

Error Handling and Interactive Programs

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- [Reading 2](#): due Friday, October 16, 11:59pm
- Homework 2: out by Friday, due Friday, October 23, 11:59pm
- Section tomorrow: testing

So far

- **Lecture 1:** a specification is a claim about every run
- **Lecture 2:** a method's specification is a contract with its client
- **Lecture 3:** a whole program's manual covers every input
 - Checks at the edge, where input arrives; preconditions inside
- **Lecture 4:** a test checks one chosen run against the specification
- **Lecture 5:** coverage says which test is next, and when you may stop
- **Today:** what a method does when it cannot do its job

A calculator

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a series of commands and their outputs in a monospaced font.

```
$ javac Calculator.java
$ java Calculator
> 1 2 + .
3
> q
```

- A **stack**: push puts a number on top; pop takes the top one off
- An **arithmetic operation** pops two numbers and pushes the answer
- The dot, `.`, pops the top number and prints it

A longer run

```
$ java Calculator
```

```
> 42
```

```
> 1 3 10 17
```

```
> p
```

```
[42 1 3 10 17]
```

```
> + p
```

```
[42 1 3 27]
```

```
> - p
```

```
[42 1 -24]
```

```
> .
```

```
-24
```

3 minus 27, not 27 minus 3: second from the top goes on the left

Interactive programs

- The stack is not the point: this program is interactive
- A **batch program** reads its input, computes, writes its output, and exits
- An **interactive program** responds to each input as it arrives
 - And keeps running until the user tells it to stop
- The calculator stops only on **q**, or at the end of the input

The loop in main

```
41  while (true) {
42      System.out.print("> ");
43      if (!in.hasNextLine()) {
44          break;
45      }
46      String line = in.nextLine();
47      String[] tokens = line.split("\\s+");
48      for (String token : tokens) {
49          if (token.equals("")) {
50              continue;
51          } else if (token.equals("q")) {
52              return;
```

A mistake does not end the run

```
$ java Calculator
> 1 +
error: need two numbers to add
> p
[1]
> x
error: don't understand 'x'
> 2 + p
[3]
> q
```

The failed + left the stack alone

Unlike Sum

- Lecture 3's `Sum` stopped at its first mistake: a message, exit status 1
- The calculator must not: the user wants to keep the stack
- So `main` must learn of every operation that could not be done
 - To tell the user, and go around the loop again

Half sheet, question 1 (4 minutes)

- 1. The user types $1 +$
 - Which method first decides that the operation cannot be done?
 - How does it say so, and who tells the user?
 - Now $1 *$: does it go the same way? Why might it not?

When a method cannot do its job

- A user's mistake: the user typed something the program cannot use
- A programmer's mistake: some code breaks a promise, a precondition
- A request that cannot be carried out
 - Pop an empty stack; Lecture 2's `indexOf` on a missing value
- The user's `1 +` reaches `add` as a request it cannot carry out
- Separation of concerns: `main` talks to the user, `CalcStack` computes
- Today's question: how does a method say no to its client?

Six ways to say no

- A precondition (*): `multiply` requires `2 <= size()`
- A special value (.): `pop` returns `null`
- A flag (+): `add` returns `false`
- A wrapper (t): `top` returns an empty `OptionalInt`
- An exception (/): `divide` throws one
- A checked exception (-): `subtract` throws one
- One per operation, on purpose; a real class picks one

Six failures, one run



```
$ java Calculator
> .
error: stack is empty
> t
error: stack is empty
> +
error: need two numbers to add
> -
error: need two numbers to subtract
⋮
```

Six failures, one run, continued

```
⋮  
> *  
error: need two numbers to multiply  
> /  
error: need two numbers to divide  
> 1 0 /  
error: cannot divide by zero  
> p  
[1 0]  
> q
```

- To the user, the six look alike; in the code, they differ

A precondition: multiply

```
176  /**
177   * @requires 2 <= size()
178   * @modifies this
179   * @effects pops the top two numbers and pushes their product
180   */
181  public void multiply() {
182      int b = pop(); int a = pop();
183      push(a * b);
184  }
```

