Perl Short Course: Second Session

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

Dept of Computing,
Imperial College London

December 2012

- In this second session, we'll start going over Perl in more detail, going through most important parts of the language, with lots of practical examples for you to try later.

- In this second session, we'll start going over Perl in more detail, going through most important parts of the language, with lots of practical examples for you to try later.
- We'll broadly follow the order of material presented in Randal Schwartz's excellent **Learning Perl** (Llama book).

- In this second session, we'll start going over Perl in more detail, going through most important parts of the language, with lots of practical examples for you to try later.
- We'll broadly follow the order of material presented in Randal Schwartz's excellent **Learning Perl** (Llama book).
- We'll look at **scalar data** (numbers, strings, variables) and **expressions**, **control structures** and **input/output**.

- In this second session, we'll start going over Perl in more detail, going through most important parts of the language, with lots of practical examples for you to try later.
- We'll broadly follow the order of material presented in Randal Schwartz's excellent **Learning Perl** (Llama book).
- We'll look at **scalar data** (numbers, strings, variables) and **expressions**, **control structures** and **input/output**.
- Next session, we'll look at more complex bits of the language: **arrays and lists** and **hashes**, **special variables** and **regular expressions**.

- In this second session, we'll start going over Perl in more detail, going through most important parts of the language, with lots of practical examples for you to try later.
- We'll broadly follow the order of material presented in Randal Schwartz's excellent **Learning Perl** (Llama book).
- We'll look at **scalar data** (numbers, strings, variables) and **expressions**, **control structures** and **input/output**.
- Next session, we'll look at more complex bits of the language: **arrays and lists** and **hashes**, **special variables** and **regular expressions**.
- In the remaining sessions, we'll look at **functions**, **references**, **modules**, **objects and classes**, take a quick tour of some of Perl's standard library, find out how to write modules and classes, then investigate some more advanced Perl topics such as functional programming, program transformations, data structures, testing and benchmarking.

- **Perl 4**, 1991-1996: the first widely used version of the **Practical Extraction and Report Language**.

- **Perl 4**, 1991-1996: the first widely used version of the **Practical Extraction and Report Language**.
- **Perl 5**, 1994-now: Larry Wall completely rewrote the perl interpreter, and added significantly to the Perl language. **Perl 5** has been maintained, patched and extended for 18 years, with new features being added every year or two.

- **Perl 4**, 1991-1996: the first widely used version of the **Practical Extraction and Report Language**.
- **Perl 5**, 1994-now: Larry Wall completely rewrote the perl interpreter, and added significantly to the Perl language. **Perl 5** has been maintained, patched and extended for 18 years, with new features being added every year or two.
- **Perl 5.10**, 2003-2010: Larry and a band of developers worked on a new version of Perl 5 for several years and released **Perl 5.10** in 2007. Perl 5.10 is installed on DoC Windows machines. To find out the version installed, do **perl -v**.

- **Perl 4**, 1991-1996: the first widely used version of the **Practical Extraction and Report Language**.
- **Perl 5**, 1994-now: Larry Wall completely rewrote the perl interpreter, and added significantly to the Perl language. **Perl 5** has been maintained, patched and extended for 18 years, with new features being added every year or two.
- **Perl 5.10**, 2003-2010: Larry and a band of developers worked on a new version of Perl 5 for several years and released **Perl 5.10** in 2007. Perl 5.10 is installed on DoC Windows machines. To find out the version installed, do **perl -v**.
- The Perl 5 developers now do annual releases of Perl 5 with even numbers (odd versions are development releases). **Perl 5.12** was released in April 2010, **Perl 5.14** in May 2011 and **Perl 5.16** in May 2012. (DoC lab Linux machines have Perl 5.14 installed).

- **Perl 4**, 1991-1996: the first widely used version of the **Practical Extraction and Report Language**.
- **Perl 5**, 1994-now: Larry Wall completely rewrote the perl interpreter, and added significantly to the Perl language. **Perl 5** has been maintained, patched and extended for 18 years, with new features being added every year or two.
- **Perl 5.10**, 2003-2010: Larry and a band of developers worked on a new version of Perl 5 for several years and released **Perl 5.10** in 2007. Perl 5.10 is installed on DoC Windows machines. To find out the version installed, do **perl -v**.
- The Perl 5 developers now do annual releases of Perl 5 with even numbers (odd versions are development releases). **Perl 5.12** was released in April 2010, **Perl 5.14** in May 2011 and **Perl 5.16** in May 2012. (DoC lab Linux machines have Perl 5.14 installed).
- **Perl 6**, 2001-????: The Perl developers are also working on a fundamental redesign of Perl - **Perl 6** - very different internally from Perl 5. However, it never seems to get finished! Will it ever?

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).
  - An arbitrarily long string.

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).
  - An arbitrarily long string.
  - A reference to some other piece of data.

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).
  - An arbitrarily long string.
  - A reference to some other piece of data.
- Integer literals may be written in decimal, octal or hexadecimal: eg. `-129, 0377 and 0xffe0.`

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).
  - An arbitrarily long string.
  - A reference to some other piece of data.
- Integer literals may be written in decimal, octal or hexadecimal: eg. `-129, 0377 and 0xffe0.`
- Real literals may be written in *exponential notation*, with or without a *decimal point*: eg. `6.27, -2.4e30, 7e-5.`

- A scalar variable is declared via `my $variablename`, the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.
- You can also write a scalar variable as `${variablename}`, just like in the Unix shell.
- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).
  - An arbitrarily long string.
  - A reference to some other piece of data.
- Integer literals may be written in decimal, octal or hexadecimal: eg. `-129, 0377 and 0xffe0.`
- Real literals may be written in *exponential notation*, with or without a *decimal point*: eg. `6.27, -2.4e30, 7e-5.`
- A string literal can be either delimited with single-quotes or double-quotes. Different rules apply inside the two kinds of string:

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.
- Each character inside the quotes is placed into the string exactly as-is - even a literal newline!

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.
- Each character inside the quotes is placed into the string exactly as-is - even a literal newline!
- There are only two exceptions to this:
  - A single-quote on it's own would terminate the current string - to embed a single quote in a single-quoted string, write `\'`.

