Name: Kevin Tran | **PID:** A16809193 | **(page 1)**

«command» < «file» ***Input redirection*** – use contents of «file» as the standard input to the command. That's what you would type!

java Read < name.txt

Read.java

```
import java.util.Scanner;
class Read {
  public static void main(String[] args) {
        System.out.print("What's your name? ");
        Scanner in;
        in = new Scanner(System.in);
        String s = in.nextLine();
        System.out.println("Hello " + s);
  }
}
```

multiple inputs --> file should have the same # of input in file

```
$ java Read
What's your name? JoeTyped
Hello JoeTyped
$ cat name.txt
JoeFromFile
$ java Read < name.txt
```

takes/reads the contents of file --> uses that as the standard input

* useful for testing programs that deals with standard input

bash has **for** statements! They can iterate over many things, including lists of paths or lines of output.

```
for VAR in <sequence>
do
  <commands; $VAR is bound to an element of sequence>
done
```

Assume we have a program with the following behavior. Underlined text is typed at standard input

```
$ javac AgeCalc.java
$ java AgeCalc
1987/6/22
You're 36 yrs old.
$ java AgeCalc
2024/7/12
You don't exist yet.
```

Assume we have a directory of files like below, where the contents of each file is in quotes next to it.

```
AgeCalc.java
check.sh <-- shellscript file
test-files/
  test1.txt "1987/6/22"
  test1.txt.expect "You're 36 yrs old."
  test2.txt "2024/7/12"
  test2.txt.expect "You don't exist yet."
  test3.txt "2023/12/5"
  test3.txt.expect "You're 0 yrs old."
  ...
```

.sh files can also use command < file

Let's write a **bash script** that will run the program on all the test files.

```
set -e

javac AgeCalc.java

for TEST in test-files/*.txt
 do
    echo "Testing: $TEST"
    cat $TEST
    java AgeCalc < $TEST
    cat $TEST.expect
done --> end loop
```

$TEST -> here comes a variable

shoves the string file $TEST with .expect to easily find expect file

bash has **`if`** statements! **`elif`**/**`else`** are optional, and there can be multiple **`elif`** clauses.

```
if <condition>
then
  <commands>
elif <condition>
then
  <commands>
else
  <commands>
fi
```
<-- ends if statement

Bash has **variables**, which are declared with

```
NAME=value
```

Bash also lets you store the ***output of a program*** in a variable, with:

```
NAME=$(some-command with some argos)
```

Bash also lets you store the ***output of a program*** in a file, with:

```
command > file
```

This last example is called **output redirection**.

Some common ***`<condition>`*s** (iff means "if and only if"):

`[[ -e <val> ]]` – **true** iff the path *val* exists
`[[ -f <val> ]]` – **true** iff the path *val* exists and is a file

`[[ _____ <val> ]]` – **true** iff the path *val* exists and is a directory

`[[ <val1> -eq <val2> ]]` – **true** iff values arithmetically equal
`[[ <val1> -ne <val2> ]]` – **false** iff values arithmetically equal
`[[ <val1> -gt <val2> ]]` – **true** iff <*val1*> is greater than <*val2*>

`[[ <val1> _____ <val2> ]]` – **true** iff <*val1*> is less than <*val2*>

`[[ <val1> == <pattern> ]]` – **true** iff val1 matches the pattern

A `<val>` here could be a use of a bash variable (like `$SOMEVAR`), or constant string values like `"0"` or `"1"`, or a written out path like `data-dir/`. Generally think of these all as being *string* values.

A <*pattern*> here could be a constant string, or something using `*` like `*Bahamas*`.

Let's update the AgeCalc script to print out whether each test matched its expectation

```
set -e

javac AgeCalc.java

for TEST in test-files/*.txt

    OUTPUT=$(java AgeCalc < test-files/test1.txt)
    EXPECTED = $(java AgeCalc < test-files/test1.txt.expect)

    if [[ $OUTPUT == $EXPECTED ]]
    then
      echo "Success"
    else
      echo "Failure"
      echo "Expected: $EXPECTED"
      echo "Got: $OUTPUT"

    fi
done
```

Name: | **PID:** **(page 1)**

Example directory structure, file contents in parentheses

```
some-files/
|-  a.txt ("hello\n")
|-  more-files/
     |-  b.txt ("hi\n")
     |-  c.java ("psvm\n")
|-  even-more-files/
     |-  d.java ("junit\ntest")
     |-  a.txt ("nested file\n")
```

`find «path»:` Recursively traverse the given path and list all files in that directory and subdirectories

`wc «file»`: Print the number of lines, words, and characters in a file or files

`grep «string» «files»:` Search a file or files for the given string, print matching lines

`«command» > «file»` Save the output of the command in the given file. Overwrites the file!

`*` (asterisk, star) Used to create **patterns**, which can refer to multiple files.
Examples: `lib/*.jar`, `*.txt`

`echo «arguments»` Print the arguments to the terminal

Which command or commands (do you think) produces the output on the right, and why? Make a guess!

```
$ ls some-files/*
```

```
$ find some-files
```

```
some-files
some-files/even-more-files
some-files/even-more-files/d.java
some-files/even-more-files/a.txt
some-files/more-files
some-files/more-files/c.java
some-files/more-files/b.txt
some-files/a.txt
```

```
$ wc some-files/a.txt
```

```
$ wc some-files/even-more-files/a.txt
```

```
 1       2       12      Joe hid a little bit of output here
```

```
$ grep "e" some-files/a.txt
```

```
$ grep "e" some-files/even-more-files/a.txt
```

```
hello
```

```
$ grep "e" */a.txt
```

```
$ grep "e" */*/a.txt */a.txt
```

```
$ grep "e" */*/a.txt
```

```
some-files/even-more-files/a.txt:nested
some-files/a.txt:hello
```

```
$ find some-files > files.txt
$ grep ".txt" some-files
```

```
$ find some-files > files.txt
$ grep ".txt" files.txt
```

```
some-files/even-more-files/a.txt
some-files/more-files/b.txt
some-files/a.txt
```