A Guide to Eobj

Eli Billauer
`elib@flextronics.co.il`

April 21, 2003

This guide applies to version 0.2 (beta release) of Eobj .

Contents

1 Introduction 5

1.1 Eobj in a nutshell . . . 5

1.1.1 What is Eobj ? . . . 5

1.2 About the project . . . 5

1.3 About the author . . . 5

1.4 Acknowledgements . . . 6

1.5 This guide's outline . . . 6

2 Writing main scripts 7

2.1 How it all works . . . 7

2.2 The `init()` call . . . 7

3 Eobj objects 9

3.1 Background . . . 9

3.2 An example . . . 9

3.3 Properties . . . 10

3.3.1 What it looks like . . . 10

3.3.2 The basics . . . 11

3.3.3 Property names . . . 11

3.3.4 Undefs and empty lists . . . 12

3.3.5 More about constant properties . . . 12

3.3.6 "Magic" callbacks . . . 13

3.3.7 The property path . . . 14

3.3.8 Methods for lists . . . 15

3.4 Creating and using objects . . . 15

3.4.1 The formalilties . . . 15

3.4.2 An example . . . 16

3.4.3 Property paths in `new` . . . 16

3.4.4 The global object . . . 17

3.5 The object dumper . . . 17

4 Eobj classes 18

4.1 How this section is organized . . . 18

4.2 Classes and inheritance . . . 18

4.2.1 Source files and classes . . . 18

4.2.2 The phases of the object system . . . 19

4.2.3 Class definition . . . 20

4.2.4 "Normal" methods . . . 20

4.2.5 Methods overriding `new` . . . 22

4.2.6 The autoloading mechanism . . . 22

4.2.7 Eobj objects vs. plain Perl objects . . . 23

4.3 Useful methods . . . 24

4.4 Error reporting . . . 24

4.4.1 Some philosophy . . . 24

4.4.2 The list of functions . . . 25
4.4.3 “Hidden” classes . . . 26
4.5 Summary: How to write classes properly . . . 26

5 Eobj main script API 28
5.1 Exported subroutines . . . 28
5.1.1 The exported subroutine `init()` . . . 28
5.1.2 The exported subroutine `inherit()` . . . 28
5.1.3 The exported subroutine `override()` . . . 30
5.1.4 The exported subroutine `underride()` . . . 31
5.1.5 The exported subroutine `definedclass()` . . . 32
5.1.6 The exported subroutine `globalobj()` . . . 33
5.2 The global variables . . . 33
5.2.1 The variable `$VERSION` . . . 33
5.2.2 The variable `$globalobject` . . . 33
5.2.3 The variable `%classes` . . . 33
5.2.4 The variable `%objects` . . . 34

6 The root class API 35
6.1 Methods . . . 35
6.1.1 The method `new()` . . . 35
6.1.2 The method `set()` . . . 35
6.1.3 The method `get()` . . . 36
6.1.4 The method `const()` . . . 38
6.1.5 The method `globalobj()` . . . 39
6.1.6 The method `who()` . . . 40
6.1.7 The method `safewho()` . . . 40
6.1.8 The method `isobject()` . . . 41
6.1.9 The method `objbyname()` . . . 42
6.1.10 The method `objdump()` . . . 43
6.1.11 The method `suggestname()` . . . 44
6.1.12 The method `addmagic()` . . . 45
6.1.13 The method `seteq()` . . . 47
6.1.14 The method `pshift()` . . . 49
6.1.15 The method `ppop()` . . . 49
6.1.16 The method `punshift()` . . . 50
6.1.17 The method `ppush()` . . . 50
6.1.18 The method `prettyval()` . . . 51
6.1.19 The method `linebreak()` . . . 52
6.2 Properties . . . 53
6.2.1 The property `name` . . . 53

A GNU FDL license **54**
A.0 PREAMBLE . . . 54
A.1 APPLICABILITY AND DEFINITIONS . . . 54
A.2 VERBATIM COPYING . . . 56
A.3 COPYING IN QUANTITY . . . 56
A.4 MODIFICATIONS . . . 57
A.5 COMBINING DOCUMENTS . . . 59
A.6 COLLECTIONS OF DOCUMENTS . . . 59
A.7 AGGREGATION WITH INDEPENDENT WORKS . . . 60
A.8 TRANSLATION . . . 60
A.9 TERMINATION . . . 60
A.10 FUTURE REVISIONS OF THIS LICENSE . . . 61

B To Do **62**
B.1 Core issues . . . 62
B.1.1 AUTOLOAD caching . . . 62
B.2 Complete the half-made . . . 62
B.2.1 The error messages . . . 62
B.3 System management . . . 62
B.3.1 Organizing classes . . . 62
B.3.2 Run options . . . 62
B.4 User Interface . . . 63
B.5 Debug tools . . . 63
B.5.1 Error trace . . . 63
B.5.2 All class loader . . . 63

1 Introduction

1.1 Eobj in a nutshell

1.1.1 What is Eobj ?

Eobj is object oriented programming in Perl for the masses. It's is a wrapper for Perl's native handling of objects, and thus it reliefs the programmer from mastering some rather advanced issues in Perl programming.

Unlike native Perl OO programming, using Eobj is based on plain Perl syntax, and the use of some special functions. Methods are defined very much like subroutines, with no need to know how to wrap them in modules. Eobj comes with one predefined root class, which consists of basic methods, including methods for accessing properties easily.

Aside from being a quick starter, Eobj also includes some special features, especially in the field of inheritance and class load on demand.

1.2 About the project

Eobj is an extraction (actually a rip-off) from a larger project, Perlilog (See `http://www.opencores.org/perlilog/`). The latter project was developed with the support of Flextronics Semiconductors in Israel, with the purpose of making Verilog IP cores easier to integrate.

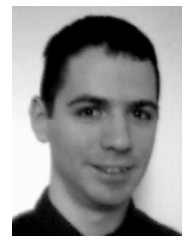
Since I couldn't find any adequate environment for OO programming (and Perlilog definitely needed one), I wrote one for myself. After finishing off Perlilog, it occured to me that the OO environment could be useful for others, so I extracted the it from Perlilog.

This became Eobj .

1.3 About the author

Eli Billauer was born in 1971, in the Israeli city Haifa, where he lives today, and works as a freelancer (recently taken picture to the right).

He received his B. Sc. in Electrical Engineering (Summa Cum Laude) in 1993 at the Technion, Haifa. He spent the six following years as a development engineer in a well-established hi-tec environment. During these years he was fortunate enough to gain experience in various fields of his profession. The main focus was on digital communication and signal processing, with MATLAB and DSP programming as the primary tools, but he was keen to learn any new field, as required. The result was knowledge in diverse subjects, such as RF on one hand, and Internet protocols on the other.



He has one year and a half of experience in managing and leading a project, which had a core of 4 engineers, and several other members for various tasks of shorter terms.

He is a freelancer since year 2000, as which he's taken projects in various fields, such as writing DSP code for GSM Layer I, signal processing of motion-detecting optical sensors, and a Sigma-Delta modulator for audio frequencies.

It was due to the last mentioned project that he learned Verilog.

Eli installed Linux on his home computer in 1998, and has been a Linux fan since. He learned Perl at about this time, originally in order to create a web site.

He is a proud member of Haifux, the Linux club of Haifa.

1.4 Acknowledgements

This project would not exist without the warm support of Flextronics Semiconductors in Israel, and Dan Gunders in particular. Lior Shtram made the connection between me and Flextronics, so I owe him thanks as well.

Erez Volk opened my eyes by introducing me to the wonderful world of GNU and Linux. He is also responsible for my first encounter with Perl.

And Larry Wall created this magic named Perl. We all owe him, I think.

1.5 This guide's outline

If you want to start working with Eobj , you're reading the wrong thing. Eobj comes with a man page, which gives exactly enough to get moving. This guide gets right down to the details.

Section 2 is short, and describes the little there is to know about the main script.

Next, section 3 explains how objects are used in Eobj . This is another stage in understanding what actually happens in the system, and it opens some additional possibilities.

Section 4 then explains how classes should be written.

Sections 5 and on are API summaries. They describe how each individual function or method should be used. It's rich with examples.

2 Writing main scripts

2.1 How it all works

Before beginning, it's important to distinguish between the main script and the class source files.

The main script is what you would usually call "the script": When we run Perl, we give some file as the script to run. That file, possibly with some other Perl modules that it runs, is the main script.

The main script is divided in two phases:

1. Declaration of classes. The main script tells what classes it wants to have, and what source file declares each class' methods.
2. Creating and using objects. This is usually where the main script does something useful: Objects are created, their methods are called, and so on.

We shift from phase 1 to phase 2 with a call to `init()`. All class declarations *must* come before `init()`, and all objects creations *must* come afterwards.

Main scripts feel like regular Perl scripts: In particular, they are by default run under a `no strict` mode (unlike Eobj classes).

2.2 The `init()` call

In all the script examples, we had a call to `init()`. This call must be made after new classes have been added to the system by the main script, but before it starts to create objects. Why is explained in section 4.2.2.

When `init()` is called, Eobj reads the `site_init.pl` file in the `sysclasses` directory. It may seem paradoxal that this file is in this directory, but there are technical reasons for this.

The file is expected to have one single subroutine, `init()`, which is called. This routine performs the following tasks:

- The global object is generated (explained in section 3.4.4).
- If an `init.pl` file exists in the main script's current directory, it is executed.

