Perl Short Course: Sixth Session

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

December 2011

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**
  - **how to write Perl classes**

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**
  - **how to write Perl classes**
  - **how to inherit classes**

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**
  - **how to write Perl classes**
  - **how to inherit classes**
- Perl's approach to modularity, information hiding, abstraction and OO is refreshingly lightweight: Perl constructs its modules and classes using about half a dozen new concepts and keywords.

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**
  - **how to write Perl classes**
  - **how to inherit classes**
- Perl's approach to modularity, information hiding, abstraction and OO is refreshingly lightweight: Perl constructs its modules and classes using about half a dozen new concepts and keywords.
- **Modularity**: splitting a large program into *separate source files*. All serious languages have some form of this capability, often linked with *separate compilation* and *control of the interface*. These separate components are variously called *modules*, *units* or (in extreme cases) *classes*.

- In this session, we'll see how we construct Perl modules:
  - **creating Perl modules**
  - **controlling symbol export/import**
  - **how to write Perl classes**
  - **how to inherit classes**
- Perl's approach to modularity, information hiding, abstraction and OO is refreshingly lightweight: Perl constructs its modules and classes using about half a dozen new concepts and keywords.
- **Modularity**: splitting a large program into *separate source files*. All serious languages have some form of this capability, often linked with *separate compilation* and *control of the interface*. These separate components are variously called *modules, units* or (in extreme cases) *classes*.
- Now, let's dive straight in, and see how easy it is to build a Perl module from scratch: suppose we're working with word frequencies in a set of files (a *corpus*), and want to know the most frequent words.

- We might speculatively write the following main program (**eg1**), using a module that doesn't exist yet. (You'll find this in the examples tarball inside the `wordfreq-v0/` directory):

```
use maxfreq;  # if it exists!

die "Usage: eg1 wordfile [wordfile...]\n" unless @ARGV;
# read all words in all files, build a frequency hash...
my %freq = ();
while( my $line = <> )
{
        chomp $line;
        $line =~ s/^\s+//; $line =~ s/\s+$//; $line = lc($line);
        my @wd = split( /\s+/, $line );
        foreach my $word (@wd)
        {
                print "word is blank\n" if $word eq "";
                $freq{$word}++;
        }
}
# tell maxfreq about our frequency data
maxfreq::forget();
maxfreq::remember( %freq );
# now maxfreq can tell us the maximum frequency in the
# whole data set, and all words with that frequency..
my( $maxfreq, @mostfreqwords ) = maxfreq::getbest();
my $str = join( ',', @mostfreqwords );
print "maximum word frequency: $maxfreq, most frequent words: $str\n";
```

- We might speculatively write the following main program (**eg1**), using a module that doesn't exist yet. (You'll find this in the examples tarball inside the `wordfreq-v0/` directory):

```
use maxfreq;  # if it exists!

die "Usage: eg1 wordfile [wordfile...]\n" unless @ARGV;
# read all words in all files, build a frequency hash...
my %freq = ();
while( my $line = <> )
{
        chomp $line;
        $line =~ s/^\s+//; $line =~ s/\s+$//; $line = lc($line);
        my @wd = split( /\s+/, $line );
        foreach my $word (@wd)
        {
                print "word is blank\n" if $word eq "";
                $freq{$word}++;
        }
}
# tell maxfreq about our frequency data
maxfreq::forget();
maxfreq::remember( %freq );
# now maxfreq can tell us the maximum frequency in the
# whole data set, and all words with that frequency..
my( $maxfreq, @mostfreqwords ) = maxfreq::getbest();
my $str = join( ',', @mostfreqwords );
print "maximum word frequency: $maxfreq, most frequent words: $str\n";
```

- Syntax check this with `perl -cw eg1` - even Perl complains about a missing module!

- Create a stub module as follows (in examples tarball in the `wordfreq-v1/` directory):

```
package maxfreq;
use strict;
use warnings;

# frequency module: record any amount of item frequency data (via "remember")
# and then report the maximum frequency, and items with that frequency.

#
# forget(); initialize/reset - forget all frequency data
#
sub forget () { print "forget(): stub call\n"; }

#
# remember( %data );
#   add more frequency data to what we remember, accumulating
#   frequencies across multiple remember() calls.  eg. if first
#   told remember freq{x}=3 and later freq{x}=7, then freq{x}=10.
#
sub remember (%) { my(%data) = @_; print "remember(): stub call\n"; }

#
# my( $maxfreq, @mostfreqitems ) = getbest();
#   report the maximum frequency in all remembered items,
#   and all items with precisely that frequency.
#
sub getbest () { print "getbest(): stub call returning 0\n"; return (0); }

1;
```

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.
  - `maxfreq.pm` starts with the declaration `package maxfreq` to give its' subroutines (and global `our` variables, if any) a private namespace. The default package is called `main` - so all our previous examples were in package `main`.

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.
  - `maxfreq.pm` starts with the declaration `package maxfreq` to give its' subroutines (and global `our` variables, if any) a private namespace. The default package is called `main` - so all our previous examples were in package `main`.
  - `maxfreq.pm` switches on strict mode and then defines several ordinary subroutines. In this case, stub implementations.

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.
  - `maxfreq.pm` starts with the declaration `package maxfreq` to give its' subroutines (and global `our` variables, if any) a private namespace. The default package is called `main` - so all our previous examples were in package `main`.
  - `maxfreq.pm` switches on strict mode and then defines several ordinary subroutines. In this case, stub implementations.
  - One weird detail is that each module must end with a spurious true value, such as '`1;`'.

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.
  - `maxfreq.pm` starts with the declaration `package maxfreq` to give its' subroutines (and global `our` variables, if any) a private namespace. The default package is called `main` - so all our previous examples were in package `main`.
  - `maxfreq.pm` switches on strict mode and then defines several ordinary subroutines. In this case, stub implementations.
  - One weird detail is that each module must end with a spurious true value, such as '`1;`'.
  - Such a module is imported into a program by the usual `use maxfreq` syntax, just like pre-written modules.

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.
  - `maxfreq.pm` starts with the declaration `package maxfreq` to give its' subroutines (and global `our` variables, if any) a private namespace. The default package is called `main` - so all our previous examples were in package `main`.
  - `maxfreq.pm` switches on strict mode and then defines several ordinary subroutines. In this case, stub implementations.
  - One weird detail is that each module must end with a spurious true value, such as '`1;`'.
  - Such a module is imported into a program by the usual `use maxfreq` syntax, just like pre-written modules.
- Now syntax check both the module (`perl -cw maxfreq.pm`) and eg1 (`perl -cw eg1`). Run `eg1 ../corpus/*` to make it analyse a small corpus of words.