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.
- Each character inside the quotes is placed into the string exactly as-is - even a literal newline!
- There are only two exceptions to this:
  - A single-quote on it's own would terminate the current string - to embed a single quote in a single-quoted string, write `\'`.
  - A backslash may be embedded in a single-quoted string as `\\`.

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.
- Each character inside the quotes is placed into the string exactly as-is - even a literal newline!
- There are only two exceptions to this:
  - A single-quote on it's own would terminate the current string - to embed a single quote in a single-quoted string, write `\'`.
  - A backslash may be embedded in a single-quoted string as `\\`.
- Apart from these exceptions, nothing is modified inside a single quoted string.

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.
- Each character inside the quotes is placed into the string exactly as-is - even a literal newline!
- There are only two exceptions to this:
  - A single-quote on it's own would terminate the current string - to embed a single quote in a single-quoted string, write `\'`.
  - A backslash may be embedded in a single-quoted string as `\\`.
- Apart from these exceptions, nothing is modified inside a single quoted string.
- In particular, `$` symbols are embedded as-is, and C-style escapes like `\n` do not function in a single-quoted string.

- A double-quoted string literal, like `"hello $name\n"`, is much more powerful. Again, it is an arbitrary sequence of characters which can be split across lines.

- A double-quoted string literal, like `"hello $name\n"`, is much more powerful. Again, it is an arbitrary sequence of characters which can be split across lines.
- However, the backslash can now specify several important control or *escape* operations as follows:

`\n`	Newline
`\t`	Tab
`\r`	Carriage Return
`\a`	Ring bell
`\072`	Any octal ASCII value (7*8+2 = 58 = ':')
`\x6d`	Any hexadecimal ASCII value (6*16+13 = 109 = 'm')
`\\`	Backslash
`\$`	Dollar
`\"`	A double-quote
`\l`	Lower-case the next letter
`\u`	Upper-case the next letter
`\L`	Start lower-casing the rest of the string
`\U`	Start upper-casing the rest of the string
`\E`	Stop `\L` or `\U`

- A double-quoted string literal, like `"hello $name\n"`, is much more powerful. Again, it is an arbitrary sequence of characters which can be split across lines.
- However, the backslash can now specify several important control or *escape* operations as follows:

`\n`	Newline
`\t`	Tab
`\r`	Carriage Return
`\a`	Ring bell
`\072`	Any octal ASCII value (7*8+2 = 58 = ':')
`\x6d`	Any hexadecimal ASCII value (6*16+13 = 109 = 'm')
`\\`	Backslash
`\$`	Dollar
`\"`	A double-quote
`\l`	Lower-case the next letter
`\u`	Upper-case the next letter
`\L`	Start lower-casing the rest of the string
`\U`	Start upper-casing the rest of the string
`\E`	Stop `\L` or `\U`

- For example, `"hello \Uthere\E how are you"` generates `"hello THERE how are you"`.

- A double-quoted string literal, like `"hello $name\n"`, is much more powerful. Again, it is an arbitrary sequence of characters which can be split across lines.
- However, the backslash can now specify several important control or *escape* operations as follows:

`\n`	Newline
`\t`	Tab
`\r`	Carriage Return
`\a`	Ring bell
`\072`	Any octal ASCII value (7*8+2 = 58 = ':')
`\x6d`	Any hexadecimal ASCII value (6*16+13 = 109 = 'm')
`\\`	Backslash
`\$`	Dollar
`\"`	A double-quote
`\l`	Lower-case the next letter
`\u`	Upper-case the next letter
`\L`	Start lower-casing the rest of the string
`\U`	Start upper-casing the rest of the string
`\E`	Stop `\L` or `\U`

- For example, `"hello \Uthere\E how are you"` generates `"hello THERE how are you"`.
- Also in a double-quoted string...

- ...$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces `$varname` with the value of that variable. This is *Variable Interpolation*.

- ...$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces `$varname` with the value of that variable. This is *Variable Interpolation*.
- Example: `"hello $name\n"` is a string, comprising:
  - the character sequence `'hello '`,

- ...$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces `$varname` with the value of that variable. This is *Variable Interpolation*.
- Example: `"hello $name\n"` is a string, comprising:
  - the character sequence `'hello '`,
  - the current value of the variable `$name` (if `$name` is not in use, then the value is the undefined value, treated as the empty string),

- ...$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces `$varname` with the value of that variable. This is *Variable Interpolation*.
- Example: `"hello $name\n"` is a string, comprising:
  - the character sequence `'hello '`,
  - the current value of the variable `$name` (if `$name` is not in use, then the value is the undefined value, treated as the empty string),
  - the ASCII newline character (symbolically written as `\n`).

- ...$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces `$varname` with the value of that variable. This is *Variable Interpolation*.
- Example: `"hello $name\n"` is a string, comprising:
  - the character sequence `'hello '`,
  - the current value of the variable `$name` (if `$name` is not in use, then the value is the undefined value, treated as the empty string),
  - the ASCII newline character (symbolically written as `\n`).
- Suppose we wish to print the value of `$n` immediately followed by a sequence of characters that could also form part of the variable name. If we write:

  ```
  "you're the $nth person today..\n"
  ```

  Perl looks for a variable called `$nth`, and ends up interpolating the empty string, swallowing `th`!

- ...$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces `$varname` with the value of that variable. This is *Variable Interpolation*.
- Example: `"hello $name\n"` is a string, comprising:
  - the character sequence `'hello '`,
  - the current value of the variable `$name` (if `$name` is not in use, then the value is the undefined value, treated as the empty string),
  - the ASCII newline character (symbolically written as `\n`).
- Suppose we wish to print the value of `$n` immediately followed by a sequence of characters that could also form part of the variable name. If we write:

  ```
  "you're the $nth person today..\n"
  ```

  Perl looks for a variable called `$nth`, and ends up interpolating the empty string, swallowing `th`!
- Fix: use `{}` around the variable name:

  ```
  "you're the ${n}th person today..\n"
  ```

- In double-quoted strings, you still need to backslash double quotes to embed them into a string. Choose your own quote character:

```
my $contentslink = qq!
<a href="$v.html">
   <img src="$v.png">
</a>
!;
```

- In double-quoted strings, you still need to backslash double quotes to embed them into a string. Choose your own quote character:

```
my $contentslink = qq!
<a href="$v.html">
   <img src="$v.png">
</a>
!;
```

- After the qq, the next character ('!') is taken as the quote character. All characters after this are placed into the string (with interpolations and backslashes working exactly as in a double-quoted string) until the next '!' is encountered.

