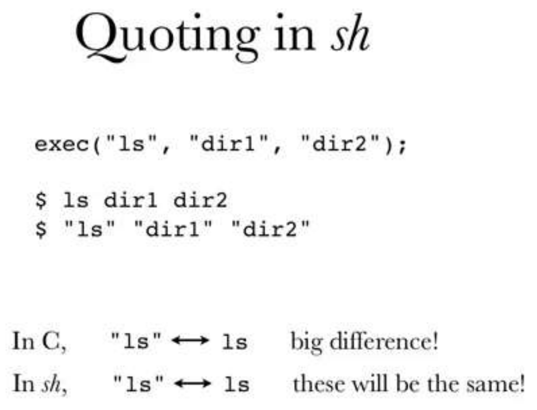
Shell programming 3 of 6

Hover over the image to see the time range in the original video. Click to play that excerpt.

This is the third video in the series about shell programming. Much of this video is about some other control constructs in *sh* which don't involve process exit status. This video is not meant to be understandable until you've already understood the previous two videos.



Quoting in *sh*

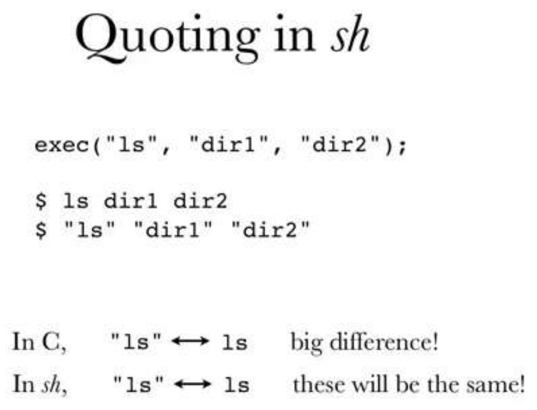
```
exec("ls", "dir1", "dir2");

$ ls dir1 dir2
$ "ls" "dir1" "dir2"
```

In C, `"ls"` ⟷ `ls` big difference!

In *sh*, `"ls"` ⟷ `ls` these will be the same!

But first I want to talk about quoting in *sh*.
In normal programming languages, a string needs quotes, most commonly double-quotes.
If we want to run the "ls" command on two directories from our C program, we're going to write something like this: `exec("ls", "dir1", "dir2")`
That's pretty cumbersome, especially as compared to the command in sh: Where the dollar sign is the shell prompt, we just write "`ls dir1 dir2`".
We really wouldn't want to have to write this at the shell prompt [`"ls" "dir1" "dir2"`], although you could argue that we should, because all three of these are strings. But we really don't want sh to work that way.
In C or Java or Python, there's a bit difference between ls with the quotes or without the quotes. With the quotes it's a string; without the quotes it's a reference to some existing variable or function or something named "ls". However, in *sh*, these will be the same. They're both strings. Everything's a string.



Quoting in *sh*

```
exec("ls", "dir1", "dir2");

$ ls dir1 dir2
$ "ls" "dir1" "dir2"
```

In C, `"ls"` ⟷ `ls` big difference!

In *sh*, `"ls"` ⟷ `ls` these will be the same!

So how do we make this work? We use a dramatically different tactic than in a normal programming language:
In a normal programming language, when you write something like "ls" the programming language processor *interprets* it. It figures out: is this a keyword, variable, function name, whatever. But in sh, the shell will interpret only some things. If we write "ls", that's a string, even without quotes. If we wanted it to be interpreted as a variable name, then we would preface it with a dollar sign.
This is a good compromise for something which is both a programming language and the way we type commands daily, but it has some odd consequences. It means we need to have a way to *suppress* the special interpretation of interesting characters.

How do we use the echo command to output an actual '>'? Suppose we want to do something like this python program.

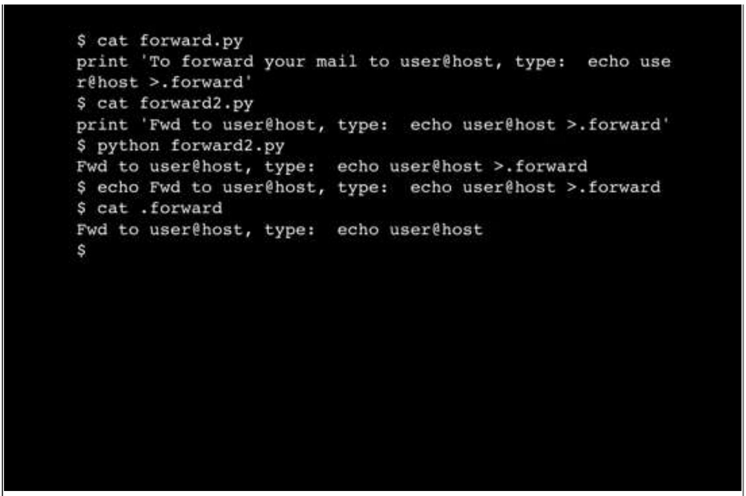
```
print 'To forward your mail to user@host, type:  echo user@host >.forward'
```

