



**Serious Perl: the absolute minimum you need to know**

Henning Koch  
hkoch@paws.cx

Henning started this series in issue 1.2 by writing about data structures and continues it this issue.

**Serious Perl**

Perl's extremely flexible syntax makes it easy to write code that is harder to read and maintain than it could be. This column describes some very basic practices I consider necessary for a clear and concise style of writing Perl.

In this installment, I cover module importing which allows you to get functions and variables from external modules into your main script. With a little magic, you can make it do even more, such as automatically create accessor methods for object data.

**Imports**

Because the packages you use get imported at compile-time you can completely change the playing field before the interpreter even gets to look at the rest of your script. Thus import mechanism can be extremely powerful.

For most cases, the Exporter module can provide the import() routine. It handles basic things such as functions and variables. You can do more fancy things with your own import(), which is just a Perl subroutine in which you can do whatever you want.

**Import parameters**

You can hand over parameters to any package you use.

```
package Student;
use Some::Package 'param1', 'param2';
```

Whenever you use() a package, perl calls the static method import() in that package with all parameters you might have given. This use() statement that

```
imports three functions from Some::Package
use Some::Package 'a', 'b', 'c';
```

is the same thing as

```
BEGIN {
    require Some::Package;
    Some::Package->import( 'a', 'b', 'c' );
}
```

Your import can do whatever you want it to do. In this case, I print the passed parameters.

```
package Some::Package;
sub import {
    my($class, @params) = @_;
    print "I got parameters @_\n";
}
```

**Who's calling?**

The caller() function lets you (among other things) find out what class was calling the current method. You don't have to hard-code the package name, which makes inheritance and sub-classing work. You can even do different things depending on the package name, if you like.

```
package Some::Package;

sub import {
    my($class, @params) = @_;
    print "Look, " . caller() .
        " is trying to import me!";
}
```

**Extending the language**

Combine what I just told you to write a simple package named "members" that sets fields for the calling package, and while it's at it produces convenient accessor methods for those fields.