- In double-quoted strings, you still need to backslash double quotes to embed them into a string. Choose your own quote character:

```
my $contentslink = qq!
<a href="$v.html">
   <img src="$v.png">
</a>
!;
```

- After the qq, the next character ('!') is taken as the quote character. All characters after this are placed into the string (with interpolations and backslashes working exactly as in a double-quoted string) until the next '!' is encountered.
- But now - you don't need to backslash double-quotes.

- In double-quoted strings, you still need to backslash double quotes to embed them into a string. Choose your own quote character:

```
my $contentslink = qq!
<a href="$v.html">
   <img src="$v.png">
</a>
!;
```

- After the qq, the next character ('!') is taken as the quote character. All characters after this are placed into the string (with interpolations and backslashes working exactly as in a double-quoted string) until the next '!' is encountered.
- But now - you don't need to backslash double-quotes.
- If the opening quote is an open bracket of some kind (round, curly or square), Perl uses the appropriate closing bracket as the closing quote - We could write the above as:

```
my $contentslink = qq(
<a href="$v.html">
   <img src="$v.png">
</a>
);
```

- Expressions in Perl are very like those of C, but without the pointer operators.

- Expressions in Perl are very like those of C, but without the pointer operators.
- Some operators expect numbers, others expect strings. Perl converts one to the other when needed. Thus the string `"17.3e7"` converts to the number `17.3e7` when needed, and the number `31.53` converts to the string `"31.53"` when needed.

- Expressions in Perl are very like those of C, but without the pointer operators.
- Some operators expect numbers, others expect strings. Perl converts one to the other when needed. Thus the string `"17.3e7"` converts to the number `17.3e7` when needed, and the number `31.53` converts to the string `"31.53"` when needed.
- Perl provides all the obvious **+**, **-**, *****, **/** arithmetic operators. Unlike C, *all operators operate in floating point*.

- Expressions in Perl are very like those of C, but without the pointer operators.
- Some operators expect numbers, others expect strings. Perl converts one to the other when needed. Thus the string `"17.3e7"` converts to the number `17.3e7` when needed, and the number `31.53` converts to the string `"31.53"` when needed.
- Perl provides all the obvious **+**, **-**, *****, **/** arithmetic operators. Unlike C, *all operators operate in floating point*.
- Unlike both C and Pascal, Perl provides the *exponentation operator* (written ******). For example: `3.25 ** 4` (ie. 3.25^4).

- Expressions in Perl are very like those of C, but without the pointer operators.
- Some operators expect numbers, others expect strings. Perl converts one to the other when needed. Thus the string `"17.3e7"` converts to the number `17.3e7` when needed, and the number `31.53` converts to the string `"31.53"` when needed.
- Perl provides all the obvious **+**, **-**, *****, **/** arithmetic operators. Unlike C, *all operators operate in floating point*.
- Unlike both C and Pascal, Perl provides the *exponentation operator* (written ******). For example: `3.25 ** 4` (ie. 3.25^4).
- Like C, Perl provides a *modulus* operator: 10%3 gives the remainder when 10 is divided by 3. Both values are truncated to integers before this operator is applied.

- Expressions in Perl are very like those of C, but without the pointer operators.
- Some operators expect numbers, others expect strings. Perl converts one to the other when needed. Thus the string `"17.3e7"` converts to the number `17.3e7` when needed, and the number `31.53` converts to the string `"31.53"` when needed.
- Perl provides all the obvious `+`, `-`, `*`, `/` arithmetic operators. Unlike C, *all operators operate in floating point*.
- Unlike both C and Pascal, Perl provides the *exponentation operator* (written `**`). For example: `3.25 ** 4` (ie. 3.25^4).
- Like C, Perl provides a *modulus* operator: `10%3` gives the remainder when 10 is divided by 3. Both values are truncated to integers before this operator is applied.
- Perl provides a set of *numeric comparison* operators just like C: `<`, `<=`, `==`, `>=`, `>` and `!=` (not equals).

- Strings may be *compared alphabetically* via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.

- Strings may be *compared alphabetically* via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.
- A common mistake is comparing numbers with **eq** or strings with ==, allowed because strings and numbers are converted back and forth freely.

- Strings may be *compared alphabetically* via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.
- A common mistake is comparing numbers with **eq** or strings with ==, allowed because strings and numbers are converted back and forth freely.
- For example, `"0" == "000"` is true, but `0 eq "000"` is false. (The string `"0"` which is not the same as `"000"`!)

- Strings may be *compared alphabetically* via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.
- A common mistake is comparing numbers with **eq** or strings with ==, allowed because strings and numbers are converted back and forth freely.
- For example, `"0" == "000"` is true, but `0 eq "000"` is false. (The string `"0"` which is not the same as `"000"`!)
- Strings can be *concatenated* with the fullstop operator (**.**), eg:

```
"hello " . $name . "\n"
```

(Warning: No space is added automatically between the strings!)

- Strings may be *compared alphabetically* via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.
- A common mistake is comparing numbers with **eq** or strings with ==, allowed because strings and numbers are converted back and forth freely.
- For example, `"0" == "000"` is true, but `0 eq "000"` is false. (The string `"0"` which is not the same as `"000"`!)
- Strings can be *concatenated* with the fullstop operator (**.**), eg:

```
"hello " . $name . "\n"
```

  (Warning: No space is added automatically between the strings!)
- Concatenation is not used as much as you would expect - interpolation is often used instead. Write the above as:

```
"hello $name\n"
```

- Strings may be *compared alphabetically* via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.
- A common mistake is comparing numbers with **eq** or strings with ==, allowed because strings and numbers are converted back and forth freely.
- For example, `"0" == "000"` is true, but `0 eq "000"` is false. (The string `"0"` which is not the same as `"000"`!)
- Strings can be *concatenated* with the fullstop operator (**.**), eg:

  ```
  "hello " . $name . "\n"
  ```

  (Warning: No space is added automatically between the strings!)
- Concatenation is not used as much as you would expect - interpolation is often used instead. Write the above as:

  ```
  "hello $name\n"
  ```
- Another useful string operator is the *repetition* operator: `"fred" x 3` gives `"fredfredfred"`
  (the right-hand argument is truncated to an integer before replication occurs).