Ok, that wraps around on this unusually narrow display in this video. So I'm going to truncate it for this video:

```
print 'Fwd to user@host, type:  echo user@host >.forward'
```

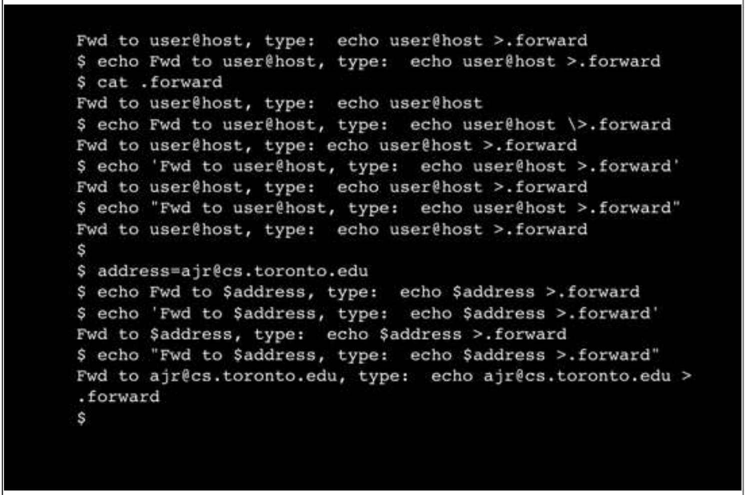
In sh, we can't just type this:

```
echo Fwd to user@host, type:  echo user@host >.forward
```



```
$ cat forward.py
print 'To forward your mail to user@host, type:  echo use
r@host >.forward'
$ cat forward2.py
print 'Fwd to user@host, type:  echo user@host >.forward'
$ python forward2.py
Fwd to user@host, type:  echo user@host >.forward
$ echo Fwd to user@host, type:  echo user@host >.forward
$ cat .forward
Fwd to user@host, type:  echo user@host
$
```

What happened to the output? It went into the file .forward!
So we need to *quote* this, to suppress the special meaning of the '>' symbol.



```
Fwd to user@host, type:  echo user@host >.forward
$ echo Fwd to user@host, type:  echo user@host >.forward
$ cat .forward
Fwd to user@host, type:  echo user@host
$ echo Fwd to user@host, type:  echo user@host \>.forward
Fwd to user@host, type: echo user@host >.forward
$ echo 'Fwd to user@host, type:  echo user@host >.forward'
Fwd to user@host, type:  echo user@host >.forward
$ echo "Fwd to user@host, type:  echo user@host >.forward"
Fwd to user@host, type:  echo user@host >.forward
$
$ address=ajr@cs.toronto.edu
$ echo Fwd to $address, type:  echo $address >.forward
$ echo 'Fwd to $address, type:  echo $address >.forward'
Fwd to $address, type:  echo $address >.forward
$ echo "Fwd to $address, type:  echo $address >.forward"
Fwd to ajr@cs.toronto.edu, type:  echo ajr@cs.toronto.edu >
.forward
$
```

As a matter of fact, there are *three* possible ways to quote this. First of all, we can use a backslash to suppress the special meaning of a single following character:

```
echo To forward your mail to user@host, type:  echo user@host >.forward
```

Or we can use single quotes:

```
echo 'To forward your mail to user@host, type:  echo user@host >.forward'
```

Or we can use double quotes.

```
echo "To forward your mail to user@host, type:  echo user@host >.forward"
```

Single and double quotes have different semantics, but we need a more-involved example to illustrate.
Let's customize the example address for the user. Suppose we have it in a variable.

```
echo To forward your mail to $address, type:  echo $address >.forward
```

