Principles of Software Construction: Objects, Design, and Concurrency

Testing and `Object` Methods in Java

Josh Bloch Charlie Garrod

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due **Today** 11:59 p.m.
  - Everyone must read and sign our collaboration policy
  - TAs will be available to help you
  - You have late days, buy you might want to save for later
- Second homework will be posted shortly

Key concepts from Tuesday

- Collections are your friend
- Interfaces-based designs are flexible
- Information hiding is crucial to good design
- Enums are also your friend

Outline

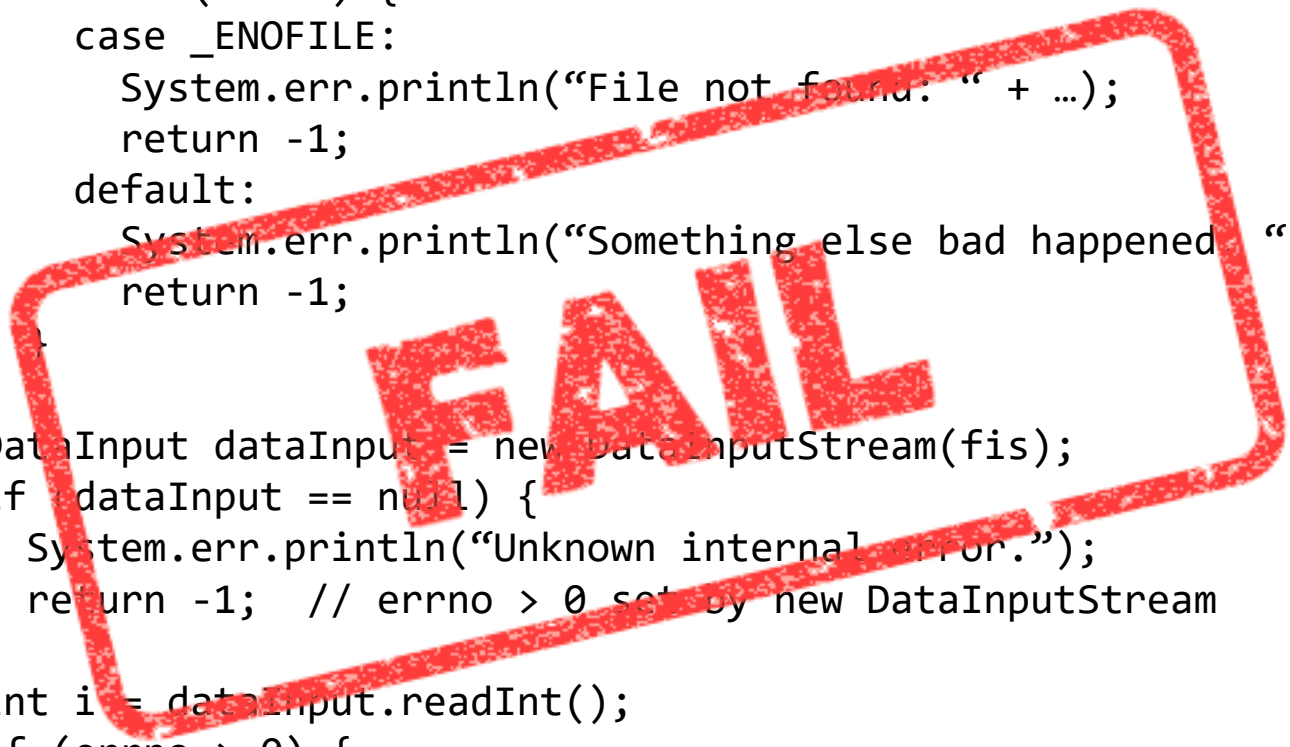
I. Exceptions
II. Specifying program behavior – contracts
III. Testing correctness – Junit and friends
IV. Overriding `Object` methods