- The most vital operator of all is assignment (=). For example:

```
$x = 37;
$y = $x * 7 + 5;
$z = $z * 3;
```

  - The first simply sets $x to 37.

- The most vital operator of all is assignment (=). For example:

```
$x = 37;
$y = $x * 7 + 5;
$z = $z * 3;
```

  - The first simply sets `$x` to 37.
  - The second calculates `$x * 7 + 5` (using the current value of `$x`) and then stores the result in `$y`.

- The most vital operator of all is assignment (=). For example:

```
$x = 37;
$y = $x * 7 + 5;
$z = $z * 3;
```

  - The first simply sets `$x` to 37.
  - The second calculates `$x * 7 + 5` (using the current value of `$x`) and then stores the result in `$y`.
  - The last example takes the current value of $z, multiplies it by three, and stores the result back in $z.

- The most vital operator of all is assignment (=). For example:

```
$x = 37;
$y = $x * 7 + 5;
$z = $z * 3;
```

  - The first simply sets `$x` to 37.
  - The second calculates `$x * 7 + 5` (using the current value of `$x`) and then stores the result in `$y`.
  - The last example takes the current value of `$z`, multiplies it by three, and stores the result back in `$z`.

- An assignment also has a value, which means you can **nest** or **chain** assignments:

```
$x = $y = 17;
$y = 5 * ($a = 7 + $x);
```

  - The first example sets both `$x` and `$y` to 17.

- The most vital operator of all is assignment (=). For example:

```
$x = 37;
$y = $x * 7 + 5;
$z = $z * 3;
```

  - The first simply sets `$x` to 37.
  - The second calculates `$x * 7 + 5` (using the current value of `$x`) and then stores the result in `$y`.
  - The last example takes the current value of `$z`, multiplies it by three, and stores the result back in `$z`.

- An assignment also has a value, which means you can **nest** or **chain** assignments:

```
$x = $y = 17;
$y = 5 * ($a = 7 + $x);
```

  - The first example sets both `$x` and `$y` to 17.
  - The second example means: evaluate 7+`$x` and store the result in `$a`, then multiply `$a` by 5 and store the final result in `$y`.

- Perl (just like C) provides a set of assignment operators that modify a single variable:

- Perl (just like C) provides a set of assignment operators that modify a single variable:
- Nearly any binary operator that computes a value can have an = sign appended on the end. For instance:

```
$z *= 3;                    $sum += $element;
```

These read as *multiply z by 3*, *add element to sum*.

- Perl (just like C) provides a set of assignment operators that modify a single variable:
- Nearly any binary operator that computes a value can have an = sign appended on the end. For instance:

```
$z *= 3;                    $sum += $element;
```

  These read as *multiply z by 3*, *add element to sum*.
- The same works for strings: `$z .= $a` appends the contents of `$a` onto the end of `$z`.

- Perl (just like C) provides a set of assignment operators that modify a single variable:
- Nearly any binary operator that computes a value can have an = sign appended on the end. For instance:

```
$z *= 3;                    $sum += $element;
```

  These read as *multiply z by 3*, *add element to sum*.
- The same works for strings: `$z .= $a` appends the contents of `$a` onto the end of `$z`.
- Just like basic assignment, all the binary assignment operators return a value too. So the following is valid:

```
$a = 3;
$b = ($a += 4) * 7;
```

- Perl (just like C) provides a set of assignment operators that modify a single variable:
- Nearly any binary operator that computes a value can have an = sign appended on the end. For instance:

```
$z *= 3;                    $sum += $element;
```

  These read as *multiply z by 3*, *add element to sum*.
- The same works for strings: `$z .= $a` appends the contents of `$a` onto the end of `$z`.
- Just like basic assignment, all the binary assignment operators return a value too. So the following is valid:

```
$a = 3;
$b = ($a += 4) * 7;
```

- **However**, never change the same variable inside two branches of the same expression:

```
$a = 3;
$b = ($a += 4) * ($a -= 2);
```

- To *increment* or *decrement* $a, write:

```
$a++;
```

```
$a--;
```

- To *increment* or *decrement* $a, write:

```
$a++;                                   $a--;
```

- Perl has a prefix and postfix form of these operators: ++$a, $a++, --$a and $a--. Both forms increment (or decrement) the variable in the same way.

- To *increment* or *decrement* $a, write:

```
$a++;                                   $a--;
```

- Perl has a prefix and postfix form of these operators: ++$a, $a++, --$a and $a--. Both forms increment (or decrement) the variable in the same way.
- Difference: when embedding in a large expression, old or new value? Consider the following two examples, first:

Perl	Simplified Form	Effect
`$a = 7;`	a=7	a:7
`$b = ++$a;`	a++; b=a	a:8, b:8
`$c = $b--;`	c=b; b--	c:8, b:7

- To *increment* or *decrement* $a, write:

```
$a++;                                   $a--;
```

- Perl has a prefix and postfix form of these operators: ++$a, $a++, --$a and $a--. Both forms increment (or decrement) the variable in the same way.
- Difference: when embedding in a large expression, old or new value? Consider the following two examples, first:

Perl	Simplified Form	Effect
$a = 7;	a=7	a:7
$b = ++$a;	a++; b=a	a:8, b:8
$c = $b--;	c=b; b--	c:8, b:7

- Second:

Perl	Simplified	Effect
$a = 7;	a=7	a:7
$b = $a++;	b=a; a++	b:7, a:8
$c = --$b;	b--; c=b	b:6, c:6

In both cases, $a ends up as 8.

- Expressions used in a boolean context are evaluated in the usual way as numbers or strings.

- Expressions used in a boolean context are evaluated in the usual way as numbers or strings.
- When the result value is produced (a number or a string), a *zero value or empty string* means *false*, and *anything else is true!*

There are a few extra boolean operators:

- ! is a unary operator meaning *not*.

- Expressions used in a boolean context are evaluated in the usual way as numbers or strings.
- When the result value is produced (a number or a string), a *zero value or empty string* means *false*, and *anything else is true!*

There are a few extra boolean operators:

- `!` is a unary operator meaning *not*.
- `||` and `&&` come straight from C – these can be written as '`or`' and '`and`' in modern Perl (but with rather confusing precedences).

