# Modularity and Reusability II

Packages and modules in Perl



# Libraries

- Imagine a world without libraries. Suppose instead that the only way to find a particular book in which you were interested was to look in all of the bookstores in your neighborhood, and then in the city, and then beyond, until you found it. Even then, you'd have to buy it instead of borrowing it.
- A library is an ecologically friendly way to share books; people know where to look for them so they do not waste time and energy,  much fewer copies of books are needed, thereby saving paper, and people do not have to spend money that they could use on other things.

# Function libraries

- Similar reasoning applies to functions.
- If you have written functions that you think you ought to save for a rainy day, you should put them someplace where you, and perhaps your friends, or even strangers, can find them.
- You could put all related functions into a single file, and if the structure of that file were standardized in a way that all programs would know how to fund the functions in it, then Perl programs that people wrote could ***borrow*** the functions defined in your file. (But unlike a book, a function can be borrowed in a reusable way -- it can be borrowed simultaneously by an unlimited number of programs.)

# Creating function libraries

- The questions that must be are:
  (1) How can we create a library of functions?
  (2) How will programs know where to look for these libraries, once we learn how to create them?
  (3) How do we know what libraries are available already, so that we do not "***reinvent the wheel***?"
- The answer to Question (2) lies in some Perl magic and in how you set up your shell environment. Question (3)'s answer comes next. The answer to Question (1) comes after that.

# Perl modules

- In Perl, function libraries are called ***modules***.
- Dozens of Perl modules are installed automatically when you install Perl on your machine.
- If you installed ***ActivePerl***, you can read the ActivePerl documentation to see the list of standard libraries, the ones that are typically installed by default. See the **`perlmodlib`** page.



# Perl modules (2)

- To see the list of ***all*** modules installed on your computer, you can enter following complicated command to the shell, ***all on one line***, (it is line-wrapped on this slide)

```
% perl -MFile::Find=find -MFile::Spec::Functions
  -Tlwe 'find { wanted => sub { print canonpath $_
  if /\.pm\z/ }, no_chdir => 1 }, @INC'
```

- Instead, I put this into a shell script called **`listperlmods`** and just type **`listperlmods`** instead.

# Learning about an installed module

- To learn what a particular installed module does, type

  **`perldoc modulename`**

  at the command line. You do not need the **`.pm`** extension on the module's filename. For example,

  **`perldoc English`**

  will display a description like a man page for the **`English`** module. Not all modules have documentation, but the standard ones do.

- Again, if you have ***ActivePerl*** installed, you can also read its documentation. See the **`perlmodlib`** page.

# Using an installed module

- If you find a module that you want to ***use*** in a program, you put the

  ```
  use ModuleName;
  ```

  declaration into your program, where ***ModuleName*** is the name of the module without the `.pm`. For example,

  ```
  use English;
  ```

  will import the **`English`** module into your program.

- Soon I will explain more about the ***use declaration***.

# Other available modules

- Hundreds modules are freely available on the Internet.
- The Comprehensive Perl Archive Network (CPAN) is an archive of Perl modules and scripts contributed by members of the Perl community. You can browse the collection of modules at http://www.perl.com/CPAN to see what is available or search for a specific module using the search engine on the site.

# Obtaining modules

- If you find a module that you would like to use in your programs, you need to download it and install it on your computer.
- All of the modules on the CPAN site are compressed archive files that contain self-installation scripts. There are instructions for how to install these modules on the CPAN page

  http://www.perl.com/CPAN/modules/INSTALL.html
- An alternative to visiting the CPAN site, on a UNIX machine with Perl installed, is to run a CPAN shell using the **`CPAN.pm`** module.

# CPAN.pm

- You can run a CPAN shell program by entering the command

  ```
  perl -MCPAN -e shell
  ```

- Answer yes to all of the questions it asks. It will create a `.cpan` directory in your home directory and fill it with various configuration files. It will eventually issue a prompt:

  ```
  cpan[1]>
  ```

- Type "**h**" for help. You will see that there are commands for searching for modules and doing many other things.

# Searching for modules

- You can search for modules on the CPAN site with the `m` command, which lets you enter a Perl regular expression, e.g.

```
cpan[1]> m /protein/
```

- The CPAN shell will list modules that match your search expression. One in the list might be

```
Bio::Align::ProteinStatistics
```

- To download a module, such as the above, use get:

```
get Bio::Align::ProteinStatistics
```

which will download files to your `.cpan` directory.



