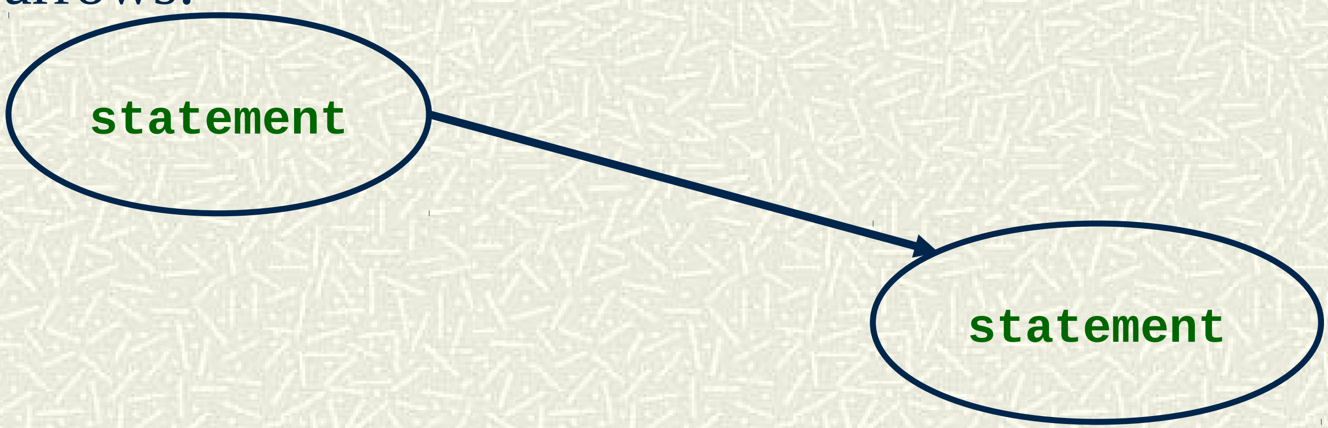
Control Structures in Perl

Controlling the Execution Flow in a Program

Control flow in programs

- A program is a collection of statements. After the program executes one statement, it "moves" to the next statement and executes that one. If you imagine that a statement is a stepping stone, then you can also think of the ***execution flow*** of the program as a sequence of "stones" connected by arrows:



Sequences

- When one statement physically follows another in a program, as in

```
$number1 = <STDIN>;
$number2 = <STDIN>;
$sum = $number1 + $number2;
```

the execution flow is a simple ***sequence*** from one statement to the next, without choices along the way. Usually the diagrams use rectangles to represent the statements:

Alteration of flow

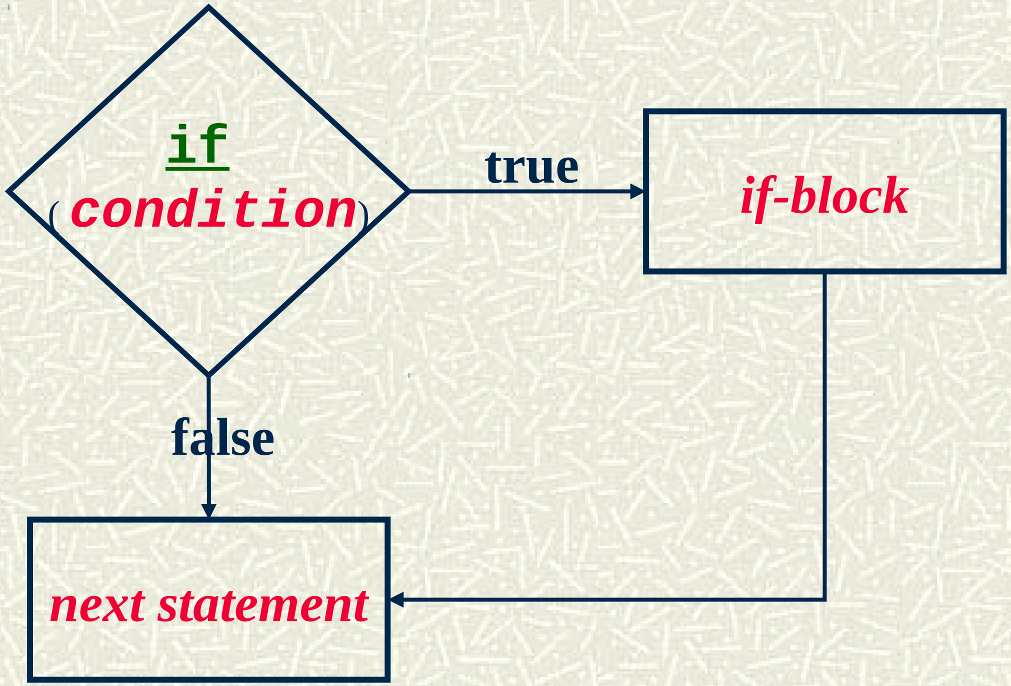
- Some statements alter the sequential flow of the program. You have already seen a few of these. The ***`if`*** statement is a type of ***selection***, or ***branching***, statement. Its syntax is

