Perl Conditional Statements

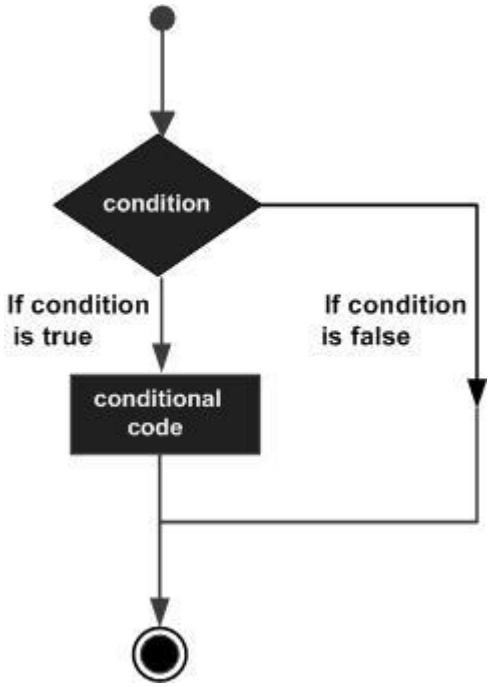
Perl programming language provides the following types of conditional statements.

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**.
3	if...elsif...else statement An **if statement** can be followed by an optional **elsif statement** and then by an optional **else statem**
4	unless statement An **unless statement** consists of a boolean expression followed by one or more statements.
5	unless...else statement An **unless statement** can be followed by an optional **else statement**.
6	unless...elsif..else statement An **unless statement** can be followed by an optional **elsif statement** and then by an optio **statement**.
7	switch statement With the latest versions of Perl, you can make use of the **switch** statement. which allows a simple comparing a variable value against various conditions.

If else statement in Perl

Perl conditional statements helps in the decision making, which require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages −



The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

In the last tutorial, we learnt about if statements in perl. In this tutorial we will learn about if-else conditional statement. This is how an if-else statement looks:

```
if(condition) {
   Statement(s);
}
else {
   Statement(s);
}
```

The statements inside “if” would execute if the condition is true, and the statements inside “else” would execute if the condition is false.

Example:

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

printf "Enter any number:";
$num = <STDIN>;
```

```
if( $num >100 ) {
   # This print statement would execute,
   # if the above condition is true
   printf "number is greater than 100\n";
}
else {
   #This print statement would execute,
   #if the given condition is false
   printf "number is less than 100\n";
}
```

Output:

```
Enter any number:120
number is greater than 100
```

if-elsif-else statement in perl

BY CHAITANYA SINGH

if-elsif-else statement is used when we need to check multiple conditions. In this statement we have only one “if” and one “else”, however we can have multiple “elsif”. This is how it looks:

```
if(condition_1) {
   #if condition_1 is true execute this
   statement(s);
}
elsif(condition_2) {
   #execute this if condition_1 is not met and
   #condition_2 is met
   statement(s);
}
elsif(condition_3) {
   #execute this if condition_1 & condition_2 are
   #not met and condition_3 is met
   statement(s);
}
.
.
.
else {
   #if none of the condition is true
   #then these statements gets executed
   statement(s);
}
```

Note: The most important point to note here is that in if-elsif-else statement as soon as the condition is met, the corresponding set of statements get

executed, rest gets ignored. If none of the condition is met then the statements inside “else” gets executed.

Example:

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

printf "Enter any integer between 1 & 99999:";
$num = <STDIN>;
if( $num <100 && $num>=1) {
   printf "Its a two digit number\n";
}
elsif( $num <1000 && $num>=100) {
   printf "Its a three digit number\n";
}
elsif( $num <10000 && $num>=1000) {
   printf "Its a four digit number\n";
}
elsif( $num <100000 && $num>=10000) {
   printf "Its a five digit number\n";
}
else {
   printf "Please enter number between 1 & 99999\n";
}
```

Output:

```
Enter any integer between 1 & 99999:8019
Its a four digit number
```

Unless statement in Perl

BY CHAITANYA SINGH

unless statement in perl behaves just opposite to the if statement in perl. It executes the set of statements inside its body when the given condition is false.

```
unless(condition) {
   statement(s);
}
```

The statements inside **unless** body would execute when the condition is false.