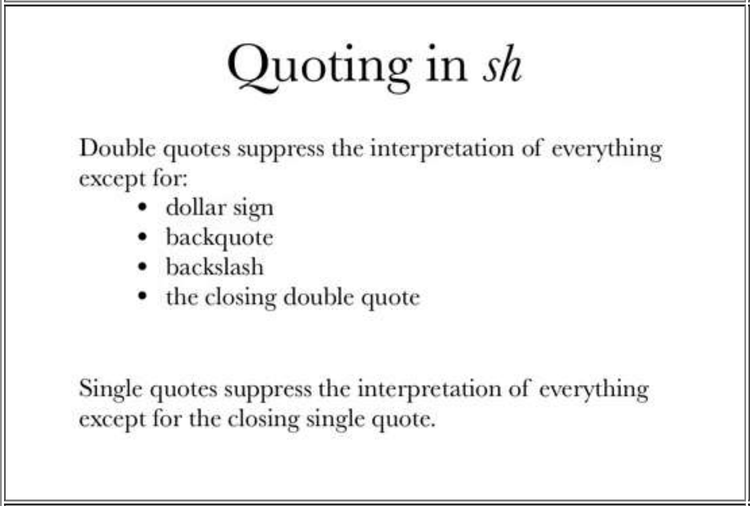
Ok, it went into the .forward file; we need some quoting to make this work.

```
echo 'To forward your mail to $address, type:  echo $address >.forward'
```

That's too *much* quoting! We want the special meaning of dollar sign to be honoured, but the special meaning of greater-than to be suppressed.

```
echo "To forward your mail to $address, type:  echo $address >.forward"
```

And double quotes do exactly that! (With some wraparound.)



Quoting in *sh*

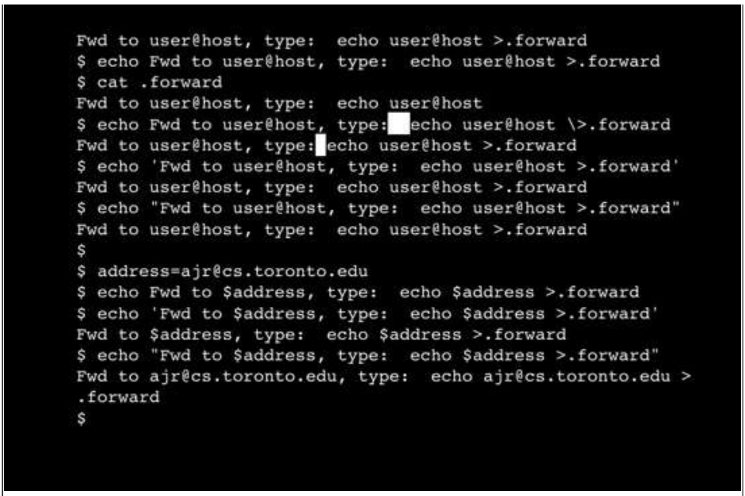
Double quotes suppress the interpretation of everything except for:

- dollar sign
- backquote
- backslash
- the closing double quote

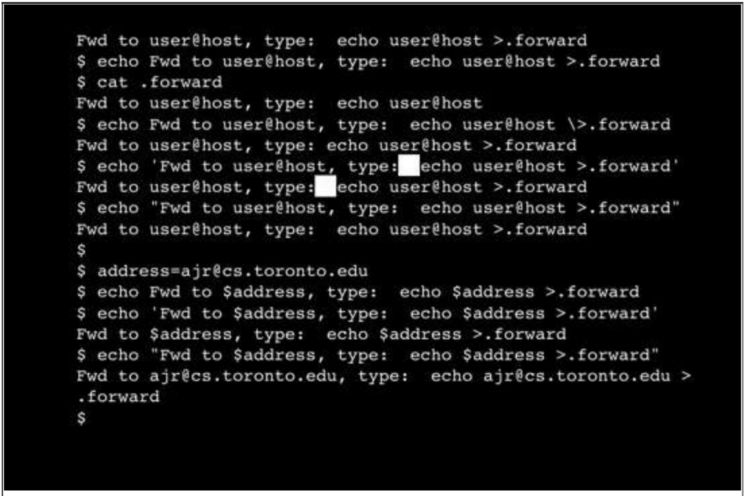
Single quotes suppress the interpretation of everything except for the closing single quote.

Double quotes suppress everything *except* for dollar sign, backquote, backslash, and the closing double quote. Whereas single quotes suppress more: everything except for the closing single quote.

So, both single and double quotes suppress the special meaning of a space! This is a bit subtle, but if we go back to the terminal window we see that this double space was collapsed to a single space in the output here,