  `if` (***condition***) { ***block*** }

  in which ***condition*** is an expression that is evaluated to determine if it is ***true*** or ***false***. If the ***condition*** is true when the statement is reached, then the ***block*** is executed. If it is false, the ***block*** is ignored. In either case, whatever statement follows the **`if`** statement in the program is executed afterwards.

The `if` statement

- The flow of control through the **`if`** statement is depicted by the following flow-chart (also called a flow diagram):



Conditions

- The condition in an **`if`** statement can be any expression. Although any expression can be used as a condition, in programs that follow good principles of software design, the condition is usually one that is built from ***relational operators*** and/or ***logical operators***.
- For example,

```
$x > $y
```

is a condition that is true if the value of **`$x`** is greater than the value of **`$y`** when the condition is reached.

Relational operators

Relational operators are operators that compare two expressions. In math we use operators like **>**, **<**, and **≠** to compare numeric expressions. There is no symbol "≤" on the keyboard, so we use a pair of symbols "**<=**" instead. There are six numeric relational operators in Perl, which are listed in the next slide.

Numeric relational operators

- The ***numeric relational operators*** in Perl are

Operator	Example	Meaning
`>`	`$x > $y`	true if `$x` is greater than `$y`
`<`	`$x < $y`	true if `$x` is less than `$y`
`==`	`$x == $y`	true if `$x` equals `$y`
`!=`	`$x != $y`	true if `$x` does not equal `$y`
`>=`	`$x >= $y`	true if `$x > $y` or `$x == $y`
`<=`	`$x <= $y`	true if `$x < $y` or `$x == $y`

Comparing words

- When you look up words in a dictionary, or sort names, you use an implicit rule for ordering strings, usually called dictionary order.
- **a** < **b** < **c** < … < **z** orders the letters, and two words ***w*** and ***v*** are ordered using the rules:

1. If first letter of ***w*** < first letter of ***v***, then ***w is less than v***
2. If the first n letters of ***w*** and ***v*** are the same, but the (n+1)st of ***w*** < (n+1)st of ***v***, then ***w is less than v***
3. If ***w*** is a prefix of ***v*** then ***w is less than v***.
4. If the words are identical, then ***w equals v***

Comparing strings

- In Perl, a different rule is used to order the characters, but the rule for words remains the same. The characters are ordered by their ASCII* values. In the ASCII ordering, all punctuation precedes digits, which precede uppercase letters, which precede lowercase letters. In UNIX, you can type "**`man ascii`**" to see the ASCII table. Thus,

```
blank < …
      < 0 < 1 < 2 < … < 9
      < … < A < … < Z
      < a < … < z
```

** (It is a bit more complex than this, but for now this is how you should think of it.)*

Comparing strings

- Examples:

`'A'`	is less than	`'a'`
`'Zoo'`	is less than	`'apple'`
`'apple'`	is less than	`'zoo'`
`'100'`	is less than	`'20'`
`'111'`	is less than	`'a'`

- The string relational operators are listed on the next page. Note that they are different from the numerical operators. You MUST use these when comparing strings.

String relational operators

- The ***string relational operators*** in Perl are

Operator	Example	Meaning
`gt`	`$x gt $y`	true if `$x` is greater than `$y`
`lt`	`$x lt $y`	true if `$x` is less than `$y`
`eq`	`$x eq $y`	true if `$x` equals `$y`
`ne`	`$x ne $y`	true if `$x` does not equal `$y`
`ge`	`$x ge $y`	true if `$x gt $y` or `$x eq $y`
`le`	`$x le $y`	true if `$x lt $y` or `$x eq $y`

