

Reading 1 Discussion and Batch Processing

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- First section: tomorrow, Tuesday, October 6
- [Homework 1](#): due Friday, October 9, 11:59pm

Reading 1 discussion

- Form groups of 3 or 4 with the people next to you
 - Fishman and Hoare readers: raise a hand, then spread out
- **Willison 2025:** watch your code work, then write a test for it
- **Fishman 1996:** the strict process of the space shuttle's software team
- **Hoare 1996:** reliable software, though almost none of it is proven

Share (5 minutes)

When you say you know your code works, what do you actually know?

How did you come to know it?

- Each of you, in turn:
 - Which article you read, and its answer
 - One thing from your reflection that you want to discuss

Two kinds of testing

- **Manual:** you run the program and watch what it does
- **Automated:** another program runs it and checks the result, every time
- A JUnit test is automated, and no AI is involved

Argue (6 minutes)

- **Willison:** a test "proves the change works"
- **Lecture 1:** testing never confirms a claim about every run
- Who is right?
- What would your article say?
- If your group agrees, have someone argue the other side, or take one:
 - Does a one-line fix need both a manual test and an automated one?
 - AI wrote it and every test passes, but nobody read it: deliver it?

Report (a few minutes)

- A couple of groups: what did you decide, and why?

Half sheet (3 minutes)

- The names of the others in your group
- What today's discussion changed in your thinking
- One new point, and the article or person it came from
- Hand it in at the door as you leave

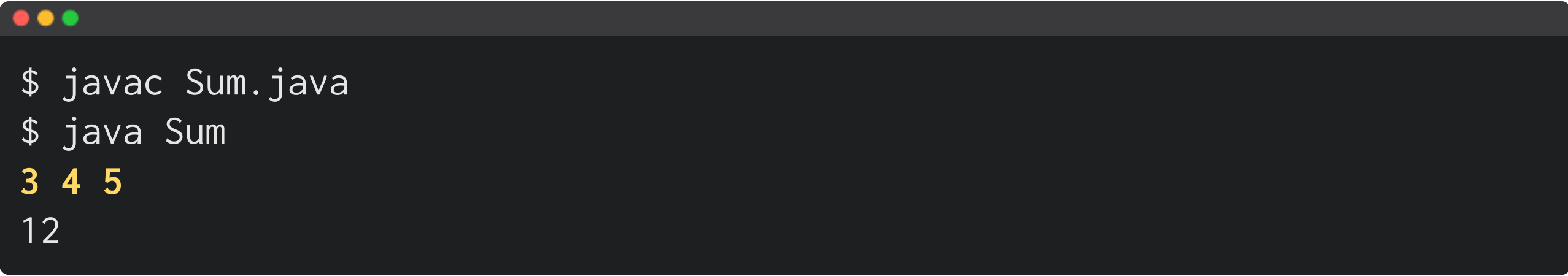
Comparing specs: stronger vs. weaker

Lecture 2's slides, from "Stronger and weaker"

A different Sum

- Lectures 1 and 2: a `Sum` that asks for numbers and adds them
- This `Sum` adds numbers too, in 99 lines
 - Most of them have nothing to do with adding
- Read it on the program sheet:
 - What does `Sum` do?
 - What is good about it?
 - What is not?

Let's run it



```
$ javac Sum.java
```

```
$ java Sum
```

```
3 4 5
```

```
12
```

End typed input with Control-D; on Windows, Control-Z, then Enter.

