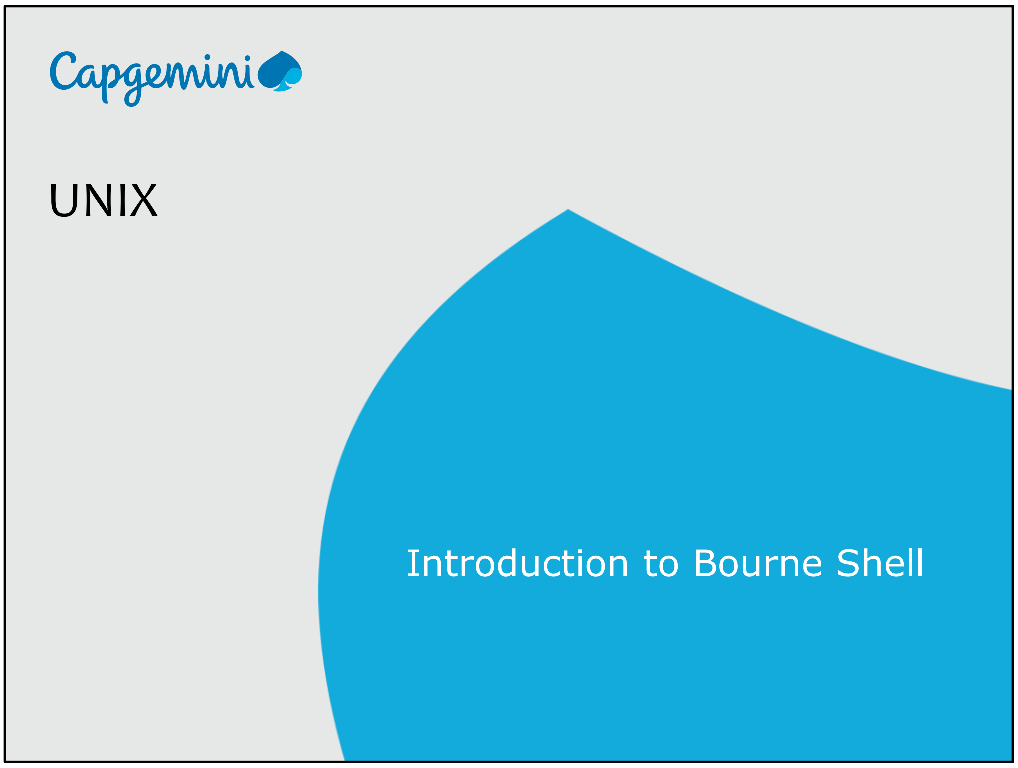
Instructor Notes:



Capgemini

UNIX

Introduction to Bourne Shell

Instructor Notes:

Lesson Objectives

To understand following topics:

- Different shell types
- Working of shell
- Bourne shell metacharacters
- Shell redirection
- Command substitution

Instructor Notes:

4.1: Shell Types

Overview

Shell is:

- The agency that sits between user and UNIX System
- Much more than command processor

Different shell types in the UNIX system are:

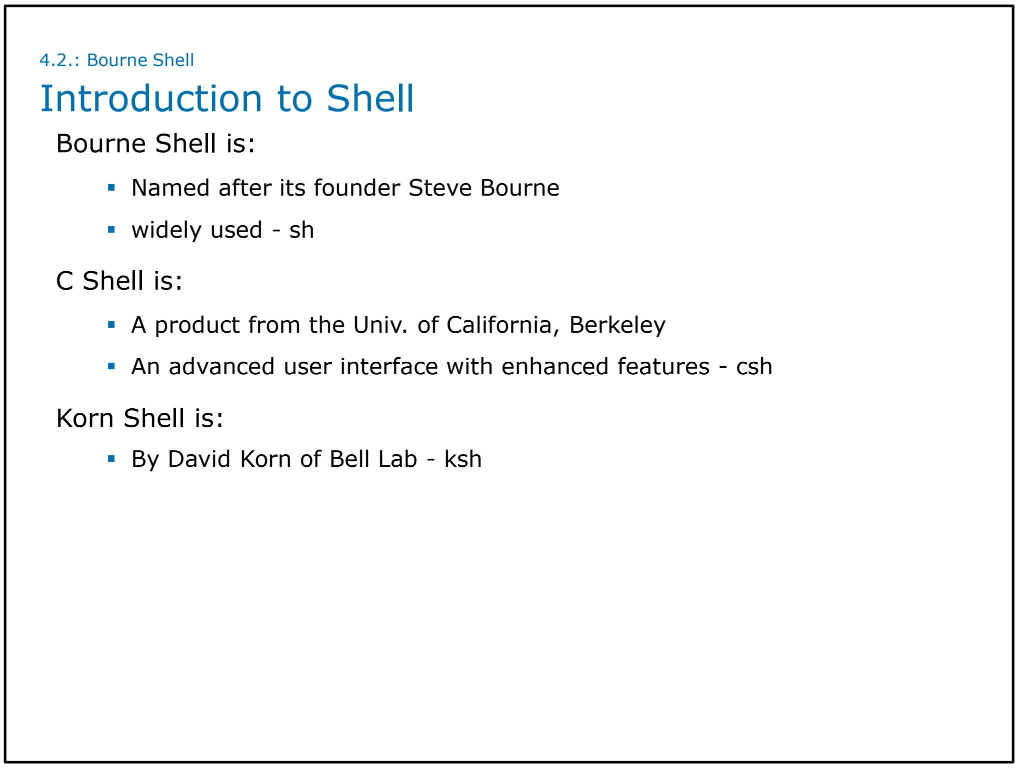
Bourne Shell	- sh
K Shell	- ksh
C Shell	- csh
Restricted Shell	- rsh

What is a Shell?

The shell is the agency that sits between the user and the Unix system. Whenever a command is issued, it is the shell that acts as the command interpreter. But the shell in Unix is much more than just a command processor. This chapter introduces some of the features of the Bourne Shell.

Shell is a process that creates an environment for you to work in.
When you login to UNIX machine you see a prompt because automatically a new shell process is started. This process will be terminated whenever user logs out.
How the command is executed
-The shell displays prompt and wait for you to enter a command
-When you type any command it scans the command line and processes all metacharacters to recreate simplified command. (e.g if the command is rm *, then * will be replaced by all file names in the current directory)
-Then it passes the command to kernel for execution
-And wait till execution of the command completes
-After command execution is complete, it displays prompt again to take up the next command

Instructor Notes:



4.2.: Bourne Shell

Introduction to Shell

Bourne Shell is:

- Named after its founder Steve Bourne
- widely used - sh

C Shell is:

- A product from the Univ. of California, Berkeley
- An advanced user interface with enhanced features - csh

Korn Shell is:

- By David Korn of Bell Lab - ksh

Introduction to the shell:

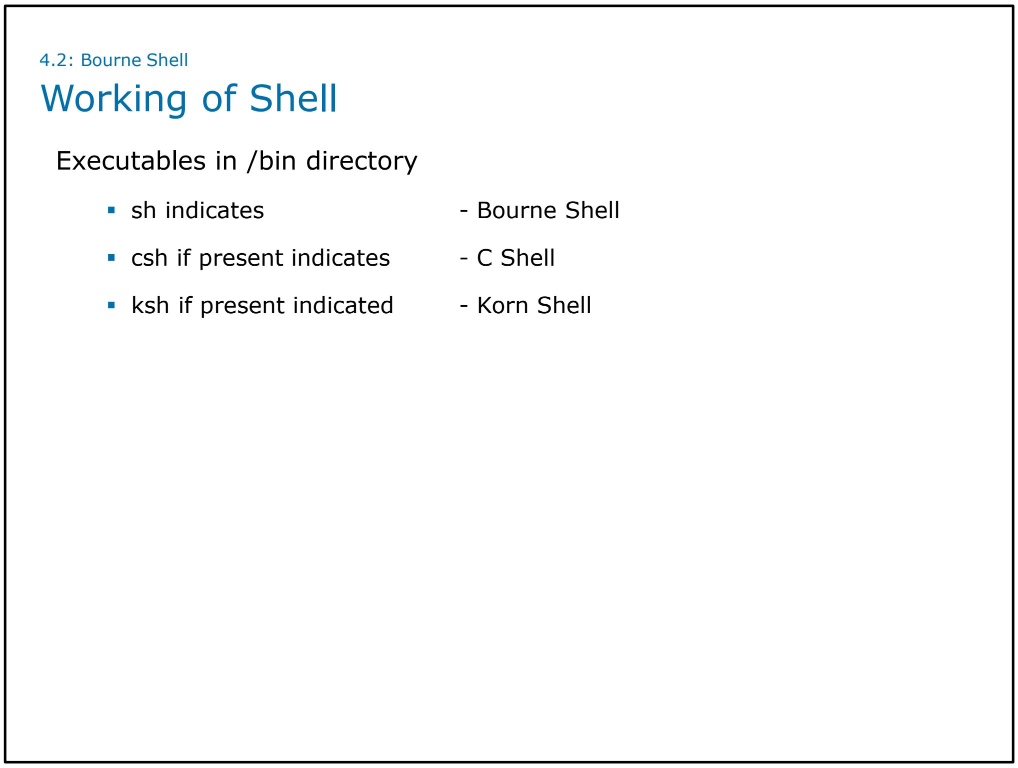
Bourne Shell is one of the earliest and most widely used Unix shells. It is named after its founder Steve Bourne. The executable program sh in the /bin directory is the Bourne shell.

There are other shells available on the Unix systems – popular amongst them are the C shell and the Korn shell.

The C shell is a product from the University of California, Berkeley. It has an advanced user interface with enhanced features. C shell, if present, is available as csh.

The Korn shell is from the Bell Labs. It is the most modern shell available currently, and is likely to become a standard. The Korn shell executable is ksh.

Instructor Notes:


4.2: Bourne Shell

Working of Shell

Executables in /bin directory

▪ sh indicates	- Bourne Shell
▪ csh if present indicates	- C Shell
▪ ksh if present indicated	- Korn Shell

Working of Shell:
The shell is a Unix command – it is a program that starts when user logs in and terminates when user logs out. The job of the shell is to accept and interpret user requests (which are nothing but other Unix commands). Besides the shell is also programmable – this will be covered later.

The shell typically performs following activities in a cycle:
Issues a $ prompt, and waits for user to enter a command
Scans and processes the command after user enters command
The command is passed on to the Kernel for execution and the shell waits for its conclusion
The $ prompt appears so that user can enter next command. Hence the shell is in a continuous sleep – waking – waiting cycle

Some of the features of the shell are explored here.

Instructor Notes:

Working of Shell (contd..)

Continuous sleep-waking-waiting cycle

Performs following activities:

- Issues a $ prompt & waits for user to enter a command.
- After user enters command, shell scans & processes the command.
- The command is passed on to the Kernel for execution & the shell waits for its conclusion.
- The $ prompt appears so that the user can enter next command.

Instructor Notes:

4.3: Metacharacters

Description

Following are the Bourne Shell metacharacters:

- * : To match any number of characters
- ? : To match with a single character
- [] : Character class; Matching with any single character specified within []
- ! : To reverse matching criteria of character class
- \ : To remove special meaning attached to metacharacters
- ; : To give more than one command at the same prompt
- All redirection operators >, <, >> are also shell metacharacters

Shell Metacharacters for pattern matching and combining commands:

Metacharacters are characters to which the shell attaches special meaning. The shell interprets these metacharacters and the command is rebuilt before it is passed on to the kernel.

Wildcards for Pattern Matching:

The wildcard * is used to match any number of characters (including none). It however, does not match patterns beginning with a dot (.).

The wildcard ? matches a single character.

Examples:

emp* : This would match all patterns that begin with emp, and may be followed by any number of characters (like emp, emppune, empttc, empseepz etc).

emp? : This would match all patterns that begin with emp and are followed by exactly one more character, which could be anything (like emp1, empa etc).