Example

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

printf "Enter any number:";
$num = <STDIN>;
```

```
unless($num>=100) {
   printf "num is less than 100\n";
}
```

Output:

```
Enter any number:99
num is less than 100
```

unless-else statement in Perl

BY CHAITANYA SINGH

Similar to unless statement, the **unless-else** statement in Perl behaves opposite to the if-else statement. In unless-else, the statements inside unless gets executed if the condition is false and statements inside else gets executed if the condition is true.

```
unless(condition) {
   #These statements would execute
   #if the condition is false.
   statement(s);
}
else {
   #These statements would execute
   #if the condition is true.
   statement(s);
}
```

Example

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

printf "Enter any number:";
$num = <STDIN>;
unless($num>=100) {
   #This print statement would execute,
   #if the given condition is false
   printf "num is less than 100\n";
}
else {
   #This print statement would execute,
   #if the given condition is true
   printf "number is greater than or equal to 100\n";
}
```

Output:

```
Enter any number:100
number is greater than or equal to 100
```

Unless elsif else statement in Perl

BY CHAITANYA SINGH

unless-elsif-else statement is used when we need to check multiple conditions. In this statement we have only one “unless” and one “else”, however we can have multiple “elsif”. This is how it looks:

```
unless(condition_1){
   #These statements would execute if
   #condition_1 is false
   statement(s);
}
elsif(condition_2){
   #These statements would execute if
   #condition_1 & condition_2 are true
   statement(s);
}
elsif(condition_3){
   #These statements would execute if
   #condition_1 is true
   #condition_2 is false
   #condition_3 is true
   statement(s);
}
.
.
.
else{
   #if none of the condition is met
   #then these statements gets executed
   statement(s);
}
```

Example

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

printf "Enter any number:";
$num = <STDIN>;
unless( $num == 100) {
   printf "Number is not 100\n";
}
elsif( $num==100) {
   printf "Number is 100\n";
}
else {
   printf "Entered number is not 100\n";
}
```

Output:

```
Enter any number:101
Number is not 100
```

Switch Case in Perl

BY CHAITANYA SINGH

switch case is deprecated in Perl 5. If you are wondering why it got deprecated, here is the answer:
Switch case may create syntax errors in other parts of code. On perl 5.10.x may cause syntax error if “case” is present inside heredoc. In general, use given/when instead. It were introduced in perl 5.10.0. Perl 5.10.0 was released in 2007. Source.

However three new keywords: given, when and default got introduced in Perl 5 that provides functionality similar to switch case. This is how it looks:

```
given (argument) {
  when (condition) { statement(s); }
  when (condition) { statement(s); }
  when (condition) { statement(s); }
  .
  .
  .
  default { statement(s); }
}
```

Example:

```
#!/usr/local/bin/perl
use v5.10;
no warnings 'experimental';
printf "Enter any number:";
$num = <STDIN>;

given($num){
  when ($num>10) {
    printf "number is greater than 10\n";
  }
  when ($num<10) {
    printf "number is less than 10\n";
  }
  default {
    printf "number is equal to 10\n";
  }
```

```
}
```

Output:

```
Enter any number:10
number is equal to 10
```

Perl Operators – Complete guide

BY CHAITANYA SINGH

An operator is a character that represents an action, for example + is an arithmetic operator that represents addition.
Operators in perl are categorised as following types:
1) Basic Arithmetic Operators
2) Assignment Operators
3) Auto-increment and Auto-decrement Operators
4) Logical Operators
5) Comparison operators
6) Bitwise Operators
7) Quote and Quote-like Operators

1) Basic Arithmetic Operators

Basic arithmetic operators are: +, -, *, /, %, **
+ is for addition: $x + $y

– is for subtraction: $x – $y

***** is for multiplication: $x * $y

/ is for division: $x / $y

% is for modulo: $x % $y
Note: It returns remainder, for example 10 % 5 would return 0

****** is for Exponentiation: $x ** $y
x to the power y

Example

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

```
$x = -4;
$y = 2;

$result = $x + $y;
print '+ Operator output: ' . $result . "\n";

$result = $x - $y;
print '- Operator output: ' . $result . "\n";

$result = $x * $y;
print '* Operator output: ' . $result . "\n";

$result = $x / $y;
print '/ Operator output: ' . $result . "\n";

$result = $x % $y;
print '% Operator output: ' . $result. "\n";

$result = $x ** $y;
print '** Operator output: ' . $result . "\n";
```

Output:

```
+ Operator output: -2
- Operator output: -6
* Operator output: -8
/ Operator output: -2
% Operator output: 0
** Operator output: 16
```

2) Assignment Operators

Assignments operators in perl are: =, +=, -=, *=, /=, %=, **=

$x = $y would assign value of variable y to the variable x
$x+=$y is equal to $x = $x+$y
$x-=$y is equal to $x = $x-$y
$x*=$y is equal to $x = $x*$y
$x/=$y is equal to $x = $x/$y
$x%=$y is equal to $x = $x%$y
$x**=$y is equal to $x = $x**$y

Example:

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

$x = 5;
$result = 10;

print "\$x= $x and \$result=$result\n";
$result = $x;
print '= Operator output: ' . $result . "\n";
```

```
print "\$x= $x and \$result=$result\n";
$result += $x;
print '+= Operator output: ' . $result . "\n";

print "\$x= $x and \$result=$result\n";
$result -= $x;
print '-= Operator output: ' . $result . "\n";

print "\$x= $x and \$result=$result\n";
$result *= $x;
print '*= Operator output: ' . $result . "\n";

print "\$x= $x and \$result=$result\n";
$result /= $x;
print '/= Operator output: ' . $result . "\n";

print "\$x= $x and \$result=$result\n";
$result %= $x;
print '%= Operator output: ' . $result . "\n";

#assigning different value to $result for this operator
$result =2;
print "\$x= $x and \$result=$result\n";
$result **= $x;
print '**= Operator output: ' . $result . "\n";
```

Output:

```
$x= 5 and $result=10
= Operator output: 5
$x= 5 and $result=5
+= Operator output: 10
$x= 5 and $result=10
-= Operator output: 5
$x= 5 and $result=5
*= Operator output: 25
$x= 5 and $result=25
/= Operator output: 5
$x= 5 and $result=5
%= Operator output: 0
$x= 5 and $result=2
**= Operator output: 32
```

3) Auto-increment and Auto-decrement Operators

++ and —
$x++ is equivalent to $x=$x+1;
$x– is equivalent to $x=$x-1;

Example:

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

$x =100;
$y =200;
$x++;
$y--;
print"Value of \$x++ is: $x\n";
print"Value of \$y-- is: $y\n";
```

Output:

```
Value of $x++ is: 101
Value of $y-- is: 199
```

4) Logical Operators

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.
Logical operators in perl are: &&, and, ||, or, not, !

"**&&**" and "**and**" are same
$x&&y will return true if both x and y are true else it would return false.

"**||**" and "**or**" are same.
$x||$y will return false if both x and y are false else it would return true.

"**!**" and "**not**" are same.
!$x would return the opposite of x, that means it would be true if x is false and it would return false if x is true.

Example

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

$x = true;
$y = false;

$result = ($x and $y);
print"\$x and \$y: $result\n";
$result = ($x && $y);
print"\$x && \$y: $result\n";

$result = ($x or $y);
print"\$x or \$y: $result\n";
$result = ($x || $y);
print"\$x || \$y: $result\n";

#point to note is that not operator works
#with 0 and 1 only.
$x=0;
$result = not($x);
```

```
print"not\$x: $result\n";
$result = !($x);
print"\!\$x: $result\n";
```

Output:

```
$x and $y: false
$x && $y: false
$x or $y: true
$x || $y: true
not$x: 1
!$x: 1
```

5) Comparison(Relational) operators

They includes: ==, eq, !=, ne, >, gt, <, lt, >=, ge, <=, le
== and eq returns true if both the left side and right side are equal
!= and ne returns true if left side is not equal to the right side of operator.
> and **gt** returns true if left side is greater than right.
< and **lt** returns true if left side is less than right side.
>= and **ge** returns true if left side is greater than or equal to right side.
<= and **le** returns true if left side is less than or equal to right side.