- What can we see immediately?
  - A Perl module called **maxfreq** is stored in a file called `maxfreq.pm`.
  - `maxfreq.pm` starts with the declaration `package maxfreq` to give its' subroutines (and global `our` variables, if any) a private namespace. The default package is called `main` - so all our previous examples were in package `main`.
  - `maxfreq.pm` switches on strict mode and then defines several ordinary subroutines. In this case, stub implementations.
  - One weird detail is that each module must end with a spurious true value, such as `'1;'`.
  - Such a module is imported into a program by the usual `use maxfreq` syntax, just like pre-written modules.
- Now syntax check both the module (`perl -cw maxfreq.pm`) and eg1 (`perl -cw eg1`). Run `eg1 ../corpus/*` to make it analyse a small corpus of words.
- Of course it doesn't produce any answers - with a stub module. We have to implement **maxfreq**!

- First, we have to decide what data structures we need. We'll certainly need a *cumulative frequency hash* to store all the items and their frequencies that we currently remember.

- First, we have to decide what data structures we need. We'll certainly need a *cumulative frequency hash* to store all the items and their frequencies that we currently remember.
- There is a choice of whether to store the *maximum frequency seen* and the *items with that maximum frequency*, or whether to calculate them on demand. We choose to store them:

```
my %freq = ();          # cumulative freqencies of everything we've seen.
my $maxfreq = 0;        # current maximum frequency of any item.
my @mostfreq = ();      # list of all items with $maxfreq.
```

- First, we have to decide what data structures we need. We'll certainly need a *cumulative frequency hash* to store all the items and their frequencies that we currently remember.
- There is a choice of whether to store the *maximum frequency seen* and the *items with that maximum frequency*, or whether to calculate them on demand. We choose to store them:

```
my %freq = ();          # cumulative freqencies of everything we've seen.
my $maxfreq = 0;        # current maximum frequency of any item.
my @mostfreq = ();      # list of all items with $maxfreq.
```

- `forget()` repeats the above initializations (without '`my`').

- First, we have to decide what data structures we need. We'll certainly need a *cumulative frequency hash* to store all the items and their frequencies that we currently remember.
- There is a choice of whether to store the *maximum frequency seen* and the *items with that maximum frequency*, or whether to calculate them on demand. We choose to store them:

```
my %freq = ();          # cumulative freqencies of everything we've seen.
my $maxfreq = 0;        # current maximum frequency of any item.
my @mostfreq = ();      # list of all items with $maxfreq.
```

- `forget()` repeats the above initializations (without '`my`').
- `remember(%data)` is implemented by:

```
while( my($k,$v) = each(%data) )
{
        $freq{$k} += $v;
        updatemaxfreq( $k, $freq{$k} );
}
```

- First, we have to decide what data structures we need. We'll certainly need a *cumulative frequency hash* to store all the items and their frequencies that we currently remember.
- There is a choice of whether to store the *maximum frequency seen* and the *items with that maximum frequency*, or whether to calculate them on demand. We choose to store them:

```
my %freq = ();          # cumulative freqencies of everything we've seen.
my $maxfreq = 0;        # current maximum frequency of any item.
my @mostfreq = ();      # list of all items with $maxfreq.
```

- `forget()` repeats the above initializations (without '`my`').
- `remember(%data)` is implemented by:

```
while( my($k,$v) = each(%data) )
{
      $freq{$k} += $v;
      updatemaxfreq( $k, $freq{$k} );
}
```

- `updatemaxfreq($k,$f)` is a private routine implemented by:

```
if( $f == $maxfreq )
{
      push @mostfreq, $k;     # add to most frequent list
} elsif( $f > $maxfreq )
{
      $maxfreq = $f;          # new maximum frequency
      @mostfreq = ( $k );     # only $k is most frequent
}
```

- Finally, `getbest()` is implemented by:

```
return ( $maxfreq, @mostfreq );
```

- Finally, `getbest()` is implemented by:

```
return ( $maxfreq, @mostfreq );
```

- You'll find the full version of **maxfreq** in the examples tarball inside the `wordfreq-v2/` directory.

- Finally, `getbest()` is implemented by:

```
return ( $maxfreq, @mostfreq );
```

- You'll find the full version of **maxfreq** in the examples tarball inside the `wordfreq-v2/` directory.
- After syntax checking, if we rerun `eg1 ../corpus/*` it should tell us the most frequent single word. Any guesses what it will be beforehand?

- Finally, `getbest()` is implemented by:

```
return ( $maxfreq, @mostfreq );
```

- You'll find the full version of **maxfreq** in the examples tarball inside the `wordfreq-v2/` directory.
- After syntax checking, if we rerun `eg1 ../corpus/*` it should tell us the most frequent single word. Any guesses what it will be beforehand?
- If we wanted to know the top 10 (most frequent) words, an obvious extension would be to extend **maxfreq**, adding:

```
#
# delbest();
#   delete all the "most frequent" items, recalculate the
#   new maximum frequency and most frequent items
#
sub delbest ()
{
  # implement me..
}
```

- Then extend **eg1** to say:

```
my $howmany = 10;
print "top $howmany word-sets by word frequency:\n\n";
foreach (1..$howmany)
{
      my( $maxfreq, @freqwords ) = maxfreq::getbest();
last if $maxfreq < 1;
      my $str = join( ',', @freqwords );
      print "word frequency: $maxfreq, words: $str\n";
      maxfreq::delbest();
}
```

- Then extend **eg1** to say:

```
my $howmany = 10;
print "top $howmany word-sets by word frequency:\n\n";
foreach (1..$howmany)
{
      my( $maxfreq, @freqwords ) = maxfreq::getbest();
last if $maxfreq < 1;
      my $str = join( ',', @freqwords );
      print "word frequency: $maxfreq, words: $str\n";
      maxfreq::delbest();
}
```

- You can find the extended versions of `eg1` and `maxfreq.pm` in the examples tarball inside the `wordfreq-v3/` directory.

- Then extend **eg1** to say:

```
my $howmany = 10;
print "top $howmany word-sets by word frequency:\n\n";
foreach (1..$howmany)
{
      my( $maxfreq, @freqwords ) = maxfreq::getbest();
last if $maxfreq < 1;
      my $str = join( ',', @freqwords );
      print "word frequency: $maxfreq, words: $str\n";
      maxfreq::delbest();
}
```

- You can find the extended versions of `eg1` and `maxfreq.pm` in the examples tarball inside the `wordfreq-v3/` directory.
- Running this version, we get the top 10 words:

```
word frequency: 440, words: the
word frequency: 190, words: of
word frequency: 98, words: and
word frequency: 94, words: in
word frequency: 68, words: to
word frequency: 57, words: a
word frequency: 54, words: is
word frequency: 49, words: binding
word frequency: 46, words: energy
word frequency: 41, words: potential
```

- Then extend **eg1** to say:

```
my $howmany = 10;
print "top $howmany word-sets by word frequency:\n\n";
foreach (1..$howmany)
{
      my( $maxfreq, @freqwords ) = maxfreq::getbest();
last if $maxfreq < 1;
      my $str = join( ',', @freqwords );
      print "word frequency: $maxfreq, words: $str\n";
      maxfreq::delbest();
}
```

- You can find the extended versions of `eg1` and `maxfreq.pm` in the examples tarball inside the `wordfreq-v3/` directory.
- Running this version, we get the top 10 words:

```
word frequency: 440, words: the
word frequency: 190, words: of
word frequency: 98, words: and
word frequency: 94, words: in
word frequency: 68, words: to
word frequency: 57, words: a
word frequency: 54, words: is
word frequency: 49, words: binding
word frequency: 46, words: energy
word frequency: 41, words: potential
```

- Anyone care to guess what specialised subject the input documents referred to?