Patterns can be made more restrictive by using character class, represented by []. Any number of characters can be specified within the [], but the matching would occur for a single character within a class.

emp[abc]: This would match with patterns that begin with emp followed by a or b or c any one of it. (like empa, empb or empc)

Contd..

Instructor Notes:

Escaping with the Backslash:

In order to remove the special meaning attached to metacharacters, the backslash can be used. For example, the expression emp* would match only with the pattern emp*. In the given example \ removes special meaning of *(i.e. 0 or more characters) and treat it as a character '*'

Combining command using the semicolon:

It is possible to give more than one command at the same prompt so that they will be executed in sequence one after the other. This is done with the character ; as in the example below:

Example: Using the wild cards:

```
$ ls -l file2.txt ; chmod u+x file2.txt ; ls -l file2.txt
-rwxr--r--   1 deshpavn group        61 Mar 29 10:44 file2.txt
$
```

Here it is possible to assign permissions and subsequently check the same.

Instructor Notes:

4.3: Redirections

Shell Redirections

Every Unix command has access to:

- Standard input
- Standard output
- Standard error

Shell can redirect I/p, o/p or error to any physical file using meta characters "<", ">" & "2>"

Redirections by Shell:

Many commands work with character streams. The default is the keyboard for input (standard input, file number 0), and terminal for the output (standard output, file number 1). In case of any errors, the system messages get written to standard error (file number 2), which defaults to a terminal.

Unix treats each of these streams as files, and these files are available to every command executed by the shell. It is the shell's responsibility to assign sources and destinations for a command. The shell can also replace any of the standard files by a physical file, which it does with the help of meta characters for redirection.

In order to take the standard input from a file, the character < is used.

```
$ wc < file1.txt
```

```
  2    12    60
```

To redirect the output to a file, > is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, >> is used.

```
$ wc < file1.txt > result
$ cat result
  2    12    60
```

Instructor Notes:

Shell Redirections (contd..)

Examples:

```
$ ls > temp
$ wc < file1.txt > result
$ cat nonexistantfile 2> err
```

Using redirections:

```
$ wc < cfile1.lst >> result
$ cat result
   2   12   60
   4    4    8
```

As shown above, it is possible to combine redirection operators on a single command line. The order of the redirection symbols does not affect the working of the command.

Using redirections:

To redirect the standard error, 2> is used.

```
$ cat xyzfile
cat: cannot open xyzfile: No such file or directory (error 2)
$ cat xyzfile 2> errfile
$ cat errfile
cat: cannot open xyzfile: No such file or directory (error 2)
$ cat xxfile 2>> errfile
$ cat errfile
cat: cannot open xyzfile: No such file or directory (error 2)
cat: cannot open xxfile: No such file or directory (error 2)
```

Instructor Notes:

4.3: Redirections

Building Block Primitives

Pipe - allows stream of data to be passed between reader & writer process.

O/p of first command is written into pipe and is input to the second command.

- $ who | wc -l
- $ ls | wc –l
- $ ls | wc -l > fcount
- $cat file1.txt | wc –l (To display number of lines in file file1.txt)

Connecting commands with Pipes:
The shell has a special operator called pipe (|), using which the output of one command can be sent as an input to another. The shell sets up the interconnection between commands. It eliminates the need of temporary files for storing intermediate results.
Creating a Pipeline:

```
$ who | wc -l
        8
```

When a sequence of commands are combined this way, a pipeline is said to have been formed.
It is possible to combine redirection along with the pipe.

Combining pipe with redirection:

```
$ ls | wc -l > output
$ cat output
   13
```

