

# Pearls of Perl

Sadiq Jaffer, Tim Retout and David Flynn

November 8, 2005

# Chapter 1

## Welcome

These are the accompanying notes to a short set of three lectures given and produced by Sadiq Jaffer, Tim Retout and David Flynn with the intent to provide the reader with an insight into what Perl can be used for and how to accomplish it.

### What is Perl?

Perl is a tool, an alternative to the curmudgeon of the single task, narrow scope UNIX tools. Traditional building block tools such as awk, sed, grep and sh while being powerful single task tools, can be unwieldy for solving a relatively complex task that uses bits from each tool.

It was originally written (and is still maintained) by Larry Wall, when awk didn't have enough grunt to solve the problem he was working on. Larry Wall is a lexicologist, he designed perl to be a language that has a grammar as opposed to designing a strict syntax. This means that the language if programmed well is very capable of being read like english, resulting in there being more than one way to do it.

### What is Perl good for?

Perl is a great tool for whipping up quick'n'dirty (often throw away) scripts to accomplish a single task. It is especially good with manipulating data - specifically "text", it inherits this from its awk beginnings. It is commonly used in web applications.

### What we assume

We assume the reader has a useful understanding of another programming language, it doesn't need to be object orientated, everything presented here will be procedural; we shall be making comparisons to Java throughout this text. We also assume a familiarity with the standard UNIX tools and concepts introduced in CS120 (Programming Lab).

# Chapter 2

## Perl basics

### Dogma

From the start of the course it is an excellent idea to learn to do things properly and safely. Perl provides two useful mechanisms that will be of great assistance to you.

#### **use strict;**

Place the directive `use strict;` near the top of your program. This makes the Perl interpreter strict about declaring variables.<sup>1</sup>

The added bonus of using `use strict;` is that it shows you when you are using an undeclared variable, normally through a typing mistake.

#### **use warnings;**

Place the directive `use strict;` near the top of your program. It will cause perl to emit warnings when you are doing something potentially silly or wrong.

### The first line

All scripts must start with a hash-bang-interpreter line; the Perl interpreter is normally `/usr/bin/perl`, thus all our Perl scripts should start:

```
#!/usr/bin/perl
```

### Comments

As in most UNIX scripting languages, any text after a `#` to the end of the line is ignored by the interpreter.

---

<sup>1</sup>In fact all variables have to be declared (not initialized mind you) using the syntax `my variable ;`. More sane scoping rules apply to variables that are declared with the `my` statement - in fact they are just like you would expect in Java.

## Statements

Like Java, statements in Perl are ended with a semi-colon, eg:

```
#!/usr/bin/perl
#
# a traditional hello world script

use warnings;
use strict;

print "Hello World!\n"; # another comment here
```

produces:

```
$ perl helloworld.pl
Hello World!
$
```

## A quick way to do Perl in one line

Rather than having to write a script to do menial one or two line tasks, it is possible to do it on the command line. The following example would become:

```
$ perl -e 'print "Hello World!\n!";'
Hello World!
$
```

# Chapter 3

## Scalars

### What is a scalar?

Scalars are one of Perl's three data types. They refer to single quantities of data, eg. a number (just like Java really). A string too is a scalar quantity. Perl differs from Java here; in Perl the whole string is considered to be a single value, as opposed to Java where it is considered to be an array of characters.

A scalar variable is just a container to store a single piece of scalar data. As Perl is a very weakly typed language, there is no limit as to what type of scalar data may be stored in the variable. To denote a scalar variable, the variable name (which conforms to the same rules as Java or C) is prefixed with a dollar sign `$`. Scalar variables are not declared as they would be in Java, they merely come into existence when you first refer to them and are always initialised to 'nothing'.

### Assignment of scalar data

Assignment with scalars is exactly the same in Perl as it is in Java, for example:

```
$fish    = "salmon"; # $fish contains the string salmon
$level   = 4;       # $level contains the number four
$level += 1;       # $level is incremented and now contains five
$level   = $level * 2 # $level is multiplied by two, now contains ten
```

### Weak typing

So far in your programming life you know about Java, you may too know about SML, each being strongly typed languages. One may only store integers in Java's *int* type, and to print out that integer it has to be converted to a String (although this may be done behind the scenes). Weak typing means is that the scalar data is interpreted in a way that is useful to the context it is being used in.