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.
- We've mentioned that we can declare a different type of shared variables - called *package variables* - using the syntax '`our`'.

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.
- We've mentioned that we can declare a different type of shared variables - called *package variables* - using the syntax '`our`'.
- So what's the difference for shared variables in modules?

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.
- We've mentioned that we can declare a different type of shared variables - called *package variables* - using the syntax '`our`'.
- So what's the difference for shared variables in modules?
- '`my`' variables are associated with the lexical scope - they are only accessible from the single Perl source file defining the module, from the point of declaration down to the bottom.

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.
- We've mentioned that we can declare a different type of shared variables - called *package variables* - using the syntax '`our`'.
- So what's the difference for shared variables in modules?
- '`my`' variables are associated with the lexical scope - they are only accessible from the single Perl source file defining the module, from the point of declaration down to the bottom.
- Hence, only the package subroutines (in the rest of the file) can see these '`my`' variables. They are shared between those subroutines - and are *truly private to them*.

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.
- We've mentioned that we can declare a different type of shared variables - called *package variables* - using the syntax '`our`'.
- So what's the difference for shared variables in modules?
- '`my`' variables are associated with the lexical scope - they are only accessible from the single Perl source file defining the module, from the point of declaration down to the bottom.
- Hence, only the package subroutines (in the rest of the file) can see these '`my`' variables. They are shared between those subroutines - and are *truly private to them*.
- '`our`' variables belong to the *package* not the *file*. They are accessible outside the package via naughty people writing (for example) `push @maxfreq::mostfreq, "hello"`.

- We've seen that we can declare shared variables in modules via the usual '`my`' syntax, near the top of the package.
- We've mentioned that we can declare a different type of shared variables - called *package variables* - using the syntax '`our`'.
- So what's the difference for shared variables in modules?
- '`my`' variables are associated with the lexical scope - they are only accessible from the single Perl source file defining the module, from the point of declaration down to the bottom.
- Hence, only the package subroutines (in the rest of the file) can see these '`my`' variables. They are shared between those subroutines - and are *truly private to them*.
- '`our`' variables belong to the *package* not the *file*. They are accessible outside the package via naughty people writing (for example) `push @maxfreq::mostfreq, "hello"`.
- In summary, use '`my`' variables most of the time. Only in special cases - where other parts of Perl need to be able to inspect package variables - should you use '`our`'.

- All this `maxfreq::remember` stuff is rather longwinded, in most languages the module designer can *control the interface* - choose which (public) symbols to export, and the module user can choose which symbols to import.

- All this `maxfreq::remember` stuff is rather longwinded, in most languages the module designer can *control the interface* - choose which (public) symbols to export, and the module user can choose which symbols to import.
- How do we do it in Perl? By using a Perl special module called `Exporter` which exists for precisely this purpose:

- All this `maxfreq::remember` stuff is rather longwinded, in most languages the module designer can *control the interface* - choose which (public) symbols to export, and the module user can choose which symbols to import.
- How do we do it in Perl? By using a Perl special module called `Exporter` which exists for precisely this purpose:
- `Exporter` defines three conceptual sets, which must be defined by `our` variables:
  - The set of symbols exported from a module and imported into a client by default (`our @EXPORT`).

- All this `maxfreq::remember` stuff is rather longwinded, in most languages the module designer can *control the interface* - choose which (public) symbols to export, and the module user can choose which symbols to import.
- How do we do it in Perl? By using a Perl special module called `Exporter` which exists for precisely this purpose:
- `Exporter` defines three conceptual sets, which must be defined by `our` variables:
  - The set of symbols exported from a module and imported into a client by default (`our @EXPORT`).
  - The set of additional symbols exported from a module which a client can choose to import (`our @EXPORT_OK`).

- All this `maxfreq::remember` stuff is rather longwinded, in most languages the module designer can *control the interface* - choose which (public) symbols to export, and the module user can choose which symbols to import.
- How do we do it in Perl? By using a Perl special module called `Exporter` which exists for precisely this purpose:
- `Exporter` defines three conceptual sets, which must be defined by `our` variables:
  - The set of symbols exported from a module and imported into a client by default (`our @EXPORT`).
  - The set of additional symbols exported from a module which a client can choose to import (`our @EXPORT_OK`).
  - The set of named *tags*, each of which represents a set of symbols which may be imported via the tag name (`our %EXPORT_TAGS`).

- All this `maxfreq::remember` stuff is rather longwinded, in most languages the module designer can *control the interface* - choose which (public) symbols to export, and the module user can choose which symbols to import.
- How do we do it in Perl? By using a Perl special module called `Exporter` which exists for precisely this purpose:
- `Exporter` defines three conceptual sets, which must be defined by `our` variables:
  - The set of symbols exported from a module and imported into a client by default (`our @EXPORT`).
  - The set of additional symbols exported from a module which a client can choose to import (`our @EXPORT_OK`).
  - The set of named *tags*, each of which represents a set of symbols which may be imported via the tag name (`our %EXPORT_TAGS`).
- We will cover the first two - see **perldoc Exporter** for all the gory details (tagged symbol sets, importing symbols matching a regex, etc).

- On the client side, we control what is imported via variations on the `use` syntax:

`use module;`	*import the default set of symbols* - everything on the module's `@EXPORT` list.
`use module ();`	*import no symbols.*
`use module qw(A B C);`	*import only symbols* `A, B` *and* `C` - these symbols must either be on the default list `@EXPORT` or the optional list `@EXPORT_OK`.
`use module qw(:DEFAULT A B C);`	*import the default set and symbols* `A, B` *and* `C` from the optional list `@EXPORT_OK`.

- On the client side, we control what is imported via variations on the `use` syntax:

`use module;`	*import the default set of symbols* - everything on the module's `@EXPORT` list.
`use module ();`	*import no symbols.*
`use module qw(A B C);`	*import only symbols* `A, B` *and* `C` - these symbols must either be on the default list `@EXPORT` or the optional list `@EXPORT_OK`.
`use module qw(:DEFAULT A B C);`	*import the default set and symbols* `A, B` *and* `C` from the optional list `@EXPORT_OK`.

