

Principles of Software Construction: Objects, Design, and Concurrency

Specifications and unit testing, exceptions

Claire Le Goues **Bogdan Vasilescu**



Remember this
discussion from
last week?

Encapsulation / Information hiding

- Well designed objects project internals from others
 - both internal state and implementation details
- Well-designed code hides all implementation details
 - Cleanly separates interface from implementation
 - Modules communicate only through interfaces
 - They are oblivious to each others' inner workings
- Hidden details can be changed without changing client!
- Fundamental tenet of software design

Who's to blame?

```
Algorithms.shortestDistance(g, "Tom", "Anne");
```

```
> ArrayOutOfBoundsException
```

Who's to blame?

```
Algorithms.shortestDistance(g, "Tom", "Anne");
```

```
> -1
```

Who's to blame?

```
Algorithms.shortestDistance(g, "Tom", "Anne");
```

```
> 0
```

Who's to blame?

```
class Algorithms {  
    /**  
     * This method finds the  
     * shortest distance between two  
     * vertices. It returns -1 if  
     * the two nodes are not  
     * connected. */  
    int shortestDistance(...) {...}  
}
```

Who's to blame?

```
class Algorithms {  
    /**  
     * This method finds the  
     * shortest distance between two  
     * vertices. It returns -1 if  
     * the two nodes are not  
     * connected. */  
    int shortestDistance(...) {...}  
}
```

Think of this (textual)
specification as a “contract”

What is a contract?

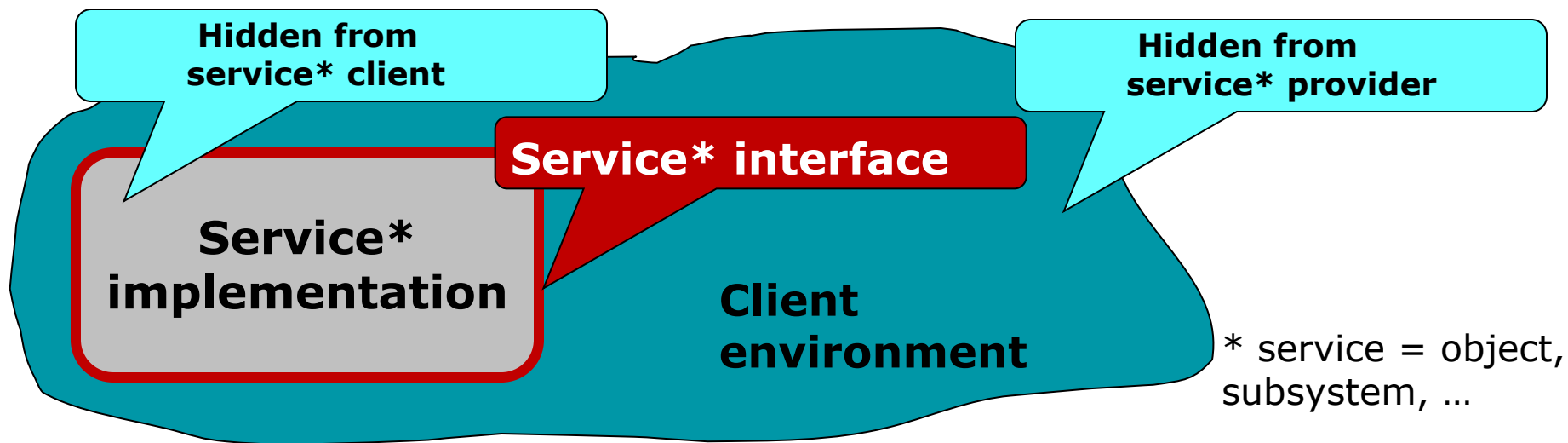
- Agreement between an object and its user
 - What object provides, and user can count on
- Includes:
 - Method signature (type specifications)
 - Functionality and correctness expectations
 - Sometimes: performance expectations
- **What** the method does, not **how** it does it
 - **Interface** (API), not **implementation**

Method contract details

- Defines method's and caller's responsibilities
- Analogy: legal contract
 - If you pay me this amount on this schedule...
 - I will build a room with the following detailed spec
 - Some contracts have remedies for nonperformance
- Method contract structure
 - Preconditions: what method requires for correct operation
 - Postconditions: what method establishes on completion
 - Exceptional behavior: what it does if precondition violated
- Defines correctness of implementation – we'll come back to this later today

Most real-world code has a contract

- Imperative to build systems that scale!
- This is why we:
 - Encode specifications
 - Test



Today

1. Exception Handling
2. Unit Testing
3. Specifications

Exceptions

What does this code do?