What does this code do?

```
FileInputStream fis = new FileInputStream(fileName);
if (fis == null) {
  switch (errno) {
    case _ENOFILE:
      System.err.println("File not found: " + …);
      return -1;
    default:
      System.err.println("Something else bad happened: " + …);
      return -1;
  }
}
DataInput dataInput = new DataInputStream(fis);
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;
```

What does this code do?

```
FileInputStream fis = new FileInputStream(fileName);
if (fis == null) {
  switch (errno) {
    case _ENOFILE:
      System.err.println("File not found: " + …);
      return -1;
    default:
      System.err.println("Something else bad happened " + …);
      return -1;
  }
}
DataInput dataInput = new DataInputStream(fis);
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;
```



There's a better way: *exceptions*

```
FileInputStream fileInput = null;

try {
    fileInput = new FileInputStream(fileName);
    DataInputStream dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (IOException e) {
    System.err.println("Could not read int from file: " + e);
    return DEFAULT_VALUE;
}
```

Exceptions

- Inform caller of problem by transfer of control
- Semantics
  - Propagates up call stack until exception is caught, or `main` method is reached (terminates program!)
- Where do exceptions come from?
  - Program can throw explicitly using `throw`
  - Underlying virtual machine (JVM) can generate

Control-flow of exceptions

```
public static void main(String[] args) {
    try {
        test();
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println"("Caught index out of bounds exception: " + e);
    }
    System.out.println(“Done");
}

public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42; // Index is too high; throws exception
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size exception: " + e);
    }
}
```

Benefits of exceptions

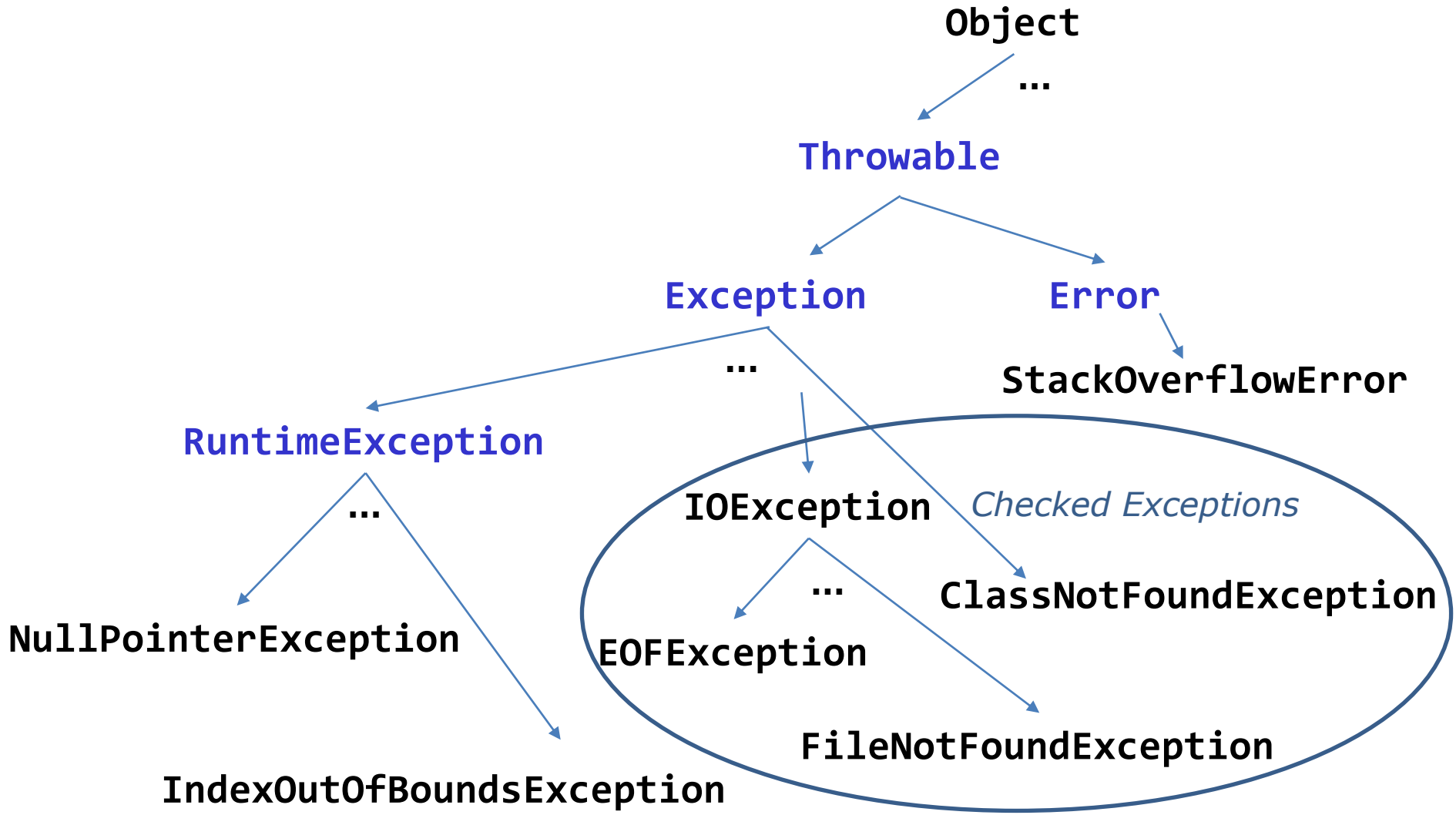
- You can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Provide high-level summary of error
  - Compare: core dump in C/C++
- Improve code structure
  - Separate normal code path from exceptional
  - Error handling code is segregated in catch blocks
- Ease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