- To make **maxfreq** an `Exporter` module, add:

```
use Exporter qw(import);

our @EXPORT = qw(forget remember getbest delbest);
our @EXPORT_OK = qw();
```

- On the client side, we control what is imported via variations on the `use` syntax:

`use module;`	*import the default set of symbols* - everything on the module's `@EXPORT` list.
`use module ();`	*import no symbols.*
`use module qw(A B C);`	*import only symbols* `A, B` *and* `C` - these symbols must either be on the default list `@EXPORT` or the optional list `@EXPORT_OK`.
`use module qw(:DEFAULT A B C);`	*import the default set and symbols* `A, B` *and* `C` from the optional list `@EXPORT_OK`.

- To make **maxfreq** an `Exporter` module, add:

```
use Exporter qw(import);

our @EXPORT = qw(forget remember getbest delbest);
our @EXPORT_OK = qw();
```

- You'll find the final exporter-friendly version of `maxfreq.pm` and `eg1` (with the `maxfreq::` prefixes removed) inside the tarball's `wordfreq-v4/` directory. Syntax check and rerun.

What can/should we Export?

- The *information hiding principle* says that you should hide as much as possible, exporting as little as possible,

- On the client side, we control what is imported via variations on the `use` syntax:

`use module;`	*import the default set of symbols* - everything on the module's `@EXPORT` list.
`use module ();`	*import no symbols.*
`use module qw(A B C);`	*import only symbols* `A, B` *and* `C` - these symbols must either be on the default list `@EXPORT` or the optional list `@EXPORT_OK`.
`use module qw(:DEFAULT A B C);`	*import the default set and symbols* `A, B` *and* `C` from the optional list `@EXPORT_OK`.

- To make **maxfreq** an `Exporter` module, add:

```
use Exporter qw(import);

our @EXPORT = qw(forget remember getbest delbest);
our @EXPORT_OK = qw();
```

- You'll find the final exporter-friendly version of `maxfreq.pm` and `eg1` (with the `maxfreq::` prefixes removed) inside the tarball's `wordfreq-v4/` directory. Syntax check and rerun.

What can/should we Export?

- The *information hiding principle* says that you should hide as much as possible, exporting as little as possible,
- A sensible recommendation is: *export only public subroutines.*

- When you've decided that something should be exported, there's still the choice of whether to export it by default (in `@EXPORT`) or optionally (in `@EXPORT_OK`).

- When you've decided that something should be exported, there's still the choice of whether to export it by default (in `@EXPORT`) or optionally (in `@EXPORT_OK`).
- The *namespace pollution principle* suggests that as little as possible should be in `@EXPORT`. Put most in `@EXPORT_OK`.
- The basic rule of thumb is that it should be "safe" to import the default set without causing problems.
- Name clashes: If two modules both export symbol X, and a single client script tries to import X from both modules, you get a perl warning: `Subroutine packagename::X redefined`; the second `X` is used!
- The client can always choose whether or not to import that symbol via specifying an import list. But it's particularly unpleasant if the client can no longer import the default set!

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.
- The main concepts involved here are *objects*, *classes*, *methods (object and class)* and *inheritance*. Here's a rough set of Perlish definitions:
  - A *class* is a Perl module, usually exporting nothing, containing class and object methods obeying the following conventions.

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.
- The main concepts involved here are *objects*, *classes*, *methods (object and class)* and *inheritance*. Here's a rough set of Perlish definitions:
  - A *class* is a Perl module, usually exporting nothing, containing class and object methods obeying the following conventions.
  - An *object* is some piece of reference data - usually a hashref - which remembers the name of it's own class. This is called a *blessed reference*.

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.
- The main concepts involved here are *objects*, *classes*, *methods (object and class)* and *inheritance*. Here's a rough set of Perlish definitions:
  - A *class* is a Perl module, usually exporting nothing, containing class and object methods obeying the following conventions.
  - An *object* is some piece of reference data - usually a hashref - which remembers the name of it's own class. This is called a *blessed reference*.
  - A *class method* (such as the *class constructor*) is a subroutine that takes the class name as it's first argument. The *class constructor* is conventionally called `new`, *destructors* are possible.

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.
- The main concepts involved here are *objects*, *classes*, *methods (object and class)* and *inheritance*. Here's a rough set of Perlish definitions:
  - A *class* is a Perl module, usually exporting nothing, containing class and object methods obeying the following conventions.
  - An *object* is some piece of reference data - usually a hashref - which remembers the name of it's own class. This is called a *blessed reference*.
  - A *class method* (such as the *class constructor*) is a subroutine that takes the class name as it's first argument. The *class constructor* is conventionally called `new`, *destructors* are possible.
  - An *object method* takes the object (`$self`) as the first argument.

- The basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour.
- We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.
- The main concepts involved here are *objects*, *classes*, *methods (object and class)* and *inheritance*. Here's a rough set of Perlish definitions:
  - A *class* is a Perl module, usually exporting nothing, containing class and object methods obeying the following conventions.
  - An *object* is some piece of reference data - usually a hashref - which remembers the name of it's own class. This is called a *blessed reference*.
  - A *class method* (such as the *class constructor*) is a subroutine that takes the class name as it's first argument. The *class constructor* is conventionally called `new`, *destructors* are possible.
  - An *object method* takes the object (`$self`) as the first argument.
  - *Single and multiple inheritance* are provided by a simple package search algorithm used to locate method subroutines.

```
package Person;
use strict;
use warnings;

my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26);
# the object constructor
sub new {
        my( $class, %arg ) = @_;
        my $obj      = bless( {}, $class );
        $obj->{NAME} = $arg{NAME} // $default{NAME};
        $obj->{SEX}  = $arg{SEX}  // $default{SEX};
        $obj->{AGE}  = $arg{AGE}  // $default{AGE};
        return $obj;
}

# get/set methods - set the value if given extra arg
sub name {
        my( $self, $value ) = @_;
        $self->{NAME} = $value if defined $value;
        return $self->{NAME};
}

sub sex {
        my( $self, $value ) = @_;
        $self->{SEX} = $value if defined $value;
        return $self->{SEX};
}

sub age {
        my( $self, $value ) = @_;
        $self->{AGE} = $value if defined $value;
        return $self->{AGE};
}

1;
```

- Here's **eg2**, the main program that uses **Person**:

```
use Person;

sub printperson ($)
{
        my( $person ) = @_;
        my $class = ref($person);
        my $name = $person->name;
        my $age = $person->age;
        my $sex = $person->sex;
        print "$class: name=$name, age=$age, sex=$sex\n";
}
my $dunc = Person->new( NAME => "Duncan",
                        AGE  => 42,
                        SEX  => "m" );
printperson( $dunc );
$dunc->age( 20 );
$dunc->name( "Young dunc" );
printperson( $dunc );
```

- Here's **eg2**, the main program that uses **Person**:

```
use Person;

sub printperson ($)
{
        my( $person ) = @_;
        my $class = ref($person);
        my $name = $person->name;
        my $age = $person->age;
        my $sex = $person->sex;
        print "$class: name=$name, age=$age, sex=$sex\n";
}
my $dunc = Person->new( NAME => "Duncan",
                        AGE  => 42,
                        SEX  => "m" );
printperson( $dunc );
$dunc->age( 20 );
$dunc->name( "Young dunc" );
printperson( $dunc );
```

- Note that the function `ref()` applied to a blessed reference returns the name of the package the reference was blessed into.