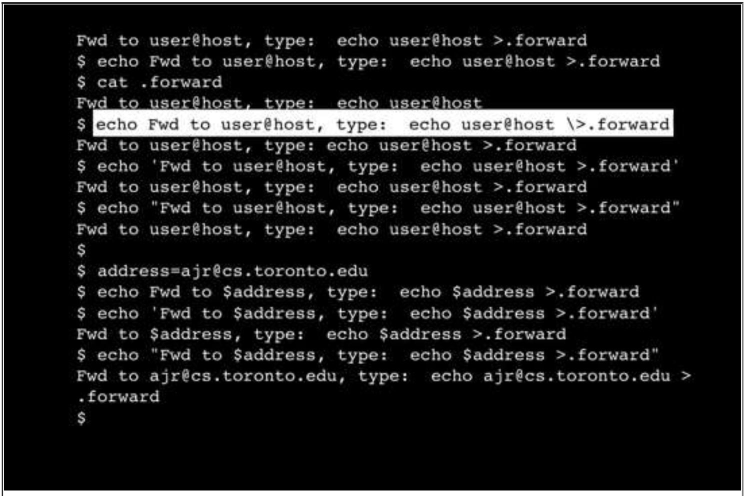


```
Fwd to user@host, type:  echo user@host >.forward
$ echo Fwd to user@host, type:  echo user@host >.forward
$ cat .forward
Fwd to user@host, type:  echo user@host
$ echo Fwd to user@host, type:  echo user@host \>.forward
Fwd to user@host, type: echo user@host >.forward
$ echo 'Fwd to user@host, type:  echo user@host >.forward'
Fwd to user@host, type:  echo user@host >.forward
$ echo "Fwd to user@host, type:  echo user@host >.forward"
Fwd to user@host, type:  echo user@host >.forward
$
$ address=ajr@cs.toronto.edu
$ echo Fwd to $address, type:  echo $address >.forward
$ echo 'Fwd to $address, type:  echo $address >.forward'
Fwd to $address, type:  echo $address >.forward
$ echo "Fwd to $address, type:  echo $address >.forward"
Fwd to ajr@cs.toronto.edu, type:  echo ajr@cs.toronto.edu >
.forward
$
```



```
Fwd to user@host, type:  echo user@host >.forward
$ echo Fwd to user@host, type:  echo user@host >.forward
$ cat .forward
Fwd to user@host, type:  echo user@host
$ echo Fwd to user@host, type:  echo user@host \>.forward
Fwd to user@host, type: echo user@host >.forward
$ echo 'Fwd to user@host, type:  echo user@host >.forward'
Fwd to user@host, type:  echo user@host >.forward
$ echo "Fwd to user@host, type:  echo user@host >.forward"
Fwd to user@host, type:  echo user@host >.forward
$
$ address=ajr@cs.toronto.edu
$ echo Fwd to $address, type:  echo $address >.forward
$ echo 'Fwd to $address, type:  echo $address >.forward'
Fwd to $address, type:  echo $address >.forward
$ echo "Fwd to $address, type:  echo $address >.forward"
Fwd to ajr@cs.toronto.edu, type:  echo ajr@cs.toronto.edu >
.forward
$
```

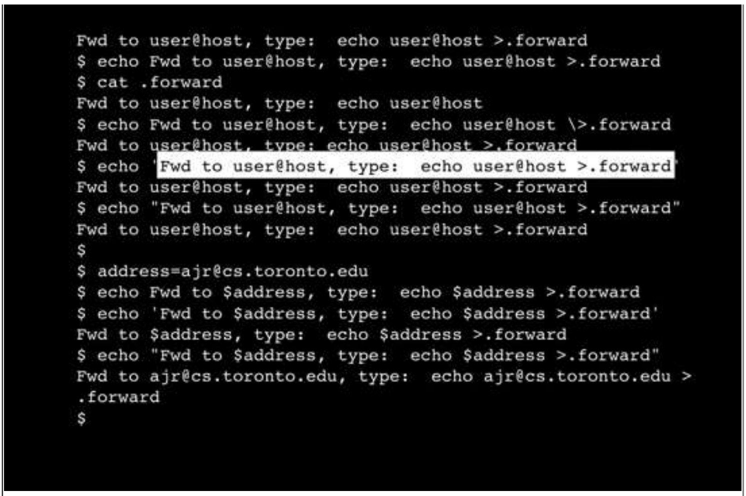
... but was preserved when quoted.