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

Java's exception hierarchy

is a big, stinking mess



Design choice: checked vs. unchecked

- Unchecked exception
  - Programming error, other unrecoverable failure
- Checked exception
  - An error that every caller should be aware of and handle
- Special return value (e.g., `null` from `Map.get`)
  - Common but atypical result (not erroneous!)
- **Do not use error codes** – too easy to ignore
- **Avoid null return values**
  - Never return null instead of zero-length list or array

Defining & using your own exception types

```
class SpanishInquisitionException extends RuntimeException {
    SpanishInquisitionException(String detail) {
        super(detail);
    }
}

public class HolyGrail {
    public void seek() {
        ...
        if (heresyByWord() || heresyByDeed())
            throw new SpanishInquisitionException("heresy");
        ...
    }
}
```

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
- Throw exceptions appropriate to abstraction (EJ Item 73)

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 IlegalArgumentException("Quantity must be positive: " + quantity);`
- Don't ignore exceptions (EJ Item 77)

```
// Empty catch block IGNORES exception – Bad smell in code!
try {
    ...
} catch (SomeException e) {
}
```

Remember this from beginning of lecture?

```
FileInputStream fileInput = null;

try {
    fileInput = new FileInputStream(fileName);
    DataInputStream dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (IOException e) {
    System.err.println("Could not read int from file: " + e);
    return DEFAULT_VALUE;
}
```

There's one part we didn't show you: cleanup

```
FileInputStream fileInput = null;

try {
    fileInput = new FileInputStream(fileName);
    DataInputStream dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (IOException e) {
    System.err.println("Could not read int from file: " + e);
    return DEFAULT_VALUE;
} finally { // Close file if it's open
    if (fileInput != null) {
        try {
            fileInput.close();
        } catch (IOException ignored) {
            // No recovery necessary (or possible)
        }
    }
}
```

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) {
     System.err.println("Could not read file: " + e);
     return DEFAULT_VALUE;
}
```

File copy with manual cleanup