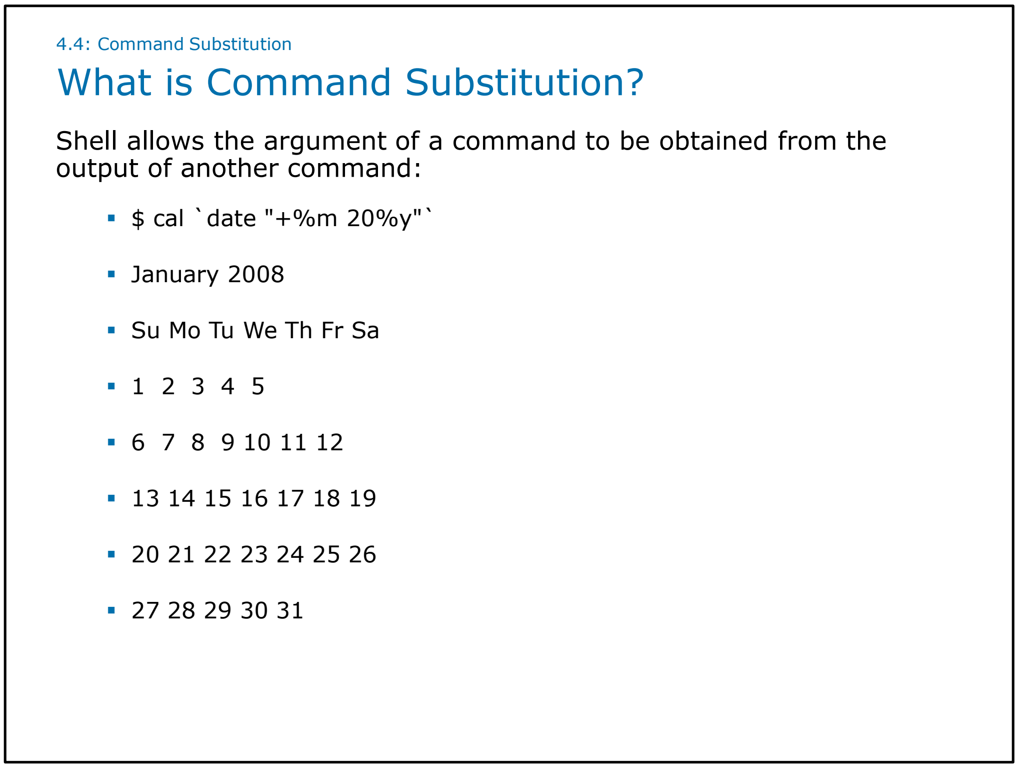
Instructor Notes:

Building Block Primitives (contd..)

| - pipe symbol

Any number of commands can be combined together to make a single command.

Instructor Notes:



4.4: Command Substitution

What is Command Substitution?

Shell allows the argument of a command to be obtained from the output of another command:

- $ cal `date "+%m 20%y"`
- January 2008
- Su Mo Tu We Th Fr Sa
- 1 2 3 4 5
- 6 7 8 9 10 11 12
- 13 14 15 16 17 18 19
- 20 21 22 23 24 25 26
- 27 28 29 30 31

What is Command Substitution?

The shell allows the argument of a command to be obtained from the output of another command – this feature is called command substitution. This is done by using a pair of backquotes. The backquoted command is executed first. The output of command is substituted in place of command. Then outer command will get executed.

Example:

```
$ cal `date "+%m 20%y"`
   March 2001
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Instructor Notes:

4.5: Shell Script

What is Shell Script?

Group of commands that need to be executed frequently can be stored in a file, called as a shell script or a shell program.

```
$ cat script2.sh                    O/P:$ script2.sh
echo 'Enter your name:                    Enter your name:
read uname                          xyz
echo "Hi $uname"                           Hi xyz
```

To assign values to variables, use the set command.