This is Java code

```
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
    switch (errno) {
        case _ENOFIL:
            System.err.println("File not found: " + ...);
            return -1;
        default:
            System.err.println("Something else bad happened: " + ...);
            return -1;
    }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
    System.err.println("Unknown internal error.");
    return -1; // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
} // The Slide lacks space to close the file. Oh well.
return i;
```

Compare to:

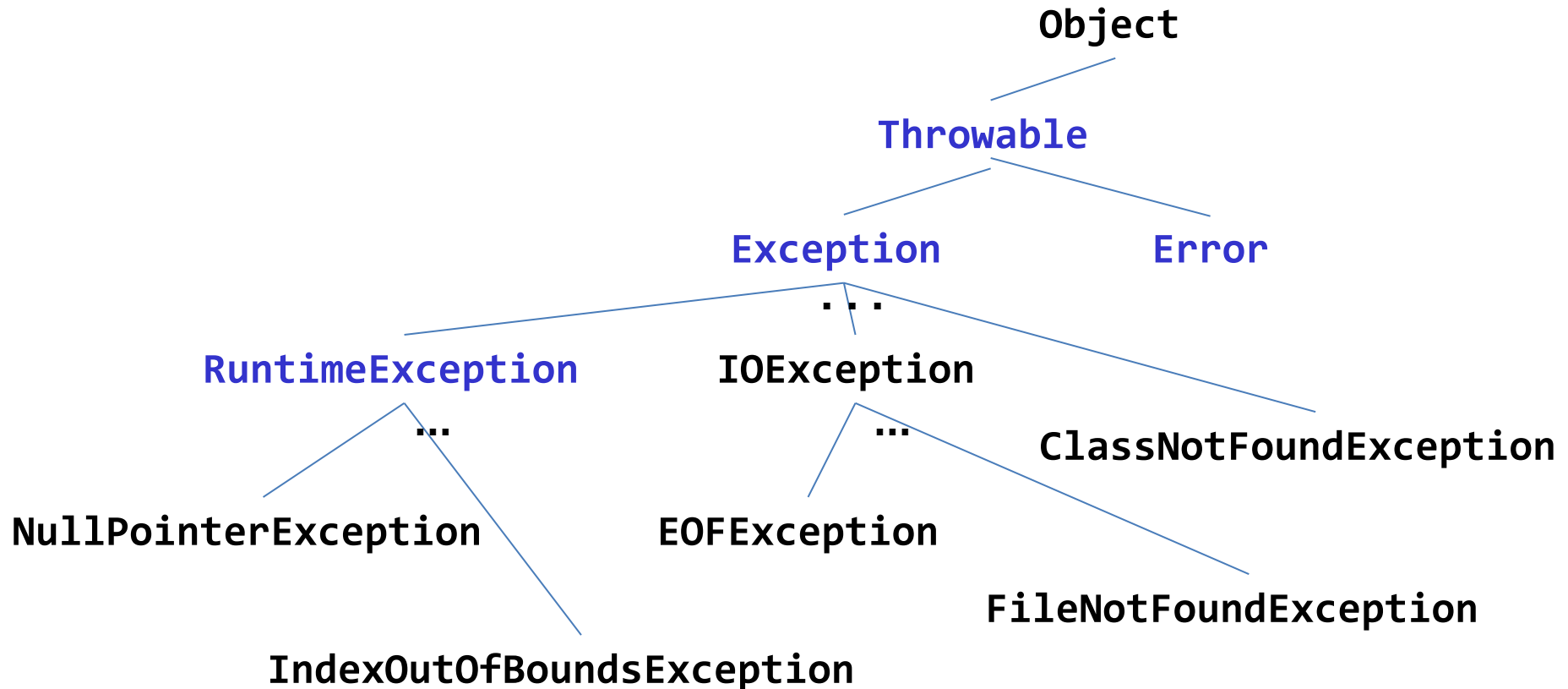
This is Java code

```
FileInputStream fileInput = null;
try {
    fileInput = new FileInputStream(fileName);
    DataInput dataInput = new
DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " +
fileName);
} catch (IOException e) {
    System.out.println("Couldn't read file: " + e);
} finally {
    if (fileInput != null) fileInput.close();
}
```

Exceptions

- Split control-flow into a “normal” and an “erroneous” branch
 - Compare “if/else”
- Inform caller of problem by transfer of control
- Where do exceptions come from?
 - Program can throw explicitly using throw
 - Underlying virtual machine (JVM) can generate
- Semantics
 - Propagates up call stack until exception is caught, or main method is reached (terminates program!)