- The client promised, so `multiply` checks nothing
- Or check anyway and throw, as Lecture 4 did: **defensive programming**

main checks first

```
82  } else if (token.equals("*")) {  
83      if (stack.size() < 2) {  
84          System.out.println(  
85              "error: need two numbers to multiply");  
86      } else {  
87          stack.multiply();  
88      }
```

- Lecture 3's rule: the check at the edge, in `main`; a precondition inside
- Lecture 2's reason: a user owes nothing; a client can keep a promise
- Suits a client that can check cheaply: here, one comparison

A special value: pop

```
125  /**
126   * @modifies this
127   * @effects removes the top number from the stack, if the stack is
128   *      non-empty
129   * @return the number removed, or null if the stack was empty
130   */
131  public Integer pop() {
132      if (items.size() == 0) { return null; }
133      int last = items.remove(items.size() - 1);
134      return last;
135  }
```

null says no number, as Lecture 2's -1 did; main checks on line 57

int and Integer

- A special value must be one the method never returns otherwise
 - `-1` for an empty stack's top? The top may be `-1`
- A primitive type, such as `int` or `boolean`, holds a plain value
- A reference type may hold `null`, which refers to nothing
- `Integer`: the `ints`, plus `null`
- Java converts between them by itself
 - `int` to `Integer` always works; `Integer` to `int` fails on `null`

A flag: add

```
146  /**
147   * @modifies this
148   * @effects if the stack has at least two numbers, pops the top
149   *         two and pushes their sum; otherwise leaves the stack
150   *         unchanged
151   * @return true if the top two numbers were replaced with their
152   *         sum, or false if the stack had fewer than two numbers
153   */
154  public boolean add() {
155      if (items.size() < 2) { return false; }
156      int b = pop(); int a = pop();
157      push(a + b);
158      return true;
159  }
```

A wrapper: top

```
137  /**
138   * @return an OptionalInt holding the top number of the stack, or
139   *         an empty OptionalInt if the stack is empty
140   */
141  public OptionalInt top() {
142      if (items.size() == 0) { return OptionalInt.empty(); }
143      return OptionalInt.of(items.get(items.size() - 1));
144  }
```

- A flag cannot carry an answer too: a method returns one value
- An `OptionalInt` holds one `int`, or is empty
 - A list of length zero or one

main asks first

```
62  } else if (token.equals("t")) {  
63      OptionalInt top = stack.top();  
64      if (top.isEmpty()) {  
65          System.out.println("error: stack is empty");  
66      } else {  
67          System.out.println(top.getAsInt());  
68      }
```

Line 64 asks isEmpty; only then does line 67 ask getAsInt

An exception: divide

```
196 public void divide() {
197     if (items.size() < 2) {
198         throw new TooFewOperandsException(
199             "need two numbers to divide");
200     }
201     int b = pop(); int a = pop();
202     if (b == 0) {
203         push(a); push(b);
204         throw new DivideByZeroException("cannot divide by zero");
205     }
206     push(a / b);
207 }
```

Catching divide's exceptions

```
89     } else if (token.equals("/")) {  
90         try {  
91             stack.divide();  
92         } catch (TooFewOperandsException e) {  
93             System.out.println(  
94                 "error: " + e.getMessage());  
95         } catch (DivideByZeroException e) {  
96             System.out.println(  
97                 "error: " + e.getMessage());  
98         }
```

try and catch

- `throw` ends the method at once
- Java looks for code that will **catch** it; with none, a stack trace
- **No exception:** the `try` block runs, and that is all
- **An exception named in a `catch`:** its block runs, not a crash
- Think of it as an `if` on "did the block throw?", with `catch` as the `else`

An exception too: subtract

```
161  /**
162   * @modifies this
163   * @effects if the stack has at least two numbers, pops the top
164   *          two (first and second) and pushes second - first; otherwise
165   *          does nothing
166   * @throws StackException if the stack had fewer than two numbers
167   */
168  public void subtract() throws StackException {
169      if (items.size() < 2) {
170          throw new StackException("need two numbers to subtract");
171      }
172      int b = pop(); int a = pop();
173      push(a - b);
174  }
```