- Here's **eg2**, the main program that uses **Person**:

```
use Person;

sub printperson ($)
{
        my( $person ) = @_;
        my $class = ref($person);
        my $name = $person->name;
        my $age = $person->age;
        my $sex = $person->sex;
        print "$class: name=$name, age=$age, sex=$sex\n";
}
my $dunc = Person->new( NAME => "Duncan",
                        AGE  => 42,
                        SEX  => "m" );
printperson( $dunc );
$dunc->age( 20 );
$dunc->name( "Young dunc" );
printperson( $dunc );
```

- Note that the function `ref()` applied to a blessed reference returns the name of the package the reference was blessed into.
- When syntax checked and run, **eg2** produces:

```
Person: name=Duncan, age=42, sex=m
Person: name=Young dunc, age=20, sex=m
```

- Here's **eg2**, the main program that uses **Person**:

```
use Person;

sub printperson ($)
{
        my( $person ) = @_;
        my $class = ref($person);
        my $name = $person->name;
        my $age = $person->age;
        my $sex = $person->sex;
        print "$class: name=$name, age=$age, sex=$sex\n";
}
my $dunc = Person->new( NAME => "Duncan",
                        AGE  => 42,
                        SEX  => "m" );
printperson( $dunc );
$dunc->age( 20 );
$dunc->name( "Young dunc" );
printperson( $dunc );
```

- Note that the function `ref()` applied to a blessed reference returns the name of the package the reference was blessed into.
- When syntax checked and run, **eg2** produces:

```
Person: name=Duncan, age=42, sex=m
Person: name=Young dunc, age=20, sex=m
```

- But why did we put `printperson` in the main program - it obviously should have been a method in class `Person`! How hard is converting a normal subroutine to a method?

- Perl has an advanced feature called *operator overloading*. We can use this to specify how to automatically convert a `Person` object to a string.

- Perl has an advanced feature called *operator overloading*. We can use this to specify how to automatically convert a `Person` object to a string.
- First, split out the formatting code from `printperson` into a separate method called `as_string`:

```
# new method
sub as_string {
      my( $self ) = @_;
      my $class = ref($self);
      my $name = $self->name;
      my $age = $self->age;
      my $sex = $self->sex;
      return "$class: name=$name, age=$age, sex=$sex\n";
}
```

- Perl has an advanced feature called *operator overloading*. We can use this to specify how to automatically convert a `Person` object to a string.
- First, split out the formatting code from `printperson` into a separate method called `as_string`:

```
# new method
sub as_string {
      my( $self ) = @_;
      my $class = ref($self);
      my $name = $self->name;
      my $age = $self->age;
      my $sex = $self->sex;
      return "$class: name=$name, age=$age, sex=$sex\n";
}
```

- Now, `printperson` is simply:

```
print $self->as_string;
```

- Perl has an advanced feature called *operator overloading*. We can use this to specify how to automatically convert a `Person` object to a string.
- First, split out the formatting code from `printperson` into a separate method called `as_string`:

```
# new method
sub as_string {
      my( $self ) = @_;
      my $class = ref($self);
      my $name = $self->name;
      my $age = $self->age;
      my $sex = $self->sex;
      return "$class: name=$name, age=$age, sex=$sex\n";
}
```

- Now, `printperson` is simply:

```
print $self->as_string;
```

- Now, add the magic pragma:

```
use overload '""' => \&as_string;
```

and the `printperson` method becomes:

```
print $self;
```

- Perl has an advanced feature called *operator overloading*. We can use this to specify how to automatically convert a `Person` object to a string.
- First, split out the formatting code from `printperson` into a separate method called `as_string`:

```
# new method
sub as_string {
      my( $self ) = @_;
      my $class = ref($self);
      my $name = $self->name;
      my $age = $self->age;
      my $sex = $self->sex;
      return "$class: name=$name, age=$age, sex=$sex\n";
}
```

- Now, `printperson` is simply:

```
print $self->as_string;
```

- Now, add the magic pragma:

```
use overload '""' => \&as_string;
```

  and the `printperson` method becomes:

```
print $self;
```

- Now, *delete the* `printperson` *method entirely* - **eg2** can now simply print each object itself.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.
- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.
- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.
  - Otherwise, perform a depth first search of the first parent class named in `@ISA`.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.
- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.
  - Otherwise, perform a depth first search of the first parent class named in `@ISA`.
  - If still not found, perform a depth first search of the second parent class named in `@ISA`.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.
- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.
  - Otherwise, perform a depth first search of the first parent class named in `@ISA`.
  - If still not found, perform a depth first search of the second parent class named in `@ISA`.
  - And so on through the remaining `@ISA` elements.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.
- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.
  - Otherwise, perform a depth first search of the first parent class named in `@ISA`.
  - If still not found, perform a depth first search of the second parent class named in `@ISA`.
  - And so on through the remaining `@ISA` elements.
  - If still not found, report an error.

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:
- A Perl class can name one or more parent classes as an ordered list, in package variable `our @ISA`.
- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. Here's the method search algorithm for a method (say `hello`):
  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.
  - Otherwise, perform a depth first search of the first parent class named in `@ISA`.
  - If still not found, perform a depth first search of the second parent class named in `@ISA`.
  - And so on through the remaining `@ISA` elements.
  - If still not found, report an error.
- Note that this search algorithm is even used for constructors - unlike many other OO languages, only one constructor method is called automatically. Do your own *constructor chaining*..

- Let's create a `Programmer` subclass of `Person`, with an additional property - a hashref storing language skills (each skill is a language name and an associated competence level).

- Let's create a `Programmer` subclass of `Person`, with an additional property - a hashref storing language skills (each skill is a language name and an associated competence level).
- It's good practice when subclassing to check that an empty subclass doesn't break things, before adding new stuff.