For instance, consider the following code:

```

$var = "4"           # $level contains four
$var += "1"         # $level is incremented and now contains five
$var = $level * 2   # $level is multiplied by two, now contains ten

```

This gives exactly the same result as the previous example. It allows the programmer to not worry about the exact type of the data - the interpreter will use what ever is appropriate for what instruction it is given.

ie. "4" and 4 are exactly the same.

Or more simply, everything is treated as a string unless an operator would like the data to be a number.

### 3.0.1 Interpolation

Remember those strings in the last chapter? Well, wouldn't it be nice to print what is in our variables? This can be done in two ways, it can be done the 'Java' way:

```

$a_nice_variable = "hello"

print "our nice variable says: " . $a_nice_variable . "\n";
# the above line would print:
# our nice variable says: hello

```

All the work is happening on the line of the print function. Instead of using Java's + operator to join (concatenated) the scalar data <sup>1</sup> it uses Perl's . (concatenate) operator. The "\n" is there simply to make the output more readable - try running it on a terminal without it.

The above is all well and good, but it isn't the Perl way. Just like the Bourne shell, Perl can interpolate variables inside double quoted string literals. All one must do is use the variable's name. eg:

```

$a_nice_variable = "hello";

print "our nice variable says: $a_nice_variable\n";
# the above line would print:
# our nice variable says: hello

```

This is called interpolation, it can only be done with double quoted string literals - remembering that everything in single quoted string literals is used verbatim, ie, no interpolation takes place.

---

<sup>1</sup>remember that a string is a scalar, as is a scalar variable

# Chapter 4

## Lists

Lists form the second of Perl's three data-types. Lists are like arrays in Java. However they hold only one data-type - the scalar. So a list is an array of scalars. There is no need for the scalars to hold the same 'type' of information - that is something for the programmer to decide.

Unlike Java, Perl manages the storage of the list, you can just add elements to it - at any position and it will be made larger.

### List literals

To form a literal list in a program, the list is specified using parentheses with a comma separated list of scalars. For example:

```
(1, 2, 3, 4, 5)           # a list containing numbers one, two and three
("goldfish", "cod", "pike") # a list containing strings
($a, $b, $c)             # a list containing the values of scalars
```

### Ranges

A list of numbers can be formed using the range  $a..b$  operator. It forms a list of integer numbers changing by one in the range of  $a$  to  $b$  inclusive, eg:

```
(1 .. 5)           # a list of numbers in the range of one through five inclusive
print 1 .. 5;     # prints "1 2 3 4 5"
```

### @

To denote a list variable in Perl, the variable name is prefixed with an @ (at). The normal variable naming rules apply, although \$foo is not the same as @foo (ie, it is legal to have both of them).

## Assignment

Assignment of a list to a list variable is simple:

```
@fishtypes = ("gold", "cod", "pike");
```

## Joining two lists

As a list is just a sequence of scalars then, to join two lists, one just places the two lists in another list, viz.

```
@a = (1 .. 5);  
@b = (6 .. 10);  
@ab = (@a, @b);
```

## Copying a list

To copy a list, just assign it to another list. This makes a *completely separate* copy of the list. There is nothing that joins the copy to the original.

```
@a = (1 .. 10);  
@b = @a;          # @b contains a copy of @a
```

## Accessing elements

To access list elements is the same as in Java, using brackets and an offset from the start of the list. However to confuse the issue, the leading @ is changed for a \$ and then prefixed with the index in brackets.

```
@fishtypes = ("gold", "cod", "pike");  
print "the second most useless fish is $fishtypes[1]";
```

However one should not confuse

```
$ni = "shrubbery";  
@ni = ("a small privet hedge", "make it larger");
```

```
print $ni[0];
```

The square brackets at the end of `$ni[0]` tell the interpreter to use the list called `@ni` not the scalar called `$ni`.

## A matter of context

Perl tries to act like a natural language, so it has what is known as ‘context’. Up until now we have had the scalar context, everything has been a scalar. With the introduction of the list we have the list context. This would be fine if everything were separated; however, you can refer to a list in both a scalar context and a list context; it will behave very differently depending on which it is in.

Simply put, in a list context, a list acts as described above. In a scalar context, it will return a scalar number of the largest element index<sup>1</sup>. By default lists will be handled in a list context.