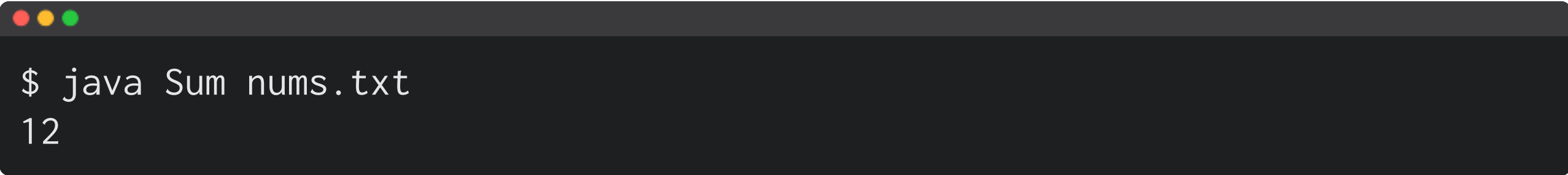
The average, with -a

```
$ java Sum -a
```

```
3 4 5
```

```
4.0
```

Numbers from a file

A terminal window with a dark background and a light gray title bar. The title bar has three colored window control buttons (red, yellow, green) on the left. The terminal displays a command prompt followed by a command and its output.

```
$ java Sum nums.txt  
12
```

nums.txt contains 3 4 5.

A word among the numbers

Lecture 1's `Sum` added 1 and 2 and ignored the rest.

```
$ java Sum
```

```
1 2 three 4
```

```
Sum: not an integer: three
```

It stops at the word, and says which word

Two numbers

Lecture 1's `Sum` printed -2147483648.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays the command `$ java Sum` and its output, which is split across two lines: `2147483647 1` in yellow and `2147483648` in white.

```
$ java Sum
2147483647 1
2147483648
```

Line 57: `sum` is a `long`, Java's 64-bit type for whole numbers

Batch programs

A batch program reads its input, computes, writes its output, and exits.

- Lecture 1's [Sum](#) and this one are both batch programs
- A web browser is not: it keeps running until told to stop
 - A later lecture takes up programs of that kind

Input

- **Command-line arguments:** the words after `java Sum`
 - Input too: the user chooses them, and they change what `Sum` does
- **Options:** arguments that start with a dash, like `-a`
- **A file:** named by any other argument
- **Standard input:** otherwise; by default, the keyboard
 - Lines 89 and 90 choose between them

A command line, word by word

```
$ java Sum -a --precision=2 nums.txt
```

`java Sum`: starts Sum's main

`-a`: an option, since it starts with a dash

`--precision=2`: an option with a value

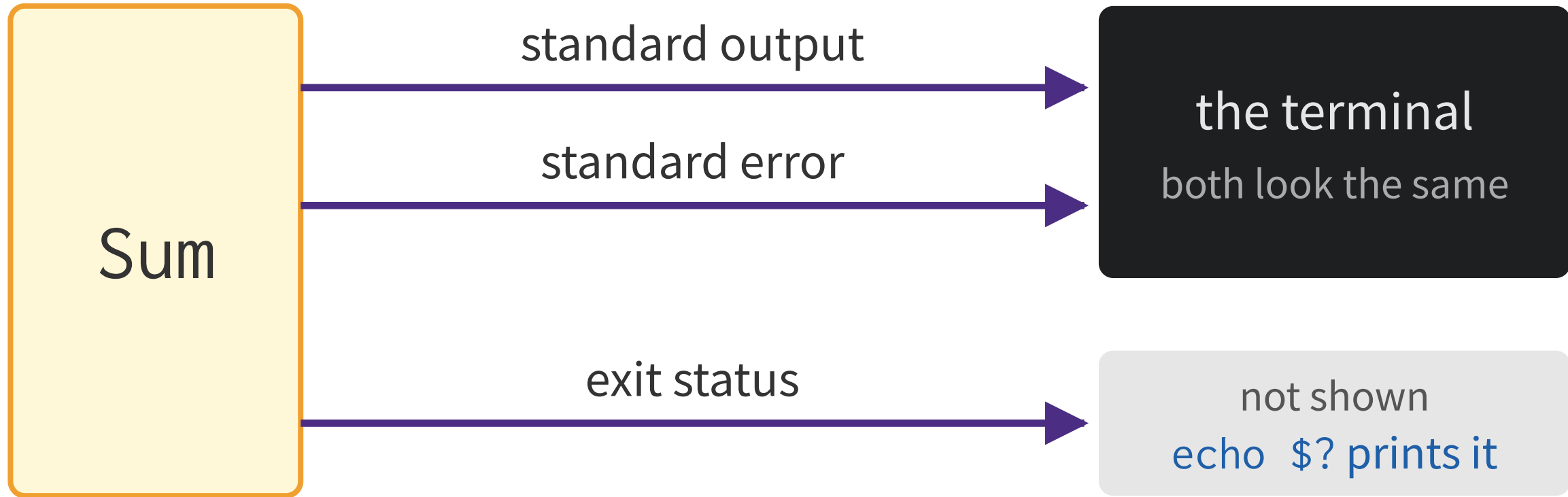
`nums.txt`: any other argument is a file to read