Logical values

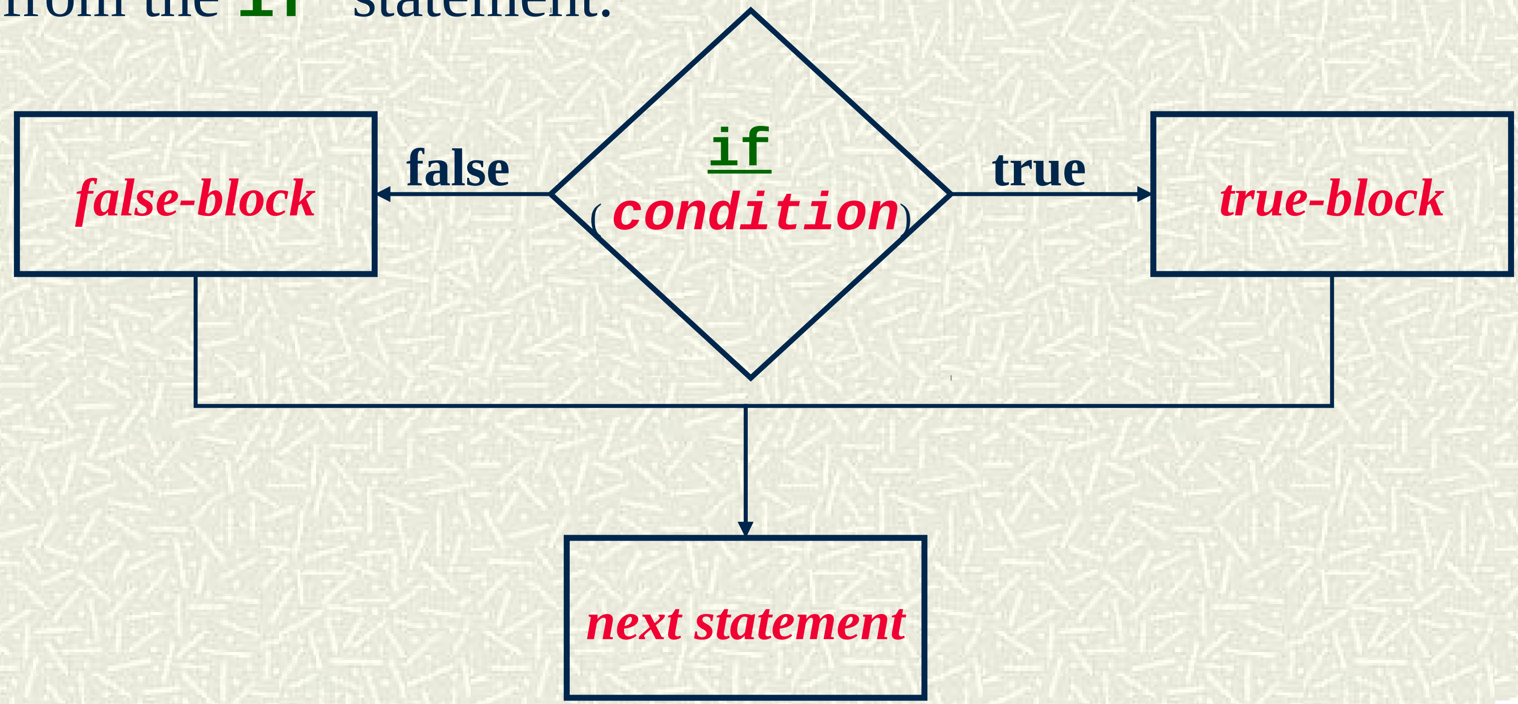
- Perl will convert all expressions to true and false, even if they have no relational operators in them. The rules are:
  - Any number other than 0 is true; 0 is false.
  - The empty string (`''` or `""`) is false.
  - A string containing only a zero, i.e., `"0"` or `'0'`, is false.
  - Anything that is ***undefined*** is false.
- Thus, the following expressions are true:

  `"hello"   62   "00"   "\t"   "0.0"`

  and these are false: `"0"   '0'   0`

The **`if-else`** Statement

- The flow of control through the **`if-else`** statement is depicted by the following flow-chart. Notice how it differs from the **`if`** statement.



The `if-else` statement

- The **`if`** statement allows an optional else clause:

  `if (` ***condition*** **`) {`** ***true block*** **`} else {`** ***false block*** **`}`**

  Its meaning is:

  If the ***condition*** is true when the statement is reached, then the ***true-block*** is executed; otherwise the ***false-block*** is executed. In either case, whatever statement follows the **`if`** statement in the program is executed afterwards.

Example 1

```
my ($a, $b);
print "Enter 2 numbers, one per line\n";
chomp($a = <STDIN>);
chomp($b = <STDIN>);
if ( $a < $b ) {
    print "$a then $b\n";
}
else {
    print "$b then $a\n";
}
```

What does this do?

Example 2

- What about this one?

