  my ($class, $val) = @_;
  bless sub { @_ ? ($val = shift) : $val }, $class;
}
my $num = Code->new(10);
print "num is ", $num->(), "\n";
$num->(20);
print "num is ", $num->(), "\n";
```

each sub gets its own `$val`
returns a blessed subroutine ref

prints `num is 10`

prints `num is 20`

More non-hash objects

- I mean any type

```
my $globject = bless do { \local *sym }, 'Foo';
print "$globject\n";
```

prints `Foo=GLOB(0xcc37c)`

- Any type at all

```
my $iobject = bless do { *STDOUT{IO} }, 'Bar';
print "$iobject\n";
```

iorefs start out blessed into `IO::Handle`*, though—might want to leave 'em alone*

- Even very strange things can be blessed

```
$baz = "Testing one two\n";
my $lvobject = bless do { \substr($baz, 8, 3) }, 'Baz';
print "$lvobject\n";
$$lvobject = "ONE";
print $baz;
```

ACK! GAG!

prints `Baz=LVALUE(0xcc3d0)`

want to see what an `LVALUE` *does, blessed or otherwise?*

`Testing ONE two`

- You can't bless a `qr//`, though—it's hardwired into `Regex`

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

The (non?-)utility of non-hash objects

- Of the non-hash types, blessed scalars and globrefs are probably the most useful
- Scalars work for single values (obviously); blessed globrefs can hold one of everything, so surely they're good for something
- Blessed coderefs might seem to give you a way to create private member data, because data hidden inside closures is definitely private

```
sub Code2::new {
  my ($class, $a, $b) = @_;
  bless sub { ($a, $b) }, $class;
}
```

each sub gets its own untouchable `$a` *and* `$b`

- In this example, each anonymous sub's `$a` and `$b` are set in the constructor
- They're private, all right—too private!
- Only the anonymous sub above has access to `$a` and `$b`—ordinary member functions don't!
- Therefore you can't write a member function capable of changing `$a` and `$b`

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

More about those pesky coderef objects

- You might try rewriting the object subroutine so that you can change the values of its bound data through it—the `perltoot` man page has one of these
- But the data is no longer hidden, because if a member function can change it through the object subroutine's interface, so can any other subroutine in the program!
- You might try to implement "access control" in the subroutine with `caller` to ensure that only certain subroutines can change the data, but that's imperfect in a language where a client can define (or redefine) a subroutine in any package
- So we're forced to consider awkward schemes like the following (it's *deja vu* all over again)

```
BEGIN {
  my %PRIVATE = {};
  sub Code2::set_a {
    my ($self, $new_a) = @_;
    ${$PRIVATE{$self}} = $new_a;
  }
```

`%PRIVATE` *can be seen only within the* `BEGIN` *block*

`$self` *is the coderef, gets stringified as a hash key look up and change* `$self`*'s* `$a`

More about those pesky coderef objects (cont'd)

```
  sub Code2::new {
    my ($class, $a, $b) = @_;
    my $self = bless sub { ($a, $b) }, $class;          same as before
    $PRIVATE{$self} = \$a;              save a pointer to this $a in %PRIVATE
    $self;                                        return coderef, same as before
  }
  sub Code2::DESTROY { delete $PRIVATE{shift}; }          clean up %PRIVATE
}
my $o = Code2->new(10, 20);
print "o is ", join ' ', $o->(), "\n";                  prints o is 10 20
$o->set_a(30);
print "o is ", join ' ', $o->(), "\n";                  prints o is 30 20
```

- Works great—but it's inefficient and awkward, and no better than simpler non coderef schemes
- Whatever utility coderef objects have, it isn't in the area of hiding member data—at least not data that can be shared readily with member functions

In summary

- This could go on a while
- If you're into this sort of thing, it's easy to get sucked into trying to create the "perfect" framework for your next object-oriented design
- Surely there must be one more cool thing that can be done with closures ...
- Surely there must be a different way to handle class data ...
- Is it good or bad that Perl doesn't constrain you in your efforts?
- Balance the complexity of your solution against the complexity of your problem
- Understand the difference between "the right way" and "a right way"
- There's more than one way to do it!

Patterns

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

What are patterns?

- “Patterns” is the term popularized by the authors of Design Patterns to describe characteristic architectures present in reusable code
- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”—Christopher Alexander, quoted in the Introduction
- Patterns are, of course, language-independent, although individual languages have more affinity for some patterns, and less than others
- Recently, I took a look at how some design patterns could be rendered in Perl
- Curiously, many patterns have very succinct expression in Perl
- One wonders if this is because Larry designed Perl to have an affinity for solution architectures
- This isn’t intended to be an exhaustive tour of patterns in Perl ... no time for that ...

PROTOTYPE

- The essence of the Prototype pattern is that objects are created by cloning prototypical objects
- Cloning (shallow copying of objects) is easily expressed in Perl
- You can add a cloning interface to classes via a mixin class