Example

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

$x = 3;
$y = 6;

if( $x == $y ){
  print "\$x and \$y are equal\n";
}else{
  print "\$x and \$y are not equal\n";
}

if( $x != $y ){
  print "\$x and \$y are not equal\n";
}else{
  print "\$x and \$y are equal\n";
}

if( $x > $y ){
  print "\$x is greater than \$y\n";
}else{
  print "\$x is not greater than \$y\n";
}

if( $x >= $y ){
  print "\$x is greater than or equal to \$y\n";
}else{
  print "\$x is less than \$y\n";
}
```

```
if( $x < $y ){
  print "\$x is less than \$y\n";
}else{
  print "\$x is not less than \$y\n";
}

if( $x <= $y){
  print "\$x is less than or equal to \$y\n";
}else{
  print "\$x is greater than \$y\n";
}
```

Output:

```
$x and $y are not equal
$x and $y are not equal
$x is not greater than $y
$x is less than $y
$x is less than $y
$x is less than or equal to $y
```

6) Bitwise Operators

There are six bitwise Operators: &, |, ^, ~, <<, >>
$x = 11; #00001011
$y = 22; #00010110

Bitwise operator performs bit by bit processing.
$x & $y compares corresponding bits of x and y and generates 1 if both bits are equal, else it returns 0.
In our case it would return: 2 which is 00000010 because in the binary form of x and y only second last bits are matching.

$x | $y compares corresponding bits of x and y and generates 1 if either bit is 1, else it returns 0.
In our case it would return 31 which is 00011111

$x ^ $y compares corresponding bits of x and y and generates 1 if they are not equal, else it returns 0.
In our example it would return 29 which is equivalent to 00011101

~$x is a complement operator that just changes the bit from 0 to 1 and 1 to 0
In our example it would return -12 which is signed 8 bit equivalent to 11110100

<< is left shift operator that moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0.
In out case output is 44 which is equivalent to 00101100

Note: In the example below we are providing 2 at the right side of this shift operator that is the reason bits are moving two places to the left side. We can change this number and bits would be moved by the number of bits specified on the right side of the operator. Same applies to the right side operator.

>> is right shift operator that moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0.
in our case output is 2 which is equivalent to 00000010

Example:

```
#!/usr/local/bin/perl
use integer;

$x = 11; #00001011
$y = 22; #00010110

$result = $x & $y;
print "\$x & \$y: $result\n";

$result = $x | $y;
print "\$x | \$y: $result\n";

$result = $x ^ $y;
print "\$x ^ \$y: $result\n";

$result = ~$x;
print "~\$x = $result\n";

$result = $x << 2;
print "\$x << 2 = $result\n";

$result = $x >> 2;
print "\$x >> 2 = $result\n";
```

Output:

```
$x & $y: 2
$x | $y: 31
$x ^ $y: 29
~$x = -12
$x << 2 = 44
$x >> 2 = 2
```

7) Quote and Quote-like Operators

There are several quote like operators in perl like q{ }, qq{ }, qw{ }, m{ }, qr{ }, s{ }{ }, tr{ }{ }, y{ }{ }.
However only two of them are frequently used: q{ } is for single quote and qq{ } is for double quote.

Example

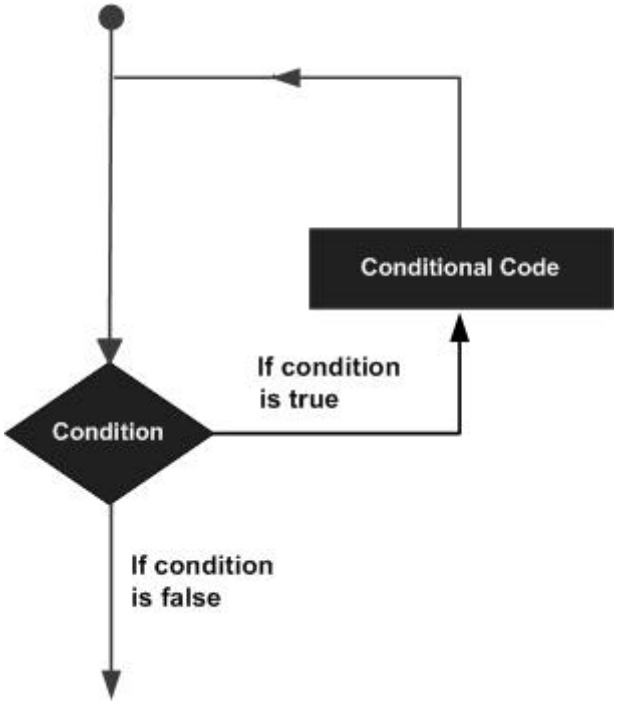
q{Welcome to beginnersbook} would return ‘Welcome to beginnersbook’
qq{Welcome to beginnersbook} would return “Welcome to beginnersbook”

Perl - Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



Perl programming language provides the following types of loop to handle the looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	until loop Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.
3	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	foreach loop The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.
5	do...while loop Like a while statement, except that it tests the condition at the end of the loop body
6	nested loops You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change the execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Perl supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description
1	next statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
2	last statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
3	continue statement A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.
4	redo statement The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed.
5	goto statement Perl supports a goto command with three forms: goto label, goto expr, and goto &name.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty.

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

for( ; ; ) {
   printf "This loop will run forever.\n";
}
```

You can terminate the above infinite loop by pressing the Ctrl + C keys.