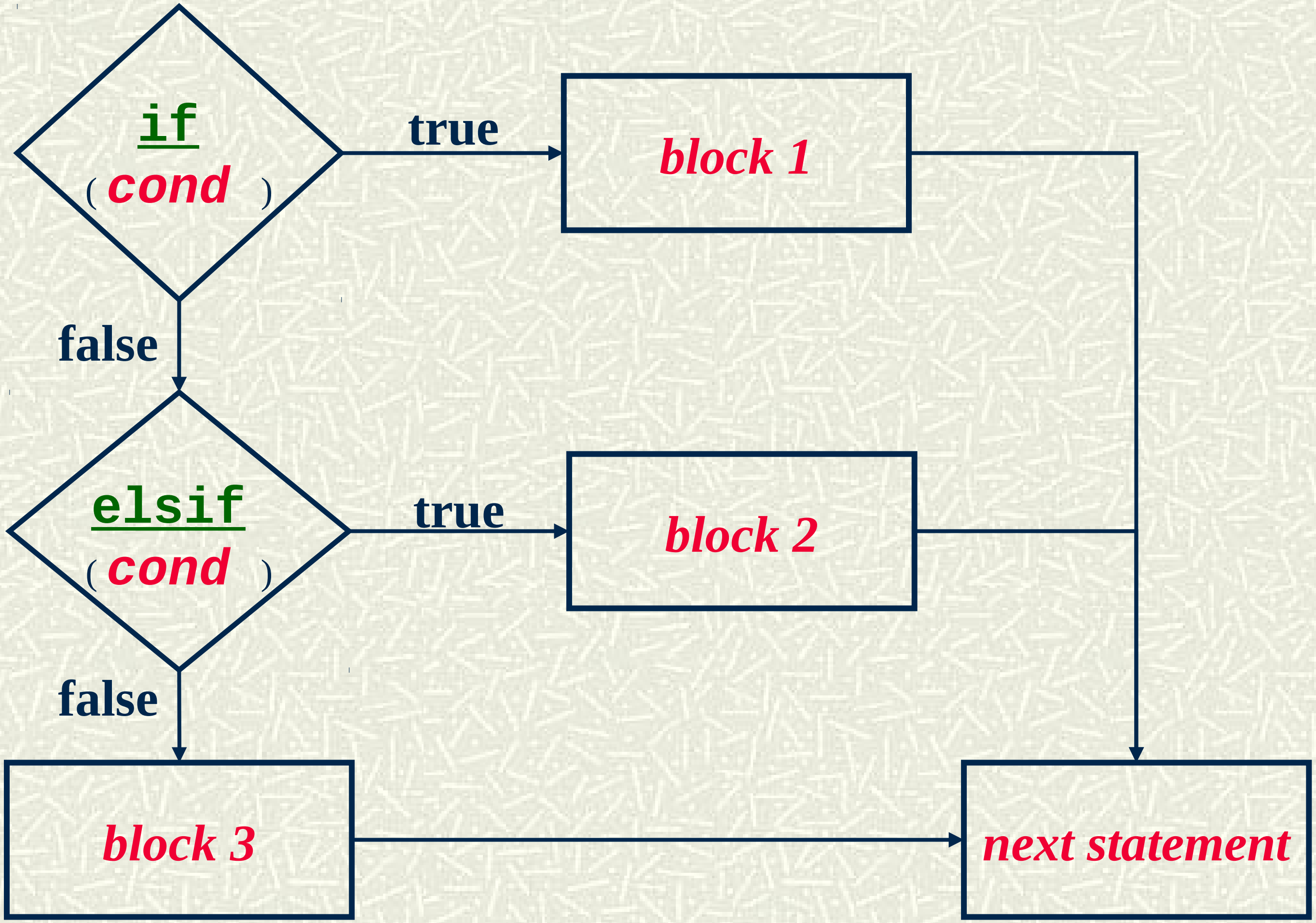
```
my ($a, $b, $c);
print "Enter 3 increasing numbers, 1 per line\n";
chomp($a = <STDIN>);
chomp($b = <STDIN>);
chomp($c = <STDIN>);
if ( $a**2 + $b**2 == $c**2 ) {
     print "Right triangle\n";
}
else {
     print "Not right triangle\n";
}
```

Optional `elsif` Clauses

- The **`if-else`** statement can be augmented with **`elsif`** clauses, which are like a shortcut for "**`else if …`**". Notice there is no '**`e`**' before the '**`i`**' in **`elsif`**.
- The syntax is

```
if    ( condition ) { block 1 }
elsif ( condition ) { block 2 }
elsif ( condition ) { block 3 }
# and so on …
else  {block N }
```

The `if-elsif-else` Flowchart



Example

```
my ($a, $b);
print "Enter 2 numbers, one per line\n";
chomp($a = <STDIN>);
chomp($b = <STDIN>);
if ( $a < $b ) {
    print "$a is less than $b\n";
} elsif ( $b < $a ) {
    print "$b is less than $a\n";
} else {
    print "The numbers are equal.\n";
}
```

Repetition statements

- A ***repetition statement*** is a statement that allows a block to be executed repeatedly, under the program's control. These statements are called ***loops***. When you see the flow charts for them you will understand why.
- What if you want to write a program to print out the squares of the first 100 integers? Without repetition statements, your program would require 100 print statements (assuming one number per statement.)
- Now what if you wanted to print the first 1000 squares, or 10,000 squares?