main gets args: `"-a"`, `"--precision=2"`, `"nums.txt"`

Output

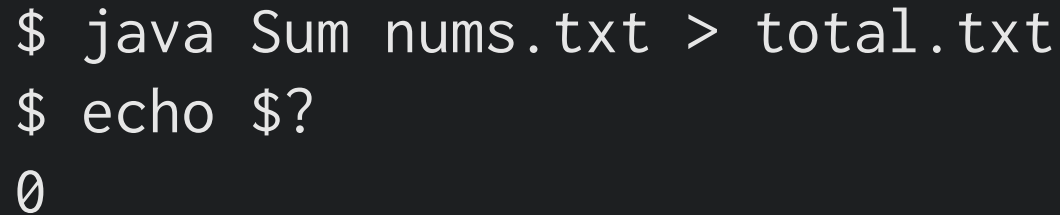
- **Standard output:** the results (lines 78 to 83)
- **Standard error:** the messages (line 96, in `fail`)
- **Exit status:** a number that says whether the run succeeded
 - 0 means success; anything else, failure
 - `fail` exits with status 1 (line 97)
- **In a terminal:** results and messages look the same
 - And the exit status does not show at all

Where the output goes



Why keep them apart?

> sends standard output to a file.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays the following text:

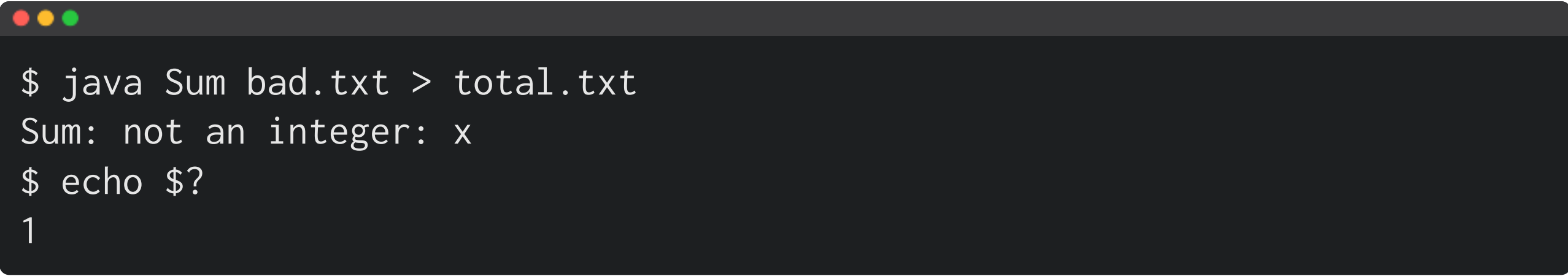
```
$ java Sum nums.txt > total.txt  
$ echo $?  
0
```

Nothing on the terminal, and 12 in total.txt

Exit status 0: the run succeeded

The last exit status: `echo $?` on macOS and Linux, `$LASTEXITCODE` in Windows PowerShell.