The really interesting part about this is that a per-project initialization file can be generated, since the user's `init.pl` is looked for in the project home directory (unless the home directory is changed by the main script prior to `init()`).

Both `site_init.pl` and `init.pl` are executed by a call to `init()`, which must be written in a Eobj class style. How to write classes is widely explained in section 4.2, but the following rules should be enough to get a descent `init.pl` file:

- The initialization script is written in a subroutine named `init()`.
- All variables that are defined in the subroutine must be localized with `my`[1]. The subroutine (unlike the main script) is run in a `use strict` mode.
- Calls to Eobj -specific routines must be done with explicit package name, as in

```
&Eobj::inherit('myclass', "myclass.pl", 'interface');
&Eobj::interfaceclass('myclass');
```

[1] If you have to, use complete `$Foo::Bar`-like identifiers for global variables

3 Eobj objects

3.1 Background

Even though the plain Perl environment supports objects as a natural part of its core, the syntax for using objects in Perl is quite cumbersome. This is probably a result of not wanting to make objects a special creature in Perl, but a natural follower of packages, references and hashes. The result is a very powerful and flexible mechanism that supports all the goodies that you could expect from object-oriented programming, and a few more tricks that are beyond the tradition. On the other hand, the path to object-oriented programming includes learning a few advanced topics in Perl, which is probably the reason why few scripts actually incorporate objects in Perl for real-life applications.

Eobj includes an environment for object-oriented programming which is, in fact, a wrapper of the native environment. It makes use of the powerful features that Perl gives in this field to create a very simple syntax to define classes, generate objects, call methods and access properties. Some features were added to support functionalities special for Eobj .

The Eobj object environment frees the programmer from awareness to references and blessed such in particular, packages and the need to know how to write Perl modules. The knowledge needed to use Eobj classes, as well as generating them, includes only plain Perl programming and the acquaintance with some additional functions, that are described in the sequel. Knowledge of general object-oriented programming techniques and concepts are needed as well, of course.

3.2 An example

Before getting into the complicated explanations, let's consider a simple example, which shows the basics of using objects in Eobj .

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";
}
```

This file is used to declare a class that is used in the following script:

```
use Eobj;

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

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

We now explain this script briefly, only to give the general picture:

The first line in the script, `use Eobj;`, makes Perl load the Eobj environment.

Then we have an `inherit` command. This tells the Eobj environment, that a new class, with the name `myclass` should be declared, the methods are defined in `myclass.pl` (listed just above), and that the class should be derived from the Eobj basic class, `root`.

We may now view `myclass.pl`, and see that it consists exactly of a subroutine definition, in well-known Perl syntax. In effect, this subroutine will become the method `sayit` in the `myclass` class. Note that the subroutine's first argument goes to a variable named `$self`, which is a handle to the called object. The argument that is given by the caller will reach the variable `$what`.

We now return to the main script. After the `inherit` declaration, Eobj is called with `init`. As this function implies, it causes the Eobj environment to initialize.

Having done this, we generate a new object of class `myclass`, give it the name `MyObject`, and put its handle (actually reference) in `$object`.

Finally, we call this object's method `sayit`, with the argument `'hello'`. As a result, the text `I was told to say hello` will be printed.

3.3 Properties

3.3.1 What it looks like

Properties are read and written with method calls such as:

```
$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');
```

3.3.2 The basics

The properties in Eobj are divided into two types: Settable and constant. Settable properties are just like any variable: They can be assigned a value at any time, and the value can be changed as often as desired. Settable constants are created with `set()`.

Constant properties, as the name implies, are assigned a value only once. Any attempt to change their value will result in a fatal error. This restriction is useful whenever changing the property's value will violate some basic assumption. For example, the `name` property is constant for good reasons: It is the object's identifier, and as a such, the connection between the name and the object's handle (reference) is recorded in the system. Constant properties are created with `const()`.

The properties are accessed mainly with three basic methods (defined in the `root` class): `get()` for retrieving properties' values, `set()` for setting the values of settable properties, and `const()` to assign values to constant properties. It goes something like:

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

Properties are also assigned values (as constants) when creating a new object using `new`. See section 3.4 for more about this.

The properties in Eobj are in general lists. Scalars are considered as a list with one item. `get` behaves in such a way, that you'll get back exactly what you gave `set` or `const`, no matter if it was a list or a scalar.

Or a bit more detailed: If a property is assigned a scalar, and `get()` is called in list context, you simply get back a list with one element, which is the value assigned. But if a property is assigned a list, and `get()` is called in scalar context, only the first element is returned.

A detailed example is shown in section 6.1.3.

3.3.3 Property names

Property names behave like (and are, in fact) hash keys: They are case-sensitive, and all characters are allowed. It is recommended not to have newlines (`\n`) in the names, since these have a special meaning in the internal structure, and may cause strange property collisions.

For normal, practical purposes, case-sensitive alphanumeric names is the right thing to do. Don't make the names too long either, only as a matter of convenience.

See section 3.3.7 regarding the directory-like structure, in which the properties can be organized.

3.3.4 Undefs and empty lists

Sometimes a property is not set, and sometimes we want to set its value to “nothing”. The Eobj API attempts to return a sensible value, depending on the situation and the context.

If a property has not been set (or has been removed), `get` will return an undefined value (a.k.a. `undef`) if it is called in scalar context, and an empty list if it's called in list context. It is perfectly proper (no warnings) to call `get` on an undefined value.

For example, if we assume that the property `'property'` was not defined when the following code snipped was run,

```
$scalar = $test -> get('property');
@list = $test -> get('property');

$elementscount = $#list + 1;
print "Was not defined and had $elementscount elements\n"
  unless (defined $scalar);
```

it will print “Was not defined and had 0 elements”.

We may also remove a property by assigning it with either an undef value, an empty list, or a list containing only one undef value. The property will be removed, and will for all purposes behave as if it was never assigned a value. Naturally, a constant property can't be removed, but using `const` with an undefined value on an undefined property, will simply have no effect.

Here are a few examples:

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

$test -> set('property', (undef, undef)); # 'property' is set to a value!
```

3.3.5 More about constant properties

Constant properties differ from settable properties in two respects:

- Constant properties can't be changed after they are set

- A callback subroutine may be triggered when the constant is assigned a value.

The second issue of callbacks is described in the next section (sec. 3.3.6). We now focus on the issue of the exact meaning of the value being “constant”.

If a property is not assigned a value, there is, of course, no problem with assigning a such. If the property has been assigned a value with `const`, Eobj allows `const` to be called again for the same property, only if the value to assign the property is “the same”. We shall see what “the same” means below. It may not seem much of a privilege to be allowed to call `const` more than once with the same value, but this turns out to be useful in callback routines.

We now define what we mean with “the same”: We recall that scalars are considered by the system as a list with one item, so we may reduce the entire discussion to lists. Two lists are “the same” if (and only if) they have the same number of elements, and each of the elements in the first list is Perl `eq` with the corresponding element in the second.

The choice of `eq` may not fit all properties. For some properties, a different criterion may be in place, in spirit of what means “the same” for that certain property. In that case, the `seteq` method (see section 6.1.13 for a description and example) may be used to substitute `eq` comparison with some other criterion.

When `const` is called for an already assigned property, it will never affect the existing value, and no magic callbacks will occur, even if the new and existing value are different (this statement is relevant when `seteq` was used to soften the criterions for “the same”).

3.3.6 “Magic” callbacks

“Magic” callbacks are possible only on constant properies. The motivation behind this mechanism is to force some properties to have the same value. These properties may belong to the same object, or to different objects.

Suppose that we want property A and B to have the same value. One way of doing this is to look for all places in the script where A and B are set, and make sure that if one of the properties is set, so is the other one.

Another, safer and more elegant way to accomplish this, is to tell the environment, that we want some piece of script to be executed, as soon as A is assigned a value. This script will then copy the value of A to B. Then we do the opposite thing with B, so that the value of B is copied to A.

A “magic” callback request (done with the `addmagic()` method), is precisely asking Eobj to run some codepiece, as soon as some constant property is assigned a value. Naturally, if the property already has a value, the callback will be fired off immediately.

The technical details of this mechanism are widely described in section 6.1.12.

A few noteworthy points about “magic” callbacks:

- “Magic” callbacks are never executed more than once, due to the nature of constant properties.
- When using “magic” callbacks for its original purpose, the callback script `get`’s the value of A and `const`’s B with that value. And vice versa.
- Usually, a callback will be set for both A and B, so that they are mutually dependent. So if A is assigned a value, B will be assigned a value by virtue of a callback. As a result, B’s callback script will be fired off, resulting in A being `const`’ed again. If all is done properly, A will be re-`const`’ed with the same value, so nothing will actually happen on this second callback.
- If property A and B are tied equal with a pair of callbacks, and B and C are tied in the same way, all three are, in fact, forced to have the same value: As soon as one of them is assigned a value, it will spread to all three by a chain effect.
- “Magic” callbacks may be used to force relations between properties, that are not necessarily equality. Complex relations between values of properties may be forced by writing the script snippets to maintain these relations.
- “Magic” callbacks are not necessarily used to force values on other properties, but can be used whenever we want a notification about some property’s assignment.

3.3.7 The property path

Similar to paths of directories and subdirectories in filesystems, there are paths to the properties in Eobj . When referring to a property by a string (like in `$object -> get('property')`), we relate to a property at root level. We may access properties at a deeper level by something like `$object -> get(['my_things', 'property'])`. In this example, `'my_things'` functions as a “directory”, and `'property'` is a property within that directory.

Property paths may be used in all places, where a property name is expected.

There is nothing special about putting properties into deeper paths, except that it lessens the chances to collide property names. There is no method to get a list of properties within a path[2]. You’re supposed to know what you’re doing.

In fact, the entire path structure can be viewed as a multidimensional array, in which the key consist of a number of scalars rather than one. A property is recognized by the exact sequence in the list, which defines its “name”. Viewing this structure as directory-like is merely a recommendation to have the properties organized better.

We may thus conclude the following facts:

- `$object -> get('property')` and `$object -> get(['property'])` are exactly the same thing.

[2]except for the `objdump()` method, which is a debugging tool. See section 3.5

- It is OK to have a property with the name of what appears to be a “directory”. We may thus call `$object -> set('like_dir', $value1)` and after that `$object -> set(['like_dir', 'like_node'], $value2)` and get two distinct and legal properties. This may be a confusing, but yet legal setting.

Some Perl programmers recognize that the squared brackets, [and], create a reference to a list, so the `\@`-type of reference can also be used, but this is usually not needed – it is enough to know that the path is a list of nodes to walk in the tree, with square brackets instead of round brackets. This list can be arbitrarily long, and it always begins from the root, and describes the nodes on the way to the desired property.

As with simple properties, it's proper to `get` undefined properties in unknown paths. An `undef` or empty list is returned, and no more fuss is made about that.

3.3.8 Methods for lists

Since the properties are considered as lists, there are four methods in the `root` class, which allows some easier access to the values: `pshift`, `punshift`, `ppush` and `ppop`. These methods are described in sections 6.1.14–6.1.17. In general, they behave very similar to their plain-Perl siblings. It is worth to mention, that these methods can't be used on constant properties, since they change the property's value.

3.4 Creating and using objects

3.4.1 The formalilties

Objects are created by a call of the format:
class `-> new(`*Initial properties*`);`
Such a call returns a handle (if you care, a blessed reference to a hash), which is usually kept in a scalar variable for future calls to the object.

When creating a new object in Eobj , the `name` property must always be set, or a fatal error will be issued. The object's name property is its identifier in the system, and must therefore be case-insensitively unique (the `suggestname` method is useful to create unique names, see section 6.1.11).

The initial properties are set by passing a hash, where the keys are the property names, and the values are their values. If we want to set a property to a list, the value of the relevant key in the passed hash should be a reference to a list, containing these values (possibly an anonymous list).

Note that properties that are assigned values with `new` are constant properties. See section 3.3 for more about constant properties and properties in general.

In Eobj as in common Perl, there is nothing really special about the word `new`, except that it happens to be the name of a method, that creates new objects. All classes in the system should create their objects by inheriting the `new` method from the `root` class. Extensions are allowed by using the `SUPER::` prefix on overriding methods of `new` (see section 4.2.5).

3.4.2 An example

While the rules above may appear complicated, this is how it really is:

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

In this example, we create a new object of the `root` class, and we set four properties: `name`, `theKey`, `myList` and `Five`. It is important to note that the name of the properties don't need quotation mark (even though they are allowed), but the properties' values are indeed quoted, or given as any literal value in Perl. Moreover, note the square brackets in giving the value of `myList`. This property will assigned a plain list with the elements given, not a reference to a list.

For sake of clarity, if we continued this section's example with:

```
print $object->get('theKey')."\n";
@list=$object->get('myList');
print join(',', @list)."\n";
```

We would get:

```
TheValue
One,Two,Three
```

3.4.3 Property paths in `new`

In section 3.3.7 we mentioned a directory-like structure of properties. It is possible to initialize properties deeper than the root level with `new` by using references to hashes. Some Perl programmers may recall, that an anonymous reference to a hash can be created by using curly brackets, { and }, so the following example should make it clear:

```
$object = myclass->new(name => 'MyObject',
                       MyDir => {MyKey => 'TheValue',
                                 MyOther => 'TheOther'}
                      );
```

In this case, `$object -> get(['MyDir', 'MyKey'])` returns `TheValue`.

The rule is that if the value is a reference to a hash, then the key becomes a "directory", and the hash is interpreted as pairs of property names and their values, within that "directory". This may be recursively repeated to achieve unlimited depth in the path.

3.4.4 The global object

The global object is useful when:

- We want to keep global information somewhere – the global object's property table is easily accessed.
- We want to run some basic method, but we don't have an object at hand. The global object is derived from the `global` class, which is derived from the `codegen` class, so it supports quite a few methods.

There is a special function to get its handle from the main script, `globalobj`. Within method declarations, `$self -> globalobj` should be used instead. See section 3.5 for an example of using the global object to reach a method.

The Global Object's properties function as the system's global variables. Different classes may add properties as necessary to this object. Care should be taken to avoid name collisions between property names: Use class-specific and unique names in the Global Object.

For example, if objects of some class need to be aware of each other, a property in the Global Object may be a list of handles to objects of that certain class. To do this, each object of that certain class will need to add itself to the list when it's created.

3.5 The object dumper

A nice method of the `root` class, is the object dumper, which is intended for debugging. When called, it prints human-readable information about all or some of the objects. This information consists of the object's name, a brief description, its class and most important, all it properties. The data is presented in a format that is practical for human reading, so information is truncated occasionally where it would take too much space.

See section 6.1.10 for a complete description of this method. But trying

```
globalobj->objdump();
```

at the end of some main script should make things clear.

4 Eobj classes

4.1 How this section is organized

The purpose of this section is to supply the knowledge which is necessary to write code classes.

Section 4.2 is a tutorial on writing Eobj classes in general.

Section 4.4 presents how errors should be reported in from within a class. In a complex system, this may help the user to get a clue of what really went wrong.

A very short section 4.5 offers a checklist summary of the main things to keep in mind when writing a class.

4.2 Classes and inheritance

4.2.1 Source files and classes

In Eobj , a class is defined by telling the system to read a file, and to consider the subroutine definitions in it as the methods of the named class. For example, `inherit('myclass', 'myclass.pl','root')` means to read the source file named `myclass.pl`, and create a new class, `myclass`. Methods are inherited from the `root` class, as specified by the third argument.

Note that there is no necessary connection between the name of the file and the name of the class. Even more important, it is not specified in the file from what class `myclass` should inherit methods. The file is only a list of subroutine (in fact, method) declarations, whose object-oriented context is given by the script.

As is seen in the example, the word `myclass` may then be used as `myclass -> new(...)` in order to generate a new object.

An even more important feature of this special class definition scheme, is the possibility to enrich an existing class with new methods, without changing its name. This is done with the `override` function. This function can be used with either two or three arguments. In the two-argument case, we have something like `override ('myclass', 'otherfile.pl')`, assuming that we have already declared the `myclass` class with `inherit`. As a result of calling `override`, the given file (`otherfile.pl` in this example) will be read. If new methods are encountered, they are simply added to the class. If methods that collide with existing methods are found, those in the given file will override those that were defined before.

Any class in the system, including the `root` class may be enriched with `override`, which opens the ability make changes in the basic classes, without needing to "tell" the other classes, and still have our new code executed by them.

In some cases, the class we want to override may not exist. Since the purpose of overriding an existing class is merely to make sure that some specific methods are supported in this class, it makes sense to create a new class with the same name if the class doesn't exist already. This is the purpose of the three-argument format of `override`: If `override` is called with three arguments, and the class mentioned doesn't exist, `override` behaves exactly like `inherit` with the same arguments. Hence, it declares a new class, with the third argument as the class to inherit from.

If `override` is called with only two arguments, and the overridden class isn't defined, an error is issued.

As can be expected from a proper object-oriented environment, overriding doesn't necessarily mean cancelling out the previous functionality of the overridden method. By using the `SUPER::` prefix, we call the original method, if it exists (this syntax is the common Perl way to reach the overridden method). See section 4.2.3.

See sections 5.1.2 and 5.1.3 for specific information about these two functions, as well as examples of using them.

A third function `underride` works opposite to `override`. Its purpose is to catch calls to methods that are not already defined in the class, or calls via the `SUPER::` prefix within the class. As the name of this function implies, all existing methods in the class will override those that appear in the file given by `underride`.

4.2.2 The phases of the object system

The inheritance relations between classes are set during the execution, and is not predefined in the classes themselves. Setting up the class tree during run-time makes the definition of classes more flexible, but altering the class definition for existing objects could also be an opening for exotic bugs. Because of this (and also due to the way the classes are loaded), there is a clear distinction between three phases in the main script:

- Declaration of classes and their relationships. During this phase, `inherit` and `override` are called in order to set up the class tree. No objects should be generated during this phase.
- A call to `init`. Some internal variables are set up during the execution of this function, and the Global Object is generated.
- Creation and use of objects – during this phase, anything except attempt to alter the class tree is allowed.

In essence, this means that the classes and their position in the inheritance tree are declared before any of them is used. This is enough freedom to allow the system to choose one set of classes over another, or fine-tune the functionality of some methods, depending on run-time parameters. A Eobj script might thus build a different class structure depending on the target device, the synthesis tool or whether some other external tools should be used.

But these manipulations can be done until the first object is created, which happens when `init` is called.

Even though it is possible to stretch these limits a bit without getting error messages, it is highly recommended to comply with this phase structure.

4.2.3 Class definition

All Eobj classes, including the "built-in" classes, are defined by source files, in which the class' methods are defined. These files are perfectly readable as plain Perl files, but special rules apply to make them useful method declarations.

The strongest rule, is that all variables inside the subroutines must be declared "locally", that is, using Perl's `my`. The source files are read in a "use strict" context, so if a variable is mentioned without being explicitly `my`'ed, a fatal error will be issued at the loading of the class.

This is more than a technical rule for programming: It is strongly recommended not to have any global variables other than properties of the Global Object. Even though a global variable in the source file (the package, actually) might be a tempting shortcut, it may cause strange bugs as the Eobj environment plays freely with namespaces. A conservative "local variable" approach ensures steady functionality.

In short, a class defintions consists of method definitions, one after the other, where each method definition is a subroutine definition with no use of Perl global variables.

We now divide the method definitions to two groups: "Normal" methods and methods overriding `new`. Their distinction is in the fact that "normal" methods are called in association with an exisiting object, while methods that override `new` are called in order to create one, so there isn't necessarily any associated object.

4.2.4 "Normal" methods

In the example in section 3.2, a method called `say` was declared and used within an object. The general format of a method declaration is as follows:

```
sub name of method {
  my $self = shift;
  Your code here...
}
```

Writing a method is very much like writing a regular subroutine. The main difference is that a handle (reference) to the called objects is given as the first argument. It is customary to put this handle in the scalar `$self`, usually with `$self = shift`. There is nothing special about the name `$self`, except that it's easier to read the code when it follows this convention. After this `shift` operation, `@_` is the list of arguments, so the rest of the code can be written exactly like a normal subroutine. There are plenty of examples in this guide.

One important choice we have to make for each method we write, is if we want our method to override the functionality of a possibly inherited method, or if it should extend it. In Eobj as in plain Perl, defining a method means that it comes instead of an already defined, possibly inherited method, if it existed. If we want the method to do more than it possibly did before, we call the inherited method with a `SUPER::` prefix.

To demonstrate this, assume that we want the `donothing` method to be defined (in order to avoid an undefined method error), but we don't want to make any changes if it was already defined in the class we inherit methods from. The following method declaration will do the job:

```
sub donothing {
  my $self = shift;
  $self -> SUPER::donothing(@_);
}
```

Note that we explicitly call the inherited method. As opposed to normal method calls, a `SUPER` call does not generate any error if it doesn't exist (that is, if we actually didn't override any method, but declared it for the first time).

It is very important to be careful about the arguments that we pass in the SUPER call. We must always `shift` away the first argument, and thus make the inherited method see exactly the same arguments as ours (the "self" handle will be added again by Perl due to the call). On the other hand, we must be sure not to alter `@_` before making this call, or make a copy of the argument list before changing it (it is very popular to read arguments by using `shift`, which changes `@_`). All this is true, of course, if we don't want to intentionally change the argument list before passing it on.

Since we control the stage in which the inherited method is called, we may perform this call either before or after doing some functionality of our own method. Where it doesn't seem to matter, it is highly recommended to call the inherited method as soon as possible (immeditately after `$self = shift`). This will minimize the chances to generate odd bugs as a result of mistakenly changing `@_`.

See section 5.1.2 for a more extensive example of SUPER calls.

A last remark about writing methods, which goes for any object-oriented programming: In places where you would write a subroutine in normal Perl, write another method in the class, and use it as a method by calling it via the `$self` object. Don't be paranoid about someone else changing the way your class will work. It's a good feature, and there are plenty of ways to screw things up anyhow.

4.2.5 Methods overriding `new`

The `root` class supplies a `new` method, which creates a new object of a given class. This method must never be overridden with any other method, but it may be extended. This is useful to add special properties to any new object, or any other operation that is needed whenever a new object comes to life.

This is done by having a piece of code as follows in the respective class source file:

```
sub new {
  my $this = shift;
  my $self = $this->SUPER::new(@_);
  Your code here...
  return $self;
}
```

Note, that unlike the "normal" methods, we don't get `$self` by `shift`'ing `@_`, but rather by calling the inherited `new`, which generates a new object for us. With this `$self` at hand, we may use it exactly like in any other method. `@_` is also exactly like in a normal method after `shift`'ing off `$this`, but it will be useless in many cases, since it consists of the initial properties, which will be initialized by `root`'s `new` method.

A word about `$this`: In almost all cases, `$this` will hold the name of the class. This is true when `new` is called in the *class* `-> new(...)` format. But it is also allowed to call `new` from an existing object, which will result in a new object of the same class.

For this reason, don't use `$this` as the class' name, but rather `ref($self)`. If you want to know the class' name before calling `new`, use `$class = ref($this) || $this`

The `new()` method usually returns `$self`, so except for in rare cases, this should be done by the extension as well.

4.2.6 The autoloading mechanism

The Eobj environment attempts to read source files as late as possible. In effect, each source file is read when an object, whose methods are based on the class, is created. The source file is read once, when it is needed for the first time.

This mechanism allows the declaration of more classes than are needed for execution, with minimal overhead for unused classes.

The autoload mechanism is transparent to the programmer. The only thing that needs to be taken into consideration, is that a source file may contain bugs or syntax errors, that will not be detected until the class is actually used. If a Eobj script is executed successfully, it does not indicate that all source files are free from syntax errors, but only those who were used.

4.2.7 Eobj objects vs. plain Perl objects

If you happen to know object programming in plain Perl, then you have more knowledge than is actually needed, and you may possibly make some mistakes because of that. This section is here to help you avoid them.

Otherwise, you may skip this section with no worries.

The class source files are not read directly by the Perl interpreter. Rather, their content is `eval()`'ed after adding a header (this is done in memory, while the source file is left untouched). This header includes the well-known declarations for a proper Perl class (the Perl `package` pragma, and setting `@ISA`, if you care). These are transparent to both class writers and users. In particular, the package name may be different from the class name (even though the class name is used in the common way to generate new objects of the class).

The Eobj object environment is merely a wrapper for the common Perl objects, so the behavior is very similar. The main pitfall for experienced Perl object programmers, is to manipulate variables that the system assumes that the user is not aware of.

In general, the class source files should be as plain as possible: A list of subroutines, nothing else in the file. Every diversion from this is an opening to strange behaviours, and that shouldn't be needed except for rare cases.

So here are a few don't:

- Do not use the `package` pragma, or attempt to access a package by its name, using the `package::name` format. Eobj feels free to change package names without any notice, so you don't know what package name your class source file will get, even if you access the methods with a known class name. If is forgivable to access the `Eobj` or `PLerror` namespaces directly, but this should be avoided, even at the cost of slower execution.
- Never access an object's properties directly. In other words, never make use of the fact that an object is actually a reference to a hash. It's not only that the way the data is stored may change without notice in future versions, but you may also corrupt the data structure.
- Never use `bless`. The objects that Eobj creates are properly blessed, and there should be no reason to rebless them. It is heavily assumed that all objects in the system were generated by the `root` class' `new` or one of its derivatives. Don't make a `new` of your own.
- Don't declare global variables within the package (that is, your class source file). It will work nicely at first, but will mess up things quite soon.
- Don't hassle with `@ISA`, `@EXPORT` and friends. These are global variables, so the previous "don't" should have been enough, but playing with these is even worse.

- Avoid `use` statements (except for `use Eobj` once in the main script). Specifically, `use` should not be used as a pragma (like `use warnings`), while reading modules with `use` should be done very carefully.

4.3 Useful methods

There is a comprehensive listing of methods at the end of this manual. What follows is merely a tip-off regarding methods that is good to know about, and weren't mentioned yet.

- The `who()` and `safewho()` methods are good to get a short and concise string that describes the current object in a way that humans understand. This is useful when creating error messages. Also, when writing a class, consider to override this method with something that will describe the objects better, but be sure to make something similar to the existing `who()` methods (overriding `who()` is enough). See sections 6.1.6 and 6.1.7 for more about this.
- The `globalobj()` method is useful to find the global object of the system. It is useful within method-defining code, since the common `globalobj()` command doesn't exist in the relevant namespace. Thus, a self-call to this method is the solution. See section 6.1.5.
- The `isobject()` method can be used to verify if some scalar is a handle to an object. This verification should be done if there is any doubt about this, before attempting to call a method of the alleged object. Otherwise an ugly Perl error may occur as a result of trying to use a non-object item as an object. See section 6.1.8.

4.4 Error reporting

4.4.1 Some philosophy

In a perfect world, errors tell the user or programmer what should be fixed. In the world where programmers rule, error messages don't exist at all, or they say what went wrong, which doesn't necessarily say anything about what should be fixed.

Eobj is intended to be an environment in which the code of many different people runs together. For this reason, it is important to supply the programmer with a variety of options to report that something is wrong, in a way that reflects the severeness of the error and also gives the reader of the error as much help as possible.

To begin with: `die` and `warn` should never be used within Eobj . Instead, other functions are supplied as follows. These functions can be used exactly in the way that `die` and `warn` would be used, both in the “main script” and in class definitions.

In particular: Unless the error message ends with a newline (\n), the line number on which the function was called will be appended to the error message.

The debug interface will hopefully develop way beyond this. But while writing both scripts and classes, it's compulsory to use the following functions for diagnostic reporting.

4.4.2 The list of functions

- `blow()` – This is similar to `die()`, and should be used when the scripts seem to be OK, but some unrecoverble error occured (such as failing to open a file). The error message should make sense to a user knowing nothing about the internals.
- `puke()` – This will work like `blow()`, but will also present a hopefully concise call stack dump. The ugly name of this function reflects what the output looks like and how graceful it is. To be used in reporting errors that occur only as a result of a bug. When the system `puke()`s on you, the error comes from the guts of the system. Accordingly, whoever will read the error message is expected to be either disgusted or having a good knowledge of the internals.
- `wiz()` – Use this function to throw warnings that will most probably not be understood by anyone else than yourself and a handfull of people. Use this mainly for testing your own class. These warning will be ignored in normal runs.
- `wizreport()` – This function should be used to detect conditions that you don't expect to happen, even after your class is in common use. Running this function will dump a call stack trace into a report file, and ask the user to send it to you. It still functions as a warning, so if execution must be stopped as a result of this condition, a call to `blow()` should take place after this one.
- `fishy()` – Generates a warning like the one you'd expect to get during the Sanity Check stage, but it may become before or after that stage. The contents of the warning text should be clear to a non-Perl user.
- `wrong()` – Like `fishy()`, but will set a flag to abort code generation, or abort immediately if in the middle of it. This is a way to report fatal conditions, and give other objects a chance to file their complaints before halting.
- `say()` – This is for general logging. This is basically reporting what you're doing, if you think it can interest someone.
- `hint()` – Like `say()`, but meant for more verbose information. The intention is to include information that may help debugging. It's plausable that these messages will be ignores unless the system is run in some verbose mode.
- `wink()` – This message is reserved for all of us who usually debug by putting meaningless `print`s in our code to mark that some milestone has been crossed. The messages will appear immeditely and visibly.

4.4.3 "Hidden" classes

Some of the error-reporting classes mentioned above make a call stack dump. There are cases, when we want to hide our class or package from this dump, in order to avoid confusing data from appearing. This applies mainly for system packages, and should not be used on "real" classes. If we want a class to be hidden from stack dumps, we define a global variable with `our $errorcrawl='skip'`, usually at the top of the file. This is a summary of possible values for `$errorcrawl`:

- `skip` – Causes the current package (or class) to be invisible in stack dumps. This was originally intended for the error-reporting package itself, so it wouldn't report its own internal calls.
- `halt` – Like `skip`, but applies also for any calls that the current class performed. In other words, the current class, and any calls it made are invisible.
- `system` – This causes the error message to be shortened slightly, by omitting the file name and line number, and saying "by System" instead. This is a slightly arrogant way to tell the reader of the error message, that the call was indeed performed, but it's useless to try find the error there.

Note that cleaning the error trace is nice as long as the information isn't needed, but it's very annoying if the error had to do with your code. Therefore, it is warmly recommended to avoid this kind of tricks unless you're absolutely confident about your code, and if it really fits the category of "system code".

4.5 Summary: How to write classes properly

This is a short checklist of things to keep in mind while writing a class in Eobj . There is nothing here that isn't mentioned elsewhere in this manual in detail.

- A class file should look like a clean list of subroutines. There should be nothing outside the subroutine blocks.
- Declare all variables with `my`. Don't use global variables.
- Use the global object where there is need for global variables.
- Always consider using the `SUPER::` prefix to call the overridden method. Make sure that all parameters are passed as is, unless intentionally doing otherwise. Make sure that the name of the SUPER'ed method is the same as your own.
- Use `blow()`, `puke()` and `wrong()` as appropriate instead of `die()`. Use `fishy()` and `wiz()` instead of `warn()`. In the error message, use `$self -> who()` to in-dentify your own object, and `$self -> safewho($other)` for some other object.

- Don't use direct subroutine calls, but method calls with as in `$self -> method()`.
- Access properties only with the standard methods (`get`, `set`, `const` and the built-in methods for list) or methods that make use of these. Never attempt to use the object as a hash.
- Be strict about the Eobj conventions. Always assume that your class must be able to work with other people's classes. Make no shortcuts.

5 Eobj main script API

5.1 Exported subroutines

5.1.1 The exported subroutine `init()`

Synopsis:

```
init;
```

Syntax:

```
init;
```

Description:

`init` must be executed before objects are created, and after the class tree is defined by using `inherit`, `override` and `underride`.
The routine sets up the global object and other environmental items.

Return value:

Not to be used

Example:

See the example in section 5.1.2

5.1.2 The exported subroutine `inherit()`

Synopsis:

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

Syntax:

`inherit(`name of new class`,` Perl file`,` class to inherit`);`

Description:

The inherit routine registers a new class into the class tree. This registration includes giving the name of the new class, the Perl file which includes the class' methods, and the class from which methods will be inherited.
The Perl file is not read during the execution of `inherit`, nor is there need for the class to inherit to be declared when the call to `inherit` is performed. `inherit` only verifies that the class that is declared doesn't exist already, and that the third argument is

given.
A detailed description of class declaration is given in section 4.2.1.

Return value:

Always returns `1`

Example:

```
use Eobj;

inherit('class_a', 'myclass_a.pl', 'root');
inherit('class_b', 'myclass_b.pl', 'class_a');

init;

print "--- Now playing with object A --\n";
$objA = class_a -> new(name => 'AObject');
$objA -> Asayhello();
$objA -> benice();

print "--- Now playing with object B --\n";
$objB = class_b -> new(name => 'BObject');
$objB -> Asayhello();
$objB -> Bsayhello();
$objB -> benice();
```

We assume that the file `myclass_a.pl` is:

```
sub Asayhello {
  my $self = shift;
  print "This is class A as ".$self->who()." saying hello\n";
}

sub benice {
  my $self = shift;
  $self -> SUPER::benice(@_);
  print "This is class A saying hello after being nice\n";
}
```

and `myclass_b.pl` is

```
sub Bsayhello {
  my $self = shift;
  print "This is class B as ".$self->who()." saying hello\n";
}

sub benice {
```

```
  my $self = shift;
  $self -> SUPER::benice(@_);
  print "This is class B saying hello after being nice\n";
}
```

The script will thus print out:

```
--- Now playing with object A --
This is class A as object 'AObject' saying hello
This is class A saying hello after being nice
--- Now playing with object B --
This is class A as object 'BObject' saying hello
This is class B as object 'BObject' saying hello
This is class A saying hello after being nice
This is class B saying hello after being nice
```

We note that since class B inherited methods from class A, we could call both `Asayhello()` and `Bsayhello()` on object B. Furthermore, when calling the method `benice()` on object B, class A's `benice()` was called as well by virtue of `$self -> SUPER::benice(@_)`.

5.1.3 The exported subroutine `override()`

Synopsis:

```
override('theclass', 'myclass.pl');
override('theclass', 'myclass.pl', 'root');
```

Syntax:

`override(`name of class`,` Perl file`[,` class to inherit`]);`

Description:

`override` causes the given Perl file's method declarations to override those of the already declared class. Note that unlike common practice in object-oriented programming, this allows to override particular methods without changing the class' name, despite the fact that strictly speaking, this operation generates a new class.
If the class, which is named in the first argument doesn't exist, `override` behaves exactly like `inherit`, and thus a third argument is needed. This third argument is optional, and functions as a "backup" in case the desired class does not exist, and hence needs to be generated with `inherit`.
A detailed description of class declaration is given in section 4.2.1.

Return value:

Always returns `1`

Example:

```
use Eobj;

inherit('class_a', 'myclass_a.pl', 'root');
override('root', 'myclass_b.pl');

init;

$objA = class_a -> new(name => 'AObject');
$objA -> Asayhello();
$objA -> Bsayhello();
$objA -> benice();
```

We assume that the files `myclass_a.pl` and `myclass_b.pl` are the same as in the example for `inherit` in section 5.1.2. This will print out:

```
This is class A as object 'AObject' saying hello
This is class B as object 'AObject' saying hello
This is class B saying hello after being nice
This is class A saying hello after being nice
```

This example shows an override of the root class, which will affect all classes in the system. It's usually not necessary to go that deep down.
Note that the root class was overridden after the declaration of `class_a`, and still `class_a` inherited the methods from `myclass_b.pl` via the inheritence from the root class.

5.1.4 The exported subroutine `underride()`

Synopsis:

`underride('theclass', 'myclass.pl');`

Syntax:

`underride(`name of class`, `Perl file`);`

Description:

`underride` works like `override`, only in the opposite way: It will give the Perl file's method the lowest predecense in the inheritence chain. In other words, the class tree will be set up like it would if the current Perl file was the one that generated the class, and all other declarations of the same class came afterwards as `override()`s.
If another `underride()` is called twice on the same class, the second one will be closer to the root class.
This routine is merely intended for debugging purposes, and is not recommended for standard use.
A detailed description of class declaration is given in section 4.2.1.

Return value:
Always returns `1`

5.1.5 The exported subroutine `definedclass()`

Synopsis:
`$status = definedclass('class');`

Syntax:
<u>class status</u> `= definedclass(`<u>class name</u>`);`

Description:
`definedclass` accepts a class name as argument, and returns a value that reflects the class' status.
A detailed description of class declaration is given in section 4.2.

Return value:
`definedclass` returns `0` if the class has not been defined. `1` is returned if the class is defined, but its Perl code has not been loaded. `2` is returned if the class is defined, and the Perl code has been loaded.

Example:

```
use Eobj;

print "In the beginning, the status was ".definedclass('class_a')."\n";

inherit('class_a', 'myclass_a.pl', 'root');
print "Afterwards, the status was ".definedclass('class_a')."\n";

init;
print "After init the status was ".definedclass('class_a')."\n";

$obj = class_a -> new(name => 'TheObject');
print "After usage the status was ".definedclass('class_a')."\n";
```

This will print:

```
In the beginning, the status was 0
Afterwards, the status was 1
After init the status was 1
After usage the status was 2
```

Note that the status remained `1` even after `init`: The class' Perl code was read only when an object was generated from the class.

5.1.6 The exported subroutine `globalobj()`

Synopsis:

`$GlobObj = globalobj;`

Syntax:

object reference = `globalobj;`

Description:

`globalobj` returns an object reference to the global object. It must not be called before `init()`, since the global object doesn't exist before that.
More information about the global object is given in section 3.4.4.

Return value:

A reference to the global object.

5.2 The global variables

5.2.1 The variable `$VERSION`

Description:

The variable `$VERSION` is the version number. It is declared as is commonly done in Perl modules. Note that it can be used as a number.

Example:

```
use Eobj;

if ($Eobj::VERSION < 1.00) {
  print "We are running on a pre-release version!\n";
}
```

5.2.2 The variable `$globalobject`

Description:

A reference to the global object is stored in `$Eobj::globalobject`. The value of this variable is returned when calling the `globalobj()` routine.

5.2.3 The variable `%classes`

Description:

This hash's keys are names of classes. To be more accurate, these are the names of the Perl packages that will be generated during the loading of class Perl files. The

values are either references to lists, or the scalar value `1`. The latter signifies that the class has been loaded (`definedclass` would return `2`).
When the value is a reference to a list, it is a list of three items: The first item is the name of the Perl file associated with the class. The second is the class which the current class should be derived from (in other words, the value of this class' `@ISA`). The last item is the class name that was used when creating this class. It may be different from the package name due to class override.

5.2.4 The variable `%objects`

Description:
This hash links between object names and their references. The keys are names of objects, and the values are their references.

6 The root class API

6.1 Methods

6.1.1 The method `new()`

Synopsis:

```
$theobj = theclass -> new(name=>'thename');
```

Syntax:

class -> `new(`property hash`);`

Description:

The `new` method creates a new object of a given class. Its initial properties are passed as a hash.

The `new` method must be called with at least one property, the `name` property set. The value of `name` property must be different from that of any other previously created object, even when making case-insensitive comparison.

Return value:

A reference ("handle") to the new object.

Example:

```
use Eobj;
init;
$obj1=root->new(name=>'theObject');
```

Note that that `parent` is assigned `$obj1`, and not the parent's name.

6.1.2 The method `set()`

Synopsis:

```
$obj->set($property, $scalar);
$obj->set($property, @list);
$obj->set(\@path, $scalar);
$obj->set(\@path, @list);
```

Syntax:

object -> set(property, new value);

Description:

The set method sets the value of a property. If property needs not to exist prior to calling set, but it must not have been created by const (see section 6.1.4).
The value is in general a list. Scalars are handled as lists with a single item. Even so, the set and get pair can be used with scalars in a straightforward way, as is shown in the example of section 6.1.3.
To delete a property, set it to undef.
If the property name is a reference to a list, this is considered as the property's path. Note that paths are easily expressed with square brackets, [and] (see example). Paths and their rules are described in section 3.3.7.

Return value:

Always returns 1

Example:

See the example in section 6.1.3

6.1.3 The method get()

Synopsis:

```
$scalar=$obj->get($property);
@list=$obj->get($property);
$scalar=$obj->get(\@path);
@list=$obj->get(\@path);
```

Syntax:

object -> get(property);

Description:

The get method looks for the required property, and returns its value if it exists.
If the property name is a reference to a list, this is considered as the property's path. Note that paths are easily expressed with square brackets, [and] (see example). Paths and their rules are described in section 3.3.7.
The get property is suitable for reading properties defined by set and const.
The set and get pair are coordinated in such a way, that the programmer can assign a list, a scalar or a hash to a property, and get the value later on in the easiest possible way. This is best explained in the example that follows, but the formal rules are hereby described for the sake of formality:
In scalar context: The first element in the list is returned. If a scalar was used to set the property, this arrangement makes sure that get returns what set (or const) got. If the property wasn't defined, undef is returned.
In list context: A list is returned. If the property wasn't defined, it's an empty list.

Note that if the property contains a list, and `get` is evaluated in a scalar context, it does NOT return the number of elements, like some Perl programmers would expect.

Return value:

The value of the requested property.

Example:

```
use Eobj;
init;
$object=root->new(name=>'theObject',
                  foo => 'bar');
print "My name is ".$object->get('name')."\n";
print "Foo is ".$object->get('foo')."\n";

$object->set('myscalar', 'scalarvalue');
$value = $object->get('myscalar');
print "My scalar is $value\n";
print "My scalar as a list: ".join(",", $object->get('myscalar'))."\n";

$object->set('mylist', 'listitem1', 'listitem2', 'listitem3');
@listvalue=$object->get('mylist');
print "My list is ".join(",", @listvalue)."\n";
print "My list (scalar context!): ".$object->get('mylist')."\n";

@thelist=(1, 2, 3);
$object->set('one_two_three', @thelist);
print "Let's count: ".join(",", $object->get('one_two_three'))."\n";

$object->set(['my', 'node'], "This is my node");
$object->set(['my', 'other'], "This is another node");
print "My node: ".$object->get(['my', 'node'])."\n";
print "My other: ".$object->get(['my', 'other'])."\n";
```

This script prints out the following:

```
My name is theObject
Foo is bar
My scalar is scalarvalue
My scalar as a list: scalarvalue
My list is listitem1,listitem2,listitem3
My list (scalar context!): listitem1
Let's count: 1,2,3
My node: This is my node
My other: This is another node
```

6.1.4 The method `const()`

Synopsis:

```
$obj->const($property, $scalar);
$obj->const($property, @list);
$obj->const(\@path, $scalar);
$obj->const(\@path, @list);
```

Syntax:

object -> `const(`property`,` scalar value`);`

Description:

The `const` method is exactly like `set` (see section 6.1.2), only it sets the value of a property as a constant. If the property already exists, the new value value must be equal (stringwise, in the Perl `eq` sense) to the value that the property already has. The sense of equality may be changed from stringwise `eq` to any arbitrary sense by using the `seteq` method detailed in section 6.1.13.

When dealing with lists, equality means equality in the number of element in the list and that each element is stringwise equal (or as chosen with `seteq`).

Either way, the previous value, if assigned, must have been set by `const` (and not `set`).

If the above conditions are not met, a fatal error occurs.

Setting a constant value may trigger off a callback mechanism. See section 6.1.12.

For more information about the constant property, see section 3.3.5.

If the property name is a reference to a list, this is considered as the property's path. Note that paths are easily expressed with square brackets, `[` and `]` (see example). Paths and their rules are described in section 3.3.7.

Return value:

Not to be used.

Example:

```
use Eobj;
init;
$object=root->new(name=>'theObject');

$object->const('myconstant', 'Stay Forever');
$value = $object->get('myconstant');
print "I say $value\n";

$object->const('myconstant', 'Stay Forever'); # This is OK
$object->set('myconstant', 'Stay Forever'); # This is an error
$object->const('myconstant', 'Change!'); # This is an error

$object->set('myscalar', 'I am not a constant');
$object->const('myscalar', 'I am not a constant'); # Error again!
```

This will result in

```
I say Stay Forever
```

and then an error will be reported, because of the attempt to use `set` on a constant value (it doesn't matter that the value would be the same).
The example shows other possible mistakes: Trying to change the value of `myconstant`, or using `const` on a property that is already assigned with `set`.

6.1.5 The method `globalobj()`

Synopsis:

`$theglobal = $anyobject -> globalobj();`

Syntax:

object `-> globalobj();`

Description:

The `globalobj` method returns a handle to the global object of the system. The return value is identical for any object that is an instantiation of a class derived from root, without having this method overrides. In simple words, it doesn't matter which object you run this method on, as long as it's supported.
The purpose of this method is to allow an easy access to the global object from within subroutines that define methods. From the main script, simply use the `globalobj()` routine.
See section 3.4.4 for details about the global object.

Return value:

A reference ("handle") to the global object

Example:

```
use Eobj;
init;
$object=root->new(name=>'theObject');

$theglobal = $object->globalobj();
print "The object's name is ".$theglobal->get('name')."\n";

$shortcut = globalobj();
```

Note that `$shortcut` will have the same value as `$theglobal`. This shorter format is only possible because of `use Eobj`, and hence it can't be used in class declarations.

6.1.6 The method `who()`

Synopsis:

`print "This is ".$object->who."\n";`

Syntax:

object -> `who();`

Description:

The `who` method returns a short and concise identification of the object, so it is readily recognized by humans. It is commonly used in error messages and alike.
Classes that are derived from `root` often override this method to give a better description. The object's name is always mentioned somehow by convention.
Note that by using `who`, we ask the object for some information, so we assume that `$object` (in the synopsis) is a proper object reference. This is assumption should be avoided, especially when handling error messages, due the unexpected nature of errors. See `safewho` in section 6.1.7 for a solution.

Return value:

A short identifier of the object, helpful for humans.

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');

print "This is ".$object->who."\n";
```

Running this:

```
This is object 'theObject'
```

6.1.7 The method `safewho()`

Synopsis:

`print "This is ".$safeobject->safewho($object)."\n";`

Syntax:

object -> `who(`object in question`);`

Description:

`safewho` calls `who` on the object that is passed as an argument, after verifying (using `isobject`) that the object reference is proper.
This is especially useful in code that define methods, since we know for sure that the

object's self reference is proper. We may thus call ourself with the `safewho` method, when attempting to identify another object.