## List operations

### scalar

To drive that last point home: if you want a list to be in a scalar context, but everything is happening in a list context, use the `scalar` operator:

```
@foo = (1 .. 10); # numbers one to ten inclusive
$foo[100] = 1;    # set the 101th element to 1

print scalar @foo; # print @foo in scalar context, results in 101
```

### reverse

The `reverse` operation takes a list and produces a new list with all the elements reversed. It does not modify the source list.

```
@ascending = (1 .. 10);
@descending = reverse @ascending;

# ascending is 1 2 3 4 5 6 7 8 9 10
# descending is 10 9 8 7 6 5 4 3 2 1
```

### pop and push

Treating a list as a stack is a very important part of data processing. The `pop` and `push` operators make this happen. `push @A $s` puts or ‘pushes’ a scalar value `$s` onto the end<sup>2</sup> of the list `@A`. Conversely, `pop @A` removes or ‘pops’ the last element of list `@A` and returns it as a scalar.

These operations do modify the list they work on.

---

<sup>1</sup>This is not a zero offset number, so normally it’s just the number of elements in the list

<sup>2</sup>The end is the part with the biggest index; alternatively, furthest to the right - which ever takes your fancy (both are the same)

```

push @list, "flibble";
push @list, "flobble";

print pop @list; # prints flobble

```

A more complicated example:

```

@list = (8, 3, "+"); # some maths in postfix notation

$op = pop @list;
if ($op eq "+") {
    $val = (pop @list) + (pop @list);
}

print "answer: $val\n" # prints the answer 11.

```

### shift and unshift

Just like `push` and `pop`, `unshift` and `shift` work on the other end of the list, that is, the beginning or left hand side. `shift @A` can be thought of shifting the whole list `@A` left one place and the left most value falling off and being returned as a scalar, conversely `unshift @A $s` shifts the scalar `$s` back on to the beginning of the list `@A`

These operations do modify the list they work on.

### qw

Rather than having to use the tedious C like style of literal Lists (arrays in C), there is a rather nifty quick way to do it:

```
@fishtypes = qw/cod gold place tuna/;
```

All the words (separated by spaces) in between the two forward slashes `/` form the elements of the list that the `qw` operation returns. The list contains the elements in the order that they were specified to `qw`

As with string literals, the bounds that the `qw` operator works over is limited by the `/` two slashes. If your string were to contain a slash, it would need to be escaped using a backslash (`\`). However, to avoid excessive escaping you may choose a different delimiting character, so long as it is the same at the start and the end. A very common delimiting character is `!` as you do not get many of them in text.

# Chapter 5

## Hashes

### is it a drug?, is it a symbol?

...no, it's an associative array. Still confused? So far we have had lists, we can use them as arrays with indices and a value is assigned to a particular place in the array - denoted by the index. However, wouldn't it be nice to be able to store the value by a name (called *keys* and then to look up that value based on the key later.

Hashes are Perl's third data type, they are what makes the language so very useful and popular, traditionally to create a 'hash table'<sup>1</sup> required a lot of work and some maths and then some scratching of heads, in Perl all this is done by the interpreter; therefore a hash can be truly conceptualised as an associative array.

There is only one caveat - no ordering information is preserved about the way elements were inserted into the hash.

There is only one limitation - the keys must be unique.

### The hash variable

The hash variable is identified by the leading % symbol, the normal naming rules then apply.

For the moment we'll put that aside because the most common way for a hash to be used is to address the individual key-value pairs. This is done using the syntax `$hashname{key}`. The key is any scalar value of variable (it will always be treated as a string though).

---

<sup>1</sup>this is their proper name in other languages to avoid ambiguity with the mathematical term hash.

## Assignment

To assign something to the hash, use the above syntax and assign the value to that key 'position'. eg:

```
$names{"sadiq"} = "jaffa";
$names{"flynn"} = "david";
```

To then get the value given a key, use the same syntax:

```
$name = "flynn";
print "$name is $names{$name}\n";
print "sadiq is $names{'sadiq'}\n";
```

If you try to access a key that doesn't exist, it will return `undef`.

## The whole hash

Using the `%` to address the hash refers to the whole hash. We may assign the whole hash to another hash variable (ie copy it), assign an empty hash to it, in the same way as with lists. This handily happens because a hash in list context is considered a list - where the elements of the list are the key value pairs, in the form (*key, value, key, value, key, value, ...*).

