Control Structures

Important Semantic Difference

- In all of these loops we are going to discuss, the braces are ALWAYS REQUIRED.
- Even if your loop/block only has one statement, you must include the braces.

if-elsif-else

- semantically the same as C/C++
- syntactically, slightly different.
  - note the spelling of elsif

```
if ($a > 0){
  print "\$a is positive\n";
} elsif ($a == 0){
  print "\$a equals 0\n";
} else {
  print "\$a is negative\n";
}
```

unless

- another way of writing **`if (!…) {…}`**
- analogous to English meaning of "unless"
- **`unless (EXPR) BLOCK`**
  - "do BLOCK unless EXPR is true"
  - "do BLOCK if EXPR is false"
- can use **`elsif`** and **`else`** with unless as well
  - caution - **`elsif`** is still an **`if`**. It's not an elsunless.

while/until loops

- **`while (EXPR) BLOCK`**
  - "While EXPR is true, do BLOCK"
  - Check the condition. If it's true, do the block. Repeat.
- **`until (EXPR) BLOCK`**
  - "Until EXPR is true, do BLOCK"
  - "While EXPR is false, do BLOCK"
  - another way of saying **`while (!…) {…}`**
  - Check the condition. If it's false, do the block. Repeat.

while magic

- If and only if the only thing within the condition of a while loop is the readline operator: **`while (<$fh>) { }`**
- perl automatically translates this to: **`while (defined($_ = <$fh>)) { }`**
  - **`$_`** holds the line that was just read.
  - When **`<$fh>`** returns **`undef`** (ie, file completely read), loop terminates.
- This does NOT happen anywhere else!
  - **`<$fh>;`**
    - Reads a line and throws it away, does NOT assign to $_
  - **`while (<$fh> and $x > $y) { }`**
    - Reads a line, throws it away. Compares $x to $y. Does not assign to $_

do

- Execute all statements in following block, and return value of last statement executed
- When modified by while or until, run through block once before checking condition

```
do {
  $i++;
} while ($i < 10);
```

for loops

- Perl has 2 styles of **for**.
- First kind is virtually identical to C/C++
- for (INIT; TEST; INCREMENT) { }
-
```
for (my $i = 0; $i < 10; $i++){
   print "\$i = $i\n";
}
```
- perform the initialization. Check the test. If it's true, do the block, then increment. Check the test. etc.

foreach loops

- Second kind of for loop in Perl
  - no equivalent in core C/C++ language
- **foreach VAR (LIST) {}**
- each member of LIST is assigned to VAR, and the loop body executed
-
```
my $sum;
foreach my $value (@nums){
  $sum += $value;
}
```

More About for/foreach

- **`for`** and **`foreach`** are synonyms
  - Anywhere you see "for" you can replace it with "foreach" and viceversa
    - Without changing ANYTHING ELSE
  - they can be used interchangeably.
  - usually easier to read if conventions followed:
    - `for (my $i = 0; $i<10; $i++) {}`
    - `foreach my $elem (@array) {}`
  - but this is just as syntactically valid:
    - `foreach (my $i = 0; $i<10; $i++) {}`
    - `for my $elem (@array) {}`

foreach triviata

- **`foreach VAR (LIST) {}`**
- while iterating through list, VAR becomes an *alias* to each member of LIST
  - Changes to VAR within the loop affect LIST
  - ```
    my @array = (1, 2, 3, 4, 5);
    foreach my $num (@array) {
      $num *= 2;
    }
    ```
  - @array now ➔ (2, 4, 6, 8, 10)
- if VAR omitted, $_ used instead
  - ```
    my $sum = 0;
    foreach (@array) {
      $sum += $_;
    }
    ```