# Installing the module

- The steps to install the downloaded module are to run three CPAN commands

```
make Bio::Align::ProteinStatistics
test Bio::Align::ProteinStatistics
install Bio::Align::ProteinStatistics
```

- The first customizes the module to your environment and creates necessary files from the templates in the module. The second tests to make sure everything was built properly. The third puts the module and its man pages in the appropriate directories on your machine.

# Problems using CPAN.pm

- Unless this is your own machine and you can obtain root permissions, you will probably encounter problems installing the module using the CPAN shell.
- The make files will require root permissions to install the module in all likelihood.
- But you can still use the CPAN shell for searching, and then visit the CPAN site to download and install the module yourself manually, as is described next.

# Manually installing modules

- The simpler method is to download the module directly from the CPAN site, unpack the module (which is either a **`.tar.gz`** file for UNIX or a **`.zip`** on Windows), and **`cd`** into the unpacked directory, where you will run the commands:

```
mkdir ~/perllib
perl Makefile.pm PREFIX=~/perllib
make
make test
make install
```

- These will install the module in your home directory in the directory **`perllib`**.

# How programs find modules

- The previous slide suggested that you create a directory **`~/perllib`** into which you could put any modules that you download and install.
- How will your program find that directory? Not by magic.
- Perl maintains an array called **`@INC`** (which is short for "***includes***"), that is a list of directories in which it searches for modules that your program uses via the **`use`** declaration. We can see what this list is by typing

  **`perl -V`**

  at the command line.

# The `@INC` array

- Alternatively, you can run the following Perl script:

```
print join("\n", @INC )
```

  which prints the directories in the `@INC` array, one per line.

- You will not find `~/perllib` in the list. To get Perl to look there, you can either add a command line switch when running Perl, or better, add the line

```
PERL5LIB=$HOME/perllib
```

- to your `.bashrc` file. Remember to run your `.bashrc` file (by typing `. ~/.bashrc`) to make the change immediate.

# Creating your own module

- We have answered Questions (2) and (3) from Slide 4. The question remains,  how can you create a module?
- We could answer this question very mechanically, with a list of steps, but you ought to understand a bit about why these steps are necessary.
- We begin by explaining the concept of a ***package***.

# Modules are packages

- A module is a set of related functions in a file, not just any file, but a special kind of file called a ***package***.
- In order to understand how to create modules, you first need to understand packages.
- Perl packages solve the problem of ***name conflicts*** in programs. We begin there.

# Name conflicts

- Imagine that we have figured out how to create files with borrowable functions, and that programs can somehow "import" these functions to within their "borders."
- What would happen if the functions used the same variable names as the program.
- For example, what if the function file used a variable **`$count`** to keep track of fruit flies on the loose, and your program also had a variable named **`$count`** that counted bananas. Then as your program increased **`$count`**, the imported functions might think there was a fruit fly plague.

# Namespaces

- Many modern programming languages utilize the concept of a ***namespace*** to avoid name conflicts. A ***namespace*** is a container that stores names that are unique within the container.
- For example, ***Music*** could be a namespace containing ***bow*** and ***Ships*** could be a different namespace also containing ***bow***.
- To refer to the bow from Music, we would write ***Music::bow***, and to refer to the bow from Ships, we would write ***Ships::bow***.
-

# Packages

- In Perl, a namespace is called a ***package***. A ***package*** encapsulates the names declared within it. Every package has a ***symbol table*** that contains the names declared within it, as well as information about those names.
- Your Perl program is automatically called **`package main`**.
- You create a package using the package declaration:

  **`package`** ***`PACKAGENAME`*****`;`**

  where ***`PACKAGENAME`*** is a legal Perl identifier that becomes the name of your package
- Everything from that point until the end of the innermost enclosing block or the next package declaration becomes part of the package.

# Package example

- The following declares a package named **SimpleStuff** containing two function definitions.

```
package SimpleStuff;
sub pi {return atan2(1.0, 1.0)*4;}
sub e  {return exp(1.0); }
```

- The functions **pi** and **e** are contained in **SimpleStuff** and could be called using the notation

```
SimpleStuff::pi();
SimpleStuff::e();
```

# Packages and name conflicts

- Suppose that we have two packages named **`Euclidean`** and **`Manhattan`**, and each contains a function named **`distance`**. A program that imports these functions for its own use (we'll see how shortly) would distinguish between them using the *qualified names*

```
Euclidean::distance;
Manhattan::distance;
```

- In other words, the package name becomes part of the function name, separated by the "`::`".