To make assignment a bit easier, we can use this to our benefit, compare with the earlier example:

```
%names = ("sadiq", "jaffa", "flynn", "david");
```

The values can then be looked up as before. However this is a little unwieldy, it is difficult to see what is a key and what isn't. This gives rise to the ...

## Big arrow

The big arrow is `=>`. There is nothing special about the big arrow, apart from the fact Perl treats it as a comma. Thus it is conventional to use it to separate the value from the key; if one were to want an English 'translation' it would be "gives rise to". eg:

```
%names = (
    "sadiq" => "jaffa",
    "flynn" => "david",
);
```

## Operations

### keys

If one wants all the keys in the hash, the `keys` function will return them as a list. The keys will be in no particular order and the ordering must not be relied upon.

```
%names = ("sadiq" => "jaffa", "flynn" => "david");  
  
print "$_ => $names{$_}\n" foreach (keys %names);
```

### values

Similar to `keys`, except that `values` returns a list of all the values in a hash.

### delete

As before with lists, if one wishes to delete a key value pair, assigning the key the value of `undef` will not do what you want<sup>TM</sup>; instead, that key will be associated with the value `undef`. Instead, one must use the `delete` function.

```
%names = ("sadiq" => "jaffa", "flynn" => "david");  
  
delete $hash{'sadiq'};  
  
# $hash{'sadiq'} will now return undef.
```

### exists

It has previously been pointed out that trying to get the associated value of a nonexistent key will return `undef`. This is not particularly good if one wishes to test that a key exists, as the key could actually be associated with the value `undef`. To get around this is the `exists` function, it simply returns true if the key does exist in the hash.

```
if (!exists $hash{'foobar'}) {  
    print "foobar is not a key to the \%hash\n";  
}
```

# Chapter 6

## Output

Outputting text is one of the most useful things a script can do, either to communicate with the user, or to act as the input to a pipe. It is also a great aid when debugging code, trying to work out what is going on can be greatly simplified by printing out state variables or simply that the execution has got to a specific point.

### Strings

Like the Bourne shell (and less like Java), Perl has two ways of representing strings. Formally called single quoted and double quoted string literals:

```
'hello world' # single quoted
"hello world" # double quoted
```

### The print function

First, before we go into the details of strings, a quick note on how to display them. Built into Perl there is a print function, it is similar to Java's *System.out.print()* (NB. not *System.out.println()*!) it is also similar to the Bourne shell echo function. Simply put, it is *print <thestring>* where *<thestring>* is a string literal that you are about to discover.

### Single quoted strings

Single quoted strings are the simplest form, where all the characters that occur between the beginning and terminating quote character are included verbatim. Well, that's the rule, there is of course an exception; if one wished to include a single quote in the string, it would be impossible as the quote would terminate the string. This is all made possible using the backslash (sometimes known as an escape character); simply put a backslash in front of the quote you wish to escape.

```
'It\'s my chimp!'    # Will be: It's my chimp
'I\'d like fish\n\\' # Will be: I'd like fish\n\
```

The above example raises a second point - what if a backslash is required at the end of the string? If it were a single backslash, it would escape the terminating quote and the string will carry on and everything gets very messy; to solve this, simply escape the backslash, ie `\\`.<sup>1</sup>

## Double quoted strings

These behave like the string literals you are used to in Java or the Bourne shell, where the backslash character and another letter can be used to represent control characters (newline, tab, backspace, etc.) For example:

```
$ perl -e 'print "line 1\nline 2 and here is a tab ->\t<-"
line 1
line 2 and here is a tab ->      <-
$
```

Some control characters:

<code>\n</code>	a newline (ends the current line)
<code>\t</code>	tab
<code>\\</code>	backslash ( <code>\</code> )
<code>\0nn</code>	any ASCII value given by the octal number <code>nn</code>
<code>\xnn</code>	any ASCII value given by the hexadecimal number <code>nn</code>

### 6.0.2 String operators

There is only one string operator, the concatenate `.` operator. It is used to join two strings together, eg:

```
"hello " . 'world' # results in the string "hello world" although, just
                  # as in addition or subtraction, neither operand is
                  # modified
```

---

<sup>1</sup>A word of advice, if you want to have a literal backslash, always escape it, it will always do what you think then.

