Simple Variables

- In Perl, we refer to simple variables (i.e., something that holds one of something) as a *scalar*
- Scalars are easy to spot, they start with the "$" character

```
$total = 0;
```

- Aside: Just to be exact, note that the semicolon in Perl is a *statement terminator*
- Assigning a value to a scalar is easy (as shown above)
- Other examples of valid assignment are:

```
$total = $total + 1;     # increment the scalar total.
$subtotal = $total = 3;  # set the scalars subtotal & total to 3.
```

More on Scalars

- Note: there's no need to declare a scalar!!
- Scalars are case-sensitive, so `$total`, `$Total`, and `$TOTAL` all refer to *different* scalars, so be *very careful* …
- If you use `$Total` and you really mean `$total`, Perl will not complain (and your script will not work at all correctly)
- Some help is on hand, however, and is available via the `-w` switch:

```
#!/usr/bin/perl -w

while (<>)
{
    do something ...
}
```

More Scalar Examples

- A scalar can hold either a string or a number:

```
$hello = "Hello World!";
$prime = 13;
```

- The following is legal, but very dangerous:

```
$while = "Wait a while ... ";
```

- Imagine code like this (Yuk!):

```
while ($while < 6) { ... }
```

- Even though the "$" in front of `while` allows Perl to treat it as a scalar (as opposed to a while loop), for us humans, such a practice can only lead to *bad things happening* …

Scalars and Numbers

- It looks strange, but it is quite okay to do this in Perl:

```
$myVar = "Hello";
print $myVar, "\n";
$myVar = 42;
print $myVar, "\n";
```

- Remember: *Perl has no real notion of type*, so a scalar can contain any value, string, or number *at any time*

- By default, Perl treats all number as double precision floating point

- By placing "`use integer;`" near the top of your script, you can indicate that integer arithmetic should be used as the default (which can sometimes improve execution speed)

More Numbers

- Hex numbers can be written in the familiar C/C++/Java notation:

```
$myHexNumber = 0xffb2;
```

- Octal numbers can be written this way:

```
$my_octal_number = 0377;
```

- Long numbers can be made more readable by the use of the underscore, so:

```
1993245890
```

- can also be written as:

```
1_993_245_890
```

Perl Operators (1 of 3)

```
++     increment
--     decrement
**     exponential
~      complement
!      logical negation
*      multiplication
/      division
%      remainder
```

Perl Operators (2 of 3)

```
+      addition

-      subtraction

<<     shift left

>>     shift right

<      less than

>      greater than

<=     less than or equal to

>=     greater than or equal to

==     equal to
```

Perl Operators (3 of 3)

```
!=     not equal to

<=>    comparison

&      bitwise AND
|      bitwise OR
^      bitwise XOR

&&     logical AND
||     logical OR

=      assignment

not    logical negation
and    logical AND
or     logical OR
```

Perl's Built-in Arithmetic Functions

```
atan ($x, $y)
abs ($x)

cos ($x)
exp ($x)

int ($x)
log ($x)

rand ($x)
rand

sin ($x)
sqrt ($x)
```

Perl's Boolean Type

- There isn't one!
- A numerical value is considered `FALSE` if it evaluates to zero
- A string value is `FALSE` if it contains the empty string, or if it contains the single character `0` (but not `00+`)
- Note that (like C/C++/Java) testing for equality is performed by the `==` operator
- The single `=` operator is used for *assignment*
- To avoid confusion, think of `=` as meaning "becomes equal to"

Using Operators

`$this <=> $that`

- will return `-1`, `0`, or `1` depending on whether the value of the scalar `$this` is less than, equal to, or greater than the value of the scalar `$that`
- We refer to the && and `||` operators as lazy
- The second part of the operator expression is only evaluated if it needs to be (this is sometimes also referred to as "short circuiting")
- As these two operators return as their value the last operand they evaluated, this behaviour (side-effect) can sometimes be very important

Watch Out! Precedence About!

- The `not`, `and`, and `or` operators behave like the `!`, `&&`, and `||` operators except that their precedence is as low as it can go:

```
open DATAFILE, 'data.txt' or die "Could not open data file\n";
```

- will open the file `data.txt`, if it exists, and assign it to the filehandle `DATAFILE` - or - the program will die ...