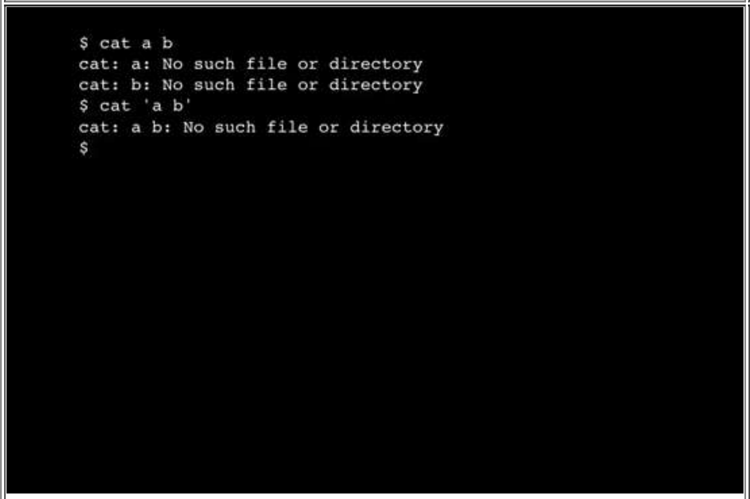
```
Fwd to user@host, type:  echo user@host >.forward
$ echo Fwd to user@host, type:  echo user@host >.forward
$ cat .forward
Fwd to user@host, type:  echo user@host
$ echo Fwd to user@host, type:  echo user@host \>.forward
Fwd to user@host, type: echo user@host >.forward
$ echo 'Fwd to user@host, type:  echo user@host >.forward'
Fwd to user@host, type:  echo user@host >.forward
$ echo "Fwd to user@host, type:  echo user@host >.forward"
Fwd to user@host, type:  echo user@host >.forward
$
$ address=ajr@cs.toronto.edu
$ echo Fwd to $address, type:  echo $address >.forward
$ echo 'Fwd to $address, type:  echo $address >.forward'
Fwd to $address, type:  echo $address >.forward
$ echo "Fwd to $address, type:  echo $address >.forward"
Fwd to ajr@cs.toronto.edu, type:  echo ajr@cs.toronto.edu >
.forward
$
```

That's because in *this* command, the first argument to 'echo' is just Fwd; the second argument is to; the third argument is user@host; and so on,

... whereas in *this* command, there is only one argument to echo, which is this string.



```
Fwd to user@host, type:  echo user@host >.forward
$ echo Fwd to user@host, type:  echo user@host >.forward
$ cat .forward
Fwd to user@host, type:  echo user@host
$ echo Fwd to user@host, type:  echo user@host \>.forward
Fwd to user@host, type: echo user@host >.forward
$ echo 'Fwd to user@host, type:  echo user@host >.forward'
Fwd to user@host, type:  echo user@host >.forward
$ echo "Fwd to user@host, type:  echo user@host >.forward"
Fwd to user@host, type:  echo user@host >.forward
$
$ address=ajr@cs.toronto.edu
$ echo Fwd to $address, type:  echo $address >.forward
$ echo 'Fwd to $address, type:  echo $address >.forward'
Fwd to $address, type:  echo $address >.forward
$ echo "Fwd to $address, type:  echo $address >.forward"
Fwd to ajr@cs.toronto.edu, type:  echo ajr@cs.toronto.edu >
.forward
$
```

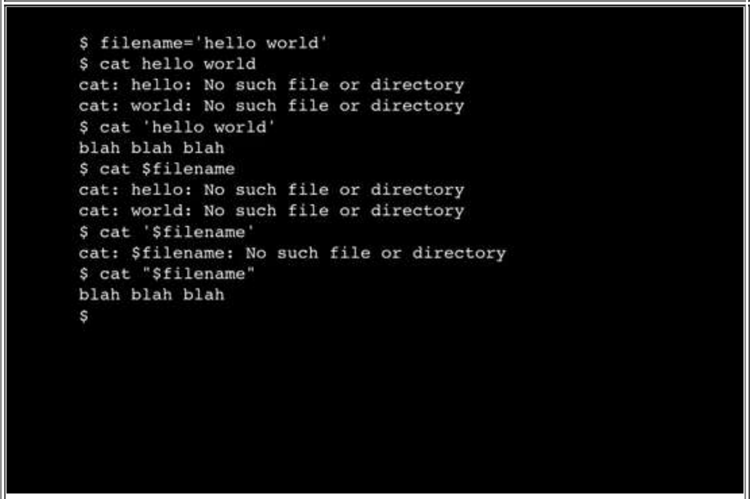


```
$ cat a b
cat: a: No such file or directory
cat: b: No such file or directory
$ cat 'a b'
cat: a b: No such file or directory
$
```

In any case, *echo* outputs all of its arguments, separated by spaces, so the result is very similar. But compare

```
cat a b
cat 'a b'
```

In the second case, the file name is actually the three characters a space b. That three-character string is all one argument to cat. So, 'space' is also a special character to the shell, and its special meaning is also suppressed by quoting.