# Chapter 7

## Numbers

Numbers in Perl can be used just as they can in any other language, although as with everything in Perl, there are some improvements.

### Maths

Most of the time it is possible when writing Perl scripts to not pay attention to the actual storage of numbers in Perl, they just work as you want them to when you want them to. However for cases where precision or large numbers are required, a basic understanding is very useful.

Perl attempts to represent all numbers as signed integers, if a larger or more precise number is required the interpreter will use floating point numbers. Conversion is automatic and mostly transparent.

### Operators

Operators in perl are very similar to that of Java and C.

**‘++’ and ‘--’** work as in C, That is, if placed before the variable, they act before the value of the variable is returned. Conversely if placed after the variable, they act after the variable’s value has been returned. Unlike C and Java, they are capable of working on characters in special circumstances. Like C and Java, avoid their use if you can and ensure that only one is used per statement.

### Multiplicative

**‘\*\*’** As per FORTRAN, returns the left operand raised to the power of the right.

**‘\*’** As per C/Java, multiplies the two operands.

‘/’ As per C/Java, floatingpoint divides the left operand by the right.

‘%’ As per C/Java, returns the modulus of two numbers.

### **Additive**

‘+’ As per C/Java, returns the sum of two numbers.

‘-’ As per C/Java, returns the difference of two numbers.

‘.’ Concatenates two strings.

### **Shift**

‘>>’ and ‘<<’ As per C/Java.

### **Bitwise**

‘|’ As per C/Java, returns the operands ORed together bit by bit.

‘^’ As per C/Java, returns the operands XORed together bit by bit.

‘&’ As per C/Java, returns the operands ANDed together bit by bit.

# Chapter 8

## Program Flow

Since both Java and Perl share some plenty of syntactical inspiration from C, you'll notice that many of the operators and program flow statements are pretty similar.

### Comparison operators

Like the Bourne shell, Perl has two equivalence operators. One for testing the equality of strings and another for numerical comparisons. The way the scalar data is interpreted is dependent on which operator is used. A common pitfall is to use the wrong one - while they seem to do the same job and yield the same results, there are cases when they don't. The `eq` operator tests for the equality of two scalars interpreted as strings. Thus:

```
"yes" eq "no" # Is false
"sue" eq "sue" # Is true
```

The `==` operator tests for the numerical equality of two scalars interpreted as numbers. Thus:

```
5 == 10      # Is false
2 == 2       # Is true
"2" == "2"   # Is true
```

However, this is why you should take care always to use the one you want:

```
"01" == "1"   # Is true (both strings are interpreted as the number 1)
```

*However*

```
"01" eq "1"   # Is false (the strings are not the same)
```

Perl has other numeric comparison operators which behave as they would in Java:

<code>\$a &gt; \$b</code>	<code>\$a</code> is numerically greater than <code>\$b</code>
<code>\$a &lt; \$b</code>	<code>\$a</code> is numerically less than <code>\$b</code>
<code>\$a &gt;= \$b</code>	<code>\$a</code> is numerically greater than or equal to <code>\$b</code>
<code>\$a &lt;= \$b</code>	<code>\$a</code> is numerically less than or equal to <code>\$b</code>
<code>\$a == \$b</code>	<code>\$a</code> is numerically equal to <code>\$b</code>
<code>\$a != \$b</code>	<code>\$a</code> is numerically not equal to <code>\$b</code>

For string comparison, the following operators exist, they compare strings ASCIIbetically. What is this one might ask? To sort something alphabetically you look at what position the letters are in the alphabet, ie, 'a' is alphabetically less than 'b', however in computing the alphabet is a small part of the ASCII character table<sup>1</sup> (8bit characters gives 256 different characters). Hence to sort something ASCIIbetically you look at the value of a character, eg, as all numbers occur before the letters in ASCII, any number is always ASCIIbetically less than any letter. Here is the table of the string comparisons in Perl:

<code>\$a gt \$b</code>	<code>\$a</code> is ASCIIbetically greater than <code>\$b</code>
<code>\$a lt \$b</code>	<code>\$a</code> is ASCIIbetically less than <code>\$b</code>
<code>\$a ge \$b</code>	<code>\$a</code> is ASCIIbetically greater than or equal to <code>\$b</code>
<code>\$a le \$b</code>	<code>\$a</code> is ASCIIbetically less than or equal to <code>\$b</code>
<code>\$a eq \$b</code>	<code>\$a</code> is ASCIIbetically equal to <code>\$b</code>
<code>\$a ne \$b</code>	<code>\$a</code> is ASCIIbetically not equal to <code>\$b</code>