- Expressions used in a boolean context are evaluated in the usual way as numbers or strings.
- When the result value is produced (a number or a string), a *zero value or empty string* means *false*, and *anything else is true!*

There are a few extra boolean operators:

- `!` is a unary operator meaning *not*.
- `||` and `&&` come straight from C – these can be written as '`or`' and '`and`' in modern Perl (but with rather confusing precedences).
- Just like C and Java (but unlike Pascal), they *short-circuit*:
  - Consider:
    ```
    $a < 7 || $b > 6
    ```
    If `$a` is less than 7, then the whole expression is bound to be true. There is no point in evaluating the rest of the expression - so Perl doesn't!

- Expressions used in a boolean context are evaluated in the usual way as numbers or strings.
- When the result value is produced (a number or a string), a *zero value or empty string* means *false*, and *anything else is true!*

There are a few extra boolean operators:

- `!` is a unary operator meaning *not*.
- `||` and `&&` come straight from C – these can be written as '`or`' and '`and`' in modern Perl (but with rather confusing precedences).
- Just like C and Java (but unlike Pascal), they *short-circuit*:
  - Consider:
    ```
    $a < 7 || $b > 6
    ```
    If `$a` is less than 7, then the whole expression is bound to be true. There is no point in evaluating the rest of the expression - so Perl doesn't!
  - Similarly, consider:
    ```
    $s eq "hello" && $t ne "bonjour"
    ```
    If `$s` is not `"hello"` then the whole expression must be false. Again, there's absolutely no point in evaluating the second half.

- Any expression can become a statement by appending '`;`': `3;` means *evaluate expression* (result: 3) and *discard the result*.

- Any expression can become a statement by appending `';'`: `3;` means *evaluate expression* (result: 3) and *discard the result*.
- Usually, expression-statements have side-effects:
  - Assignments (the basic =, all the binary assignment operators like *=, and the ++ and -- operators).

- Any expression can become a statement by appending `';'`: `3;` means *evaluate expression* (result: 3) and *discard the result*.
- Usually, expression-statements have side-effects:
  - Assignments (the basic =, all the binary assignment operators like *=, and the ++ and -- operators).
  - Function/procedure calls (eg. print - or your own functions).

- Any expression can become a statement by appending `';'`: `3;` means *evaluate expression* (result: 3) and *discard the result*.
- Usually, expression-statements have side-effects:
  - Assignments (the basic =, all the binary assignment operators like *=, and the ++ and -- operators).
  - Function/procedure calls (eg. print - or your own functions).
- Perl also provides *modifiers* for single statements:

```
<statement> if <expr> ;
<statement> unless <expr> ;
<statement> while <expr> ;
<statement> until <expr> ;
```

(Perl allows `unless` to be used as a synonym for `if !`, ie. *if not true*, and `until` as a synonym for `while !`).

- Any expression can become a statement by appending '`;`': `3;` means *evaluate expression* (result: 3) and *discard the result*.
- Usually, expression-statements have side-effects:
  - Assignments (the basic =, all the binary assignment operators like *=, and the ++ and -- operators).
  - Function/procedure calls (eg. print - or your own functions).
- Perl also provides *modifiers* for single statements:

```
<statement> if <expr> ;
<statement> unless <expr> ;
<statement> while <expr> ;
<statement> until <expr> ;
```

  (Perl allows `unless` to be used as a synonym for `if !`, ie. *if not true*, and `until` as a synonym for `while !`).
- A sequence of statements may be enclosed inside `{}` braces forming a *block*. NB: no '`;`' after the '`}`' of a block.

- Perl provides a conventional **if.. elsif.. elsif.... else** statement, to choose between two or more alternatives:

```
if( $i < 20 || $j > 7.4 )
{
        print "case one\n";
} elsif( $i > 40 && $j > 0 )
{
        print "case two\n";
} else
{
        print "case three\n";
}
```

Note that the brackets – both round and curly – are compulsory on an **if** - and the loop below.

- Perl provides a conventional **if.. elsif.. elsif.... else** statement, to choose between two or more alternatives:

```
if( $i < 20 || $j > 7.4 )
{
        print "case one\n";
} elsif( $i > 40 && $j > 0 )
{
        print "case two\n";
} else
{
        print "case three\n";
}
```

  Note that the brackets – both round and curly – are compulsory on an **if** - and the loop below.

- In a rare omission, Perl did **NOT** provide a special case/switch statement for multi-way comparisons. An add-on module adds this.

- Perl provides a conventional **if.. elsif.. elsif.... else** statement, to choose between two or more alternatives:

```
if( $i < 20 || $j > 7.4 )
{
        print "case one\n";
} elsif( $i > 40 && $j > 0 )
{
        print "case two\n";
} else
{
        print "case three\n";
}
```

  Note that the brackets – both round and curly – are compulsory on an **if** - and the loop below.

- In a rare omission, Perl did **NOT** provide a special case/switch statement for multi-way comparisons. An add-on module adds this.
- Perl also provides a conventional *test at the top* while loop:

```
my $w = $x; my $h = 1;
while ( abs($w-$h) > 0.001 )
{
        $w = ($w+$h)/2;
        $h = $x/$w;
}
```

- Perl provides a conventional **if.. elsif.. elsif.... else** statement, to choose between two or more alternatives:

```
if( $i < 20 || $j > 7.4 )
{
        print "case one\n";
} elsif( $i > 40 && $j > 0 )
{
        print "case two\n";
} else
{
        print "case three\n";
}
```

  Note that the brackets – both round and curly – are compulsory on an **if** - and the loop below.

- In a rare omission, Perl did **NOT** provide a special case/switch statement for multi-way comparisons. An add-on module adds this.
- Perl also provides a conventional *test at the top* while loop:

```
my $w = $x; my $h = 1;
while ( abs($w-$h) > 0.001 )
{
        $w = ($w+$h)/2;
        $h = $x/$w;
}
```

- The above algorithm happens to find the square root of $x!