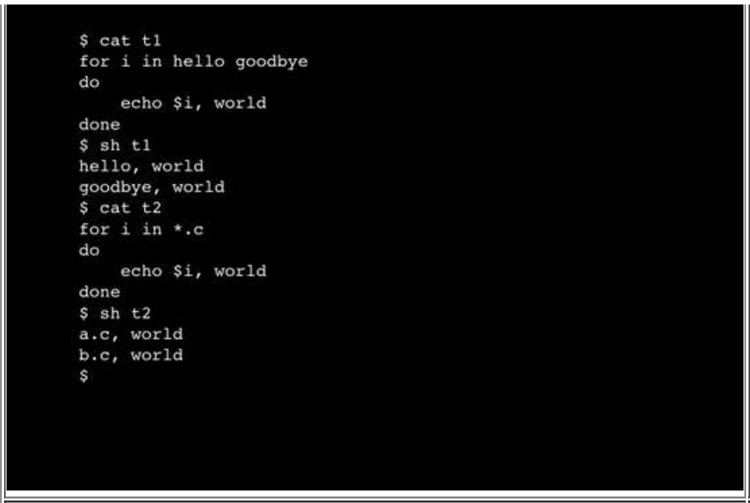
```
$ filename='hello world'
$ cat hello world
cat: hello: No such file or directory
cat: world: No such file or directory
$ cat 'hello world'
blah blah blah
$ cat $filename
cat: hello: No such file or directory
cat: world: No such file or directory
$ cat '$filename'
cat: $filename: No such file or directory
$ cat "$filename"
blah blah blah
$
```

Suppose we have a file named "hello world", with a space in the file name. We might have a variable containing this file name...
If we wanted to cat the file normally, we couldn't do "cat hello world", because of course *cat* would get two arguments, neither of which is the file name.
Instead, as we now know, we have to use quotes:

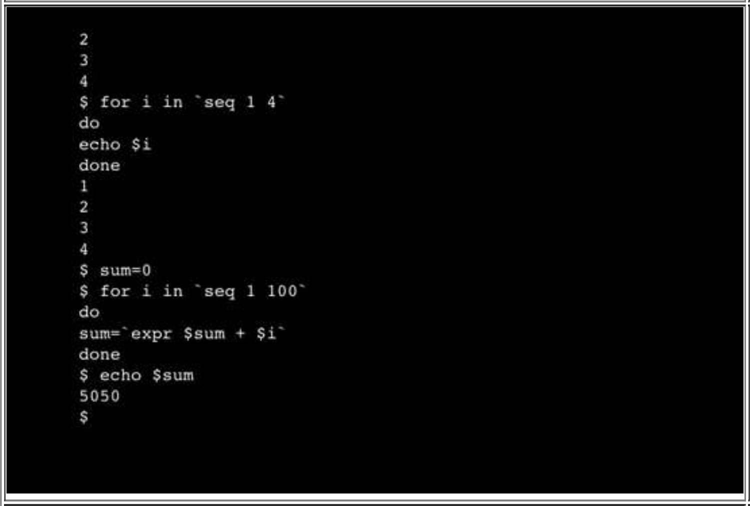
```
cat 'hello world'
```

Similarly, if we want to use the variable, we can't write "cat $filename", because the shell interpolates the variable value, then takes the spaces as separating arguments...
But again, single quotes suppress too *much*...
So we want double quotes...
The double quotes suppress the interpretation of the space, but they don't suppress the interpretation of the dollar sign. So almost any time we have a variable whose value might include a space, we are going to want to put that variable interpolation in double quotes.

Ok, let's talk about more control constructs. The 'for' construct is more like the one in Python than the one in C or Java.
The 'for' loops over any number of strings. Of course, you don't always specify the list of strings literally.
Here we loop over all file names ending with ".c".



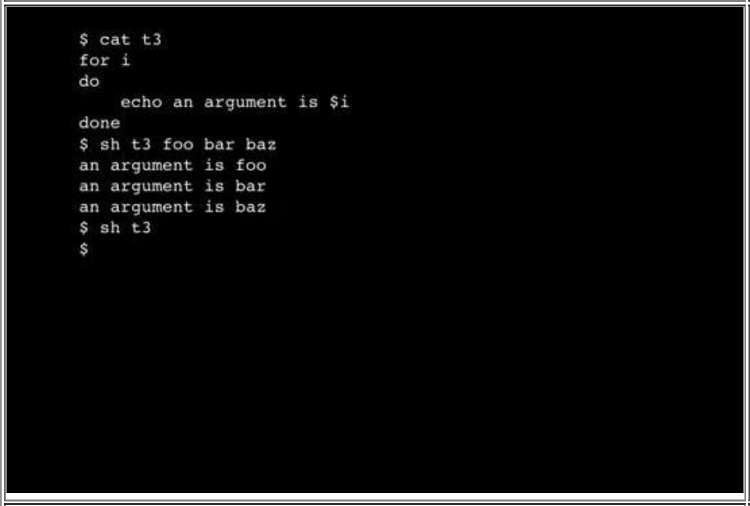
```
$ cat t1
for i in hello goodbye
do
    echo $i, world
done
$ sh t1
hello, world
goodbye, world
$ cat t2
for i in *.c
do
    echo $i, world
done
$ sh t2
a.c, world
b.c, world
$
```



```
2
3
4
$ for i in `seq 1 4`
do
echo $i
done
1
2
3
4
$ sum=0
$ for i in `seq 1 100`
do
sum=`expr $sum + $i`
done
$ echo $sum
5050
$
```