Repetition statements

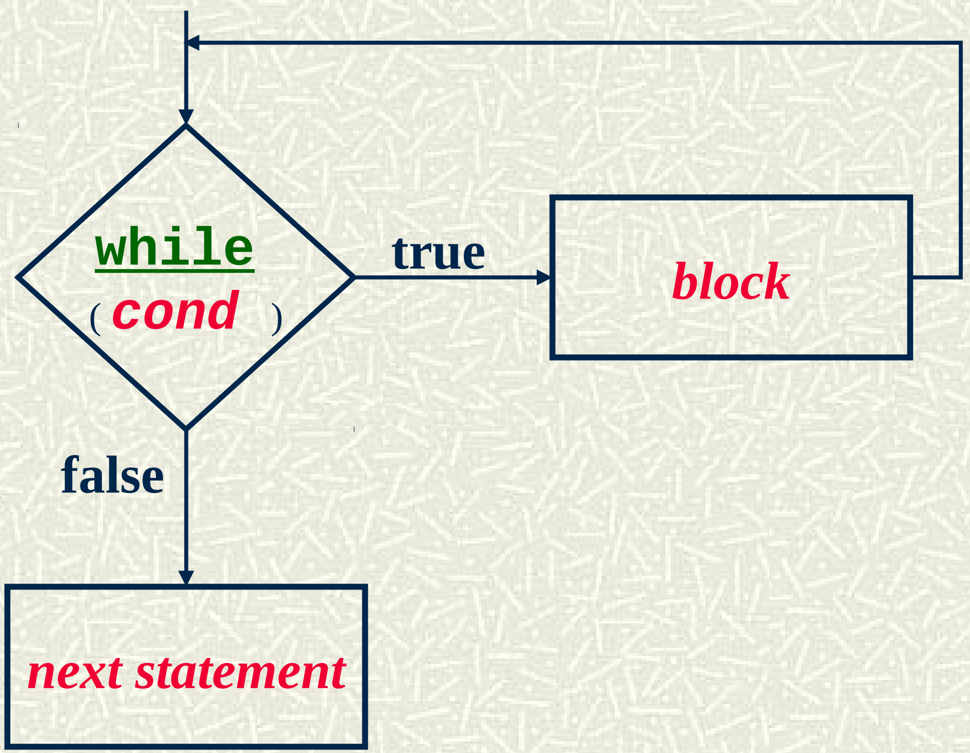
- Repetition statements allow your program to process more data without increasing in size. They also allow it to process unknown amounts of data.
- The first loop we examine is the `while` loop:

  `while (` ***condition*** **`) {`** ***block*** **`}`**

- When the statement is first reached, the condition is evaluated. If it is true, the block is executed and the process repeats -- the condition is evaluated and if true, the block is executed and the process repeated. This goes on until the condition is false.

The `while` loop flowchart

- The `while` loop flowchart looks like this:



`while` loop: Example

- Here is a simple example:

```
my  $rabbits = 2;
while ( $rabbits < 1000 ) {
    $rabbits = int (1.618*$rabbits);
    print "There are now $rabbits rabbits.\n";
}
```

- This loop will test whether `$rabbits < 1000.` If so, it sets `$rabbits` to a larger value, the smallest integer in `1.618*$rabbits,` prints out a message, and continues.

`while` loop: Example 2

- Here is another example:

```
my $sqrt = 0;
while ( $sqrt * $sqrt <= $number ) {
     $sqrt = $sqrt + 1;
}
```

- Given a non-negative number `$number`, this loop exits when `$sqrt` is the smallest integer value such that `$sqrt*$sqrt > $number`. In other words, `($sqrt-1)*($sqrt-1)<= $number` and `$number < (sqrt)*($sqrt).` Therefore `$sqrt-1` is the largest number whose square is at most `$number`. This is the integer part of `sqrt($number)`.

Use of `while` loops

- The `while` loop is most useful when you do not know how much data there is in the input source. In particular, it is perfect for reading data from input sources whose size is unknown in advance, such as files and the keyboard. You will learn this soon.

