Repetition structures

- Repetition structures allow us to repeat some action or set of actions.
- This is useful if we would like to execute an action more than once, until a condition becomes true or while a condition is true.
- The number of times a repetition structure, or loop, executes can be controlled by the program/programmer or by the user.
- One time through a loop is called an iteration.

Types of Repetition

- There are two main repetition control structures in Perl:
  - `while`
  - `for`
- There are two types of repetition:
  - counter-controlled
  - sentinel-controlled
Counter-Controlled Repetition

- Counter-controlled repetition is used when you know ahead of time, or the program can calculate while it is running, the exact number of times an action will need to be repeated.
- Both types of looping structures except special versions of the `for` can implement counter-controlled repetition, though typically you will opt for a `for` loop.
- Counter-controlled repetition is also known as definite or finite looping.

Sentinel-Controlled Repetition

- Sentinel-controlled repetition is used when you do not know ahead of time, or the program cannot calculate while it is running, the exact number of times an action will need to be repeated.
- Sentinel-controlled looping can be implemented using both of the looping structures except special versions of the `for`, though you will typically use a `while`.
- Sentinel-controlled repetition is also known as indefinite looping.

Looping Requirements

- Both counter-controlled and sentinel-controlled repetition have the following requirements to create good, working loops:
  1. The name of a loop control variable (LCV) to store either the counter or sentinel value
  2. The initial value of the LCV, either set explicitly or read from the user
  3. The modification of the LCV each time through the loop; in a counter-controlled loop, it's typically an increment or a decrement while in a sentinel-controlled loop it's typically a new value input by the user.
  4. The condition that tests the LCV for the final value (i.e., whether looping should continue)
The `while` loop

```
while (CONDITION) {BODY}
```

- Requires curly braces `{}` for the body of the loop.
- Pre-test loop, meaning that the loop will test the condition first before possibly executing the action.
- Pre-test loops may never execute if `CONDITION` fails the first time it is tested.

### while, counter-controlled

1. `#!/usr/bin/perl`
2. `use Modern::Perl;`
3. `my $start = 1;`
4. `while ( $start <= 15 ) {`
5. `say "Counting: $start";`
6. `$start++;
7. }`
8. `# print the last value of the $start variable`
9. `say "The last value is : $start";`

- This program simply counts from 1 through 15 and outputs the numbers along the way.
- This is a counter-controlled loop, executing an exact number of times.

### while, counter-controlled

1. `#!/usr/bin/perl`
2. `use Modern::Perl;`
3. `my $start = 15;`
4. `while ( $start >= 1 ) {`
5. `say "Counting: $start";`
6. `$start--;
7. }`
8. `# print the last value of the $start variable`
9. `say "The last value is : $start";`

- This program simply counts from 15 down through 1 and outputs the numbers along the way.
- This is a counter-controlled loop, executing an exact number of times.
### While, counter-controlled

```perl
#!/usr/bin/perl
use Modern::Perl;
my $start = 15;
while ($start >= 1) {
  say "Counting: $start";
  $start -= 2;
}
# print the last value of the $start variable
say "The last value is: $start";
```

- This program simply counts from 15 down through 1 and outputs the numbers along the way.
- This is a counter-controlled loop, executing an exact number of times.

### While, counter-controlled

```perl
#!/usr/bin/perl
use Modern::Perl;
print 'How many times should we loop? ';
chomp (my $num = <>);
my $start = 1;
while ($start <= $num) {
  say "Counting: $start";
  $start ++;
}
# print the last value of the $start variable
say "The last value is: $start";
```

- This program asks the user to enter a number and then, starting at 1, loops until the user-entered number is reached.
- This is a counter-controlled loop, executing an exact number of times.

### While, counter-controlled

```perl
#!/usr/bin/perl
use Modern::Perl;
my ( $total, $counter, $grade, $average ) = ( 0, 1, 0, 0 );
while ( $counter <= 5 ) {
  print "Please enter grade $counter: ";
  chomp ( $grade = <> );
  $total += $grade;
  $counter ++;
}
$average = $total / 5;
say "The average of the grades was $average";
```

- This is a counter-controlled loop that reads five grades from the user and then prints their average at the end.
Sentinel-controlled looping information, part 1

- Sentinel-controlled loops are obviously controlled by something called a sentinel instead of a counter.
- A sentinel value is a nonsensical value that does not occur in the normal range of input values and "flags" the end of input.
- Consequently, a sentinel value is also sometimes called a flag value.
- For a range of grades from 0 to 100, -1 is a good choice although -1234 or 1234787 would work, also.
- When inputting letter grades ('A', 'B', 'C', 'D' or 'F'), good choices might be 'Z' or '!'.