When a run fails

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays the execution of a Java program named 'Sum' with a file named 'bad.txt' as input. The program outputs an error message and an exit status.

```
$ java Sum bad.txt > total.txt
Sum: not an integer: x
$ echo $?
1
```

The message skips the file, and total.txt is empty

Exit status 1: the run failed

bad.txt contains 3 x.

Why standard error exists

- At first, every message went to standard output
 - Pipes then sent the messages into the next program
 - Typesetting runs printed only `cannot open file foobar`
- Soon after, Dennis Ritchie added standard error
- Then each message had to name its program
 - With several programs in a pipeline, who said it?

Stephen Johnson, 2013; M. D. McIlroy, 1987.

What a user sees

- **Three parts:** standard output, standard error, exit status
 - `Sum`'s specification has to say what each is, on every run
- A **person** at the terminal reads the first two
- A **script:** a program that runs other programs
 - It can read all three
 - Like a client in Lecture 2: it relies on `Sum`'s promises
 - Unlike a method's client: it owes `Sum` nothing

A maxim from 1978

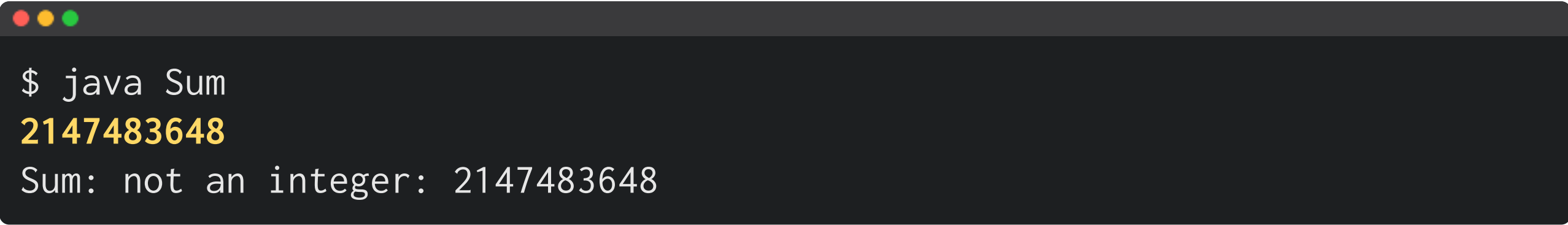
*Expect the output of every program to become
the input to another, as yet unknown, program.
Don't clutter output with extraneous information.*

- **Clutter:** a message among the results
 - Standard error keeps it out
- **As yet unknown:** [Sum](#)'s author cannot know its scripts
 - What scripts rely on has to be in the specification

M. D. McIlroy, E. N. Pinson, and B. A. Tague, 1978; often credited to McIlroy alone.

Testing the user manual

- Lines 5 to 19: a manual for whoever runs `Sum`
- It promises to read integers



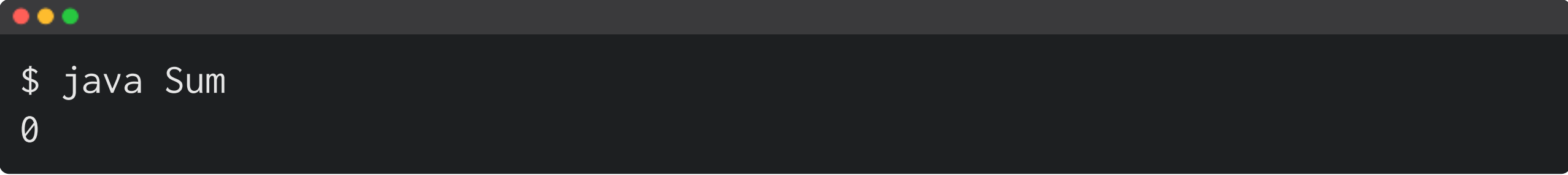
```
$ java Sum
```

```
2147483648
```

```
Sum: not an integer: 2147483648
```