foreach loop caveat

- ```
  my $num = 'alpha';
  for $num (0..5) {
    print "num: $num\n";
  }
  ```
- Upon conclusion of the loop, $num goes back to 'alpha'!
  - The for loop creates its own lexical variable, even though you didn't specify `for my $num (0..5) { … }`
- For this reason, it is always preferred to *explicitly* declare the variable lexical to the loop, to avoid the possible confusion.
  ```

Best Practice

- There is *rarely* any need to use C-style for loops in Perl
- If you want to iterate over a count value:
  - **`foreach my $count (1..$total) { }`**
  - **`foreach my $i (0..$#array) { }`**
- Only time you need a C-style for loop is to increment by something other than 1:
  - **`for (my $v=0; $v < $tot; $i+=3) { }`**

`foreach (<$fh>){ }` vs `while (<$fh>){ }`

- Both constructs appear to do the same thing.
  - Assign each line of **`$fh`** to **`$_`**, execute loop body
- The difference is internal
  - **`foreach`** takes a LIST
    - evaluates **`<$fh>`** in list context, once
  - **`while`**'s condition is **`defined($_ = <$fh>)`**
    - evaluates **`<$fh>`** in scalar context, repeatedly
- **`foreach`** will read the entire file into memory at once
  - each element of resulting list is then assigned to **`$_`**
- **`while`** will read the file line by line, discarding each line after it's read
  - FAR more efficient. Always use **`while (<$fh>) { }`**

given (5.10 exclusive)

- Compare one variable against a series of different values/conditions

```
use feature ':5.10';
given ($foo) {
    when (5) { say '$foo == 5'; }
    when ('abc') { say '$foo eq "abc"'; }
    when (undef) { say '$foo is undefined'; }
    when (@nums) { say '@nums contains $foo'; }
    when (%age_of) { say '$foo is key in %age_of; }
    when ($_ > 100) { say '$foo > 100'; }
    when ($_ lt 'xyz') { say '$foo lt "xyz"'; }
    default { say 'I know nothing about $foo; }
}
```

- given an array, can check for identical array or an element
- given a hash, can check for identical keys or a key's existence

given fallthrough

- Fallthrough does not happen by default. As soon as one **when** matches, **given** terminates.
- To continue checking other conditions, end with **continue;**
-
```
given (@students) {
    when ('Joe') {
       say 'Joe is in the class';
       continue;
    }
    when ('Mary') {
       say 'Mary is in the class';
       continue;
    }
    when ('Adam') {
       say 'Adam is in the class';
    }
}
```
- To terminate the **when** and entire **given** blocks early, use **break**
- Warning - these are only valid for given/when - they don't have the same effects you'd expect from the C/C++ versions.

Reading it in English

- Perl has a cute little feature that makes simple loop constructs more readable
- If your **if**, **unless**, **while**, **until**, or **foreach** block contains only a single statement, you can put the condition at the end of the statement:
- **if ($a > 10) {print "\$a is $a\n";}**
- **print "\$a is $a\n" if $a > 10;**
- Using this modifier method, braces and parentheses are unneeded
- This is syntactic sugar – whichever looks and feels right to you is the way to go.

Loop Control – next, last, redo

- **last** ➔ exit innermost loop
  - equivalent of C++ **break**
- **next** ➔ begin next iteration of innermost loop
  - (mostly) equivalent of C++ **continue**
- **redo** ➔ restart the current loop, without evaluating conditional
  - no real equivalent in C++
- Note that Perl does not consider **do** to be a looping block. Hence, you cannot use these keywords in a do block (even if it's modified by while)

Breaking Out of More Loops

- **next**, **last**, **redo** operate on innermost loop
- Labels are needed to break out of nesting loops

```
TOP: while ($i < 10){
  MIDDLE: while ($j > 20) {
    BOTTOM: for (@array){
              if ($_ == $i * $j){
                last TOP;
              }
              if ($i * 3 > $_){
                next MIDDLE;
              }
            }
            $j--;
          }
          $i++;
        }
```

goto

- yes, it exists.
- No, don't use it.