When the client forgets

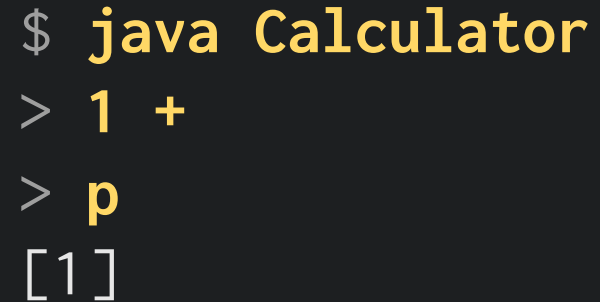
- All six work when the client checks
- **A second question:** what happens when the client forgets?
 - What would you want to happen?
- Copies of the calculator, each without one of `main`'s checks
 - Blank lines left behind, so every other line keeps its number

Forgetting the flag

```
69  } else if (token.equals("+")) {  
70      boolean ok = stack.add();  
71      if (!ok) {  
72          System.out.println(  
73              "error: need two numbers to add");  
74      }
```

Only the call on line 70 is left: main never looks at ok

Forgetting the flag, the run

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The text inside the terminal is as follows:

```
$ java Calculator  
> 1 +  
> p  
[1]
```

- `add` left the stack as it was: to the user, `+` added 0
- `main` ignored the flag, so nobody told the user

Forgetting the precondition

```
82  } else if (token.equals("*")) {  
83      if (stack.size() < 2) {  
84          System.out.println(  
85              "error: need two numbers to multiply");  
86      } else {  
87          stack.multiply();  
88      }
```

Only the call on line 87 is left: main calls multiply without checking

Forgetting the precondition, the run

```
$ java Calculator
```

```
> 1 *
```

```
Exception in thread "main" java.lang.NullPointerException:  
    Cannot invoke "java.lang.Integer.intValue()" because the return value of "CalcStack.pop()" is null  
at CalcStack.multiply(Calculator.java:182)  
at Calculator.main(Calculator.java:87)
```

- The second `pop` returned `null`, stored in the `int a`

Forgetting the special value

```
55  } else if (token.equals(".")) {  
56      Integer top = stack.pop();  
56      int top = stack.pop();  
57      if (top == null) {  
58          System.out.println("error: stack is empty");  
59      } else {  
60          System.out.println(top);  
61      }
```

No check, and pop's answer in an int: does it compile?

Forgetting the special value, the run

```
$ java Calculator
```

```
> .
```

```
Exception in thread "main" java.lang.NullPointerException:  
    Cannot invoke "java.lang.Integer.intValue()" because the return value of "CalcStack.pop()" is null  
    at Calculator.main(Calculator.java:56)
```

- `null` into an `int`: the same `intValue` crash as `multiply`'s, now in `main`
- With `Integer` kept, `.` prints `null` and the run goes on

Forgetting the exception

```
89  } else if (token.equals("/")) {  
90      try {  
91          stack.divide();  
92      } catch (TooFewOperandsException e) {  
93          System.out.println(  
94              "error: " + e.getMessage());  
95      } catch (DivideByZeroException e) {  
96          System.out.println(  
97              "error: " + e.getMessage());  
98      }
```

Only line 91 is left: no try around `stack.divide()`

Forgetting the exception, the run

```
$ java Calculator
```

```
> 1 /
```

```
Exception in thread "main" TooFewOperandsException:  
    need two numbers to divide  
    at CalcStack.divide(Calculator.java:198)  
    at Calculator.main(Calculator.java:91)
```

- A stack trace for the user, which Lecture 3's rule forbids
- But it gives `divide`'s own message, and the line that threw

Forgetting the wrapper

```
62  } else if (token.equals("t")) {  
63      OptionalInt top = stack.top();  
64      if (top.isEmpty()) {  
65          System.out.println("error: stack is empty");  
66      } else {  
67          System.out.println(top.getAsInt());  
68      }
```

Line 67 asks `getAsInt` with no `isEmpty` first: does it compile?