Line 61 asks `hasNextInt`: only the integers that fit in an `int`

Nothing at all

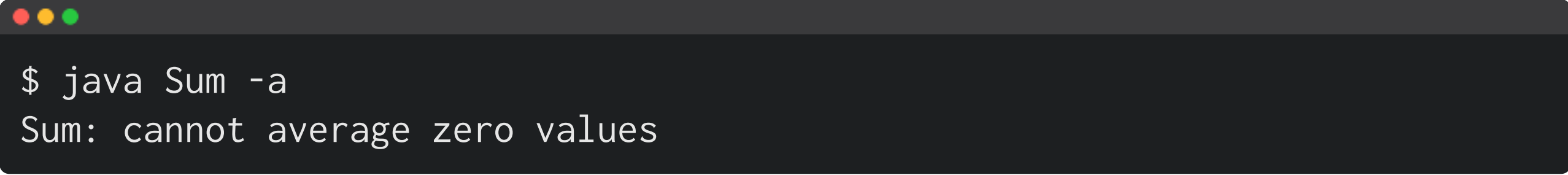


```
$ java Sum  
0
```

Type nothing; end the input at once.

Covered by the manual, if the sum of no numbers is 0

Nothing at all, with -a



```
$ java Sum -a  
Sum: cannot average zero values
```

Type nothing; end the input at once.

The manual says nothing about this run

Every error so far

```
95     private static void fail(String message) {  
96         System.err.println("Sum: " + message);  
97         System.exit(1);  
98     }
```

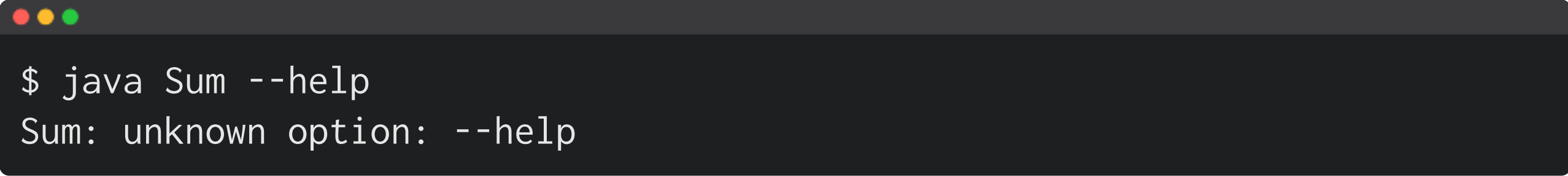
A message on standard error, starting with the program's name

Then exit status 1

- Each one went through `fail`
 - So each looks the same, to a person and to a script

Lecture 1's standard

- A stack trace is never the response: no run so far ends in one
- Every input gets a described response: `Sum` fails
 - The manual describes no message and no exit status
- And the manual sits in the source:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a command and its output.

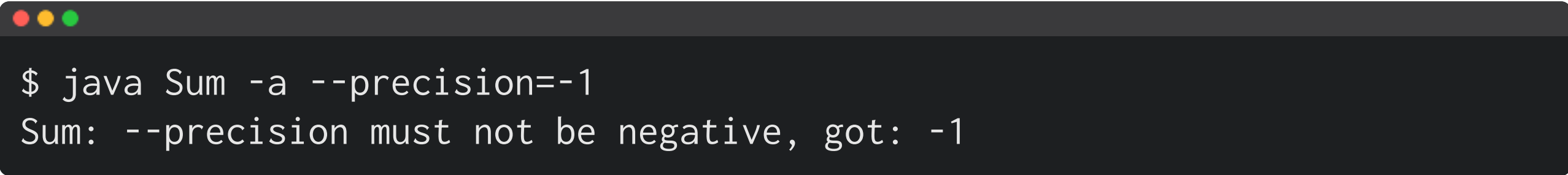
```
$ java Sum --help  
Sum: unknown option: --help
```

What are the checks for?