Looking ahead: the `split()` function

- A very useful tool for processing input text is the **`split()`** function. **`split()`** is used to break apart a string into a list of substrings. To be precise:

  `split( /`*pattern*`/, `*string*`)`

  returns a list consisting of the parts of ***string*** that don't match ***pattern***. E.g.,

```
my $string = 'name:id:email:phone';
my @fields = split(/:/, $string);
print "@fields\n";
# prints: name id email phone
print "$fields[2]\n";
# prints: email
```

Processing tab-separated data files

- Very often, data is in tab-separated or comma-separated files. The **split()** function can extract the data into fields. Suppose that the input lines contain name, id, email, and phone data, separated by tabs.

```
while ( $line = <STDIN> ) {
    chomp($line);
    ($name,$id,$email,$phone) = split(/\t/,$line);
    print "$name has email address $email\n";
}
```

- You can extract whatever information you want. In this case I ignored **$id** and **$phone** and just used **$name** and **$email**.

Splitting on white space

- A very common task is to split lines on any amount of whitespace. To split on whitespace, use either the pattern **`/\s+/`** or the string consisting of a single blank, **`" "`**, as follows.

```
while ($line = <STDIN> ) {
    @words = split(/\s+/, $line);
    #equivalently, @words = split(" ", $line);
    $wc += @words;
}
print "There are $wc words in the files.\n";
```

-

Parsing files

- In a PDB file, lines that represent ATOM records begin like:

```
ATOM    1  N   MET A   0    24.512   8.259  -9.688
```

- We will use a new input operator to illustrate. Instead of using the **`<STDIN>`** to input a line, we will use **`<>`** without the **`STDIN`** in between the brackets. This “diamond operator” as it is called can read from files whose names are given on the command line. The next slide illustrates.

Parsing files

- In a PDB file, lines that represent ATOM records begin like:

```
ATOM    1  N   MET A   0   24.512   8.259  -9.688
```

- If we wanted to find all atoms of a specific type, say nitrogen, we could use the code

```
while ($line = <> ) {
    @fields = split(/\s+/, $line );
    if ( $fields[0] =~ /ATOM/ ) {
        if ($fields[2] eq "N" ) {
            print "Atom $fields[1] is N.\n";
        }
    }
}
```

The `until` loop

- The `until` loop

  `until (` ***condition*** **`) {`** ***block*** **`}`**

  is structurally the same as the `while` loop. The only different is that the block is repeatedly executed until the condition is true. The flowchart is the same.

`for` loops

- An alternative loop is the `for` loop, so named because it starts with the word "`for`":

  `for (` ***initialization*** `;` ***condition*** `;` ***update*** `) {` ***block*** `}`

- When the loop is first reached,

1. The ***initialization*** section is executed; then
2. The ***condition*** is tested;
3. If the ***condition*** is true, the ***block*** is executed and the ***update*** is executed, and it starts again in step 2. Otherwise the loop is exited and the following statement is executed.

Example

- The following is a simple but very typical example:

```
for ( my $count = 0; $count < 10; $count++ ) {
    print "$count\n"; # any statement works!
}
```

- When the loop is reached, **`$count = 0`** is executed and then **`$count < 10`** is tested. Since **`0 < 10`** is true, the **`print`** statement is executed, then **`$count++`** is executed, making **`$count ==1`**. Now the condition is re-tested. Since **`1 < 10`**, the **`print`** is executed again. This continues until **`$count == 10`**, when the loop exits. Therefore the **`print`** statement prints the values **`0`**, **`1`**, **`2`**, …, **`9`**.

Safety of `for` loops

- The `for` loop, when used in specific ways, is a ***safe*** loop to use because it always terminates. If you always use a `for` loop with a condition in the form `$var <` ***expression*** and the update is always `$var++`, the loop will always terminate in a predictable number of iterations. For example, in this loop:

```
for ($var = start; $var < final; $var++ ) {
     # any block of statements
}
```

the enclosed block will be executed ***final-start*** times.

Dangerous uses of `for` loops

- Sometimes, you will see programs in which `for` loops are used in precarious ways, not in the spirit of their intended use. A common use is:

```
for (; ; ) {
    # block of statements
}
```

- When the condition is empty, it is true. It cannot be changed, and so nothing will ever stop this loop. This is an example of an ***infinite loop***. ***The loop block must have a statement to force the loop to exit.*** One such statement is "`last`".

last

- The "**`last`**" statement (and I do not mean the ***last*** statement but the statement named "**`last`**") causes the innermost enclosing loop to exit immediately:

```
for (; ; ) {
    # useful stuff here
    $response = <STDIN>;
    if ( $response eq 'q' ) {
        last;
    }
}
```

- In this example, when the input is a '**`q`**' then the loop exits. The next statement executed is whatever follows the loop.