Forgetting the wrapper, the run

```
$ java Calculator
```

```
> t
```

```
Exception in thread "main" java.util.NoSuchElementException:
```

```
    No value present
```

```
    at java.base/java.util.OptionalInt.getAsInt(OptionalInt.java:130)
```

```
    at Calculator.main(Calculator.java:67)
```

- It compiles: a wrapper does not stop a forgotten check
- What it stops is using the answer as an `int`: next

Forgetting the wrapper, as an int

```
62 } else if (token.equals("t")) {  
63     OptionalInt top = stack.top();  
63     int top = stack.top();  
64     if (top.isEmpty()) {  
65         System.out.println("error: stack is empty");  
66     } else {  
67         System.out.println(top.getAsInt());  
67     System.out.println(top);  
68 }
```

No check, and top's answer in an int: does it compile?

Forgetting the wrapper, as an int, the run

```
$ javac Calculator.java
Calculator.java:63: error:
    incompatible types: OptionalInt cannot be converted to int
        int top = stack.top();
                        ^
1 error
```

- No: the client must ask with `getAsInt`, a reminder there may be none

What would you want to happen?

- Five forgotten checks: silence, a crash, `null` or a crash, a trace, a crash
- Only the wrapper in an `int` was caught before a run, by the compiler
- **To be told before the run at all**
- Java can do that for an exception, if the method's author asks

Two kinds of exception

```
222  class StackException extends Exception {  
223      public StackException(String message) {  
224          super(message);  
225      }  
226  }
```

```
228  class TooFewOperandsException extends RuntimeException {  
229      public TooFewOperandsException(String message) {  
230          super(message);  
231      }  
232  }
```

Checked and unchecked

- **Unchecked:** extends `RuntimeException`
 - Thrown with no `throws` clause; the client may catch it or not
- **Checked:** extends `Exception`, not `RuntimeException`
 - The method says so: `throws StackException` (line 168)
 - Every client catches it, or declares it too
- The compiler tracks the checked kind
- `FileNotFoundException`, which Lecture 3's `Sum` caught, is checked

Forgetting the checked exception

```
75  } else if (token.equals("-")) {  
76      try {  
77          stack.subtract();  
78      } catch (StackException e) {  
79          System.out.println(  
80              "error: " + e.getMessage());  
81      }
```

Only line 77 is left: no try around stack.subtract()

Forgetting the checked exception, the run

```
$ javac Calculator.java
Calculator.java:77: error:
    unreported exception StackException;
    must be caught or declared to be thrown
                stack.subtract();
                        ^
1 error
```

- Found before anyone ran the program
- Or `main` declares it too: next

Declaring it instead

```
38 public static void main(String[] args) {  
38 public static void main(String[] args) throws StackException {
```

```
75 } else if (token.equals("-")) {  
76     try {  
77         stack.subtract();  
78     } catch (StackException e) {  
79         System.out.println(  
80             "error: " + e.getMessage());  
81     }
```

No try, and main declares the exception too: does it compile?

Declaring it instead, the run

```
$ java Calculator
```

```
> 1 -
```

```
Exception in thread "main" StackException:  
    need two numbers to subtract  
    at CalcStack.subtract(Calculator.java:170)  
    at Calculator.main(Calculator.java:77)
```

- It compiles: `main` passed the exception on, as its clause says
- No method called `main`, so nothing catches it: a stack trace

The six, forgotten

- A flag: nothing happens, silently
- A precondition: all bets are off; here, a `NullPointerException`
- A special value: `.` prints `null`, or an `int` stops the program
- An exception: a stack trace with the method's own message
- A wrapper: a crash at the ask; as an `int`, it does not compile
- A checked exception: the compiler refuses the program

Let the compiler find the mistake

- The compiler is a **tool**: it checked one claim about every run
 - That `main` catches `StackException`, or declares it
- The author chose it: the compiler holds every client to the check
- A mistake it catches never reaches a run, a user, or a test

Half sheet, question 2 (5 minutes)

- 2. `CalcStack` says no six different ways, one per operation
 - A real class picks one way and uses it everywhere
 - Which way would you pick for `CalcStack`, and why?