The exception hierarchy in Java (messy)



Control-flow of exceptions

This is Java code

```
public static void test() {  
    try {  
        System.out.println("Top");  
        int[] a = new int[10];  
        a[42] = 42;  
        System.out.println("Bottom");  
    } catch (NegativeArraySizeException e) {  
        System.out.println("Caught negative array size");  
    }  
}  
  
public static void main(String[] args) {  
    try {  
        test();  
    } catch (IndexOutOfBoundsException e) {  
        System.out.println("Caught index out of bounds");  
    }  
}
```



Control-flow of exceptions

This is Java code

```
public static void test() {  
    try {  
        System.out.println("Top");  
        int[] a = new int[10];  
        a[42] = 42;  
        System.out.println("Bottom");  
    } catch (NegativeArraySizeException e) {  
        System.out.println("Caught negative array size");  
    }  
}  
  
public static void main(String[] args) {  
    try {  
        test();  
    } catch (IndexOutOfBoundsException e) {  
        System.out.println("Caught index out of bounds");  
    }  
}
```

Handle errors at a level you choose, not necessarily in the low-level methods where they originally occur.

Exception Handling

Undeclared

```
int divide(int a, int b) {  
    return a / b;  
}
```

vs.

Declared

```
String read(String path) throws  
    IOException {  
    return Files.lines(Path.of(path))  
        .collect(Collectors.joining("\n"));  
}
```

Exception Handling

Undeclared

```
int divide(int a, int b) {  
    return a / b;  
}
```

vs.

Declared

```
String read(String path) throws  
    IOException {  
    return Files.lines(Path.of(path))  
        .collect(Collectors.joining("\n"));  
}
```

Unchecked

```
divide(4, 3); // Compiles  
              fine
```

vs.

Checked

```
read("test.txt"); // Unhandled  
exception: java.io.IOException
```

Exception Handling

Handling unchecked exceptions is not enforced by the compiler

These are quite common

- E.g., all exceptions in C++
- In Java: any exception that extends Error or RuntimeException