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but as a programmer more commonly use the for (;;) construct to signify an infinite loop.

Perl while Loop

A **while** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

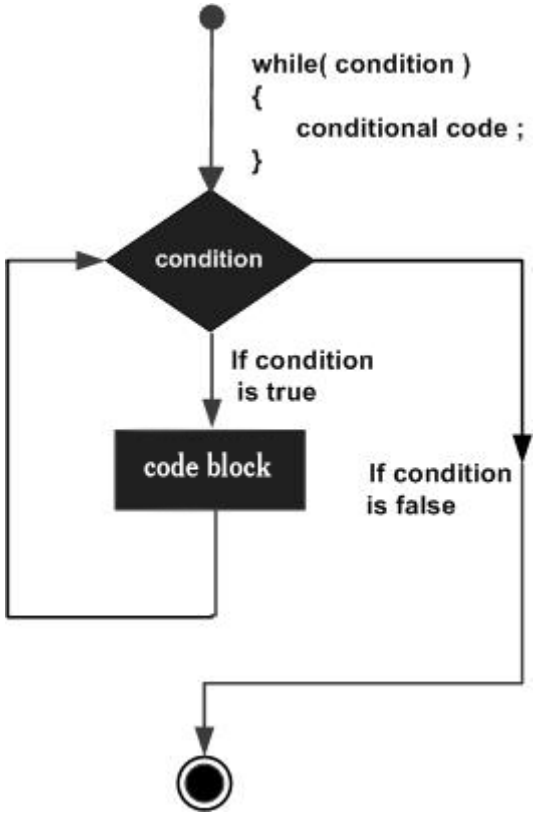
The syntax of a **while** loop in Perl programming language is −

```
while(condition) {
   statement(s);
}
```

Here **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram



Here the key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

Live Demo

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

$a = 10;

# while loop execution
while( $a < 20 ) {
   printf "Value of a: $a\n";
   $a = $a + 1;
}
```

Here we are using the comparison operator < to compare value of variable $a against 20. So while value of $a is less than 20, **while** loop continues executing a block of code next to it and as soon as the value of $a becomes equal to 20, it comes out. When executed, above code produces the following result −

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
Value of a: 19
```

Perl until Loop

An **until** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is false.

Syntax

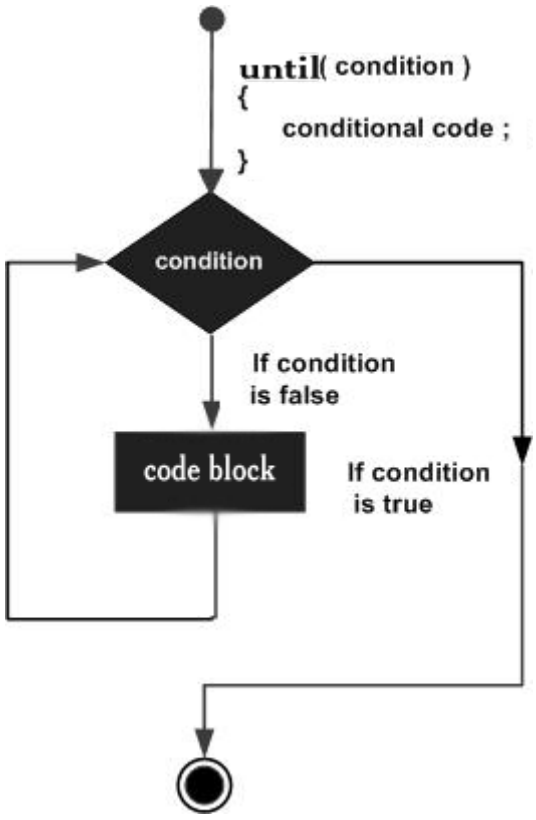
The syntax of an **until** loop in Perl programming language is −

```
until(condition) {
   statement(s);
}
```

Here **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression. The loop iterates until the condition becomes true. When the condition becomes true, the program control passes to the line immediately following the loop.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram



Here key point of the *until* loop is that the loop might not ever run. When the condition is tested and the result is true, the loop body will be skipped and the first statement after the until loop will be executed.