- You may have been tempted to do the following:

```
open DATAFILE, 'data.txt' || die "Could not open data file\n";
```

- which, due to the *precedence rules* would be interpreted as:

```
open DATAFILE, ('data.txt' || die "Could not open data file\n");
```

More Assignment Stuff

- Note: an *assignment* is an *expression*, and returns as its value the left-hand-side of the statement

- Perl supports the composite operators familiar to C/C++/Java programmers:

```
$total += 2;  # add 2 to $total.
$times *= 3;  # multiply $times by 3.
```

- Interestingly, a *substitution* operation is also an expression - its result is the number of substitutions performed:

```
$howmany = s/teh/the/g;
```

- will put the number of times '`teh`' was replaced with '`the`' into a scalar called `$howmany`

Default Behaviours

- Scalars that are used in an arithmetic context are guaranteed to be initialized to zero
- To catch usage of uninitialized variables (scalars) in your code, use the `-w` option, i.e, `#!/usr/bin/perl -w`
- Before any variable is used, it has an *undefined* value
- You can test for this using the Perl function `defined`:

```
print $howmany if defined( $howmany );
```

- will only print the value of `$howmany` if it has been defined beforehand
- You can force a variable to be undefined by using the `undef` function

Strings in Perl

- Consider the following:

```
print "You changed teh to the $howmany times\n";
```

- This (above) is an example of an *interpolated* string, which is usually enclosed in double quotes (")

```
print 'You changed teh to the $howmany times\n';
```

- This (above) is an example of a *literal* string, which is usually enclosed in single quotes (')

- The difference between the two is in how Perl treats them

Literal Strings

- These are strings that are treated as is, where almost every character stands for itself, so:

```
print 'You changed teh to the $howmany times\n";
```

- will print out the string:

```
You changed teh to the $howmany times\n
```

- i.e., all the characters, *including the ones that have special meaning in Perl*, are treated literally, including the new-line sequence \n

- To include a new-line, put it into the string as follows:

```
print 'Here is a sentence that
has a new-line in it at the end of each line
';
```

Dealing with '

- What if we want to include a `'` in our literal string?

```
How's it going?
```

- Perl provides a method for dealing with this:

```
print q[How's it going?];
```

- The q introduces a *single-quoting* character of your choice which is used to delimit the string (this is the same idea as the matching `m` delimiter used with patterns)

```
print q!How's it going?!;
```

Interpolated Strings

- Special characters are interpreted then variables are replaced with their values prior to using the string
- So, if `$howmany` has a value of `42`, then:

```
print STDOUT "You changed teh to the $howmany times\n";
```

- will print:

```
You changed teh to the 42 times
```

- including the actual new-line character
- This process is referred to as *interpolation*
- Note: in the previous example, Perl (very kindly) converted the number `42` into the string `"42"`

Dealing with "

- Just like with literal strings, we can use a Perl method for dealing with " within our interpolated strings:

```
print qq[His real name is "Zorro", I swear!\n];
```

- will provide the desired effect, and can be referred to as *double-quoting*

- Like with q, qq can take any delimiting character of our choice:

```
print qq|His real name is "Zorro", I swear!\n|;
```

- will also work for us (because, in Perl, *There's More Than One Way To Do It!*)

Perl Gotcha

- Let's say you want to print out the following string:

```
Dublin is about 100km away
```

- and that the value `100` is in a scalar called `$distance`
- We could try this:

```
print "Dublin is about $distancekm away\n";
```

- but, Perl would print;

```
Dublin is about away
```

- due to the fact that `$distancekm` is *undefined* - we should have used:

```
print "Dublin is about ${distance}km away\n";
```

Working with Strings

- We can assign strings to scalars, as follows:

```
$greeting = q#Howya Doin'?#;

$greet_everyone = "$greeting, everyone!";

$next_input_line = <MYFILE>;
```

- Concatenation of strings is performed by `.` (i.e., dot):

```
$part1 = "Hello";
$part2 = "World!";

print $part1 . " " . $part2;
```

- will print out:

```
Hello World!
```

Building Strings Incrementally

- Here's our input file:

```
Paul is teaching us all about Perl.
Excellent!
Rather than looking at syntax, we are getting real work done!
Like ... cool, man!  Trippy ...
```