# Where do package declarations go?

- A package declaration can be placed anywhere that an ordinary statement can be placed, but since our objective is to build a function library, we want to put our package declaration in its own separate file (and put that file in the **`perllib`** directory in our home directory.)
- Although we could put a  package declaration in the program file, that would not make our package reusable.
- Therefore, we outline the steps for creating a package in a separate file.

# How to create a package file

- Start the file with the **`package`** declaration, e.g.

```
package foo;
```

- Put the function definitions into the file. If these functions need to use any common shared variables, make them lexical variables (declared with the **`my`** operator).
- Although package files can have any name, since our goal is to create a module, you must give the file the same name as the package itself, with a "**`.pm`**" extension. In our case, the file should be named **`foo.pm`**.

# Example: the Euclidean package

- We create a package file named **`Euclidean.pm`** with a single function in package **`Euclidean`**:

```
package Euclidean;
sub distance
{
  my ($point1, $point2) = @_;
  my @point1 = @$point1;
  my @point2 = @$point2;
  my $distance = 0;
  for ( my $i = 0; $i < @point1; $i++ ) {
     $distance += ($point2[$i] - $point1[$i])**2;
  }
  return sqrt($distance);
}
```

# Using a package in a program

- There are two ways to include a package or a module in a program: the **`use`** declaration and the **`require`** declaration.
- The **`require`** declaration has slightly different meaning than use: ***the `require` declaration brings the package in at runtime, whereas `use` brings it in at compile-time***.
- To bring our Euclidean package with **`require`**, we would put the line

```
require "Euclidean.pm";
```

in the beginning of the program.

# Using a package in a program (2)

- Alternatively, we can write

```
use Euclidean;
```

- In either case, to use the **`distance`** function, we would write something like

```
$dist = Euclidean::distance(\@point1,\@point2);
```

where the arguments are passed as array references.

# Importing functions

- At this point, the only way to use the distance function from Euclidean in our program is to qualify its name:

```
Euclidean::distance( )
```

- If we want to add the distance function to the main program's namespace so that we do not have to qualify it, i.e., so we can write:

```
$dist = distance(\@point1,\@point2);
```

then we need to ***import*** the function from the package into the program.

# Importing functions (2)

- To convert a package into a module that exports some or all of its functions (or other symbols such as variables), we need to add several lines to the beginning of the package.

```
package Euclidean;
use vars qw( @EXPORT @EXPORT_OK @ISA );
require Exporter;
@EXPORT_OK = qw ( distance );
@ISA = qw(Exporter);
```

- The **`use vars`** line declares a few global variables that we need for exporting symbols.

# Importing functions (3)

- The line

```
require Exporter;
```

  tells Perl that we need the **Exporter** module, which is a module that can export our function definitions for us to the programs that use our module.

- The line

```
@EXPORT_OK = qw ( distance );
```

  does the actual exporting. The **@EXPORT_OK** array is filled with the names of the functions that we want to export, in this case, only the distance function.

# Importing functions (4)

- The line

```
@ISA = qw(Exporter);
```

  helps Perl to find the functions in the **`Exporter`** module, since it is not located here. The actual meaning of this **`@ISA`** array is beyond the scope of this set of slides.

- We must modify the **`use`** declaration in the main program: We need to write

```
use Euclidean 'distance';
```

  instead of

```
use Euclidean;
```

  to tell Perl that we want to bring in the **`distance`** function in particular.

# The revised main program

- The main program can now call the **`distance`** function without qualification:

```
$dist = distance(\@point1,\@point2);
```

- Of course, if for some reason, you have two modules that each contain a function named **`distance`**, you should use the fully-qualified name anyway, so that you know which module's distance you are actually using. This is a pretty rare occurrence anyway.

# Export Methods

- We used the **@EXPORT_OK** array in the preceding module. This is the safe thing to do. The symbols listed in the **@EXPORT_OK** array will only be added to the main program's namespace if they are listed in the use declaration in a comma-separated list.
- If we put symbols into the **@EXPORT** array, in contrast, they are automatically brought into the main program's namespace, without an explicit request. This "pollutes" the main program with symbols it did not necessarily want. It is best to avoid it.



# What else?

- We have barely scratched the surface of the topic of modules. However, this is enough for you to write modules and use them.
- Nonetheless, there are a few things that we must cover. One is how to document the modules properly. Another is how to use Perl's **`h2xs`** program, which can set up skeleton modules for you. The latter is not really necessary, whereas the former is.
- The topic of documentation is covered in a separate set of slides.