```
static void copy(String src, String dest) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dest);
        try {
            byte[] buf = new byte[8 * 1024];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
            } finally {
                if (out != null) out.close();
            }
        } finally {
            if (in != null) in.close();
        }
    }
}
```

File copy with `try`-with-resources

```
static void copy(String src, String dest) throws IOException {
    try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dest)) {
        byte[] buf = new byte[8 * 1024];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

Outline

I. Exceptions
II. Specifying program behavior – contracts
III. Testing correctness – Junit and friends
IV. Overriding `Object` methods

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

Formal contract specification

Java Modelling Language (JML)

```
/*@ requires array != null;
  @
  @ ensures \result ==
  @     (\sum int j;  0 <= j && j < array.length;  array[j]);
  @*/
int total(int array[]);
```

- Theoretical approach
  - Advantages
    - Runtime checks generated automatically
    - Basis for formal verification
    - Automatic analysis tools
  - Disadvantages
    - Requires a lot of work
    - Impractical in the large
    - Some aspects of behavior not amenable to formal specification

Prose specification - Javadoc

- Practical approach
- Document
  - Every parameter
  - Return value
  - Every exception (checked and unchecked)
  - What the method does, including
    - Primary purpose
    - Any side effects
    - Any thread safety issues
    - Any performance issues
- Do **not** document implementation details
  - Known as *overspecification*