Return value:

Same as `who` if the argument is a proper object. Otherwise, the string `'(non-object item)'` is returned.

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');
$safeobject = root->new(name=>'ImSafe');
$junk = "Hello";

print "This is ".$safeobject->safewho($object)."\n";
print "What is this??? ".$safeobject->safewho($junk)."\n";
```

Running this:

```
This is object 'theObject'
What is this??? (non-object item)
```

Note that trying `$junk->who` would cause a Perl error, that would be quite unhelpful.

6.1.8 The method `isobject()`

Synopsis:

`if (isobject($obj)) { ... }`

Syntax:

object `-> isobject(` scalar`);`

Description:

`isobject` identifies if its scalar argument is a Eobj object. This method is useful before attempting to call an object's method with an arrow notation ("->"). The method's result does not depend on whose object it is a method of. Only scalar in the argument matters (unless the method has been overridden, which it shouldn't).
Note that if an object has been created outside the Eobj mechanism, `isobject` will return false even though it's OK to use the argument as an object.

Return value:

True (`1`) or undefined value (`undef`).

Example:

```
use Eobj;
```

```
init;
$object = root->new(name=>'theObject');

print "Is the handle an object? ".globalobj()->isobject($object)."\n";
print "Is the name an object? ".globalobj()->isobject('theObject')."\n";
print "Can I check myself? ".$object->isobject($object)."\n";
```

Note that we use `globalobj()` just to get some object to call `isobject` on.
Running this we get:

```
Is the handle an object? 1
Is the name an object? 0
Can I check myself? 1
```

Also note that we got `0` when passing the object's name as an argument. The method will always return `0` to any string it gets.

6.1.9 The method `objbyname()`

Synopsis:

```
$obj -> objbyname('theObject');
```

Syntax:

object `-> objbyname(`name of object`);`

Description:

`objbyname()` returns the handle (reference) to the object whose name is given as an argument. If no such object exists, `undef` is returned.

Return value:

A reference to an object or `undef`.

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');

$same = globalobj->objbyname('theObject');

print("The same object!\n")
  if ($object == $same);
```

Description:

In this example, we create a new object, and puts its handle in `$object`. Then we get the same value by using `objbyname()` on the object's name, and put it in `$same` (which is useless for practical reasons, since we've already have the reference of the objects handy). Note that we use the global object, like we would if we had no other reference to an object handy.

When running this script, the `print` is executed.

6.1.10 The method `objdump()`

Synopsis:

```
$obj -> objdump;
$obj -> objdump('theObject');
$obj -> objdump($objref);
```

Syntax:

object -> `objbyname();`
object -> `objbyname(`list of names or references of objects`);`

Description:

`objdump()` is intended for debugging. It prints out information (to the standard output) about either a specific object, or all the objects that are defined in the system.

If a list of objects is given (this includes one object) as argument, these objects' information is printed out, which includes the properties.

The objects may be identified by their name or reference interchangably.

If `objdump()` is called with no arguments, all objects in the system are showed in the order that they were created.

The output format is intended for human reading, and is thus presented in what seems to be an easy way to read, with less emphasis on formal rules.

Return value:

Not to be used.

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');

globalobj->objdump($object);     # Will show 'theObject'
globalobj->objdump('theObject'); # 'theObject' again
$object->objdump(globalobj);     # Will show global object!
globalobj->objdump();            # Will show them all
```

6.1.11 The method `suggestname()`

Synopsis:

```
$safename = $obj->suggestname($IWantThis);
$obj = AnyClass->new(name => $safename);
```

Syntax:

object -> `suggestname(`desired name`)`

Description:

`suggestname()` will check the given string against the names of already existing objects. If the name is OK for a new object, it will simply return the string. If a new object can't be named with the given string, it is altered lightly (see example). Either way, the returned string is a legal name for a new object, which will be close enough to the original.

Note, that `suggestname` may suggest the same name more than once, if it isn't used to create a new object – it checks uniqueness against existing objects, not its own suggestions.

Return value:

A string with a name for a new object.

Example:

```
use Eobj;
init;
$global = globalobj();
$name1 = $global -> suggestname('theObject');
$object = root->new(name => $name1);
print "The first object was called $name1\n";

$name2 = $global -> suggestname('THEOBJECT');
$object = root->new(name => $name2);
print "The next object was called $name2\n";

$name3 = $global -> suggestname('theobject');
print "Now we were suggested the name $name3\n";
$name4 = $global -> suggestname('theobject');
print "We didn't use it, so we got $name4 again\n";

$name5 = $global -> suggestname('theobjects');
print "We added one 's', and got $name5 -- it's unique\n";
```

We run this:

```
The first object was called theObject
The next object was called THEOBJECT_1
```

```
Now we were suggested the name theobject_2
We didn't use it, so we got theobject_2 again
We added one 's', and got theobjects -- it's unique
```

6.1.12 The method `addmagic()`

Synopsis:

```
$object->addmagic($property, \&callback);
$object->addmagic($property, sub { ... });
$object->addmagic(\@path, \&callback);
$object->addmagic(\@path, sub { ... });
```

Syntax:

<u>object</u> `-> addmagic(`<u>property</u>`,` <u>subroutine reference</u>`);`

Description:

The `addmagic` method queues a callback subroutine, which will be called upon when the respective property is assigned a value by using `const`. If The property's value is already set, the subroutine will be called immediately.

If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 3.3.7.

`addmagic` is loop-safe: The callback mechanism assures that infinite callback loops will not occur. This is done by removing each callback entry from the queue before performing the callback itself, so each callback entry is run at most once. In particular, the callback subroutine may include a call to the `const` of another object, which may trigger off another callback. In fact, this is the intended use of `addmagic` (see examples).

Using `addmagic` and callback subroutines requires an understanding of the scope under which the subroutines are run (that is, when variables are evaluated, and what variable space is applied). A lack of such understanding may lead to strange bugs. We shall address a few of the issues in the examples below.

Return value:

Not to be used.

Example:

```
use Eobj;
init;
$object1 = root->new(name=>'First');
$object2 = root->new(name=>'Second');

$copy1to2 = sub {
  print "Callback to copy1to2\n";
  my $val = $object1 -> get('TheProperty');
  $object2 -> const('TheProperty', $val);
```

```
};

$copy2to1 = sub {
  print "Callback to copy2to1\n";
  my $val = $object2 -> get('TheProperty');
  $object1 -> const('TheProperty', $val);
};

$object1->addmagic('TheProperty', $copy1to2);
$object2->addmagic('TheProperty', $copy2to1);

$object1->const('TheProperty', 'TheValue');
print "The value is ".$object2->get('TheProperty')."\n";
```

And when running this, we get:

```
Callback to copy1to2
Callback to copy2to1
The value is TheValue
```

The heart of this example is that we ran `const` on `$object1`, but read the property of `$object2`. This demonstrates how one object can "learn" the value of another one as soon as it is set.
We see that by calling `const` on `$object1` triggered off the callback to `copy1to2`. In `copy1to2` there's `const` is called on `$object2`, and thus `copy2to1` is triggered off. In `copy2to1` we call `const` on `$object1` again, on a constant property that already has a value. This is OK, since we attempt to assign the same value that the property already has.
No more callbacks take place, since we've exhausted the queues. Note that each of the two objects watch the other. We could have ran `const` on `$object2`, and read it from `$object1` as well. In fact, this callback setup assures that both object's properties will be equal, both by copying the value, and by not allowing unequal values to be set.
In the example above, the subroutines could have been defined as `sub copy1to2 { ... }` (and not `$copy1to2 = sub { ... }`), achieving the same results for this specific example. (In this case `\&copy1to2` would be given to `addmagic`, rather than `$copy1to2`). Even so, it's still higly recommended to follow the example's subroutine definition format, in order to assure the correct scoping. This is especially important if the subroutines are defined as part of some method routine.
We now see another example, which demonstrates a variable scoping issue. Suppose that we want 10 objects, whose property `X` is equal on all:

```
use Eobj;
init;
@l=();
for ($i=0; $i<10; $i++) {
  $l[$i]=root->new(name=>'MyName'.$i);
```

```
  my $j=$i;
  if ($l[$j+1]) {
    $l[$j]->addmagic('X',
                     sub { $l[$j+1]->const('X', $l[$j]->get('X') ); });
  }
  if ($l[$j-1]) {
    $l[$j]->addmagic('X',
                     sub { $l[$j-1]->const('X', $l[$j]->get('X') ); });
  }
}

print "Before, the value is '".$l[3]->get('X')."'\n";
$l[8]->const('X', 'what we want');
print "After, the value is '".$l[3]->get('X')."'\n";
```

Running this, we get:

```
Before, the value is ''
After, the value is 'what we want'
```

In this example, we pass an anonymous subroutine (`sub {...}`) to `addmagic`. It is important to note, that even though the loop index is `$i`, we create a copy of it, `my $j=$i;` and use it within the callback.
The reason for this, is that the variables in the subroutine are evaluated only upon execution, but the values are those that the variables have at the moment of execution. Thus, we couldn't use `$i` in the callback subroutine, because it would have the value `10` (the final value) for all subroutines. By making a "local copy" with `my`, within a Perl block, we get the right `$j` for each callback.
This example may be confusing, but it shown the importance of knowing the scoping issue well.

6.1.13 The method `seteq()`

Synopsis:

```
$object->seteq($property, \&compare);
$object->seteq($property, sub { ... });
$object->seteq(\@path, \&compare);
$object->seteq(\@path, sub { ... });
```

Syntax:

<u>object</u> `-> seteq(`<u>property</u>`,` <u>subroutine reference</u>`);`

Description:

`seteq` changes the meaning of equalilty for a certain property. This meaning is effective when `const` is used on a property that already has a value, which is when `const` verifies that the new value and the old one are "the same". The exact meaning of being "the same" is given by the argument to `seteq`, which is a reference to a comparing subroutine.
The comparing subroutine shall accept two arguments, and return `1` if the arguments are "the same" in the relevant sense, `0` otherwise.
`const` does not update the propery nor run callbacks when executed on a property that has a value, even if it is considered equal. If the property needs to be updated, the use of constant properties should be reconsidered.
By default, stringwise `eq` is used to compare the value given to `const`, as if `seteq($property, sub {return shift eq shift;})` had been run on every property.
If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 3.3.7.

Return value:

Not to be used.

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');

$object -> seteq('theProperty', \&ignorecase);

$object -> const('theProperty', 'THEVALUE');

#The next line would cause an error if it wasn't for seteq above
$object -> const('theProperty', 'thevalue');

print "The value is ".$object->get('theProperty')."\n";

sub ignorecase {
  my ($a, $b) = @_;
  return (lc($a) eq lc($b));
}
```

Running, this:

```
The value is THEVALUE
```

In this example, we make the comparison case-insensitive by applying the subroutine `ignorecase` on `seteq`. Unlike `addmagic` (see section 6.1.12), it's proper to use named subroutines (defined as `sub ignorecase { ... }`), since the result of the subroutine

depends only on its arguments, and hence scoping issues are irrelevant.
We can see that the value of the property did not change, but no error was reported.

6.1.14 The method `pshift()`

Synopsis:

```
$scalar=$obj->pshift($property);
$scalar=$obj->pshift(\@path);
```

Syntax:

object `->` `pshift(`property`);`

Description:

`pshift` treats the given property as a list, and performs a Perl `shift` operation on that list: It removes the first item of the list, and returns its value. If the property is undefined, or it's an empty list, `undef` is returned.
The property must not have been defined with `const`.
If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 3.3.7.

Return value:

Same as Perl's `shift` on a list.

Example:

See the example in section 6.1.17

6.1.15 The method `ppop()`

Synopsis:

```
$scalar=$obj->ppop($property);
$scalar=$obj->ppop(\@path);
```

Syntax:

object `->` `ppop(`property`);`

Description:

`ppop` treats the given property as a list, and performs a Perl `pop` operation on that list: It removes the last item of the list, and returns its value. If the property is undefined, or it's an empty list, `undef` is returned.
The property must not have been defined with `const`.
If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 3.3.7.

Return value:

Same as Perl's `pop` on a list.

Example:

See the example in section 6.1.17

6.1.16 The method `punshift()`

Synopsis:

```
$obj->punshift($property, $scalar);
$obj->punshift($property, @list);
$obj->punshift(\@path, $scalar);
$obj->punshift(\@path, @list);
```

Syntax:

object -> `punshift(`property`,` list items`);`

Description:

`punshift` treats the given property as a list, and performs a Perl `unshift` operation on that list: It adds the given items at the beginning of the list. If the property is undefined, it is created with `set`.
The property must not have been defined with `const`.
If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 3.3.7.

Return value:

Not to be used.

Example:

See the example in section 6.1.17

6.1.17 The method `ppush()`

Synopsis:

```
$obj->ppush($property, $scalar);
$obj->ppush($property, @list);
$obj->ppush(\@path, $scalar);
$obj->ppush(\@path, @list);
```

Syntax:

object -> `ppush(`property`,` list items`);`

Description:

`ppush` treats the given property as a list, and performs a Perl `push` operation on that list: It adds the given items at the end of the list. If the property is undefined, it is created with `set`.

The property must not have been defined with `const`.
If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 3.3.7.

Return value:
Not to be used.

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');

$object -> set('myList', 5, 6, 7, 8);

print "This is five: ".$object -> pshift('myList')."\n";
print "This is eight: ".$object -> ppop('myList')."\n";

$object -> punshift('myList', 1, 2);
$object -> ppush('myList', 'Finito!');

print "My list is ".join(', ', $object -> get('myList'))."\n";
```

Which prints when runned:

```
This is five: 5
This is eight: 8
My list is 1, 2, 6, 7, Finito!
```

6.1.18 The method `prettyval()`

Synopsis:
`print "I have ".$self->prettyval(@things)."\n";`

Syntax:
object `-> prettyval(`list`);`

Description:
`prettyval()` accepts items in a list, and returns a string with the items presented in a way that humans understand. This should be used in error messages and similar cases, when we want to present the value in a message.

`prettyval()` handles lists by printing a few of the first elements in parantheses. What appears to be non-numerical strings is enclosed with quote marks. Object references are translated into their `who()` representation, enclosed in curled brackets.

This method may be used in conjunction with `linebreak()` in order to handle long lines.

Return value:
A well-formatted string

Example:
See the example in section 6.1.19

6.1.19 The method `linebreak()`

Synopsis:

```
print $self->linebreak($A_long_string);
print $self->linebreak($A_long_string, ' ');
```

Syntax:
<u>object</u> `-> linebreak(`<u>string</u>`[,` <u>indent string</u>`]);`

Description:
The `linebreak()` method searches the string for lines that are over 80 characters in length (newline to newline), and attempts to cut them wisely by adding newlines.

If a second argument is given, that string will be put after each newline that is inserted. If the string is a few whitespaces, this will turn out to be the indentation of each line break that the method creates.

With “newline” we mean a Perl `\n`.

Return value:
A (hopefully) better formatted string

Example:

```
use Eobj;
init;
$object = root->new(name=>'theObject');

@thelist = ('Foo', 5, $object, globalobj(), 'Bar', 'FooBar');

print globalobj->linebreak("I don't know what to do with ".
   globalobj->prettyval(@thelist)."\n");
```

This will print out:

```
I don't know what to do with ('Foo', 5, {object 'theObject'},
{The Global Object}, ...)
```

Note that the line break before the forth element is a result of the `linebreak()` method. We can also see that only the first four elements were actually displayed.

The rest are chopped out.

6.2 Properties

6.2.1 The property `name`

Description:

The `name` property is a unique ASCII identifier of the object. It must be set when creating a new object.
The first character of the property string must be an underscore or a letter (upper case or lower case). The rest of the string must match `\w*` (as a Perl regular expression).
`MyName`, `Hello_9`, and `_underscored` are legal names. `2good`, `-myname`, `%Name` and `/slash` are illegal.
The `name` property must be unique in a case-insenstive sense.
See 6.1.1 and 6.1.11 for more details.

A GNU FDL license

GNU Free Documentation License
Version 1.2, November 2002

A.0 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as ”you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A ”Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A ”Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The ”Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The ”Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A ”Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not ”Transparent” is called ”Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The ”Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ”Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section ”Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as ”Acknowledgements”, ”Dedications”, ”Endorsements”, or ”History”.) To ”Preserve the Title” of such a section when you modify the Document means that it remains a section ”Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.2 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section A.3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections A.2 and A.3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled ”History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled ”History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the ”History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled ”Acknowledgements” or ”Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled ”Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled ”Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled ”Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section A.4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ”History” in the various original documents, forming one section Entitled ”History”; likewise combine any sections Entitled ”Acknowledgements”, and any sections Entitled ”Dedications”. You must delete all sections Entitled ”Endorsements”.

A.6 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ”aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section A.3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.8 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section A.4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled ”Acknowledgements”, ”Dedications”, or ”History”, the requirement (section A.4) to Preserve its Title (section 1) will typically require changing the actual title.

A.9 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

B To Do

This section includes issues that are still to be done.

B.1 Core issues

B.1.1 AUTOLOAD caching

Change the AUTOLOAD mechanism, so it puts a subroutine in the namespace of the caller for standard routines (error reporting and such). Then, enrich the group of subroutines that are supported from anywhere ("exported").

B.2 Complete the half-made

B.2.1 The error messages

Make the different error messages (`puke`, `wiz`, `hint` and so on) actually generate some different things. Make `wrong` actually set flags and/or stop in time.

B.3 System management

B.3.1 Organizing classes

Define and implement means for an easy installation of new classes. For example, search some path recursively for some certain filename pattern, and run the file as a script during `init`. This will allow class enrichment easily, and the classes could be loaded depending on some condition.

B.3.2 Run options

A means for defining the execution options (target device, debug modes and so on), as well as holding it comfortably in the system is needed.

B.4 User Interface

B.5 Debug tools

B.5.1 Error trace

Due to the complexity of the system, it is hard to give a concise error message. The error may be detected an reported, but it says nothing about the reason for it. For example, if a constant property gets a new value, it's obviously an error, but it reported at the attempt to change it, without saying anything about when and where the first value got there.

This should be remedied as follows: The system should be able to run in a error-trace mode, in which every call to distinguished methods (or all?) is logged, along with the stack trace. This can be done by automatically overriding all or some of the classes with envelope classes, which log every call and exit from methods. This can be helpful for human reading, but even better, it can help to resolve what happened, and who was trying to do what.

This error-trace mode will be slower, but it allows a rerun when something goes wrong (which could be done automatically by a wrapper such as a GUI tool).

B.5.2 All class loader

This general function is useful to verify that all declared classes are indeed OK. This would mean "load classes now".