Exception Handling

Handling unchecked exceptions is not enforced by the compiler

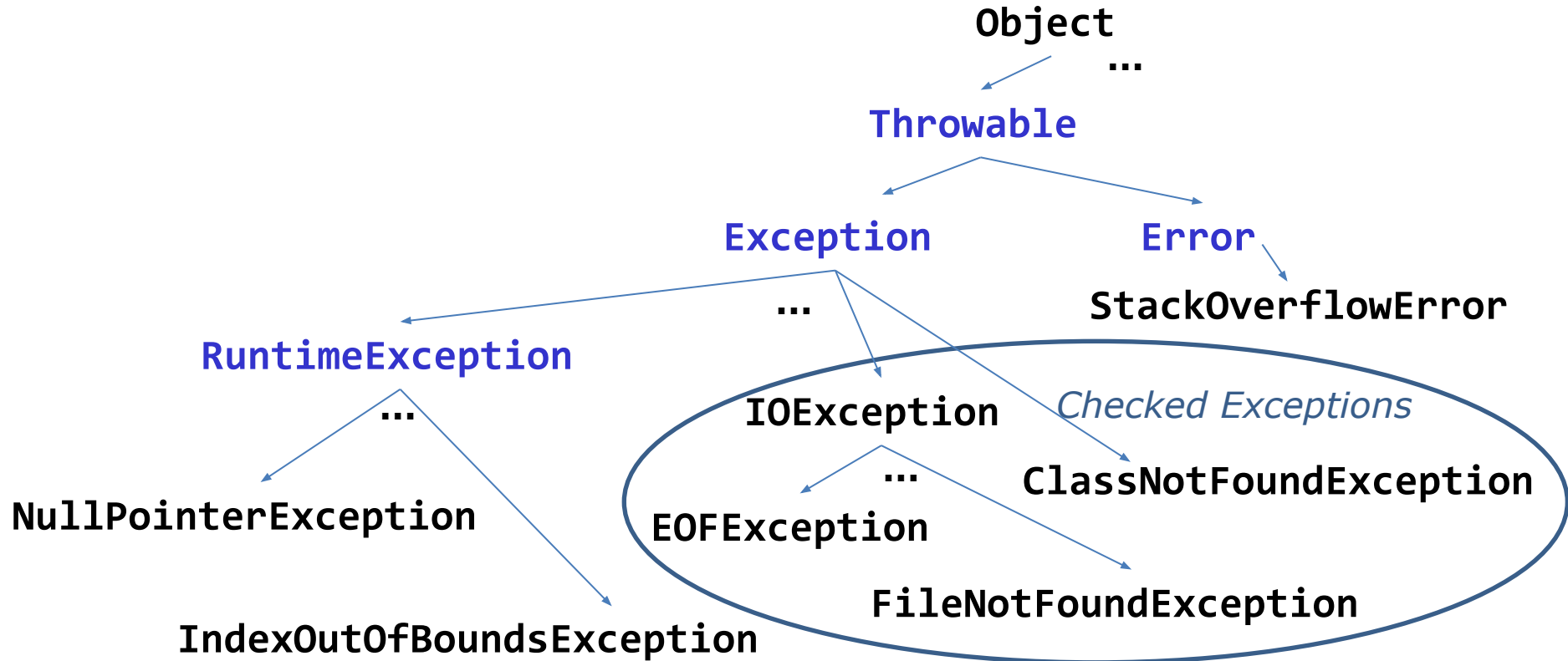
These are quite common

- E.g., all exceptions in C++
- In Java: any exception that extends Error or RuntimeException
 - E.g.:

```
int divide(int a, int b) throws ArithmeticException {  
    return a / b;  
}  
divide(4, 3); // Compiles fine
```

- **Note:** we don't typically declare unchecked exceptions.

Java's exception hierarchy (messy)



Checked vs. unchecked exceptions

- **Checked exception**

- Must be caught or propagated, or program won't compile
- **Exceptional condition that programmer must deal with**

- **Unchecked exception**

- No action is required for program to compile...
 - But uncaught exception will cause failure at runtime
- Usually indicates a **programming error**

- **Error**

- Special unchecked exception typically thrown by VM
- Recovery is usually impossible

Benefits of exceptions (summary)

- You can't forget to handle common failure modes
 - Explicit > implicit
 - Compare: using a flag or special return value
- Provide high-level summary of error
 - Compare: core dump in C/C++
- Improve code structure
 - Separate normal code path from exceptional
 - Error handling code is segregated in catch blocks
- Ease task of writing robust, maintainable code

Defining & using Exception Types

```
class BufferBoundsException extends Throwable {  
    public BufferBoundsException(String message) {  
        ...  
    }  
}  
  
void atIndex(int[] buff, int i) throws BufferBoundsException {  
    if (buff.length <= i)  
        throw new BufferBoundsException("...");  
    return buff[i];  
}
```

Exception Handling

- It's still wise to guard for “obvious” unchecked exceptions

```
if (arr.length > 10)  
    return arr[10];
```

- Or explicitly signal the problem, recall:

```
if (buff.length <= i)  
    throw new BufferBoundsException("...");  
return buff[i];
```

- Why is this better than letting the index fail?

Exception Handling

- It's still wise to guard for “obvious” unchecked exceptions

```
if (arr.length > 10)
    return arr[10];
```

- Or explicitly signal the problem, recall:

```
if (buff.length <= i)
    throw new BufferBoundsException("...");
return buff[i];
```

- Why is this better than letting the index fail?
 - BufferBoundsException can be a checked exception!
 - Which forces someone to handle it
 - Here, we declared: `atIndex(int[] buff, int i) throws BufferBoundsException`
 - So every calling method must handle it, or throw it on

Guidelines for using exceptions (1)