- Perl provides a *test at the bottom* loop, two forms:

```
do {
        $y *= $x;
        $x += 2;
} while( $x < 15 );
```

```
do {
        $y *= $x;
        $x += 2;
} until( $x >= 15 );
```

- Perl provides a *test at the bottom* loop, two forms:

```
do {
        $y *= $x;
        $x += 2;
} while( $x < 15 );
```

```
do {
        $y *= $x;
        $x += 2;
} until( $x >= 15 );
```

- A C-style *counting loop*:

```
for( my $sum = 0, my $i = 1; $i <= 10; $i++ )
{
        $sum += $i;
}
```

- Perl provides a *test at the bottom* loop, two forms:

```
do {
        $y *= $x;
        $x += 2;
} while( $x < 15 );
```

```
do {
        $y *= $x;
        $x += 2;
} until( $x >= 15 );
```

- A C-style *counting loop*:

```
for( my $sum = 0, my $i = 1; $i <= 10; $i++ )
{
        $sum += $i;
}
```

- A very useful *for each element in a list/array* loop:

```
foreach my $x ( 1, 3, 7, 5, 4, 54 )
{
        $total += $x;
}
```

(We'll see more about arrays next session).

- Perl provides a *test at the bottom* loop, two forms:

```
do {
        $y *= $x;
        $x += 2;
} while( $x < 15 );
```

```
do {
        $y *= $x;
        $x += 2;
} until( $x >= 15 );
```

- A C-style *counting loop*:

```
for( my $sum = 0, my $i = 1; $i <= 10; $i++ )
{
        $sum += $i;
}
```

- A very useful *for each element in a list/array* loop:

```
foreach my $x ( 1, 3, 7, 5, 4, 54 )
{
        $total += $x;
}
```

(We'll see more about arrays next session).

- Exercise: Try writing a little Perl program that reads in a number and computes the square root using the while loop on the previous slide, printing out a message like `"The square root of $x is $w"`.

Lets consider getting input from the keyboard, and reporting results to the screen.

- The <> operator (pronounced *diamond* or *getline*) fetches a line of input from a filehandle - `STDIN` is a filehandle, which represents the standard input (normally the keyboard). So:

  ```
  $name = <STDIN>;
  ```

  reads an entire line of input, up to and including a newline, and then stores the input in the variable (including the newline).

Lets consider getting input from the keyboard, and reporting results to the screen.

- The <> operator (pronounced *diamond* or *getline*) fetches a line of input from a filehandle - `STDIN` is a filehandle, which represents the standard input (normally the keyboard). So:

  ```
  $name = <STDIN>;
  ```

  reads an entire line of input, up to and including a newline, and then stores the input in the variable (including the newline).
- After reading a line, you usually want to get rid of the trailing newline. Two operators:
  - Perl 4 provided `chop($name)` which removes the last character from the named variable and returns the character removed.

Lets consider getting input from the keyboard, and reporting results to the screen.

- The <> operator (pronounced *diamond* or *getline*) fetches a line of input from a filehandle - `STDIN` is a filehandle, which represents the standard input (normally the keyboard). So:

  ```
  $name = <STDIN>;
  ```

  reads an entire line of input, up to and including a newline, and then stores the input in the variable (including the newline).
- After reading a line, you usually want to get rid of the trailing newline. Two operators:
  - Perl 4 provided `chop($name)` which removes the last character from the named variable and returns the character removed.
  - Perl 5 added `chomp($name)` which deletes a trailing newline or does nothing.

Lets consider getting input from the keyboard, and reporting results to the screen.

- The <> operator (pronounced *diamond* or *getline*) fetches a line of input from a filehandle - `STDIN` is a filehandle, which represents the standard input (normally the keyboard). So:

  ```
  $name = <STDIN>;
  ```

  reads an entire line of input, up to and including a newline, and then stores the input in the variable (including the newline).
- After reading a line, you usually want to get rid of the trailing newline. Two operators:
  - Perl 4 provided `chop($name)` which removes the last character from the named variable and returns the character removed.
  - Perl 5 added `chomp($name)` which deletes a trailing newline or does nothing.
- Use `chomp` everywhere; it's safer and more portable (it will remove the newline from the string even if the newline convention on the OS you're using is not a single character!).

- There is a nice simple way of printing some results:

```
print expression;
```

- There is a nice simple way of printing some results:

```
print expression;
```

- Combined with the variable interpolation we've seen in strings, this gives us all the formatting power we need for now:

```
print "debug: x=$x, y=$y, z=$z\n";
```

- There is a nice simple way of printing some results:

```
print expression;
```

- Combined with the variable interpolation we've seen in strings, this gives us all the formatting power we need for now:

```
print "debug: x=$x, y=$y, z=$z\n";
```

- Here's an example (**eg1**), allowing you to investigate different expressions:

```
print "Please enter x: ";
my $x = <STDIN>;
chomp $x;
print "Please enter y: ";
my $y = <STDIN>;
chomp $y;
my $z = $x + $y;
print "\n$x + $y = $z\n";
```

- There is a nice simple way of printing some results:

```
print expression;
```

- Combined with the variable interpolation we've seen in strings, this gives us all the formatting power we need for now:

```
print "debug: x=$x, y=$y, z=$z\n";
```

- Here's an example (**eg1**), allowing you to investigate different expressions:

```
print "Please enter x: ";
my $x = <STDIN>;
chomp $x;
print "Please enter y: ";
my $y = <STDIN>;
chomp $y;
my $z = $x + $y;
print "\n$x + $y = $z\n";
```

- You may like to spend some time trying some examples of expressions, starting with the above example and incorporating different, more complex, expressions.

- There is a nice simple way of printing some results:

```
print expression;
```

- Combined with the variable interpolation we've seen in strings, this gives us all the formatting power we need for now:

```
print "debug: x=$x, y=$y, z=$z\n";
```

- Here's an example (**eg1**), allowing you to investigate different expressions:

```
print "Please enter x: ";
my $x = <STDIN>;
chomp $x;
print "Please enter y: ";
my $y = <STDIN>;
chomp $y;
my $z = $x + $y;
print "\n$x + $y = $z\n";
```

- You may like to spend some time trying some examples of expressions, starting with the above example and incorporating different, more complex, expressions.
- Try various += and ++ types of operators to get clear exactly what they do.

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.
- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.
- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!
- So does the keyboard: On Unix systems, end-of-input is signalled by the user typing CTRL-D on a line on its own.

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.
- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!
- So does the keyboard: On Unix systems, end-of-input is signalled by the user typing CTRL-D on a line on its own.
- No CTRL-D character (ASCII code 4) is delivered; CTRL-D is a signal that input has ended.