Example

Live Demo

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

$a = 5;

# until loop execution
until( $a > 10 ) {
   printf "Value of a: $a\n";
   $a = $a + 1;
}
```

Here we are using the comparison operator > to compare value of variable $a against 10. So until the value of $a is less than 10, **until** loop continues executing a block of code next to it and as soon as the value of $a becomes greater than 10, it comes out. When executed, above code produces the following result −

```
Value of a: 5
Value of a: 6
Value of a: 7
Value of a: 8
Value of a: 9
Value of a: 10
```

Perl do...while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

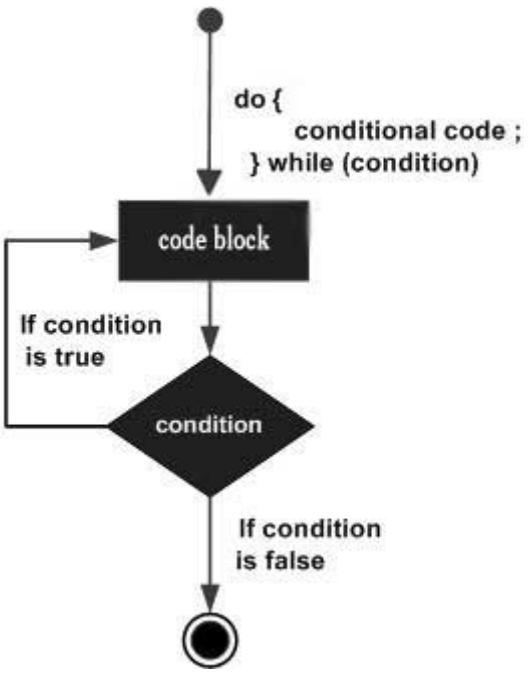
The syntax of a **do...while** loop in Perl is −

```
do {
   statement(s);
}while( condition );
```

It should be noted that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram



Example

Live Demo

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

$a = 10;

# do...while loop execution
do{
   printf "Value of a: $a\n";
   $a = $a + 1;
}while( $a < 20 );
```

When the above code is executed, it produces the following result −

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
Value of a: 19
```

Perl for Loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

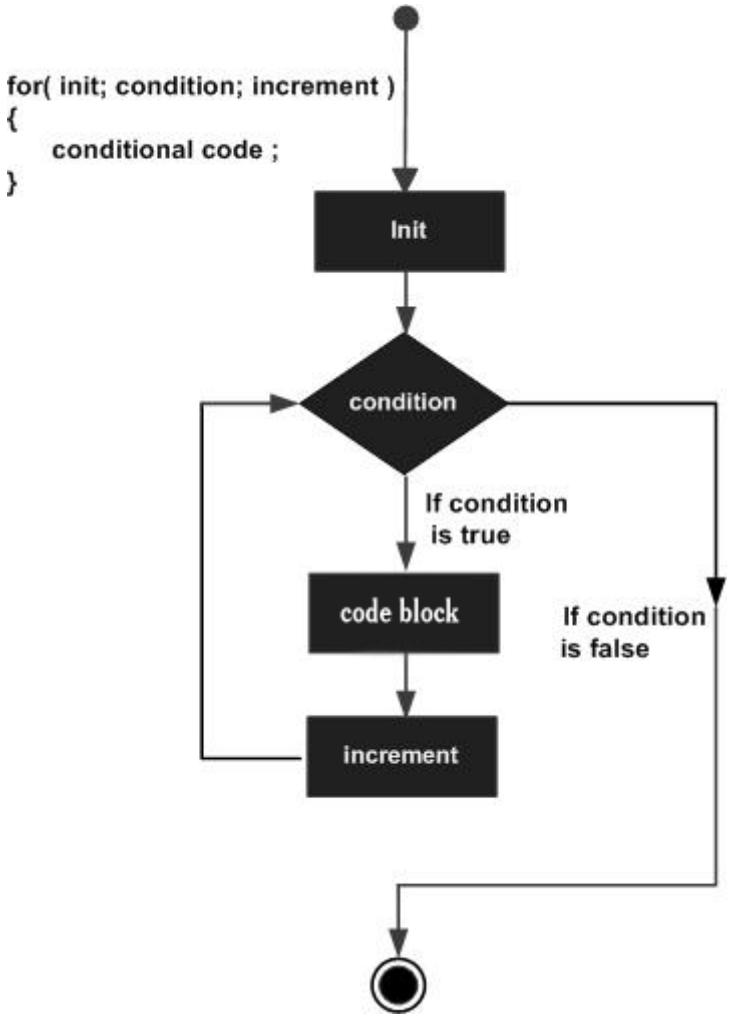
The syntax of a **for** loop in Perl programming language is −