- Avoid unnecessary checked exceptions (EJ Item 71)
- Favor standard exceptions (EJ Item 72)
 - `IllegalArgumentException` – invalid parameter value
 - `IllegalStateException` – invalid object state
 - `NullPointerException` – null param where prohibited
 - `IndexOutOfBoundsException` – invalid index param
 - `IOException` -- and its subclasses, mostly for File-related actions
- Throw exceptions appropriate to abstraction (EJ Item 73)

Guidelines for using exceptions

- Document all exceptions thrown by each method in the specification
 - Unchecked as well as checked (EJ Item 74)
 - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IllegalArgumentException(  
    "Quantity must be positive: " + quantity);
```

Guidelines for using exceptions (2)

- Document all exceptions thrown by each method
 - Unchecked as well as checked (EJ Item 74)
 - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IllegalArgumentException(  
    "Quantity must be positive: " + quantity);
```

- Don't ignore exceptions (EJ Item 77)

```
try {  
    processPayment(payment);  
}  
catch (Exception e) { // BAD!  
}
```

Cleanup

Exception handling often also supports cleaning up

```
openMyFile();  
try {  
    writeMyFile(theData); // This may throw an error  
} catch(e) {  
    handleError(e); // If an error occurred, handle it  
} finally {  
    closeMyFile(); // Always close the resource  
}
```


Manual Resource Termination

Is ugly and error-prone, especially for multiple resources

- Even good programmers usually get it wrong
 - Sun's Guide to Persistent Connections got it wrong in code that claimed to be exemplary
 - Solution on page 88 of Bloch and Gafter's Java Puzzlers is badly broken; no one noticed for years
- 70% of the uses of `close` **in the JDK itself** were wrong in 2008!
- Even the “correct” idioms for manual resource management are deficient

The solution: try-with-resources

Automatically closes resources!

```
try (DataInputStream dataInput =  
    new DataInputStream(new FileInputStream(fileName))) {  
    return dataInput.readInt();  
} catch (IOException e) {  
    ...  
}
```

Exceptions Across Languages

Alas, try-with-resources does not exist in JS/TS

- Neither does 'throws'

Exception structures differ radically across languages

- Most languages have 'try/catch' and 'throw'
 - Some have 'finally'
- Python has 'with' for resource management (since 2006)
 - C# has 'using'
 - Java's try-with-resources was added in 2011
- Go returns an error-typed value, to be checked for nullity

Exceptions Across Languages

Use what you have

- When possible, be explicit
 - Use the compiler to enforce, where possible
 - Proactively avoid corner-cases, where not
 - Unchecked exceptions, JS/TS
- Make exceptions part of your contract

Outline

1. Exception Handling
2. **Unit Testing**
3. Specifications

Functional correctness

- Compiler ensures **types** are correct (**type-checking**)
 - Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”

Functional correctness

- Compiler ensures **types** are correct (**type-checking**)
 - Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”
- How to ensure functional correctness, beyond type correctness?

One option: Formal verification

- Use mathematical methods to prove correctness with respect to the formal specification
- Formally prove that **all possible executions** of an implementation **fulfill the specification**
- Manual effort; partial automation; not automatically decidable

Another option: Testing

- Executing the program with selected inputs in a controlled environment
- Goals
 - Reveal bugs, so they can be fixed (main goal)
 - Assess quality
 - Clarify the specification, documentation
- Testing is related to contracts
 - Because we need to know what to test!

Re: Formal verification, Testing

**“Beware of bugs in the above code; I
have only proved it correct, not tried it.”**

Donald Knuth, 1977

**"Testing shows the presence, not the
absence of bugs."**

Edsger W. Dijkstra, 1969

Q: Who's right, Dijkstra or Knuth?

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1); // key not found.
17:     }
```

This is Java code

Q: Who's right, Dijkstra or Knuth?

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1); // key not found.
17:     }
```

Spec: sets mid to the average of low and high, truncated down to the nearest integer.

Fails if $\text{low} + \text{high} > \text{MAXINT} (2^{31} - 1)$
Sum overflows to negative value

A: They're both right

- There is no silver bullet!
- Use all the tools at your disposal
 - Careful design
 - Testing
 - Formal methods (where appropriate)
 - Code reviews
 - ...
- You'll still have bugs, but hopefully fewer.