- Let's create a `Programmer` subclass of `Person`, with an additional property - a hashref storing language skills (each skill is a language name and an associated competence level).
- It's good practice when subclassing to check that an empty subclass doesn't break things, before adding new stuff.
- So, here's our *empty subclass version* of `Programmer`:

```
# stub class Programmer - reuse all methods!
package Programmer;
use strict; use warnings;
use Person;
our @ISA = qw(Person);
1;
```

- Let's create a `Programmer` subclass of `Person`, with an additional property - a hashref storing language skills (each skill is a language name and an associated competence level).
- It's good practice when subclassing to check that an empty subclass doesn't break things, before adding new stuff.
- So, here's our *empty subclass version* of `Programmer`:

```
# stub class Programmer - reuse all methods!
package Programmer;
use strict; use warnings;
use Person;
our @ISA = qw(Person);
1;
```

- Let's make **eg3** a copy of our final version of **eg2**, and then change both occurrences of `Person` to `Programmer`, i.e.:

```
use Programmer;
my $dunc = Programmer->new( NAME => "Duncan",
                            AGE  => 42,
                            SEX  => 'm' );
```

- Let's create a `Programmer` subclass of `Person`, with an additional property - a hashref storing language skills (each skill is a language name and an associated competence level).
- It's good practice when subclassing to check that an empty subclass doesn't break things, before adding new stuff.
- So, here's our *empty subclass version* of `Programmer`:

```
# stub class Programmer - reuse all methods!
package Programmer;
use strict; use warnings;
use Person;
our @ISA = qw(Person);
1;
```

- Let's make **eg3** a copy of our final version of **eg2**, and then change both occurrences of `Person` to `Programmer`, i.e.:

```
use Programmer;
my $dunc = Programmer->new( NAME => "Duncan",
                            AGE  => 42,
                            SEX  => 'm' );
```

- What do we expect to happen? It should work just like before, but the object should know that it's a `Programmer`! After syntax checking, run **eg3** to see what happens:

```
Programmer: name=Duncan, age=42, sex=m
Programmer: name=Young dunc, age=20, sex=m
```

- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`@Programmer::ISA = (Person)`, so `Person` is first parent
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`@Programmer::ISA = (Person)`, so `Person` is first parent
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- `Person::new` is called with the arguments:

```
$class = "Programmer";
%arg = ( "NAME" => "Duncan", "AGE" => 42, "SEX" => "m" );
```

and then creates a new object, `blesses` it into package `$class` (i.e. `"Programmer"`), initializes it, and finally returns it.

- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`@Programmer::ISA = (Person)`, so `Person` is first parent
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- `Person::new` is called with the arguments:

```
$class = "Programmer";
%arg = ( "NAME" => "Duncan", "AGE" => 42, "SEX" => "m" );
```

  and then creates a new object, `blesses` it into package `$class` (i.e. `"Programmer"`), initializes it, and finally returns it.

- All the ordinary method calls are handled in the same way, each time they are tried in package `Programmer`, found not to exist, and then tried (and found) in package `Person`.

- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`@Programmer::ISA = (Person)`, so `Person` is first parent
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- `Person::new` is called with the arguments:

```
$class = "Programmer";
%arg = ( "NAME" => "Duncan", "AGE" => 42, "SEX" => "m" );
```

  and then creates a new object, `blesses` it into package `$class` (i.e. `"Programmer"`), initializes it, and finally returns it.
- All the ordinary method calls are handled in the same way, each time they are tried in package `Programmer`, found not to exist, and then tried (and found) in package `Person`.
- Note that stringifying our object for printing still works - so even the stringification overloading must be inherited properly.

- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`@Programmer::ISA = (Person)`, so `Person` is first parent
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- `Person::new` is called with the arguments:

```
$class = "Programmer";
%arg = ( "NAME" => "Duncan", "AGE" => 42, "SEX" => "m" );
```

  and then creates a new object, `blesses` it into package `$class` (i.e. `"Programmer"`), initializes it, and finally returns it.
- All the ordinary method calls are handled in the same way, each time they are tried in package `Programmer`, found not to exist, and then tried (and found) in package `Person`.
- Note that stringifying our object for printing still works - so even the stringification overloading must be inherited properly.
- Ok, now let's start really implementing **Programmer**.

- Add a new `skills` method and override `as_string`:

```
package Programmer;
use strict; use warnings;
use Person;
our @ISA = qw(Person);

sub skills {                    # additional get/set method
        my( $self, $value ) = @_;
        $self->{SKILLS} = $value if defined $value;
        return $self->{SKILLS};
}

sub skills_as_string {          # additional method
        my( $self ) = @_;
        my $sk = $self->skills;
        my @str = map {
                sprintf( "%s:%s", $_, $sk->{$_} )
        } sort(keys(%$sk));
        return "{" . join(", ", @str) . "}";
}

use overload '"""' => \&as_string;
sub as_string {                 # override method
        my( $self ) = @_;
        my $pers = $self->Person::as_string;
        my $skills = $self->skills_as_string;
        return "$pers\t    skills=$skills\n";
}

1;
```

- Add a new `skills` method and override `as_string`:

```
package Programmer;
use strict; use warnings;
use Person;
our @ISA = qw(Person);

sub skills {                    # additional get/set method
        my( $self, $value ) = @_;
        $self->{SKILLS} = $value if defined $value;
        return $self->{SKILLS};
}

sub skills_as_string {          # additional method
        my( $self ) = @_;
        my $sk = $self->skills;
        my @str = map {
                sprintf( "%s:%s", $_, $sk->{$_} )
        } sort(keys(%$sk));
        return "{" . join(", ", @str) . "}";
}

use overload '""' => \&as_string;
sub as_string {                 # override method
        my( $self ) = @_;
        my $pers = $self->Person::as_string;
        my $skills = $self->skills_as_string;
        return "$pers\t    skills=$skills\n";
}

1;
```

- `$self->Person::as_string` is an example of *method chaining*, which does a normal method call to `Person::as_string`.

- Here's our test harness **eg3a** which uses some of the new features:

```
use strict;
use warnings;
use Programmer;

my $dunc = Programmer->new( NAME   => "Duncan",
                            AGE    => 42,
                            SEX    => "m",
                            SKILLS => {
                                "C" => "godlike",
                                "perl" => "godlike",
                                "C++" => "ok",
                                "pascal" => "good",
                                "java" => "minimal"
                            } );
print $dunc;
$dunc->age( 20 );
$dunc->name( "Young dunc" );
$dunc->skills( { "C" => "good", "pascal" => "ok" } );
print $dunc;
```

- Here's our test harness **eg3a** which uses some of the new features:

```
use strict;
use warnings;
use Programmer;

my $dunc = Programmer->new( NAME   => "Duncan",
                            AGE    => 42,
                            SEX    => "m",
                            SKILLS => {
                                "C" => "godlike",
                                "perl" => "godlike",
                                "C++" => "ok",
                                "pascal" => "good",
                                "java" => "minimal"
                            } );
print $dunc;
$dunc->age( 20 );
$dunc->name( "Young dunc" );
$dunc->skills( { "C" => "good", "pascal" => "ok" } );
print $dunc;
```