```
for ( init; condition; increment ) {
   statement(s);
}
```

Here is the flow of control in a **for** loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Example

Live Demo

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

# for loop execution
for( $a = 10; $a < 20; $a = $a + 1 ) {
   print "value of a: $a\n";
}
```

When the above code is executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Perl foreach Loop

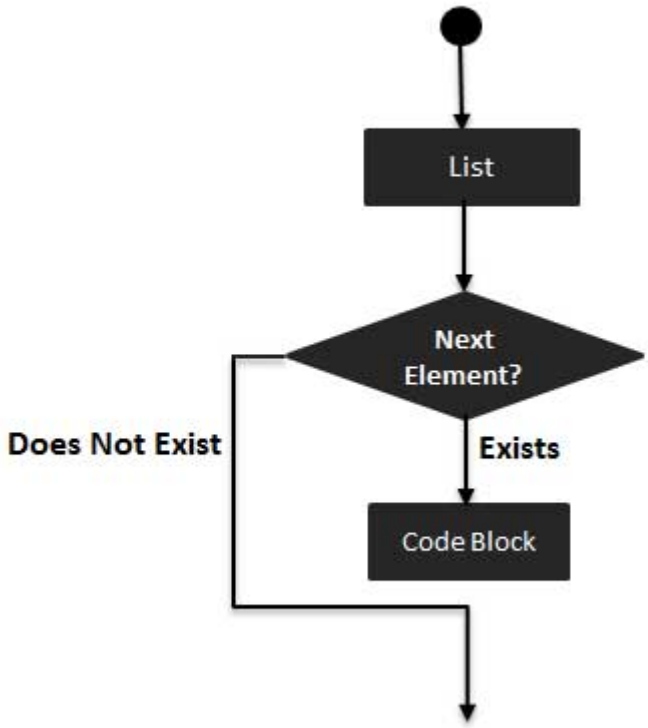
The **foreach** loop iterates over a list value and sets the control variable (var) to be each element of the list in turn −

Syntax

The syntax of a **foreach** loop in Perl programming language is −

```
foreach var (list) {
...
}
```

Flow Diagram



Example

Live Demo

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

@list = (2, 20, 30, 40, 50);

# foreach loop execution
foreach $a (@list) {
```

```
   print "value of a: $a\n";
}
```

When the above code is executed, it produces the following result −

```
value of a: 2
value of a: 20
value of a: 30
value of a: 40
value of a: 50
```

Perl next Statement

The Perl **next** statement starts the next iteration of the loop. You can provide a LABEL with **next** statement where LABEL is the label for a loop. A **next** statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

If there is a **continue** block on the loop, it is always executed just before the condition is about to be evaluated. You will see the continue statement in separate chapter.

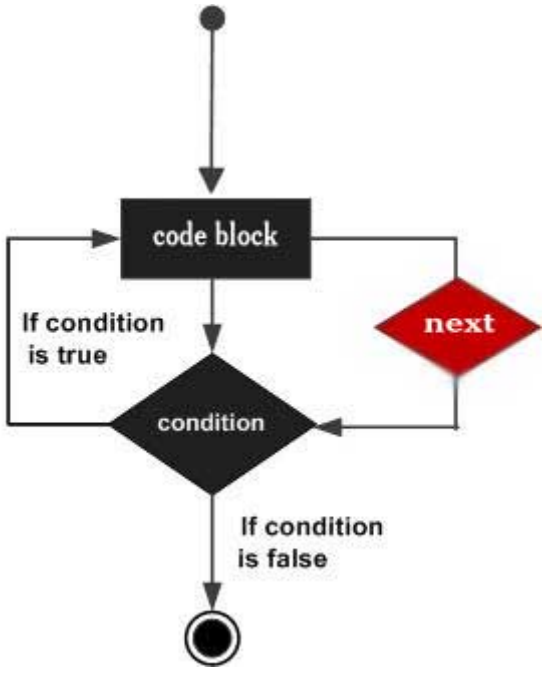
Syntax

The syntax of a **next** statement in Perl is −

