The Eobj Perl environment

Eli Billauer

elib@flextronics.co.il

http://search.cpan.org/author/BILLAUER/

Lecture overview

- Introduction: Me, `Eobj` and OO programming
- `Eobj` classes – how to write and use them
- Properties in `Eobj`
- Special goodies of `Eobj` :
  - The object dumper
  - The global object
  - Destroying objects
  - Constant properties
  - Magic callbacks
  - Property paths
  - Error handling
- Conclusion

About me

- Finished Electrical Engineering at the Technion in 1993
- Freelancer since 2000
- Main interests: Digital communication, signal processing (theory and implementation), MATLAB simulations, VLSI (writes in Verilog).
- Linux & Perl fan mainly because they get the job done

How Eobj came to life

- Object-Oriented for “the People”
- A toolkit for the Perlilog project
- Eobj has proved itself for a certain purpose
- My work with Flextronics Semiconductors lead to developing a source code integration tool
- Perlilog integrates IP cores for ASIC, written in Verilog (see http://www.opencores.org/perlilog/).
- Flextronics’ supported this project warmly
- Perlilog and Eobj are released under GPL
- Download Eobj from CPAN: http://search.cpan.org/author/BILLAUER/

Why Object Oriented programming in Perl?

- Sometimes it’s the natural choice
- Perl has an excellent support of objects and classes
- Elegancy
- Large projects
- Teamwork and code integration (don’t hack my code...)
- Flexibility

Why OO programming in Perl is uncommon

... or: The thing that **DON'T** bother you with `Eobj` ...

- Need to understand references
- Need to understand the `bless()` thing
- Need to understand Perl modules
- Properties are hash entries (scalar)
- Mess with getting `new()` right
- No elegant way to destroy objects

A short `Eobj` script

Now we use the class `myclass.pl`:

```
use Eobj;

inherit('myclass','myclass.pl','root');
init;

$object = myclass->new(name => 'MyObject');
$object->sayit('hello');
```

And then we run the script:

```
$ perl -w trymyclass.pl
I was told to say hello
```

A short `Eobj` class

Let's assume that we have a file named `myclass.pl` which is:

```
sub sayit {
  my $self = shift;
  my $what = shift;
  print "I was told to say $what\n";
}
```

Rules for writing a class

- A bunch of subroutines
- The class file does nothing (no errors) on `perl -w`
- Implicitly `strict vars`
- Careful overriding the `new()` method
- No hassle with OO-related variables allowed
- No global variables
- Package name is unknown and must not be set

Relations between classes

- The subroutines become a class' methods when declared with `inherit()` or `override()`
- `inherit()` – Inherit methods from given class
- `inheritdir()` – Scan a directory for class files. Directory tree becomes class tree.
- `override()` – Override methods of given class, and "steal" its name (!)
- Overriding vs. Extending
- `SUPER::method` calls – as usual
- The `root` class can be overridden
- `underride()` – catch leftover calls

An example with two classes

Now we have a file named `hisclass.pl`:

```
sub sayit {
  my $self = shift;

  $self->SUPER::sayit(@_); # Call original method

  my $what = shift;
  print "And he said $what back\n";
}
```

An example with two classes (cont.)

First we use plain `inherit()`:

```
use Eobj;

inherit('myclass','myclass.pl','root');
inherit('hisclass', 'hisclass.pl','myclass');
init;

$object = hisclass->new(name => 'MyObject');
$object->sayit('hello');
```

Trying it out:

```
$ perl -w tryclasses.pl
I was told to say hello
And he said hello back
```

An example with two classes (cont.)

And the script goes:

```
use Eobj;

inherit('myclass','myclass.pl','root');
override('myclass', 'hisclass.pl'); # Only difference!
init;

$object = myclass->new(name => 'MyObject');
$object->sayit('hello');
```

Trying it out (just the same):

```
$ perl -w tryclasses.pl
I was told to say hello
And he said hello back
```

Every Object has a Name

- The object's name must be unique
- `new()` returns the object's reference, not its name
- The `suggestname()` method
- The `objbyname()` method
- Reason: References can't be hash keys (string translation)
- Good for error messages
- Default name set if not given in `new()` call

```
$name = globalobj->suggestname('MyObject');
$object = myclass->new(name => $name);
# And now we make a useless sanity check:
print "Something is very wrong!\n"
  unless (globalobj->objbyname($name) == $object);
```

Dynamic loading of classes

- The classes are loaded (parsed by the Perl interpreter) only when needed
- `inherit()` and `override()` does not even verify that the class' file exists.
- A rich class library can be declared without delaying execution
- Perl's AUTOLOAD mechanism is used
- When updating a class, try it actively (create an object)
- Special handling of bareword warnings

Three stages in the execution cycle

1. Class tree declarations (`inherit()` and `override()`)
2. Calling `init()`
3. Creating and using objects

- Breaking this order is possible, but will cause bugs in the long run
- `init()` creates the global object
- `init()` looks for `init.pl` in the current directory
- If found, the `user_init` class is declared from it, and the `init()` method is called. (An object is not created – no call to `new()`)
- The user's `init()` method should not declare classes

Properties: `set()` and `get()`

In a nutshell:

```
$object->set('property', 'value');
$value = $object->get('property');
```

