Introduction

In this video, we are going to take a look at the different types of **loops** that can be used in Ruby.

Loops will repeat a piece of code over and over again according to certain conditions.

Wire Loop.

The first type of loop we're going to look at is called a **while loop**. This will repeatedly run a block of code while a certain condition is true.

```
number = 0
attempts = 0

while number < 6
  number = rand(1..6)
  puts "You rolled a #{number}"
  attempts += 1
end

puts "It took you #{attempts} attempts to roll a 6"
```

`number` and `attempts` variables are initially assigned the value of `0`. These variables are going to be used in the loop and so they need to be initialized at the start of the program before the loop is run. Otherwise, there will be an error if they are mentioned before they have a value.

The loop starts with the `while` keyword and finishes with the `end`. The condition is `number < 6`. So in English, this is basically saying that we need to keep repeating this block of code while the `number` is less than 6.

Running Code

Open up a terminal and type in

```
ruby while.rb
```

Until Loops

Next we're going to look at **until loops**. These are very similar to while loops, except they will keep running until a condition is met.

```
puts "Let's play a game! Type something"
input = ""

until input == "goodbye"
  puts input.upcase
  input = gets.chomp
end

puts "Bye! Let's play again Soon"
```

We start the loop using the keyword `until`, and again it finishes with the keyword `end`. This time, the condition is `input == "goodbye"`, so the block of code will keep on running until the value of the `input` variable is equal to "goodbye".

Remember that `input` was the empty string, so this loop is going to be run at least once.

Iterators

Now we're going to look at **iterators**. These are methods of arrays, hashes, and ranges, and they act in similar ways to loops.

The first one we're going to look at is the `each` method:

```
(1..10).each do |number|
  puts "#{number} squared is #{number**2}"
end
```

`each` will run the block of code here for each of the integers in the range. The `number` variable inside the pipe symbols (`|`), represents a sort of temporary variable that contains the number in the range during each iteration. So, it will start at 1 and then it'll enter the value of 1 into this code, and run it. Once this code has been run, it will then increment the value of number to 2, and run the code again and so on. It's like a placeholder for each number in each iteration. Note that inside the block of code we use `**` operator - it represents a power.

Map Method

The `map` method is a similar iterator method to `each`, but instead of just running the block of code, it actually replaces the originalvalue inside the array, or hash, with the results of the block of code.

```
list = [ "apples" , "bananananas" ]
list.map! { |fruit| "yummy " + fruit }
p list
```

The `map` method will iterate over the array and run the code for each item inside the array. This time we place the the block of code inside curly braces. This means we don't need to use the `end` keyword to finish off the block of code. This is very common if the code is only one line, which it is in this case.

Notice again we've got this temporary variable called `fruit` inside the pipe symbols, that will refer to each item in the array that is being passed into this block of code.

For each item in the array, we're going to be adding the string "yummy". What is going to happen is each item in the array will be replaced with a new one, so for example "apples" will become "yummy apples".

Notice I'm using the bang version of `map` - this means the array will be fully replaced. If we didn't use that, it would run the code and create a temporary array but wouldn't actually replace the array held in the `list`.

We also use `p` command to display what the array looks like. The `p` command is often used when debugging code, as it will show an object, rather than converting it into a string. It's very similar to the `puts` command, but shows us the exact object.