```
package Prototype;
sub clone {
  my $self = shift;
  my $cloned_self = eval {
    bless { %$self }, ref $self
  };
  die "can't clone: $@" if $@;
  $cloned_self;
}
```

`Prototype` *has no constructor—not meant to be instantiated*

`eval` *is handy for catching errors—good messages*

- This assumes that `$self` is a blessed hash ref—could be rewritten to accomodate more types

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

PROTOTYPE (cont'd)

- Classes that derive from Prototype can now be cloned

```
package Prototyped;
@ISA = qw(Prototype);
sub new {
  my $class = shift;
  bless { @_ }, $class;
}
```

- Here is some (ugly) code showing an object being cloned

```
package main;
my $a = Prototyped->new(a => 1, b => 2, c => 3);
my $b = $a->clone();
$b->{d} = '4';
print map("($_ => $self->{$_}) ", sort keys %$a), "\n";
print map("($_ => $self->{$_}) ", sort keys %$b), "\n";
```

PROTOTYPE (cont'd)

- Implementing cloning in Perl is straightforward because Perl objects have an exposed architecture (at least when you use hashes as objects)
- An object can be cloned through code that knows nothing about the object's or its class's architecture
- It could even be cloned by an ordinary subroutine
- Deep copying can also be implemented without an understanding of class architecture
- A simple deep copy can be implemented with `Data::Dumper`

```
use Data::Dumper;
sub deep_copy {
  local $x;
  eval Data::Dumper->new([shift], ['x'])->Purity(1)->Deepcopy(1)->Dump;
  die $@ if $@;
  $x;
}
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

FACTORY METHOD

- The Factory Method pattern describes a class or object that is capable of instantiating objects of more than one class.
- Suppose your application has to write out a hierarchy of objects to a stream and later read them back in
- You might write a method called `construct_from_stream` that takes a lump of data from the stream and instantiates an object of the appropriate class from it
- `construct_from_stream` is a sort of “virtual constructor” that is capable of creating an object of one of several classes depending upon the data it reads.
- This kind of capability is hard to express in statically-typed languages, but in Perl it’s easy!

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

FACTORY METHOD—generic, inheritable constructor

- The generic, inheritable constructor is *already* a factory method

```
package GenericObject;
sub new {
  my $class = shift;
  bless { @_ }, $class;
}
```

- This constructor can create objects of arbitrary class

```
@ClassA::ISA = qw(GenericObject);
@ClassB::ISA = qw(GenericObject);
$a = ClassA->new(a => 1, b => 2);
$b = ClassB->new(a => 1, b => 2, c => 3);
```

FACTORY METHOD—dispatching

- In some programming languages, the factory method must dispatch to subclasses in some manner

```
package Streamable;
sub construct_from_stream {
  shift;                                          ignore class
  my $stream = shift;                             looking for a globref, ioref, or something
  my $class;
  chomp($class = <$stream>);                      read class name from the stream
  if ($class eq 'Line') {                         dispatch to the appropriate class
    Line->construct_from_stream($stream);
  } elsif ($class eq 'Rectangle') {
    Rectangle->construct_from_stream($stream);
  } else {
    die "can't construct"
  }
}
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

FACTORY METHOD—dispatching

- Or, more generally:

```
package Streamable;
sub construct_from_stream {
  shift;
  my $stream = shift;
  my $class;
  chomp($class = <$stream>);
  $class->construct_from_stream($stream);          dispatch directly through class name
}
```

- In Perl, it's possible to write a single factory method for an entire hierarchy—again, because objects can be created and blessed anywhere, not just within a constructor

FACTORY METHOD—creating objects ad hoc

```
package Streamable;                          although this doesn't even have to be a class method
sub construct_from_stream {
  shift;                                                              ignore class
  my $stream = shift;                        looking for a globref, ioref, or something
  my $class;
  chomp($class = <$stream>);                                          read class name
  my $self = bless {}, $class;                                        create empty object
  while (<$stream>) {
    last unless /\S/;                                                 look for empty line
    /(\S+)\s+(.*)/;                          match space-separated key-value pair
    $self->{$1} = $2;                                                 add to the object hash
  }
  $self;                                                              return newly constructed object
}
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

FACTORY METHOD—constructing from a stream

- To construct objects with this method, you need some data, for example

```
Foo
a 1
b 2
c 3

Bar
a 1
b 2
```

- Now you can read and construct

```
open OBJECTDATA, "<objectdata" or die $!;
my $foo = Streamable->construct_from_stream(*OBJECTDATA{IO});
my $bar = Streamable->construct_from_stream(*OBJECTDATA{IO});
```

FACTORY METHOD—constructing from a stream

- Note that this capability can be expressed as a mixin class
- Let's go ahead and add a `write_to_stream` method

```
package Streamable;
sub write_to_stream {
  my ($self, $stream) = @_;
  print $stream ref($self), "\n";
  print $stream map("$_ $self->{$_}\n", sort keys %$self), "\n";
}
```

- We can make any simple class (one with scalar members) streamable now

```
package Foo;
@ISA = qw(Streamable);                                  # add the mixin class
sub new { my $class = shift; bless { @_ }, $class; }    # boring old class otherwise
```

FACTORY METHOD—constructing from a stream

- Use it like this

```
package main;
my $foo = Foo->new(a => 1, b => 2, c => 3);
open FOO, ">foodata" or die $!;
$foo->write_to_stream(*FOO{IO});
# ...
open FOO, "<foodata" or die $!;
$foo = Streamable->construct_from_stream(*FOO{IO});
```

- Of course, `Data::Dumper` already does this a thousand times better
- What this example demonstrates, though, is that Perl can instantiate objects of any class, “anytime, anywhere”
- This is an extremely useful capability, especially if the alternative is having to add individual member functions to every class in a hierarchy because you can’t construct, typecast, or whatever else “on the fly”

STATE

- The State pattern allows an object to change its behavior when its internal state changes
- The object appears to change its class
- There are many ways to implement state machines in Perl
- The technique shown here takes somewhat after the example in Design Patterns
- You might also use tables, or use hashes of code refs
- This technique is suitable for more verbose applications, where it is an advantage to implement your state machine with more “normal-looking” code
- Hashes of code refs, on the other hand, provide a more uniform but less readable appearance

STATE—architecture

- This example implements a state machine that describes the life of a bored Perl instructor stuck at home
- There are two distinct class entities in the implementation
- One is the “instructor” class, called `AtHome`
- The other is a hierarchy of states, derived from an abstract class called `State`
- Each state—`State_sleeping`, `State_websurfing`, `State_eating`—is a subclass of `State`, and is represented by a singleton instance
- `AtHome` has a `State` object as a member, which will be one of the various singletons—can be accessed through the `AtHome` object’s state method
- The `AtHome` object changes state by calling the handle method with an event (a string in this case)
- The `AtHome` object’s handle method dispatches the event to its `State` object, which does whatever it does and returns the resulting `State` (might be the same)

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

STATE—implementation

- First, the AtHome class

```
package AtHome;
sub new {
  my $pkg = shift;
  bless { state => shift }, $pkg;
}
sub handle {
  my ($self, $event) = @_;
  $self->{state} = $self->{state}->handle($event);
}
sub state { shift->{state} }
```

- The constructor takes a single argument, which is the initial state (one of the `State` singletons)
- Events are handled by the `handle` method, and the current state can be found through the `state` method