- Lines 28 to 55 check the arguments
- Line 61 checks each word of the input
- Lines 73 to 75 check that there is something to average
- Take one away, and run the same command
 - The lines go blank, so the rest keep their numbers

A negative precision

```
40  if (precision < 0) {  
41      fail("--precision must not be negative, got: "  
42          + precision);  
43  }
```



```
$ java Sum -a --precision=-1  
Sum: --precision must not be negative, got: -1
```

Without lines 40 to 43

```
$ java Sum -a --precision=-1
```

```
1 2
```

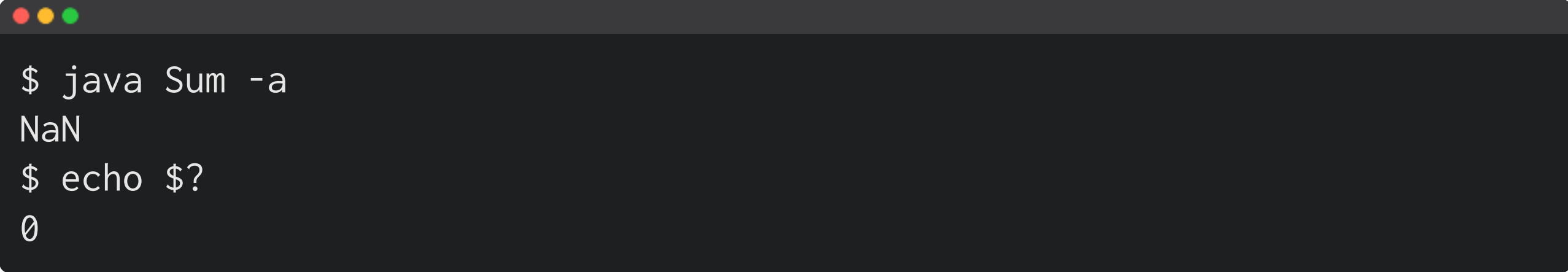
```
Exception in thread "main" java.util.UnknownFormatConversionException:  
Conversion = '.'
```

```
    at java.base/java.util.Formatter.parse(Formatter.java:2818)  
    at java.base/java.util.Formatter.format(Formatter.java:2744)  
    at java.base/java.io.PrintStream.format(PrintStream.java:1183)  
    at java.base/java.io.PrintStream.printf(PrintStream.java:1081)  
    at Sum.main(Sum.java:78)
```

Line 78 builds the format `%. -1f`, and `printf` throws

Without lines 73 to 75

```
73  if (count == 0) {  
74      fail("cannot average zero values");  
75  }
```



```
$ java Sum -a  
NaN  
$ echo $?  
0
```

Type nothing; end the input at once.

NaN, not a number: line 76 divides 0 by 0

Exit status 0: a script would take NaN for an average

The average of no numbers

- No single right answer: refuse, as `Sum` does, or print 0
- One answer is always wrong: ignore the problem
 - As the copy without lines 73 to 75 does
- **USS Yorktown**: a crew member entered a zero
 - A program divided by it, and the cruiser stopped for hours
- `Sum`'s manual should say which answer `Sum` gives

Brian Kernighan and Rob Pike, 1999.

Checks at the edge

- Each check stands between the user and the computation
 - The user owes `Sum` nothing
 - The computation should get only what it can handle
- Split `main` into methods with specifications, as in Lecture 2
 - The checks stay at the edge, in `main`
 - A method inside may then have a precondition
- Homework 1's `Gradebook` has the same shape

Coming up

- Tuesday: section, on assertions and specifications
- Wednesday and Friday: testing
- Homework 1, due Friday, October 9, 11:59pm:
 - Task 3: a user manual for [Histogram](#)
 - Task 4: checks at the edge, in [Gradebook's main](#)
- Hand in your half sheet at the door