Sentinel-controlled looping information, part 2

The basic format for constructing a valid sentinel-controlled loop looks like this:

1. "prime" the loop by prompting for and reading a value from the user to process
2. set up a looping structure, testing whether or not the sentinel value has been entered
   a) process the value that was entered by the user
   b) 'prime' the loop again, setting up the next test of the looping structure.

while, sentinel-controlled

```
#!/usr/bin/perl
use Modern::Perl;
my $input;

print "Please enter the word 'no': ";
chomp ( $input = <> ); # "prime" the loop

while ( $input ne "no" ) {
    say "Hi there. Try entering the word 'no'.";
    print "Please enter the word 'no': ";
    chomp ( $input = <> ); # "prime" the loop again
}

say "Thanks for finally typing 'no'.";
```

- This program asks the user to simply type in the word no and keeps prompting them until they do.
- This is a sentinel-controlled loop, based on the value of $input.
while, sentinel-controlled

1. $/usr/bin/perl
2. use Modern::Perl;
3. my $input;
4. print "Please enter a number between 50 and 150: ";
5. chomp ( $input = <> );
6. while ( $input < 50 || $input > 150 ) {
7. say "Hi there. Try entering a number between 50 and 150: ";
8. chomp ( $input = <> );
9. }
10. say "Thanks for finally entering $input.";

This program asks the user to simply type the word no and keeps prompting them until they do.

This is a sentinel-controlled loop, based on the value of $input.

while, sentinel-controlled

1. $/usr/bin/perl
2. use Modern::Perl;
3. my ( $total, $howmany, $grade, $average ) = ( 0, 0, 0, 0 );
4. print "Please enter a grade or -1 to end: ";
5. chomp ( $grade = <> );
6. while ( $grade != -1 ) {
7. $total += $grade;
8. $howmany ++;
9. print "Please enter a grade or -1 to end: ";
10. chomp ( $grade = <> );
11. }
12. if ( $howmany == 0 ) {
13. say "There were no grades to average."
14. }
15. else {
16. say "The average of the grades was ", $total / $howmany;
17. }

This is a sentinel-controlled loop that reads an unknown number of grades from the user, sums them all up and then prints the average, if any.

The for loop

for ( START; STOP; MODIFICATION ) { BODY }

initially executes the START statement and then repeatedly executes BODY, as long as the STOP condition remains TRUE. The MODIFICATION statement is executed after each iteration of the BODY.
A for loop example

1. #!/usr/bin/perl
2. use Modern::Perl;
3. for ( my $x = 1; $x <= 15; $x++ ) {
4.   say $x;
5. }
6. say 'We cannot access $x outside the loop!';

• Initializes $x to 1, and then checks to see if $x is less than or equal to 15.
• If it is, it does the BODY of the loop, which is to print the scalar $x using say $x;
• Then, the MODIFICATION takes place, which is to increment the scalar by one, which is the $x++ code.
• The STOP condition ($x <= 15) is checked again. As long as it's true, you keep going. When the condition becomes false, the loop stops.

A for loop example

1. #!/usr/bin/perl
2. use Modern::Perl;
3. for ( my $x = 15; $x >= 1 ; $x-- ) {
4.   say $x;
5. }
6. say 'We cannot access $x outside the loop!';

• Initializes $x to 15, and then checks to see if $x is greater than or equal to 1.
• If it is, it does the BODY of the loop, which is to print the scalar $x using say $x;
• Then, the MODIFICATION takes place, which is to decrement the scalar by one, which is the $x-- code.
• The STOP condition ($x >= 1) is checked again. As long as it's true, you keep going. When the condition becomes false, the loop stops.

Printing even numbers between two user-chosen numbers

1. #!/usr/bin/perl
2. use Modern::Perl;
3. my ( $start, $end );
4. print "Enter starting number: ";
5. chomp ( $start = <> ); # get starting number
6. print "Enter ending number more than $start: ";
7. chomp ( $end = <> ); # prime loop
8. while ( $end <= $start ) {
9.   print "Enter ending number more than $start: ";
10.  chomp ( $end = <> ); # get bigger num for end
11. }
12. say "Here are the even numbers from $start to $end: ",
13. for ( my $x = $start; $x <= $end; $x++ ) {
14.   if ( $x % 2 == 0 ) {
15.     print "$x: ", # test for even numbers
16.   }
17. }
Process grades with data validation for grades

```perl
# File: checkgrades.pl
use Modern::Perl;
my ( $total, $counter, $grade, $average, $howmany ) = ( 0, 1, 0, 0 );
print "How many grades should we process? 

chomp ( $howmany = <> );
while ( $howmany <= 0 ) {
    print "How many grades should we process? 
    chomp ( $howmany = <> );
}
for ( my $x = 1; $x <= $howmany; $x++ ) {
    print "Please enter grade $x:
    chomp ( $grade = <> );
    while ( $grade < 0 || $grade > 100 ) { # data validation
        print "Please enter grade $x again: 
        chomp ( $grade = <> );
    }$total += $grade;
$average = $total / $howmany;
print "The average of the grades was $average
```