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.
- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!
- So does the keyboard: On Unix systems, end-of-input is signalled by the user typing CTRL-D on a line on its own.
- No CTRL-D character (ASCII code 4) is delivered; CTRL-D is a signal that input has ended.
- When eof occurs, <> returns the *empty string*. This never otherwise occurs - why?

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.
- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!
- So does the keyboard: On Unix systems, end-of-input is signalled by the user typing CTRL-D on a line on its own.
- No CTRL-D character (ASCII code 4) is delivered; CTRL-D is a signal that input has ended.
- When eof occurs, <> returns the *empty string*. This never otherwise occurs - why?
- So comparing the result of <> against the empty string will reliably determine eof:

```
if( $line eq "" )...
```

- We've seen that `$line = <STDIN>` reads a whole line of input from stdin.
- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!
- So does the keyboard: On Unix systems, end-of-input is signalled by the user typing CTRL-D on a line on its own.
- No CTRL-D character (ASCII code 4) is delivered; CTRL-D is a signal that input has ended.
- When eof occurs, <> returns the *empty string*. This never otherwise occurs - why?
- So comparing the result of <> against the empty string will reliably determine eof:

```
if( $line eq "" )...
```

- More typically, you test for *not eof*, which is written `if( $line ne "" )` or simply `if( $line )`.

- We can even combine the line reading, assignment, and test for not eof into one:

```
if( $line = <STDIN> )   # if we've read a line..
```

- We can even combine the line reading, assignment, and test for not eof into one:

```
if( $line = <STDIN> )   # if we've read a line..
```

- Using a `while` instead, and adding `my` to declare `$line`, gives us the **for each line in a file** idiom:

```
while( my $line = <STDIN> )   # for each line from stdin
{
        chomp $line;
        # now process $line
}
```

- We can even combine the line reading, assignment, and test for not eof into one:

```
if( $line = <STDIN> )   # if we've read a line..
```

- Using a `while` instead, and adding `my` to declare `$line`, gives us the **for each line in a file** idiom:

```
while( my $line = <STDIN> )   # for each line from stdin
{
        chomp $line;
        # now process $line
}
```

- Let's see an simple example of that (**eg2**):

  Write a program that reads a list of numbers (one per line) from `STDIN`*, adds all these numbers up, and prints the total.*

- We can even combine the line reading, assignment, and test for not eof into one:

```
if( $line = <STDIN> )   # if we've read a line..
```

- Using a `while` instead, and adding `my` to declare `$line`, gives us the **for each line in a file** idiom:

```
while( my $line = <STDIN> )   # for each line from stdin
{
        chomp $line;
        # now process $line
}
```

- Let's see an simple example of that (**eg2**):

  Write a program that reads a list of numbers (one per line) from `STDIN`*, adds all these numbers up, and prints the total.*

- Above the while loop, we initialize: `my $sum = 0;`

- We can even combine the line reading, assignment, and test for not eof into one:

```
if( $line = <STDIN> )   # if we've read a line..
```

- Using a `while` instead, and adding `my` to declare `$line`, gives us the **for each line in a file** idiom:

```
while( my $line = <STDIN> )   # for each line from stdin
{
        chomp $line;
        # now process $line
}
```

- Let's see an simple example of that (**eg2**):

  Write a program that reads a list of numbers (one per line) from `STDIN`*, adds all these numbers up, and prints the total.*

- Above the while loop, we initialize: `my $sum = 0;`
- Processing is: `$sum += $line;`

- We can even combine the line reading, assignment, and test for not eof into one:

```
if( $line = <STDIN> )   # if we've read a line..
```

- Using a `while` instead, and adding `my` to declare `$line`, gives us the **for each line in a file** idiom:

```
while( my $line = <STDIN> )   # for each line from stdin
{
        chomp $line;
        # now process $line
}
```

- Let's see an simple example of that (**eg2**):

  Write a program that reads a list of numbers (one per line) from `STDIN`*, adds all these numbers up, and prints the total.*

- Above the while loop, we initialize: `my $sum = 0;`
- Processing is: `$sum += $line;`
- At the end, add `print "total: $sum\n"`.

- Perl can read the contents of named files (or Unix pipelines), create/overwrite a named file with new contents, and append new output on the end of a file.

- Perl can read the contents of named files (or Unix pipelines), create/overwrite a named file with new contents, and append new output on the end of a file.
- Suppose we wish to read a file called `fred`. The first step is to create a filehandle like `STDIN` but connected to the file `"fred"`:

- Perl can read the contents of named files (or Unix pipelines), create/overwrite a named file with new contents, and append new output on the end of a file.
- Suppose we wish to read a file called `fred`. The first step is to create a filehandle like `STDIN` but connected to the file `"fred"`:
- Near the top of our program, we write:

```
use IO::File;
```

- Perl can read the contents of named files (or Unix pipelines), create/overwrite a named file with new contents, and append new output on the end of a file.
- Suppose we wish to read a file called `fred`. The first step is to create a filehandle like `STDIN` but connected to the file `"fred"`:
- Near the top of our program, we write:

```
use IO::File;
```

- Then to open `fred`:

```
my $in = new IO::File( "fred" );
unless( $in )
{
        # ... handle the failure...
}
```

If the file `"fred"` doesn't already exist, **new IO::File** will fail, returning 0 - a boolean false - so check for success by testing the result. Or check for failure by testing `unless( $in )`.

- Perl can read the contents of named files (or Unix pipelines), create/overwrite a named file with new contents, and append new output on the end of a file.
- Suppose we wish to read a file called `fred`. The first step is to create a filehandle like `STDIN` but connected to the file `"fred"`:
- Near the top of our program, we write:

```
use IO::File;
```

- Then to open `fred`:

```
my $in = new IO::File( "fred" );
unless( $in )
{
        # ... handle the failure...
}
```

  If the file `"fred"` doesn't already exist, **new IO::File** will fail, returning 0 - a boolean false - so check for success by testing the result. Or check for failure by testing `unless( $in )`.
- Often, handling the failure is done by printing an error message (on `STDERR`) and exiting - use `die`:

```
my $in = new IO::File( "fred" );
unless( $in )
{
        die "can't open fred\n";
}
```

- Better still, use conditional modifier syntax:

```
my $in = new IO::File( "fred" );
die "can't open fred\n" unless $in;
```

- Better still, use conditional modifier syntax:

```
my $in = new IO::File( "fred" );
die "can't open fred\n" unless $in;
```

- Best of all, we can use the common *Do or die* idiom. This wouldn't work without short-circuit evaluation:

```
my $in = new IO::File( "fred" ) || die "can't open fred!\n";
```

- Better still, use conditional modifier syntax:

```
my $in = new IO::File( "fred" );
die "can't open fred\n" unless $in;
```

- Best of all, we can use the common *Do or die* idiom. This wouldn't work without short-circuit evaluation:

```
my $in = new IO::File( "fred" ) || die "can't open fred!\n";
```

- Now, `$in` is an open filehandle, just like `STDIN`, from which we can read data.