```
next [ LABEL ];
```

A LABEL inside the square braces indicates that LABEL is optional and if a LABEL is not specified, then next statement will jump the control to the next iteration of the nearest loop.

Flow Diagram



Example

Live Demo

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

$a = 10;
while( $a < 20 ) {
   if( $a == 15) {
      # skip the iteration.
```

```
      $a = $a + 1;
      next;
   }
   print "value of a: $a\n";
   $a = $a + 1;
}
```

When the above code is executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Let's take one example where we are going to use a LABEL along with next statement −

Live Demo

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

$a = 0;
OUTER: while( $a < 4 ) {
   $b = 0;
   print "value of a: $a\n";
   INNER:while ( $b < 4) {
      if( $a == 2) {
         $a = $a + 1;
         # jump to outer loop
         next OUTER;
      }
      $b = $b + 1;
      print "Value of b : $b\n";
   }
   print "\n";
   $a = $a + 1;
}
```

When the above code is executed, it produces the following result −

```
value of a : 0
Value of b : 1
Value of b : 2
Value of b : 3
Value of b : 4

value of a : 1
Value of b : 1
Value of b : 2
Value of b : 3
```

```
Value of b : 4

value of a : 2
value of a : 3
Value of b : 1
Value of b : 2
Value of b : 3
Value of b : 4
```

Perl last Statement

When a **last** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. You can provide a LABEL with last statement where LABEL is the label for a loop. A **last** statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

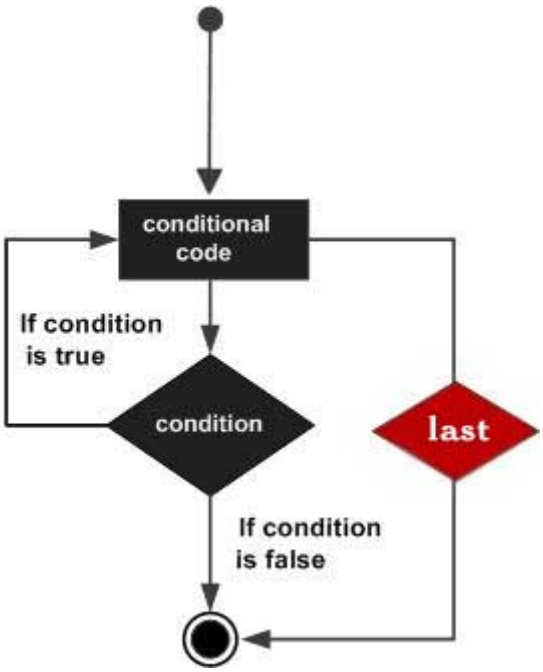
If there is any **continue** block on the loop, then it is not executed. You will see the continue statement in a separate chapter.

Syntax

The syntax of a **last** statement in Perl is −

```
last [LABEL];
```

Flow Diagram



Example 1

Live Demo

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

$a = 10;
while( $a < 20 ) {
   if( $a == 15) {
      # terminate the loop.
      $a = $a + 1;
```

```
      last;
   }
   print "value of a: $a\n";
   $a = $a + 1;
}
```

When the above code is executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

Example 2

Let's take one example where we are going to use a LABEL along with next statement −

Live Demo

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

$a = 0;
OUTER: while( $a < 4 ) {
   $b = 0;
   print "value of a: $a\n";
   INNER:while ( $b < 4) {
      if( $a == 2) {
         # terminate outer loop
         last OUTER;
      }
      $b = $b + 1;
      print "Value of b : $b\n";
   }
   print "\n";
   $a = $a + 1;
}
```

When the above code is executed, it produces the following result −

```
value of a : 0
Value of b : 1
Value of b : 2
Value of b : 3
Value of b : 4

value of a : 1
Value of b : 1
Value of b : 2
Value of b : 3
Value of b : 4

value of a : 2
```