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

STATE—implementation

- Next, the abstract State class

```
package State;
sub new { bless {}, shift }
sub handle { shift }
my %instances;
sub instance {
  my $class = shift;
  $instances{$class} ||= $class->new;
}
```

inheritable, creates a blessed empty hash ref

by default, return self

- The `State` constructor creates an empty blessed hash
- All derived classes use this constructor—`State` objects have no internal data
- `State` defines a class method called `instance`
- This is used to obtain singleton instances of each of the derived classes—each singleton is filed away under its class name in the `%instances` hash

STATE—implementation

- Finally, the State subclasses (excuse the tight fit)

```
package State_sleeping; @ISA = qw(State);
sub handle {
  my ($self, $event) = @_;
  if ($event eq 'rested') { State_websurfing->instance }
  elsif ($event eq 'hungry') { State_eating->instance }
  else { $self }
}
package State_websurfing; @ISA = qw(State);
sub handle {
  my ($self, $event) = @_;
  if ($event eq 'hungry') { State_eating->instance }
  elsif ($event eq 'sleepy') { State_sleeping->instance }
  else { $self }
}
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

STATE—implementation

- One more

```
package State_eating;
@ISA = qw(State);
sub handle {
  my ($self, $event) = @_;
  if ($event eq 'full') { State_websurfing->instance }
  else { $self }
}
```

- The various `State` subclasses respond to events by returning a new state, or sometimes by doing nothing and returning `$self`

STATE—a demonstration

- We can now set up and run the state machine—start in `State_sleeping` and take it from there

```
package main;
my $joseph = new AtHome(State_sleeping->instance);
for (;;) {
  print "you're in state ", ref($joseph->state), "\n";
  print "event? [rested hungry sleepy full quit] ";
  chomp(my $event = <STDIN>);
  last if $event =~ /^quit/i;
  $joseph->handle(lc $event);
}
```

- When running, looks like this:

```
you're in state State_sleeping
event? [rested hungry sleepy full quit] rested
you're in state State_websurfing
event? [rested hungry sleepy full quit]
```

STATE—comments

- There are many things you could add
- More methods could be delegated to the `AtHome` object's `State`
- For example, `AtHome` could have a whereis method, which each subclass of State would respond to differently
- This gives the appearance of the `AtHome` object changing its class—really, it's the aggregated `State` object that is changing
- You could also add more states
- Notice that it's pretty easy to add another state to this design—just add another subclass and modify `handle` methods as necessary
- The example also illustrates a convenient technique for managing singleton instances for an entire class hierarchy (hey, that's the Singleton pattern!)

INTERPRETER

- The Interpreter pattern takes a representation of an abstract syntax tree and performs actions on its contents according to the language of the syntax tree
- Our Interpreter example will take an algebraic expression represented as a hierarchy of binary expressions, unary expressions, and numeric values, and evaluate the expression
- It will also turn the expression into a string (like `"(2 + 3)"`)
- The Interpreter pattern isn't responsible for parsing—we begin with the syntax tree already in existence
- In the Interpreter pattern, each grammar rule is represented by a class
- Instance variables correspond to the symbols on each rule's right-hand side

```
Expr ::= Expr_Binary | Expr_Unary | Expr_Numeric
Expr_Binary ::= Expr binop Expr
Expr_Unary ::= unop Expr
Expr_Numeric ::= value
```

nevermind about precedence

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

INTERPRETER-architecture

- In our example, we define an abstract class `Expr` and three subclasses, `Expr_Binary`, `Expr_Unary`, and `Expr_Numeric`
- `Expr_Binary` contains an operator (`+`, `-`, `*`, or `/`) and two other `Expr`s
- `Expr_Unary` contains an operator (`+` or `-`) and one `Expr`
- `Expr_Numeric` contains a numeric value
- For each subclass of `Expr`, we define methods `eval` (to evaluate) and `deparse` (to turn into a string representation)
- `Expr` has the by-now-familiar simple base class look

```
package Expr;
sub new { my $pkg = shift; bless { @_ }, $pkg }
```

INTERPRETER-implementation

- `Expr_Binary` is the most complex of the classes