To give an example:

```
$r lt "abc" # True if $r is ASCIIbetically less than "abc"
$r gt "abc" # True if $r is ASCIIbetically greater than "abc"
$r le "abc" # True if $r is ASCIIbetically less than or equal to "abc"
$r ge "abc" # True if $r is ASCIIbetically greater than or equal to "abc"
$r ne "abc" # True if $r is not equal to "abc"
```

## if

Perl's `if` construct uses virtually identical syntax to the Java `if` statement. Although Perl's `if` statement's braces <sup>2</sup> are mandatory - they must be used, where as in Java you can get away without using them in special circumstances. To demonstrate:

```
if ( "perl" eq "perl" ) {
    # Is true and statements in here will be run
}
```

---

<sup>1</sup>for more information on the ASCII character set, look up the ASCII man page on any Solaris machine (eg topaz) *man ASCII* if anyone has forgotten

<sup>2</sup>braces are sometimes known as "curly brackets", they look like `{ }`

```
if ( 5 < 3 ) {
    # Is false and statements inside of these brackets will not be executed
}
```

## else

This statement's use is identical to Java's `else` statement, although again, braces are mandatory. And with the design of forcing the programmer to use braces around the blocks of code following the `if` statement, the problem of the dangling-else is solved - Yay! An example:

```
if( $status eq "available" ) {
    # this will be executed if $status only contains "available"
} else {
    # do this if it didn't
}
```

## elsif

To all the alert people out there, this is *not* a spelling mistake. You couldn't (even though Java would allow it) do:

```
if ($foo == 1) {} else if ($foo == 2)
```

The *else-if* construct is called `elsif` in Perl. And if anyone has a problem with that, Larry Wall says "If you want to spell it with a second `e`, it's simple. Step 1 - Make up your own language. Step 2 - Make it popular.". That is the way it is :-)

```
if ( $sold eq "yes" ) {
    print "Good";
} elsif ( $sold eq "no" ) {
    print "Bad";
} else {
    print "Error: Unknown status.";
}
```

## for

Perl's `for` syntax is identical to that in Java; again braces around the looping block of code is mandatory. Namely: `for(initialiser; test-condition; loop-operation) {...}`.

```
for ($a = 1; $a <= 10; $a++) {
    print "a is $a\n";
}
```

Will output:

```
a is 1
a is 2
a is 3
a is 4
a is 5
a is 6
a is 7
a is 8
a is 9
a is 10
```

## **while**

It isn't that much of a surprise that Perl's while syntax doesn't differ from Java's (yes braces are still mandatory – they always are). Being, `while(loop-condition) {...};`

```
$a = 1;
```

```
while ($a <= 10) {
    print $a;
    $a++;
}
```

Will output:

```
a is 1
a is 2
a is 3
a is 4
a is 5
a is 6
a is 7
a is 8
a is 9
a is 10
```

## An example

In the first year, you should come across the Euclid's GCD algorithm; everyone will be pleased to know, that this will not be an exception.

```
#!/usr/bin/perl

use warnings;
use strict;

my $a = 36573;
my $b = 123;

while ($a != $b) {
    if ($a > $b) {
        $a -= $b;
    } else {
        $b -= $a;
    }
}

print "The GCD is $a\n";
```

## Finding help yourself

There are two very useful resources for Perl, the first, affectionately known as the 'Llama book'<sup>3</sup> also known as 'Learning Perl' by Schwartz and Phoenix; it like most books on Perl is an O'Reilly book.

Secondly there is a significant amount of documentation available using Perl's documentation utility *perldoc*. A list of all chapters may be seen using *perldoc Perl*, then subsequent chapters may be viewed using *perldoc chaptername* eg. *perldoc perlfunc* for details on Perl's built-in functions. Help on specific functions can be found with *perldoc -f function\_name*.

A word of caution first, It probably isn't a good idea to look at the *perldoc* documentation until you've read everything here - it is a little bit hard-core.

---

<sup>3</sup>because it has a picture of a Llama on the front