- When syntax checked and run, **eg3a** produces:

```
Programmer: name=Duncan, age=42, sex=m
            skills={}
Programmer: name=Young dunc, age=20, sex=m
            skills={C:good, pascal:ok}
```

- Here's our test harness **eg3a** which uses some of the new features:

```
use strict;
use warnings;
use Programmer;

my $dunc = Programmer->new( NAME   => "Duncan",
                            AGE    => 42,
                            SEX    => "m",
                            SKILLS => {
                                "C" => "godlike",
                                "perl" => "godlike",
                                "C++" => "ok",
                                "pascal" => "good",
                                "java" => "minimal"
                            } );
print $dunc;
$dunc->age( 20 );
$dunc->name( "Young dunc" );
$dunc->skills( { "C" => "good", "pascal" => "ok" } );
print $dunc;
```

- When syntax checked and run, **eg3a** produces:

```
Programmer: name=Duncan, age=42, sex=m
            skills={}
Programmer: name=Young dunc, age=20, sex=m
            skills={C:good, pascal:ok}
```

- But... this is awful! Where have all Duncan's skills gone? Answers on a postcard please:-)

- The problem is that `Person::new` has no code to initialize a `SKILLS` field. And nor should it!

- The problem is that `Person::new` has no code to initialize a `SKILLS` field. And nor should it!
- So we must define our own `Programmer` constructor. The following would definitely work, by repeating all of `Person::new`'s initializations:

```
my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"});
sub new {                          # the object constructor
  my( $class, %arg ) = @_;
  my $self        = bless( {}, $class );
  $self->{NAME}   = $arg{NAME}   // $default{NAME};
  $self->{SEX}    = $arg{SEX}    // $default{SEX};
  $self->{AGE}    = $arg{AGE}    // $default{AGE};
  $self->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
  return $self;
}
```

- The problem is that `Person::new` has no code to initialize a `SKILLS` field. And nor should it!
- So we must define our own `Programmer` constructor. The following would definitely work, by repeating all of `Person::new`'s initializations:

```
my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"});
sub new {                          # the object constructor
  my( $class, %arg ) = @_;
  my $self        = bless( {}, $class );
  $self->{NAME}   = $arg{NAME}   // $default{NAME};
  $self->{SEX}    = $arg{SEX}    // $default{SEX};
  $self->{AGE}    = $arg{AGE}    // $default{AGE};
  $self->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
  return $self;
}
```

- Here we're breaking a cardinal rule of programmers: **Don't Repeat Yourself** - this is prone to errors.

- The problem is that `Person::new` has no code to initialize a `SKILLS` field. And nor should it!
- So we must define our own `Programmer` constructor. The following would definitely work, by repeating all of `Person::new`'s initializations:

```
my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"});
sub new {                          # the object constructor
  my( $class, %arg ) = @_;
  my $self        = bless( {}, $class );
  $self->{NAME}   = $arg{NAME}   // $default{NAME};
  $self->{SEX}    = $arg{SEX}    // $default{SEX};
  $self->{AGE}    = $arg{AGE}    // $default{AGE};
  $self->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
  return $self;
}
```

- Here we're breaking a cardinal rule of programmers: **Don't Repeat Yourself** - this is prone to errors.
- What we really want is *constructor chaining* - use `Person::new`:

```
my %default = ( SKILLS => { java => "ok" } );
sub new {
  my( $class, %arg ) = @_;
  my $obj        = Person->new( %arg );
  $obj->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
  bless( $obj, $class );
  return $obj;
}
```

- Note that `Person->new(%arg)` creates a full-blown `Person` object, blessed into package `Person`, which we then modify - add the `SKILLS` field, and *re-bless the object into* `$class`.

- Note that `Person->new(%arg)` creates a full-blown `Person` object, blessed into package `Person`, which we then modify - add the `SKILLS` field, and *re-bless the object into* `$class`.
- Give this version (inside the tarball `programmer-v3/` dir) a try.

- Note that `Person->new(%arg)` creates a full-blown `Person` object, blessed into package `Person`, which we then modify - add the `SKILLS` field, and *re-bless the object into* `$class`.
- Give this version (inside the tarball `programmer-v3/` dir) a try.
- Isn't there a better way? Well, the only thing varying per-class appears to be the set of data fields which we want to initialize, and their default values. Even better, the data fields are just the keys of the default values. Remove `Programmer's` constructor, and generalise `Person's` constructor as follows:

```
sub new {
  my( $class, %arg ) = @_;
  my $obj    = bless( {}, $class );
  my %default = $obj->_defaultvalues;
  while( my($datum,$value) = each( %default ) )
  {
    $obj->{$datum} = $arg{$datum} // $value;
  }
  return $obj;
}
```

- Note that `Person->new(%arg)` creates a full-blown `Person` object, blessed into package `Person`, which we then modify - add the `SKILLS` field, and *re-bless the object into* `$class`.
- Give this version (inside the tarball `programmer-v3/` dir) a try.
- Isn't there a better way? Well, the only thing varying per-class appears to be the set of data fields which we want to initialize, and their default values. Even better, the data fields are just the keys of the default values. Remove `Programmer's` constructor, and generalise `Person's` constructor as follows:

```
sub new {
  my( $class, %arg ) = @_;
  my $obj    = bless( {}, $class );
  my %default = $obj->_defaultvalues;
  while( my($datum,$value) = each( %default ) )
  {
    $obj->{$datum} = $arg{$datum} // $value;
  }
  return $obj;
}
```

- Now, each class defines a private `_defaultvalues` method, listing the default values of all the initializable data fields:

```
sub Person::_defaultvalues {
  return (NAME=>"Shirley", SEX=>"f", AGE=>26);
}
```

- Continuing:

```
sub Programmer::_defaultvalues {
  return ( NAME=>"Shirley", SEX=>"f",
           AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- Continuing:

```
sub Programmer::_defaultvalues {
  return ( NAME=>"Shirley", SEX=>"f",
           AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.

- Continuing:

```
sub Programmer::_defaultvalues {
  return ( NAME=>"Shirley", SEX=>"f",
           AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.
- Can we fix this? Yes, with method chaining!

```
sub Programmer::_defaultvalues {
  my $self = shift;
  my %default = $self->Person::_defaultvalues;
  $default{SKILLS} = { java => "ok" };
  return %default;
}
```

- Continuing:

```
sub Programmer::_defaultvalues {
  return ( NAME=>"Shirley", SEX=>"f",
           AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.
- Can we fix this? Yes, with method chaining!

```
sub Programmer::_defaultvalues {
  my $self = shift;
  my %default = $self->Person::_defaultvalues;
  $default{SKILLS} = { java => "ok" };
  return %default;
}
```

- More generically, we can write the chained method call as:

```
my %default = $self->SUPER::_defaultvalues;
```

to call the first available parental `_defaultvalues` method.