```
package Expr_Binary;
@ISA = qw(Expr);
sub eval {                          evaluate $self by evaluating child nodes and operating on them
  my $self = shift;
  if ($self->{op} eq '+') { $self->{op1}->eval + $self->{op2}->eval }
  elsif ($self->{op} eq '-') { $self->{op1}->eval - $self->{op2}->eval }
  elsif ($self->{op} eq '*') { $self->{op1}->eval * $self->{op2}->eval }
  elsif ($self->{op} eq '/') { $self->{op1}->eval / $self->{op2}->eval }
}
sub deparse {
  my $self = shift;                 deparse $self by deparsing child nodes and joining with op
  '(' . $self->{op1}->deparse . " $self->{op} " . $self->{op2}->deparse . ')';
}
```

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

INTERPRETER-implementation

- Here are Expr_Unary and Expr_Numeric (excuse the squeeze) ...

```
package Expr_Unary;
@ISA = qw(Expr);
sub eval {
  my $self = shift;                          evaluate child node, then negate if op is -
  if ($self->{op} eq '-') { -$self->{op1}->eval }
  elsif ($self->{op} eq '+') { $self->{op1}->eval }
}
sub deparse { my $self = shift; $self->{op} . $self->{op1}->deparse; }

package Expr_Numeric;
@ISA = qw(Expr);
sub eval { shift->{value} }                  evaluating (and deparsing) just return value
sub deparse { shift->{value} }
```

INTERPRETER-using

- To use the our Interpreter, we need to create a suitable syntax tree

```
my $expr = Expr_Binary->new(
  op1 => Expr_Unary->new(
    op => '-',
    op1 => Expr_Numeric->new(value => 2),
  ),
  op => '+',
  op2 => Expr_Numeric->new(value => 3)
);
```

- Now we can evaluate and deparse it

```
print "evaluates to ", $expr->eval, "\n";
print "deparses to ", $expr->deparse, "\n";
```

prints `evaluates to 1`

prints `deparses to (-2 + 3)`

INTERPRETER—comments

- Most programmers do not write algebraic expression parsers and interpreters
- Yet there are many opportunities to use the Interpreter pattern
- For example, you could think of operations on a tree of graphic objects as the work of an Interpreter (is “drawing” a kind of evaluation or execution? certainly ...)
- Or, more mundanely, you might read in the contents of a configuration file, parse them into a syntax tree, and interpret it

Patterns in summary

- My little “Patterns in Perl” project has been entertaining and also edifying
- One thing that I’ve learned is that Perl is extremely well suited for expressing basic programming idioms and architectures
- It’s not necessarily the best language for illustrating things like sorting algorithms (C is pretty good for that) but you can very easily use Perl to demonstrate higher order structures
- Perl is a kind of universal pseudocode—but your Perl pseudocode actually runs!
- Design Patterns rendered into Perl are extremely concise compared to their equivalents in C++, and even in Smalltalk
- —Patterns recoded in Perl are a fraction of their original size
- Perl has it all over C++ as a language for *expressing* architecture because of the inordinate amount of boilerplate and protocol that shows up in any C++ class design
- Perl beats out many other languages because it is such a *handy* environment —creating and testing and modifying small programs is fast and simple

Summary

Object oriented programming in summary

- People encountering Perl as their second or third object-oriented programming language are often unimpressed or bewildered
- —"It doesn't seem to be very complicated"
- —"It doesn't seem to have the features I need"
- —"I don't know where to start"
- Because of the slimness of its object-oriented framework, Perl allows you to be agile in your coding
- You don't really have to think out a design before you start if you don't want to—you can prototype parts of it over and over again, in a dozen or a hundred lines of code, and not waste days doing experiments that should take hours
- Once you have a design, you can refine and if necessary restructure it more quickly
- This good characteristic is due to the presence of *essential* features and the lack of *unnecessary* features in Perl's object-oriented model

Object oriented programming in summary

- Perl certainly gives you the tools to create all kinds of complicated object-oriented scaffolding
- —man pages have been written, and books are being written!
- A few things are awkward, but Perl lets you do just about anything you want, one way or another
- The best part is that you get to choose ...
- You can design interfaces with the level of formality *you* choose
- You can structure your program's innards with the level of consistency *you* choose
- You are free to do as little as necessary to make your design work, or as much as you want, and then you can go on to the next one
- So ... go on!