The *seq* command is a useful tool which outputs a range of numbers based on its arguments, very similar to *range* in Python.
This is very useful in a 'for' statement, in backquotes. If we write "`for i in `seq 1 4``" then it will run seq, seq's output will be captured, the output will be substituted in the command line, so this will be the same as writing "`for i in 1 2 3 4`". But that's more boring so let's go back to this.

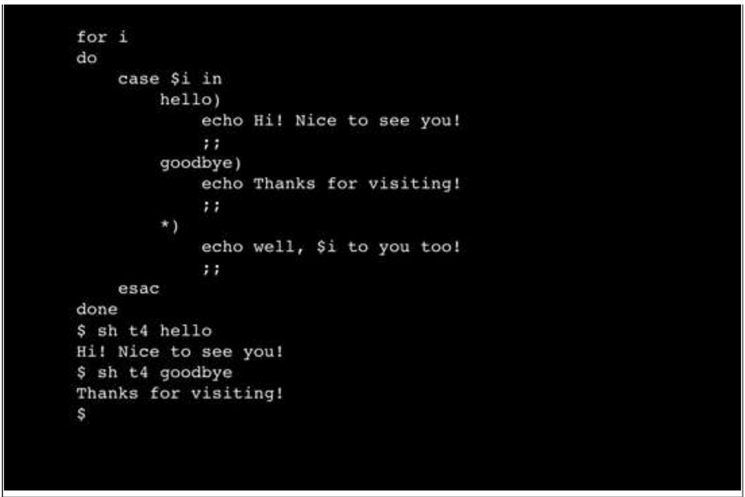
Let's add the numbers from 1 to 100.



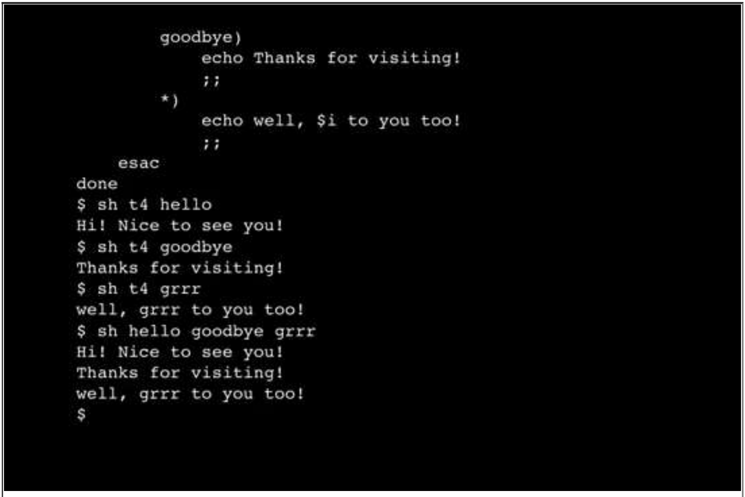
```
$ cat t3
for i
do
    echo an argument is $i
done
$ sh t3 foo bar baz
an argument is foo
an argument is bar
an argument is baz
$ sh t3
$
```

We haven't talked about processing the command-line arguments to your shell script yet, but I will mention at this point that there is a version of the 'for' statement which loops through all of the command-line arguments. You write "for" and a variable name, and that's all.
With no arguments, the loop executes zero times.

sh also has a "case" statement.
In case you were starting to miss the reversed keyword for the 'end' keyword, here's another one in the "case" command. The syntax is pretty weird with the closing parentheses in the case labels. It's meant to look like numbered items in traditional text, in which we do use a right parenthesis in this way. But in a programming language, it's weird. But not really problematic.
In case you haven't seen "case" or "switch" in other programming languages before, it's similar to a cascading if/elseif/elseif, except that there's one value being compared always -- each case compares it to a new fixed string. So if $i expands to 'hello', we execute the first case; else if it expands to 'goodbye' we execute the second case. Each particular case is terminated with the double-semicolon.