- Better still, use conditional modifier syntax:

```
my $in = new IO::File( "fred" );
die "can't open fred\n" unless $in;
```

- Best of all, we can use the common *Do or die* idiom. This wouldn't work without short-circuit evaluation:

```
my $in = new IO::File( "fred" ) || die "can't open fred!\n";
```

- Now, `$in` is an open filehandle, just like `STDIN`, from which we can read data.
- `$line = <$in>` reads the next line of input from `fred`. When there is no more input, `$line` will contain the empty string.

- Better still, use conditional modifier syntax:

```
my $in = new IO::File( "fred" );
die "can't open fred\n" unless $in;
```

- Best of all, we can use the common *Do or die* idiom. This wouldn't work without short-circuit evaluation:

```
my $in = new IO::File( "fred" ) || die "can't open fred!\n";
```

- Now, `$in` is an open filehandle, just like `STDIN`, from which we can read data.
- `$line = <$in>` reads the next line of input from `fred`. When there is no more input, `$line` will contain the empty string.
- A typical *read every line* program looks like **eg3**:

```
use IO::File;
my $in = new IO::File( "fred" ) || die "can't open fred\n";
while( my $line = <$in> )
{
        chomp $line;
        print "read '$line'\n";
}
$in->close;
```

- Better still, use conditional modifier syntax:

```
my $in = new IO::File( "fred" );
die "can't open fred\n" unless $in;
```

- Best of all, we can use the common *Do or die* idiom. This wouldn't work without short-circuit evaluation:

```
my $in = new IO::File( "fred" ) || die "can't open fred!\n";
```

- Now, `$in` is an open filehandle, just like `STDIN`, from which we can read data.
- `$line = <$in>` reads the next line of input from `fred`. When there is no more input, `$line` will contain the empty string.
- A typical *read every line* program looks like **eg3**:

```
use IO::File;
my $in = new IO::File( "fred" ) || die "can't open fred\n";
while( my $line = <$in> )
{
        chomp $line;
        print "read '$line'\n";
}
$in->close;
```

- Note: As well as `die` there's a function `warn` which prints a message to `STDERR` but doesn't exit.

- To open a file for writing (overwriting the old contents):

```
my $out = new IO::File( "> bob" ) || die "sob.. can't create bob\n";
```

- To open a file for writing (overwriting the old contents):

```
my $out = new IO::File( "> bob" ) || die "sob.. can't create bob\n";
```

- We may now write new data into bob by any number of:

```
print $out "...some data...\n"; or $out->print( "...some data...\n" );
```

- To open a file for writing (overwriting the old contents):

```
my $out = new IO::File( "> bob" ) || die "sob.. can't create bob\n";
```

- We may now write new data into bob by any number of:

```
print $out "...some data...\n"; or $out->print( "...some data...\n" );
```

- Just as with reading, don't forget to `$out->close` when you finish writing data.

- To open a file for writing (overwriting the old contents):

```
my $out = new IO::File( "> bob" ) || die "sob.. can't create bob\n";
```

- We may now write new data into bob by any number of:

```
print $out "...some data...\n"; or $out->print( "...some data...\n" );
```

- Just as with reading, don't forget to `$out->close` when you finish writing data.
- We can append data to bob, instead of overwriting:

```
my $out = new IO::File( ">> bob" ) || die "sob.. can't append to bob\n";
```

- To open a file for writing (overwriting the old contents):

```
my $out = new IO::File( "> bob" ) || die "sob.. can't create bob\n";
```

- We may now write new data into bob by any number of:

```
print $out "...some data...\n"; or $out->print( "...some data...\n" );
```

- Just as with reading, don't forget to `$out->close` when you finish writing data.
- We can append data to bob, instead of overwriting:

```
my $out = new IO::File( ">> bob" ) || die "sob.. can't append to bob\n";
```

- You can also open a pipe to/from a Unix pipeline for reading/writing:
  - `my $in = new IO::File( "ls | sort -r |" );`
    If the last character is a '|', then we can read data from the pipeline: Any output that `ls | sort -r` prints onto its STDOUT will be available for us to read via `<$in>`.

- To open a file for writing (overwriting the old contents):

```
my $out = new IO::File( "> bob" ) || die "sob.. can't create bob\n";
```

- We may now write new data into bob by any number of:

```
print $out "...some data...\n"; or $out->print( "...some data...\n" );
```

- Just as with reading, don't forget to `$out->close` when you finish writing data.
- We can append data to bob, instead of overwriting:

```
my $out = new IO::File( ">> bob" ) || die "sob.. can't append to bob\n";
```

- You can also open a pipe to/from a Unix pipeline for reading/writing:
  - `my $in = new IO::File( "ls | sort -r |" );`
    If the last character is a '|', then we can read data from the pipeline: Any output that `ls | sort -r` prints onto its STDOUT will be available for us to read via `<$in>`.
  - `my $out = new IO::File( "| expand" );`
    If the first character is a '|', then we can write data to the pipe: `expand` will perceive a stream of data coming from its STDIN, but it'll really be whatever we write to `$out`.

- Exercise: Merge **eg2** and **eg3** to sum up the leading numbers in a specific named file.
- Exercise: Write a program that reads every line from `STDIN`, lower-cases it using `lc()` and writes the lower-cased lines into a file called `lower`. You might call such a program `mklower`.
- Exercise: modify `mklower` slightly: remove the word `STDIN` from the `<STDIN>` getline call, leaving the mysterious syntax `<>`. We'll explain next session what this syntax means - for now, try to work it out for yourself by experimenting with `mklower`'s behaviour.