What a checked exception costs

- Every client must catch it or declare it
 - Even one that knows the stack has two numbers
- And so must every method between `main` and `subtract`
- **The tension:** every client pays, in a `try` or a clause
 - In return, the compiler checks every client, before any run

Choosing a way

- Can the client check first, cheaply? Then a precondition
 - And the condition sits in front of whoever reads `main`
- How convenient is the answer? An `if`, a `try`, or a `try` everywhere
- What happens when a client forgets? Silence, a crash, or the compiler
- Promising to throw strengthens the specification: hard to take back
- A later lecture returns to exceptions in depth

A rule of thumb: which kind of exception?

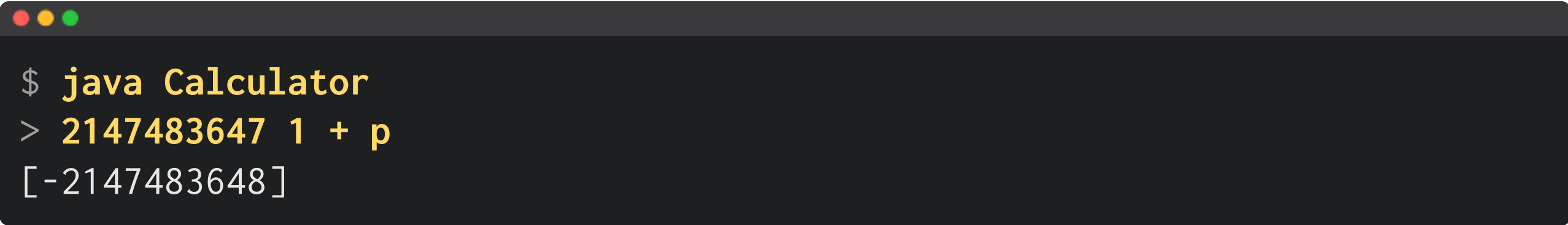
- **Bloch's rule:** who caused the problem, and who can recover?
- **The client can recover:** a checked exception (or a wrapper)
 - A file that is not there
- **Only a bug can cause it:** an unchecked exception (or a precondition)
 - A call that breaks a precondition

Joshua Bloch, *Effective Java*, 2nd ed. (2008), item 58.

A seventh way: make something up

- A bad one: say nothing, and make up an answer
 - `add` could treat a missing number as 0
- JavaScript: an array read past its end gives `undefined`
 - `undefined + 1` is `NaN`, and `NaN + 1` is `NaN`: it travels on
- Java's doubles: `NaN` is a special value; unchecked, it travels on too
- `null`, returned where the specification does not say so

The calculator makes one up, by omission

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The text inside the terminal is as follows:

```
$ java Calculator  
> 2147483647 1 + p  
[-2147483648]
```

\$ java Calculator
> 2147483647 1 + p
[-2147483648]

- The sum is past the largest `int`, so `a + b` wraps around
- `add` promised their sum: on this run it breaks its specification
- Nobody decided what `add` does then, so Java's arithmetic decided
- The fix is a decision: say no one of the six ways, or promise less

Interactive programs are harder

- You have now read one: what makes it harder to get right than `Sum`?
- **State that lasts:** a line can depend on every line before it
 - A bug may need one particular sequence of inputs
- **A user who chooses the order:** any input, at any moment
- **State the user cannot see:** the stack shows only on `p`

Testable by design

- `main` keeps the conversation: the prompt, the tokens, the messages
- `CalcStack` keeps the state and the arithmetic
- A test of `add` needs no keyboard
 - Push two numbers, call `add`, check the flag and the stack
- Lecture 5: one job per method, so the keyboard stays in `main`

Testing a @throws clause

```
17  @Test
18  void divideOnOneNumberThrows() {
19      CalcStack stack = new CalcStack();
20      stack.push(1);
21      assertThrows(TooFewOperandsException.class,
22                  () -> stack.divide());
23      assertEquals("[1]", stack.toString());
24  }
```

Your job is to deliver code that you know works.

Coming up

- **Tomorrow:** section, on testing
- **Wednesday:** reasoning, and Floyd-Hoare logic
- **Reading 2:** due Friday, October 16, 11:59pm
- Homework 2: out by Friday, due Friday, October 23, 11:59pm
- Hand in your half sheet at the door