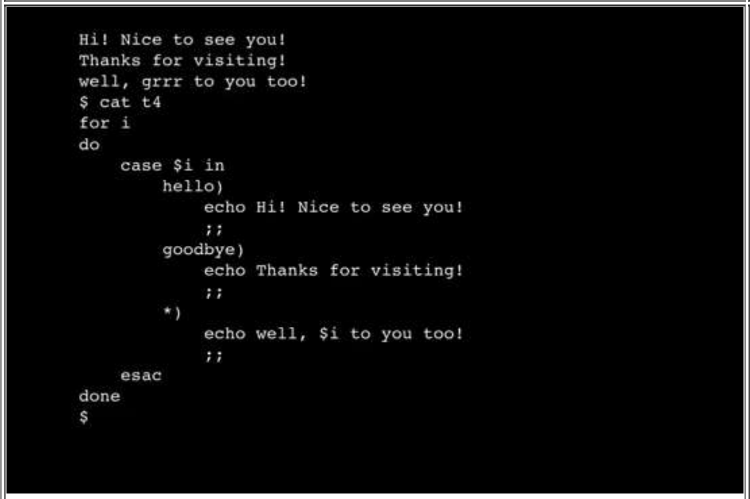
```
for i
do
    case $i in
        hello)
            echo Hi! Nice to see you!
            ;;
        goodbye)
            echo Thanks for visiting!
            ;;
        *)
            echo well, $i to you too!
            ;;
    esac
done
$ sh t4 hello
Hi! Nice to see you!
$ sh t4 goodbye
Thanks for visiting!
$
```



```
        goodbye)
            echo Thanks for visiting!
            ;;
        *)
            echo well, $i to you too!
            ;;
    esac
done
$ sh t4 hello
Hi! Nice to see you!
$ sh t4 goodbye
Thanks for visiting!
$ sh t4 grrr
well, grrr to you too!
$ sh hello goodbye grrr
Hi! Nice to see you!
Thanks for visiting!
well, grrr to you too!
$
```

Programming languages with 'case' statements generally have the possibility of a "default" case, which is executed if none of the fixed labels matches, like the 'else' at the end of a cascading if/elseif/elseif. In sh, we have a more general mechanism, which is that each of the case labels is actually in the "glob" notation, the pattern language used for filename wildcards. So for a default case, we can use an asterisk like here, which matches anything... because the case labels are tested *in order*. So it's only going to check the asterisk against the value of $i after 'hello' and 'goodbye' don't match.

And it *is* a loop, "for i"...

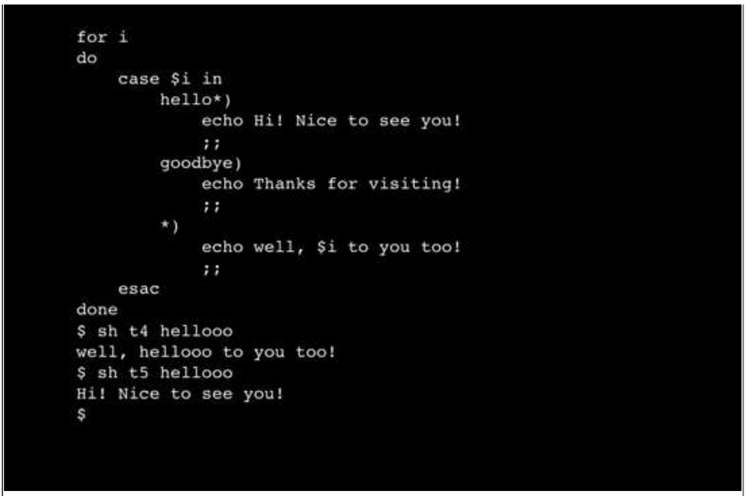


```
Hi! Nice to see you!
Thanks for visiting!
well, grrr to you too!
$ cat t4
for i
do
    case $i in
        hello)
            echo Hi! Nice to see you!
            ;;
        goodbye)
            echo Thanks for visiting!
            ;;
        *)
            echo well, $i to you too!
            ;;
    esac
done
$
```

Here's that file again; you can pause the video if you want to look at it.

Let's have a little look at other uses of the glob notation in case labels:
Change "hello" to "hello*".
Now compare the behaviour of the old version...
to the new version...

Again, please pause the video if you want to look at this more.

 ```for i / do / case $i in / hello*) / echo Hi! Nice to see you! / ;; / goodbye) / echo Thanks for visiting! / ;; / *) / echo well, $i to you too! / ;; / esac / done / $ sh t4 hellooo / well, hellooo to you too! / $ sh t5 hellooo / Hi! Nice to see you! / $```	
	That concludes the third video. The fourth video in this series is about i/o redirection and about command-line arguments.

```
for i
do
    case $i in
        hello*)
            echo Hi! Nice to see you!
            ;;
        goodbye)
            echo Thanks for visiting!
            ;;
        *)
            echo well, $i to you too!
            ;;
    esac
done
$ sh t4 hellooo
well, hellooo to you too!
$ sh t5 hellooo
Hi! Nice to see you!
$
```