- Continuing:

```
sub Programmer::_defaultvalues {
  return ( NAME=>"Shirley", SEX=>"f",
           AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.
- Can we fix this? Yes, with method chaining!

```
sub Programmer::_defaultvalues {
  my $self = shift;
  my %default = $self->Person::_defaultvalues;
  $default{SKILLS} = { java => "ok" };
  return %default;
}
```

- More generically, we can write the chained method call as:

```
my %default = $self->SUPER::_defaultvalues;
```

  to call the first available parental `_defaultvalues` method.
- That's enough OO for now! Happy OO Perl programming.

- When you say `use module` where does Perl search for the file `module.pm`?

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).
- Typically, then, we build a Perl program and some associated modules, and then want to:
  - Install the program into (say) `/homes/dcw/bin`.

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).
- Typically, then, we build a Perl program and some associated modules, and then want to:
  - Install the program into (say) `/homes/dcw/bin`.
  - Ensure that `/homes/dcw/bin` is on our path.

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).
- Typically, then, we build a Perl program and some associated modules, and then want to:
  - Install the program into (say) `/homes/dcw/bin`.
  - Ensure that `/homes/dcw/bin` is on our path.
  - Install the modules into (say) `/homes/dcw/perllib`.

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).
- Typically, then, we build a Perl program and some associated modules, and then want to:
  - Install the program into (say) `/homes/dcw/bin`.
  - Ensure that `/homes/dcw/bin` is on our path.
  - Install the modules into (say) `/homes/dcw/perllib`.
  - And have the program *know where to find the modules*.

- When you say `use module` where does Perl search for the file `module.pm`?
- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).
- Typically, then, we build a Perl program and some associated modules, and then want to:
  - Install the program into (say) `/homes/dcw/bin`.
  - Ensure that `/homes/dcw/bin` is on our path.
  - Install the modules into (say) `/homes/dcw/perllib`.
  - And have the program *know where to find the modules*.
- Perl has a list of locations that it searches, called the *include path*. The include path is available within a Perl script as the special variable `@INC`.

- You can add an extra directory (`/homes/dcw/perllib` for example) to the include path in two ways:
  - Run your Perl script via:

```
perl -I/homes/dcw/perllib ...
```

- You can add an extra directory (`/homes/dcw/perllib` for example) to the include path in two ways:
  - Run your Perl script via:

    ```
    perl -I/homes/dcw/perllib ...
    ```

  - Alternatively, near the top of your script, add:

    ```
    use lib qw(/homes/dcw/perllib);
    ```

- You can add an extra directory (`/homes/dcw/perllib` for example) to the include path in two ways:
  - Run your Perl script via:

    ```
    perl -I/homes/dcw/perllib ...
    ```

  - Alternatively, near the top of your script, add:

    ```
    use lib qw(/homes/dcw/perllib);
    ```

- This works, but it's a real pain to move to another location - because you have to change all references to `/homes/dcw/perllib` to (say) `/vol/project/XYZ/lib`.

- You can add an extra directory (`/homes/dcw/perllib` for example) to the include path in two ways:
  - Run your Perl script via:

    ```
    perl -I/homes/dcw/perllib ...
    ```

  - Alternatively, near the top of your script, add:

    ```
    use lib qw(/homes/dcw/perllib);
    ```

- This works, but it's a real pain to move to another location - because you have to change all references to `/homes/dcw/perllib` to (say) `/vol/project/XYZ/lib`.
- This becomes a serious program as your applications grow larger; imagine an application comprising 10 main programs and 50 support modules.

- You can add an extra directory (`/homes/dcw/perllib` for example) to the include path in two ways:
  - Run your Perl script via:

    ```
    perl -I/homes/dcw/perllib ...
    ```

  - Alternatively, near the top of your script, add:

    ```
    use lib qw(/homes/dcw/perllib);
    ```

- This works, but it's a real pain to move to another location - because you have to change all references to `/homes/dcw/perllib` to (say) `/vol/project/XYZ/lib`.
- This becomes a serious program as your applications grow larger; imagine an application comprising 10 main programs and 50 support modules.
- We'd like a *position independent* way of specifying where to find the modules. The standard Perl module `FindBin` helps:

```
use FindBin qw($Bin);
use lib qw($Bin/../perllib);
use MyModule;
```

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (useful for CGI programs generating dynamic images).

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (useful for CGI programs generating dynamic images).
- *Parser generators* using Perl - especially the awesome yacc-like module `Parse::RecDescent`.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (useful for CGI programs generating dynamic images).
- *Parser generators* using Perl - especially the awesome yacc-like module `Parse::RecDescent`.
- *Perl threads* - semaphores, thread queues etc.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (useful for CGI programs generating dynamic images).
- *Parser generators* using Perl - especially the awesome yacc-like module `Parse::RecDescent`.
- *Perl threads* - semaphores, thread queues etc.
- *Interfacing external C libraries into Perl* via `XS`, and embedding a Perl interpreter in other programs, eg. Apache and mod_perl.

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - *The old* `local` *keyword* - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - *Autoloading* - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the *Perl debugger* (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (useful for CGI programs generating dynamic images).
- *Parser generators* using Perl - especially the awesome yacc-like module `Parse::RecDescent`.
- *Perl threads* - semaphores, thread queues etc.
- *Interfacing external C libraries into Perl* via `XS`, and embedding a Perl interpreter in other programs, eg. Apache and mod_perl.
- And lots lots more.... Perl 5.10 new features, Perl 6, Parrot..

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.
- Our old friend **CPAN**, found at `http://www.cpan.org/`.

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.
- Our old friend **CPAN**, found at `http://www.cpan.org/`.
- The wonderful *Perl Journal* at `http://tpj.com/` which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.
- Our old friend **CPAN**, found at `http://www.cpan.org/`.
- The wonderful *Perl Journal* at `http://tpj.com/` which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.
- *The Perl Directory* at `http://www.perl.org/` is a directory of links to other Perl information and news.

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.
- Our old friend **CPAN**, found at `http://www.cpan.org/`.
- The wonderful *Perl Journal* at `http://tpj.com/` which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.
- *The Perl Directory* at `http://www.perl.org/` is a directory of links to other Perl information and news.
- *The Perl Monks* at `http://www.perlmonks.org/` is a forum-based discussion site for all matters Perlish.

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.
- Our old friend **CPAN**, found at `http://www.cpan.org/`.
- The wonderful *Perl Journal* at `http://tpj.com/` which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.
- *The Perl Directory* at `http://www.perl.org/` is a directory of links to other Perl information and news.
- *The Perl Monks* at `http://www.perlmonks.org/` is a forum-based discussion site for all matters Perlish.
- That's all folks! Enjoy your Perl programming - and remember the Perl motto:
  There's More Than One Way To Do It!