- Properties can be scalars, lists or hashes
- Unlike Perl variables, their name doesn't carry their type

Properties: Examples

Writing properties and reading them back...

```
$object->set('myscalar', 'The value');
$scalar = $object->get('myscalar');

$object->set('mylist', 'One', 'Two', 'Three');
@list = $object->get('mylist');

%hash = ('Foo' => 'Bar',
         'Daa' => 'Doo');
$object->set('myhash', %hash);
%the_hash = $object->get('myhash');
```

Rules for using properties

Forget this slide if all you need is in the previous one

- Direct access to the object's referenced hash is unallowed
- Internally, all properties are lists
- `get()` is context sensitive (uses `wantarray`)
- In scalar context, `get()` returns the first item in the list, not the number of elements!

Many ways to say nothing

- If a property is undefined, `get()` returns `undef` in scalar context, and `()` in list context.
- No warning is issued when `get()` reads an undefined property.
- There are plenty of ways to remove ("undef") a property:

```
$test -> set('property', undef);
$test -> set('property');
$test -> set('property', ());
$test -> set('property', (undef));
```

But this will *not* remove the propery, but set it to `(undef, undef)`:

```
$test -> set('property', (undef, undef));
```

`pshift()`, `punshift()` and friends

- `pshift()`, `punshift()`, `ppush()` and `ppop()` behave like their Perl siblings
- `punshift()` and `ppop()` return `undef` when called on empty lists = undefined properties
- For example:

```
$object->set('mylist', 'One', 'Two', 'Three');
print $object->pshift('mylist')."\n";
```

will print `One`

The Object dumper

- A debug tool
- Dumps basic information and the properties of one or all objects
- To dump all objects from the main script:
  `globalobj->objdump;`

The Global Object

- Generated by `Eobj` at `init()` call
- Created with the `global` class which is derived from `root`
- One global object per Perl execution
- Holds “global variables” as properties
- Used in main scripts to run methods of “just some object”
- The `globalobj()` command returns its reference
- The `root` class’ `globalobj()` method does the same

Destroying objects in `Eobj`

- An object is destroyed by calling the `destroy()` method
- Hash will be emptied
- Error message when trying to call a method on a destroyed object
- The native Perl's `DESTROY()` method is never called.
- `destroy()` may be extended. The object is stable when this method is called.
- Before exiting, all objects are `destroy()`ed in reverse order of creation
- On script termination: `survivor()` is called just before `destroy()`.

Constant properties

- Created and assigned value with the `const()` method
- Constant properties must not be changed or a fatal error will be issued
- `const()` can be called again on the property, if the “new value” is “the same”.
- If you know that some property mustn’t change, make it constant. This detects bugs.
- `seteq()` – change the meaning of “the same”

```
$object->const('Creators', 1);
$howmany = $object->get('Creators');
```

Magic callbacks

- Only on constant properties
- Execute a routine when property is assigned a value
- May fire off right away
- Maintain relations between properties
- Anonymous subroutines: Know your scope

Magic callbacks – example

```
use Eobj;
init;

$obj1 = root->new(name => 'One');
$obj2 = root->new(name => 'Two');

$obj1->addmagic('the-property', sub {
   $obj2->const('other-property',
      $obj1->get('the-property') );
                                    } );

$obj1->const('the-property', 'fire me off!');

print $obj2->get('other-property');
```

Initalizing properties with `new()`

- It's convenient to assign values to some properties at object creation
- These properties will be constant properties
- The `name` property must always be assigned a value
- Example:

```
$object = root->new(name => 'MyObject',
                    TheString => 'TheValue',
                    myList => ['One', 'Two', 'Three'],
                    Five => 5);
```

The property path

- Properties may be accessed in a “directory”-like structure
- Useful on the global object to avoid name collisions
- `$object->get('property')` and `$object->get(['property'])` is exactly the same thing.
- A better use: (`MyThings` is the “directory”)

```
globalobj->set(['MyThings', 'TheThing'], 'This');
```

Useful methods

- `who()` – An informal string about the object (error messages...)
- `isobject()` – Does some scalar consist of an object?
- `safewho()` – `who()` on another object (possibly a non-object)
- `prettyval()` – Turn value into a human-readable string
- `linebreak()` – Break (multi-line) string nicely

```
$object = myclass->new(name => 'MyObject');
print $object->who;
```

will print `object 'MyObject'`

Error messages

- `Eobj` has a set of functions to report errors
- Work like `die()` and `warn()`
- Based upon the `Carp` module
- `blow()` is used instead of `die()` when the error is “natural” (failure to open a file etc.)
- `puke()` is used instead of `die()` when the error is unexpected, and hence due to a bug or misuse of the object. A stack trace is given.
- Better than `die()` because the error message can be redirected.

```
puke("sayit called with no argument\n")
  unless (defined $what);
```

Summary

- Anyone can write object oriented in Perl
- No need to think big when the project is small
- No mysterious syntax
- No mysterious variables
- Easy to access properties

Thank you

`Eobj` can be downloaded at
`http://search.cpan.org/author/BILLAUER/`

Questions?

Slides were made with LaTeX, using the `prosper` document class