Manual testing

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select “Create new Mes- sage”	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select “Insert Picture”	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select “Send Message”	Message is correctly sent



Automated testing

- Execute a program with specific inputs, check output for expected values
- Easier to test small pieces than testing user interactions
- Set up testing infrastructure
- **Execute tests regularly**
 - After every change

Testing

How do we know
this works?

```
int isPos(int x) {  
    return x >= 1;  
}
```


Testing

How do we know
this works?

Testing

```
int isPos(int x) {  
    return x >= 1;  
}  
  
@Test  
void testIsPos() {  
    assertTrue(isPos(1));  
}
```

Are we done?

Testing

How do we know
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 1;  
}  
  
@Test  
void testIsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void testNotPos() {  
    assertFalse(isPos(-1));  
}
```

Testing

How do we know
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 0; // What if?  
}  
  
@Test  
void testIsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void testNotPos() {  
    assertFalse(isPos(-1));  
}
```

Testing

How do we know
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 0; // What if?  
}  
  
@Test  
void test1IsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void test0IsNotPos() {  
    assertFalse(isPos(0)); // Fails  
}
```

Boundary Value Testing

We cannot test for every integer.

Choose *representative* values:
1 for positives, -1 for negatives

And *boundary cases*: 0 is a likely
candidate for mistakes

- Think like an attacker

```
int isPos(int x) {  
    return x >= 0; // What if?  
}  
  
@Test  
void test1IsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void test0IsNotPos() {  
    assertFalse(isPos(0)); // Fails  
}
```

Testing

How do we know a program is correct?

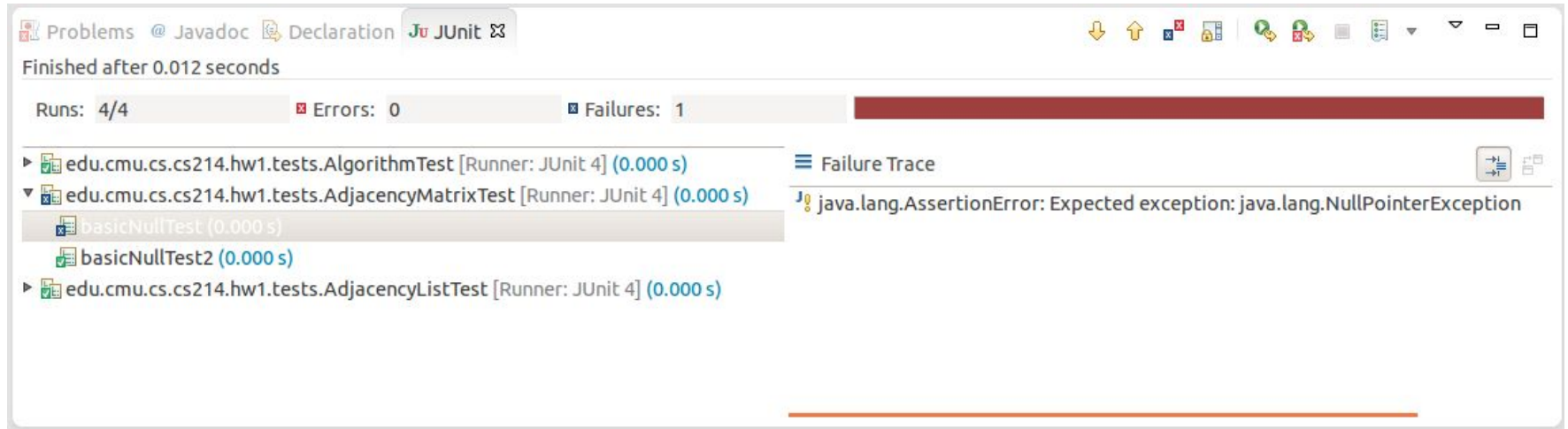
- In a perfect world (maybe): formal verification
 - Easy enough for proving that $\text{isPos}(x)$ -- the implementation is the definition
 - Tedious, cannot be done automatically
- Hence, testing

Unit Tests

- For “small” units: methods, classes, subsystems
 - Unit is smallest testable part of system
 - Test the parts before assembling them
 - Intended to catch local bugs
- Typically (but not always) written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment
- Insufficient, but a good starting point

For Java: JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available, e.g., IntelliJ integration



For Java: JUnit

Syntax:

```
import static org.junit.Assert.*;

class PosTests {

    @Before
    void setUp() {
        // Anything you want to run
        // before each test
    }