- Here's our script (what do we get as output?):

```
while (<>)
{
 /(^.)/;
 $initials .= $1;
}

print $initials, "\n";
```

Repetitive Strings

- We can *repeat* a string using the string repetition operator with is `x`:

```
print "Perl is Cool!\n" x 3;
```

- will display as output:

```
Perl is Cool!
Perl is Cool!
Perl is Cool!
```

- Warning: be careful when using `.` and `x` when working with strings. For example, this:

```
print 3.33, "\n";
```

- is not the same as (and does not mean), this:

```
print 3 . 33, "\n";
```

Comparing Strings

- <, <=, >, >=, ==, != and <=> do *arithmetic comparisons*, and can sometimes produce unexpected results when used with strings

- Perl provides a set of string comparison operators that use *lexicographical ordering*:

```
lt  -  less than
le  -  less than or equal to
gt  -  greater than
ge  -  greater than or equal to
eq  -  equals
ne  -  does not equal
cmp -  compare
```

Some (Built-In) String Functions

- Determining the length of strings:

```
$this = 'This is a long string';

print length( $this ), "\n";
```

- will print out the value `21`
- Refer to the Perl on-line documentation for more details on `length`:

```
man perlfunc
```

- will display information on all of Perl's *built-in functions*

Substrings

- Working with *substrings* has the following general form:

```
substr( $string, $offset, $count )
```

- where `$string` is the string you wish to work with, `$offset` is the location within the string you wish to start extracting the substring from, and `$count` is the size of the substring to extract, so:

```
$message = 'Take me to your leader';
$who = substr( $message, 5, 2 );
print 'The value of $who is ', $who, "\n";
```

- will print out:

```
The value of $who is me
```

- Note: that the value of the offset *starts counting at zero*

Chopping and Chomping

```
chop( $text );
```

- will take the value of `$text` and remove the last character from the end, regardless of what that character actually is

```
chomp( $line );
```

- will remove the last character from `$line` if, *and only if*, the character is a new-line
- So, what happens when we run the following script?

```
while (<>)
{
 chomp;
 print;
}
```

The Infamous $_ Variable

- In the last example, the `chomp` wasn't told what variable to `chomp` - it *magically* knew to work on the current line that we were working with!

- In Perl, if an explicit variable is not indicated, functions (as well as other things) operate on the built-in default variable, referred to as `$_`:

```
while (<>)
{
 chomp( $_ );
 print( $_ );
}
```

- If you put '`use English;`' at the top of your script, you can refer to this variable as `$ARG`

- Most Perl programmers prefer to use `$_`

$_ and Regular Expressions

- If you've been paying attention, you will have figured out that regular expressions work on `$_` by default:

```
print if /barryp/;
```

- But, what if we want to use our regular expressions with something other than `$_`?

- Perl has the answer: the `=~` operator (*string binding*):

```
print $line if $line =~ /barryp/;
```

- will print `$line` if it contains the pattern '`barryp`'

More Regular Expression Stuff

- We can also do things like this:

```
$line =~ s/teh/the/g;
```

- to do substitutions
- And we can do this:

```
$line =~ tr/A-Z/a-z/;
```

- to do translations on variables

Even More Regular Expression Stuff

- We can count the number of times a replacement is performed:

```
$line = 'this is a test of the teh teh the teh relacement';
$howmany = $line =~ s/teh/the/g;
print 'We changed teh to the ' . $howmany . ' times.', "\n";
```

- will produce the following output:

```
We changed teh to the 3 times.
```

- It is also possible to execute an expression as part of a pattern:

```
s!(\d+)km!($1*5/8).' miles'!ge;
```

- will convert all kilometre strings into miles strings - note the qualifier '`e`' at the end of the pattern, which stands for *expression*

A Complete Example: WYSIWYG

- Any idea what this small script does?

```
#!/usr/bin/perl -w

while (<>)
{
  chomp;
  s/^\W*//;
  $phrase = $_; # $ARG if we use: 'use English;'.
  $initials = '';
  while ($_)
  {
       s/^([\w']+)\W*//;
       $initials .= substr( $1, 0, 1 );
  }
  print "$phrase -> \U$initials\E\n";   # Uppercase.
}
```