The `next` statement

- The **`next`** statement causes the program to advance to the next iteration of the loop, ignoring whatever follows it in the block. In a **`for`** loop, this means that the update is executed and then the loop condition is tested. For example:

```
for ( my $i = 0; $i < 30; $i++) {
    if ( $i % 3 != 0 ) {  next; }
    print "$i\n";
}
```

- This prints multiples of 3, because when **`$i`** is not divisible by 3, the **`next`** is executed, bypassing the **`print`** statement.

Another example of **next**

```
while ($line = <STDIN> ) {
    if ( $line !~ /\b[A-Z][A-Z]+\b/ )
    {
        next;
    }
    print $line;
}
```

- This will print all lines and only lines that contain words of at least two uppercase letters, because if a line does not match the pattern **/\b[A-Z][A-Z]+\b/**, **next** causes the program to go back to the condition, bypassing the **print** statement.

The **`foreach`** statement

- This is one of the most versatile looping statements. The **`foreach`** statement syntax is

  `foreach $alias (` ***list*** **`) {`** ***block*** **`}`**

  in which **`$alias`** is a variable and ***list*** is either an array variable or a list literal.
- The meaning of the **`foreach`** statement is that it executes the block for each value in the list, setting **`$alias`** to that value. If the value in the list is a variable, then **`$alias`** is actually an ***alias*** for it, i.e., it acts like another name for the variable itself.

The `foreach` statement

- The principal advantage of the **`foreach`** statement over other looping statements is that because it iterates over all elements of the list, your program does not need any type of stopping test in the loop condition.
- You do not have to bother with counting the size of the array or knowing when the last element is reached because the **`foreach`** logic internalizes this.

Example

- Consider this example.

```
my $item;
my @array = ( 1, 2, 3, 4, 5 );
foreach $item ( @array ) {
    $item = 2 * $item;
}
print "@array\n"; # prints 2 4 6 8 10
```

- In this example, `$item` successively "becomes" `$array[0]`, then `$array[1]`, up to `$array[4]`. `foreach` "knows" when to stop. Because `$item` is changed, so is each array element; thus the array is changed as a result of executing this loop.

Another Example

```
my @vector = (1 .. 100 );
my ($item, $prev );
$prev = 1;
foreach $item ( @vector ) {
    $item = $item * $prev;
    $prev = $item;
}
print "@vector\n";
```

- This is tricky to analyze. What does it do? Try guessing before you run the demo program.

Logical Operators

- A logical operator is an operator whose operands have one of the logical values, **`true`** or **`false`**, and whose application results in a logical value as well.
- In ordinary speech, we use the words "and" and "or" as logical operators. For example, one might say,

  "If the class is open <u>or</u> you get an overtally, you can register for the class."
- You know that the "or" here means if either clause is true, the whole premise is true.

Perl's Logical Operators

- Perl has three logical operators, named **`and`**, **`or`**, and **`not`**. It also has symbolic versions of these, **`&&`** (and), **`||`**(or), and **`!`** (not).
- The symbolic versions differ in that they have higher precedence, which means that when you omit parentheses around expressions, funny things may happen. For now, we start with **`and`**, **`or`**, and **`not`**.
- The logical meanings of word versions and symbolic versions are the same. I will refer to the abstract logical operators as AND, OR, and NOT.

Logical Semantics

- Logical-AND is called ***conjunction***: if ***a*** and ***b*** are logical expressions, then

  a **`and`** ***b*** is true if and only if both ***a*** and ***b*** are true.

- Logical-OR is called ***disjunction***: if ***a*** and ***b*** are logical expressions, then

  a **`or`** ***b*** is true if and only if either ***a*** or ***b*** is true, which also includes the possibility that both are true.

- Logical-NOT is called ***negation***: if ***a*** is a logical expression, then

  `!`***a*** is true if and only if ***a*** is false.

Truth Table for Logical AND

- A logical operation can be characterized completely by a truth table, which is a table that defines the value of the operation based on its operands. The truth table for logical-AND is below:

You can see that **A AND B** is true only when both A and B are true and false otherwise.

A	B	A AND B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Truth Table for Logical OR

You can see in the table to the right that **A OR B** is true when either A or B is true and false only when both are false.

A	B	A OR B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Truth Table for Logical NOT

You can see in the table to the right that **NOT A** is true when A is false and false when A is true.