    @Test
    void test1IsPos() {
        assertTrue(isPos(1));
    }
}
```

For TS: Jest

- In particular, ts-jest
 - Many other options; your choice
- Requires a few files:
 - jest.config.js, to specify testing mode
 - package.json with (ts-)jest dependencies
- Provides useful features:
 - 'test', 'expect' (= 'assert')
 - 'toBe', 'toEqual'
 - 'fn', for Mocking (later)

```
test > TS isPos.test.ts > ...
1  import { isPos } from "../src/isPos"
2
3  test('1 is positive', () => {
4    expect(isPos(1)).toBe(true);
5  });
6
7  test('-1 is not positive', () => {
8    expect(isPos(-1)).toBe(false);
9  });
10
11 test('0 is not positive', () => {
12   expect(isPos(0)).toBe(false);
13 });
```

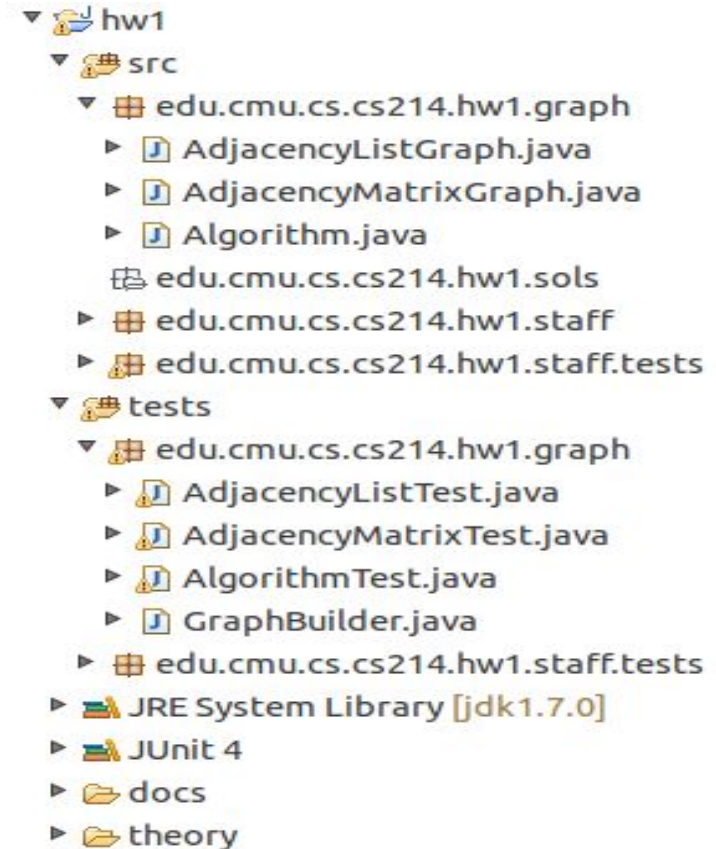
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

at Object.<anonymous> (test/isPos.test.ts:12:19)

Test Suites: 1 failed, 1 total
Tests: 1 failed, 2 passed, 3 total
Snapshots: 0 total

Test organization

- Conventions (not requirements)
- Have a test class FooTest for each public class Foo
- Have a source directory and a test directory
 - Store FooTest and Foo in the same package
 - Tests can access members with default (package) visibility



Writing Testable Code

- Think about testing when writing code
 - Unit testing encourages you to write testable code
- Modularity and testability go hand in hand
 - Same test can be used on multiple implementations of an interface!
- Test-Driven Development
 - A design and development method in which you write tests before you write the code
 - Writing tests can expose API weaknesses!

Run Tests Often

- You should only commit code that passes all tests...
- So run tests before every commit
- If test suite becomes too large & slow for rapid feedback
 - Run local package-level tests (“smoke tests”) frequently
 - Run all tests nightly
 - Medium sized projects often have thousands of test cases
- Continuous integration (CI) servers help to scale testing
 - We ask you to use GitHub Actions in this class

Reflections on Testing

“Testing shows the presence, not the absence of bugs.”

Edsger W. Dijkstra, 1969

“Functionality that can’t be demonstrated by automated test simply don't exist.”

Kent Beck

Outline

1. Exception Handling
2. Unit Testing
3. **Specifications – to be continued on Tuesday**