```
$ set uname="EveryOne"
$ echo Hi $uname
   Hi EveryOne
```

User defined shell variables and the echo command:

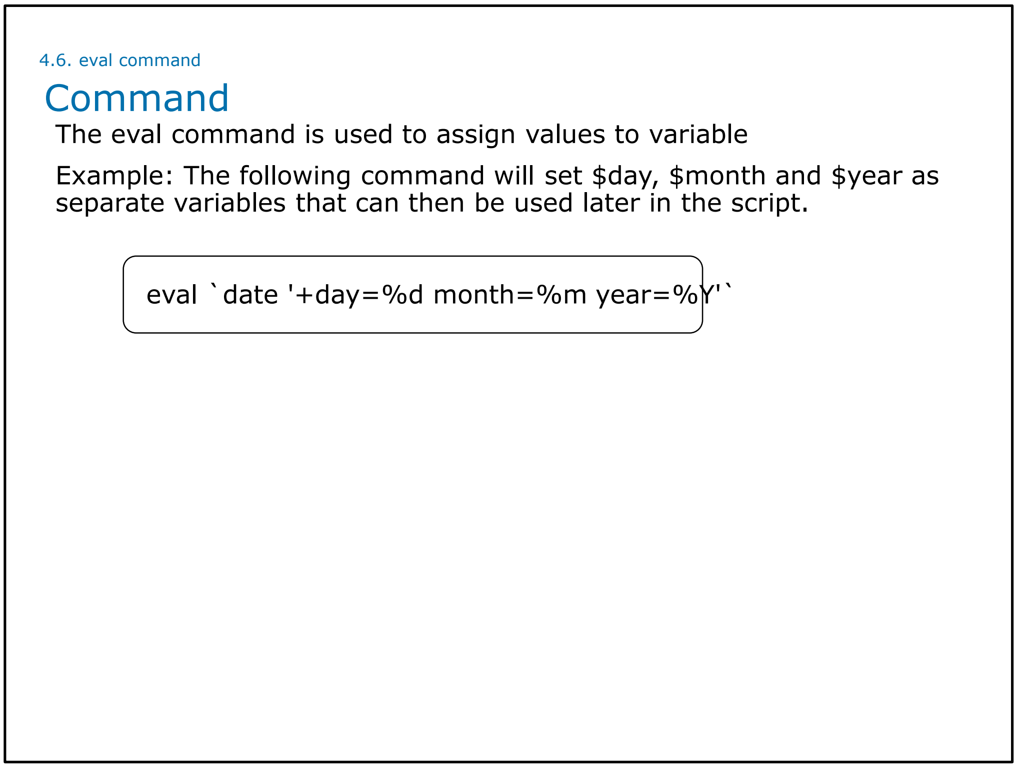
Since the shell is programmable, it is possible for users to define their own variables. No type is associated with the shell variables. The value of a variable is always string type.

Shell variables are assigned values with the = operator, in the form variable=value. [There should be no spaces on either side of =]. Variable names can be combination of letters, digits & underscore, but the first character has to be a letter. Shell is sensitive to case. Even though the value is a string, if it contains only numerals, it can be used for numerical computation.

In order to retrieve the value stored in a variable, the $ sign needs to be used. The command echo can be used to display messages, as also the values of variables.

```
$ echo hi
hi
$ echo hi there
hi there
$ echo "hi there"
hi there
$ set msg="Hi there"
$ echo $msg
 Hi there
```

Instructor Notes:


4.6. eval command

Command

The eval command is used to assign values to variable

Example: The following command will set $day, $month and $year as separate variables that can then be used later in the script.

```
eval `date '+day=%d month=%m year=%Y'`
```

Using the shell eval command:

In shell scripts it is common to set variables using the output of a command, Example:

```
variable=`command`
```

The output from the command is normally a single value. If, however, the command returns multiple values you need to use some other program (e.g. awk) to split the combined result into separate parts. Alternatively, you could pass an argument to the command to specify which result should be returned.

A neater solution is to use the shell's eval command. For example, say you wanted to return day, month and year from the date command. You'd have to write either:

```
day=`date +%d`
 month=`date +%m`
year=`date +%Y`
```

or:

```
result=`date '+%d %m %Y'`
```

and then break up $result with awk or cut for example. Instead you can write:

```
eval `date '+day=%d month=%m year=%Y'`
```

This will set $day, $month and $year as separate variables that can then be used later in the script.

Instructor Notes:

Summary

In UNIX different types of shells are available: CSH, KSH and Bourne Sh.

Redirection operator can be used to redirect i/p or o/p to files or printer.

Pipeline character can be used to send o/p of one command as i/p of another command.

Group of commands that need to be executed frequently are stored in a file, called as a shell script.

Instructor Notes:

Review Questions

Question 1: In shell, what are the different metacharacters available?

Question 2: ______ symbol is used as output redirection.

Question 3: ______ symbol is used as command substitution operator.