A	NOT A
TRUE	**FALSE**
FALSE	**TRUE**

Example

```
print "Enter 3 numbers in increasing order.\n";
chomp($x = <STDIN>);
chomp($y = <STDIN>);
chomp($z = <STDIN>);
if ( not ( $x <= $y and $y <= $z )) {
    print "The numbers are not in increasing
  order.\n";
```

- In this example, the condition is of the form ***not (A and B)***. It is true if and only if ***A*** is false or ***B*** is false, which means either **`$x > $y`** or **`$y > $z`**, implying the numbers are not in order.

DeMorgan's Law

- The condition
  `( not ( $x <= $y and $y <= $z ))`
  could also have been written
  `( $x > $y or $y > $z )`
  which is much easier to understand. The first just illustrates the use of the **`not`** and **`and`** operators.
- The equivalence is a result of ***DeMorgan's Law***, which basically states that the following are equivalent:

```
NOT ( A AND B )  = (NOT A) OR (NOT B)
NOT ( A OR B )   = (NOT A) AND ( NOT B)
```

Lazy Evaluation

- Logical AND and OR are ***lazy evaluation operators*** in Perl. This means that they do the least work necessary to get their results. Consider the statement

```
if ( a and b ) { print "true"; }
```

If ***a*** is false, the condition cannot possibly be true, so Perl does not bother to even evaluate ***b***.

- Similarly, in

```
if ( a or b ) { print "true"; }
```

if ***a*** is true, there is no need to evaluate ***b***, since the condition must be true even if ***b*** is false.

Use of Lazy Evaluation

- You can take advantage of the fact that AND and OR are lazy operators in your code. Consider this condition:

  ```
  if ( $count != 0 and $sum/$count > 10 ) { … }
  ```

- If Perl did not use lazy evaluation, this would cause a divide-by-zero error if `$count` were `0`. But with lazy evaluation, if `$count` is `0`, `$count != 0` is false and execution will never reach the division operator.

Digging Deeper

- The logical operators AND and OR do not really return logical values; they return the value of the last expression they evaluate, which is then treated as a logical value if used in a place where a logical value is expected. For example, consider

```
$count = ( $count or 1 );
```

- If `$count > 0`, then `$count` is true and because of lazy evaluation, the `or` will not evaluate the `1`. The last value the `or` evaluated was the value of `$count`, so `$count` gets its own value again. But if $`count == 0`, it is false, so the `or` will evaluate the `1`. Thus, `1` is the last value evaluated, so `$count` will be assigned `1`. No matter what, `$count > 0` afterwards!

More Lazy Evaluation

- This can be used, for example, to simplify the end of the averaging program from earlier in these slides. By using this:

```
$count = ($count or 1);
print "The average is ", $total/$count, "\n";
```

we are guaranteed that the average is either 0, if no numbers were entered (0/1 = 0) or whatever the true average is.

The symbolic logical operators

- You can use the symbolic versions of the logical operators instead of the word versions, but only if you understand that they are higher precedence operators. The previous statements can be

```
$count = $count || 1;
print "The average is ", $total/$count, "\n";
```

- because the `||` has higher precedence than the assignment operator. The `or` operator does not, which is why in the preceding slide, the expression was enclosed in parentheses.

Statement modifiers

- This is a topic of not very high importance. It is yet another instance of Perl's providing many more ways to do things than are necessary, but which are sometimes more convenient or easier to read.
- All of the control statements, i.e., the **`if`**, **`while`**, **`for`**, **`next`**, **`last`**, and so on, can be used to modify a ***simple single statement*** by appending the control statement to the end of the statement. The requirement is that it must be a ***simple statement*** that is modified, not a block and not a control statement.
- The next slide will demonstrate the syntax.

Statement modifier example 1

- Instead of writing

```
if ( $line !~ /\b[A-Z][A-Z]+\b/ ) {
        next;
}
```

you can write

```
next if  $line !~ /\b[A-Z][A-Z]+\b/ ;
```

- It does exactly the same thing. Notice that there is no semicolon after **`next`** and no parentheses enclosing the condition.

Statement modifier example 2

- Instead of writing

```
my $var = 0;
while ( $var < 10 ) {
    print $var++, "\n";
}
```

you can write

```
my $var = 0;
print $var++,"\n" while $var < 10;
```

It has the same effect. Remember that it must be a simple statement that is modified, not a block and not a control statement.

Summary

- Flowcharts are a means of displaying the execution flow in a program or group of statements.
- Conditions are expressions that are evaluated as being true or false.
- Perl has different relational operators for strings and numbers.
- The `if` statement and its variants are branching statements.
- The `while`, `for`, and `foreach` statements are looping statements.
- The `next` and `last` statements are means of altering the flow within a loop.
- Logical operators can add complexity to conditions and help solve other problems. They are lazy evaluators.
- Statement modifiers are a convenience for added program clarity; they can sometimes make programs easier to read.