if - if/else statements

```
if (conditional expression) {
      #block of if statement
}
```

Curly brackets are required regardless of the number of lines to be executed

- The block of statements can be called a compound statement

```
if (conditional expressions)
   { #if block }
else
   { #else block }
```

if/elsif/else statement

```
if (conditional expression 1)
  { if block }
elsif ( conditional expression 2 )
  { block 2 }
elsif ( conditional expression 3 )
  { block 3 }
…
else
  { else block }
```

Only one block of code can possibly be executed

else portion is optional

- If present one block (and only one block) is guaranteed to execute

Statement modifiers

Perl always requires the curly braces when using the unless, until, while, if, for and foreach statements even if your code only has one statement in it

Fortunately Perl offers some flexibility in writing control structures (there is more than one way to do it :-). If there is a single statement inside the control structure Perl allows a shorter form called the modifier form. This form can be used with the if, unless, while and until statements

To use this form simply place **one** statement before the control structure

- Any single statement can be placed before the modifier
- Reverse the order of evaluation – the modifier is always evaluated first
- No cascading modifiers

Modifier form examples

```
if ($x > 10) {
   $y = $x;
}
# same as
$y = $x if ($x > 10);


while (++$index <= $fact) {
   $sum = $sum + $index;
}
# same as
$sum += $index while (++$index <= $fact);
```

unless – unless/else statements

```
unless (conditional expression) { unless block }

Or

unless (conditional expression )
   { unless block }
else
   { else block }

Or

unless (conditional expression )
   { unless block }
elsif ( conditional expression 2 )
   { block 2 }
…
else
   { else block }
```

The unless construct has the exact opposite logic of the if construct

- If the tested condition is false then the body of the loop is executed

while loop

The while loop executes a block of code as long as the conditional expression is true

- The while loop is a pretest loop

```
while (conditional expression)
  { while block }
```

Like just about every other looping construct there must be some action within the loop that eventually causes the conditional expression to become false

until loop

The until loop executes a block of code as long as the conditional expression is false

- Works just the opposite of the while loop
- The until loop is a pretest loop

```
until (conditional expression)
  { until block }
```

Unlike just about every other looping construct there must be some action within the loop that eventually causes the conditional expression to become true

do while / do until

Both the while and until loops have posttest versions

- The body of the loops are guaranteed to execute at least once

The conditional expression ends with a semicolon

```
do
  { do while block }
while ( conditional expression );

do
  { do until block }
until ( conditional expression );
```

for loop

The for loop is literally identical to the Java or C for loop

- For loop is a pretest
- Three pieces of information can be provided
  - Initialization step – can be multiple statements separated by commas
    - Only executed once at the start of the loop
  - Test step – conditional expression that must evaluate to true or false (0 or non-zero)
    - Tested before each execution of the for body
    - Defaults to true if omitted
  - Increment step – can be multiple statements separeated by commas
    - Executed at the end of each execution of the for body
    - Acts as a stand-alone statement at the end of the for block
- All three pieces of information can be omitted but the semicolons are required

for loop examples

Basic syntax

```
for (initialization step; test step; increment step)
   { for block }
```

Infinite loop – we’ll see ways to exit it later

```
for (;;)
   { for block }
```

More traditional for loop – counts from 1 to 10

```
for ($x = 1; $x < 10; $x++)
{
   print “$x\n”;
}
```

Other looping options

Loops using subscripts tend to be slower than loops that don't, because subscripts take a significant amount of time for Perl to evaluate. Also, subscripts can be use only on named arrays.

Perl has other options (TIMTOWDI) that somewhat overlap but other unique primary purposes.

- foreach – to iterate over, and possibly modify, elements in a list
- map – creates a list based on the contents of another list
- grep – select or count elements in a list

foreach loop

The foreach command steps through an array, hash, or list one element at a time

```
foreach $controlVariable (list) {
   # foreach block
}

foreach $number ( 1..24 ) {
   print “$number\n”;
}
```

foreach specifics

$controlVariable represents a particular item in the list. Changing the value of the control variable in the body of a foreach loop modifies the element in the list.

```
@array = ( 1..10 );
foreach $number ( @array ) {  #for each element in the array
   $number **= 2;                         # double the value
}
```

If the control variable is omitted the $_ variable is used

map

map evaluates a block or expression for each item in a list and returns a list composed of the result of each evaluated item

- $_ is an alias for the current element in the iteration
- It is generally considered bad style to alter $_
- The results of a map can be passed directly to another function
- Anything written with map COULD be written with a foreach pushing items onto the results array (but likely less efficiently)
- A comma is required if delimiters other than { } are used

Generic syntax

```
@returnedArray = map { code using $_} @originalArray

#doubles each item in @values
@doubled = map { $_ * 2 } @values;
```

grep

The grep operator selects or counts elements in a list based upon supplied criteria

- `$_` is an alias for the current element in the iteration
- The returned array is a collection of the items in the original array that returned true when evaluated
- In scalar context grep returns a count of the selected elements rather than the elements themselves
- Regular expressions can be used in the evaluation code – use / delimiters rather than { }
- A comma is required if delimiters other than { } are used

Generic syntax

```
@returnedArray = grep { code using $_ that
  returns true/false} @originalArray

@itemsWithTom = grep /tom/i, @originalList;
@trueItems = grep { $_ } @originalList;
```

Altering loop control

Perl provides labels and simple control statements to alter the flow of control in a control structure

next

- Can be executed in a while, until, for or foreach structure
- Skips the remaining statements in the body of that structure and performs the next iteration of the loop
  - Works with the innermost loop (if nested loops)
  - In the while and until structures the test step in evaluated immediately after the next statement executes
  - In the for and foreach structures the increment step executes and then the test step is executed

Altering loop control

- last
  - Can be executed in a while, until, for or foreach structure
  - Causes immediate exit from the structure
    - Mostly commonly used to escape early from a loop
    - Works with the innermost loop (if nested loops)
- redo
  - Can be executed in a while, until, for or foreach structure
  - Returns to the first statement in the body of the loop without evaluating the test step or performing the increment step
    - Used to repeat a particular iteration of a loop – maybe because of invalid data
    - Works with the innermost loop (if nested loops)

Labels

- Labels are optional but can be used to control the flow of a loop
- Labels must start a line, followed by a colon
    - Labels are not executed
    - Label act as a target for the next, last and redo commands
    - Allows these commands to work with something other than the innermost loop
- Any loop block except a do/while or do/until structure can have a label
- It is suggested that labels be in all upper case letters