Specifications in the real world

Javadoc for `List`*'s* `get` *method*

```
/**
 * Returns the element at the specified position of this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations, it may run in time proportional
 * to the element position.
 *
 * @param index position of element to return;
 *   must be non-negative and less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

postcondition

(Performance)

precondition

exceptional behavior

(No side effects)

Outline

I. Exceptions
II. Specifying program behavior – contracts
III. Testing correctness – Junit and friends
IV. Overriding `Object` methods

Semantic correctness

*Adherence to **contracts***

- Compiler ensures types are correct (type-checking)
  - Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”
- Static analysis tools (e.g., SpotBugs) recognize many common problems (*bug patterns*)
- But how do you ensure semantic correctness?

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
- Requires manual effort. Can be partially automated, but not automatically decidable

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

Edsger W. Dijkstra, 1969

Testing

- Execute the program with selected inputs in a controlled environment
- Goals
  - Reveal bugs, so they can be fixed (primary goal)
  - Clarify the specification, documentation

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

Donald Knuth, 1977

Who's right, Dijkstra or Knuth?

- They're both right!
- Please see "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken"
  - Official "Google Research" blog
  - http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
- Conclusion: **There is no silver bullet**
  - Use all tools at your disposal

Manual testing?

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

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 Message"	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

- Live system?
- Lots of hardware?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

Automate testing

- Automatically execute program with specific inputs, and check output for expected values
- Set up testing infrastructure
- Execute tests regularly
  - After *every* change, before (and after) it's pushed

Unit tests

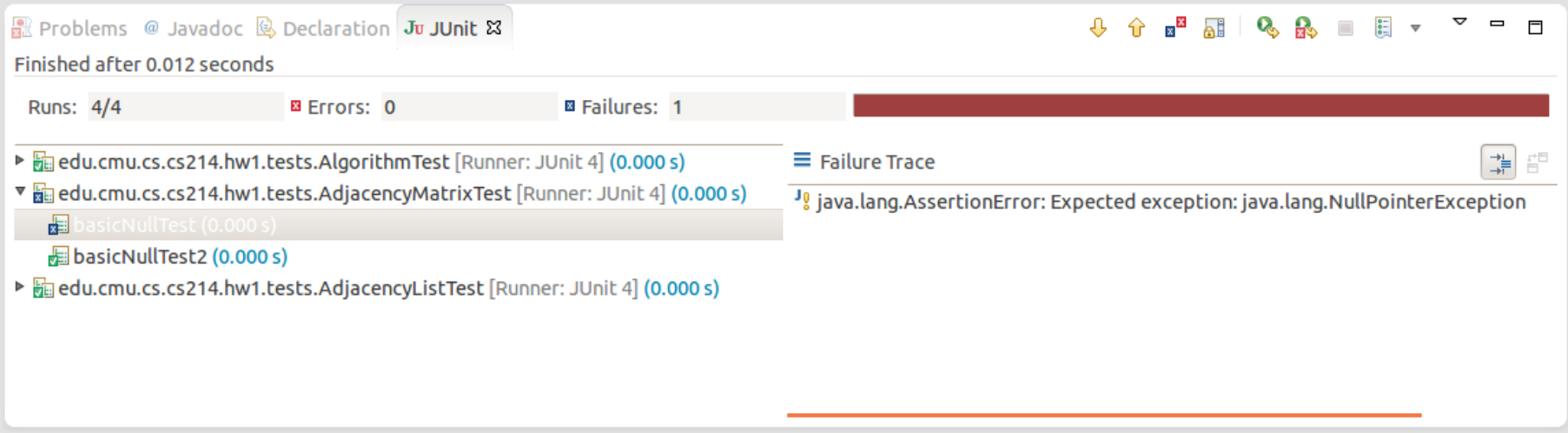
- For “small” units: methods, classes, subsystems
  - Unit is smallest testable part of system
  - **Test parts before assembling them**
  - Intended to catch local bugs
- Typically (but not always) written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system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

Selecting test cases: common strategies

- Read specification
- Write tests for
  - Representative case
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker – read spec looking for “loopholes”
  - The tester’s goal, like the hacker’s, is to find bugs!
- How many test should you write?
  - Aim to cover the entire specification
  - But work within time/money constraints

JUnit

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



Kent Beck on automated testing

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

JUnit conventions

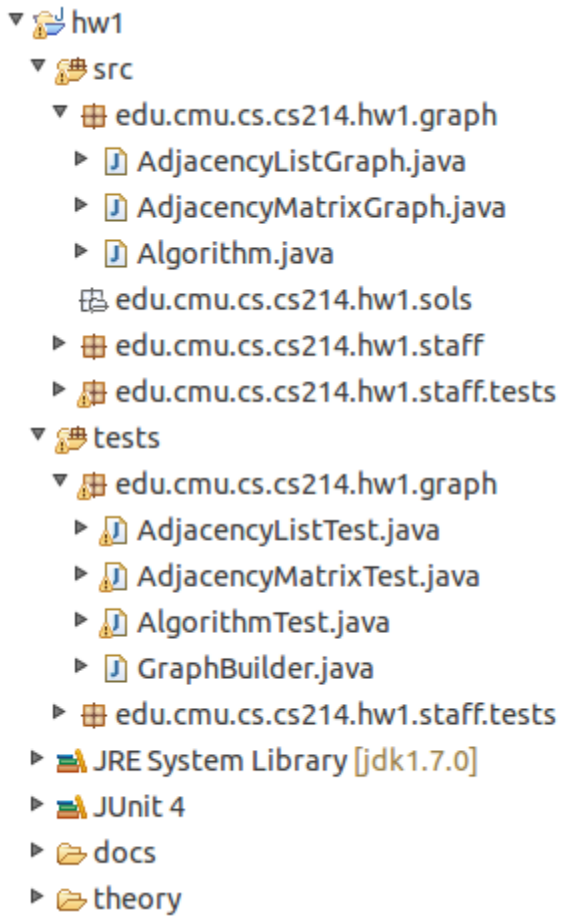
- `TestCase` collects multiple tests (in one class)
- `TestSuite` collects test cases (typically package)
- Tests should run fast
- Tests should be independent

- Tests are methods without parameters or return values
- `AssertError` signals failed test (unchecked exception)

- Test `Runner` knows how to run JUnit tests
  - (uses reflection to find all methods with `@Test` annotation)

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



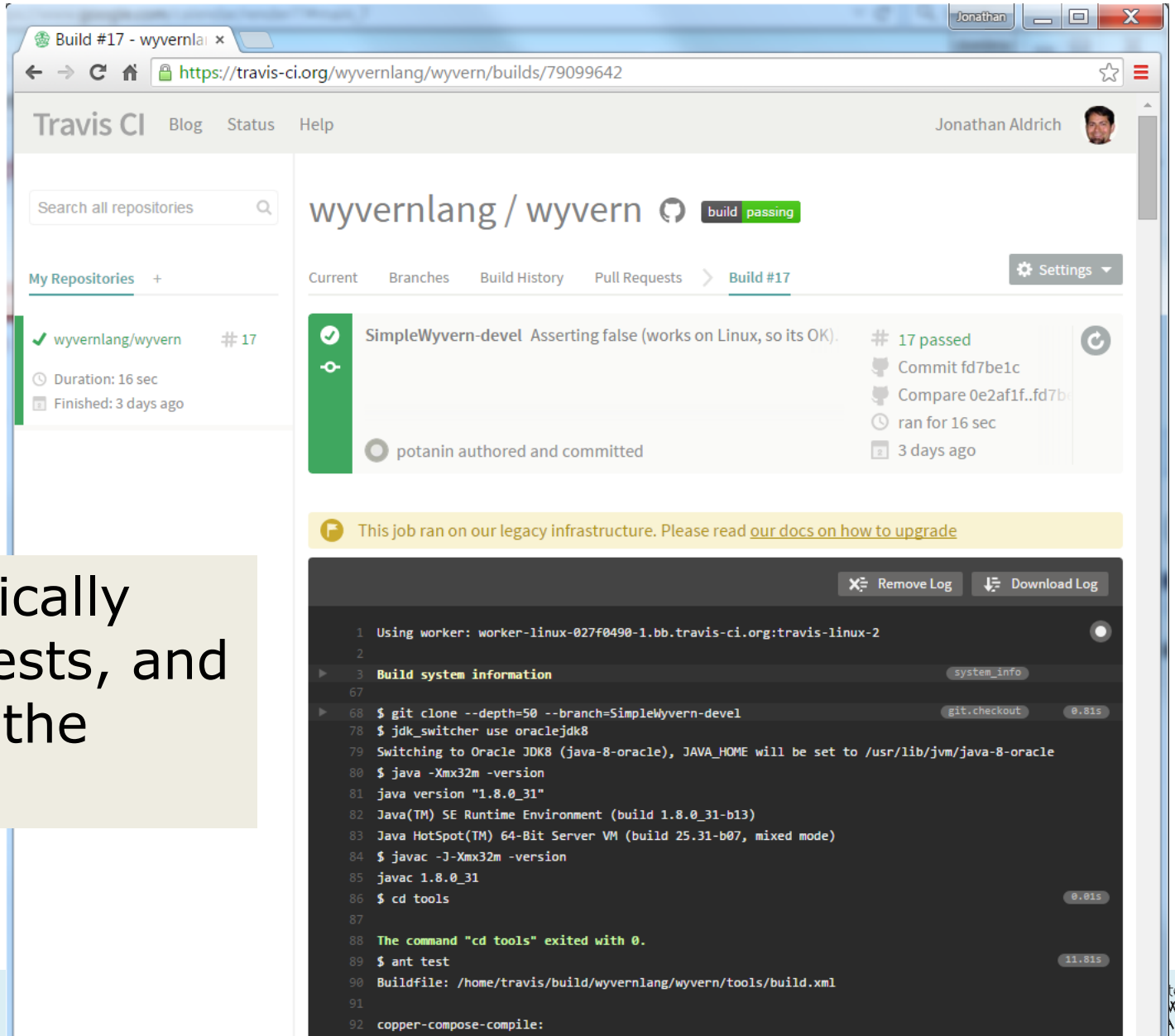
Write testable code

- **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 frequently

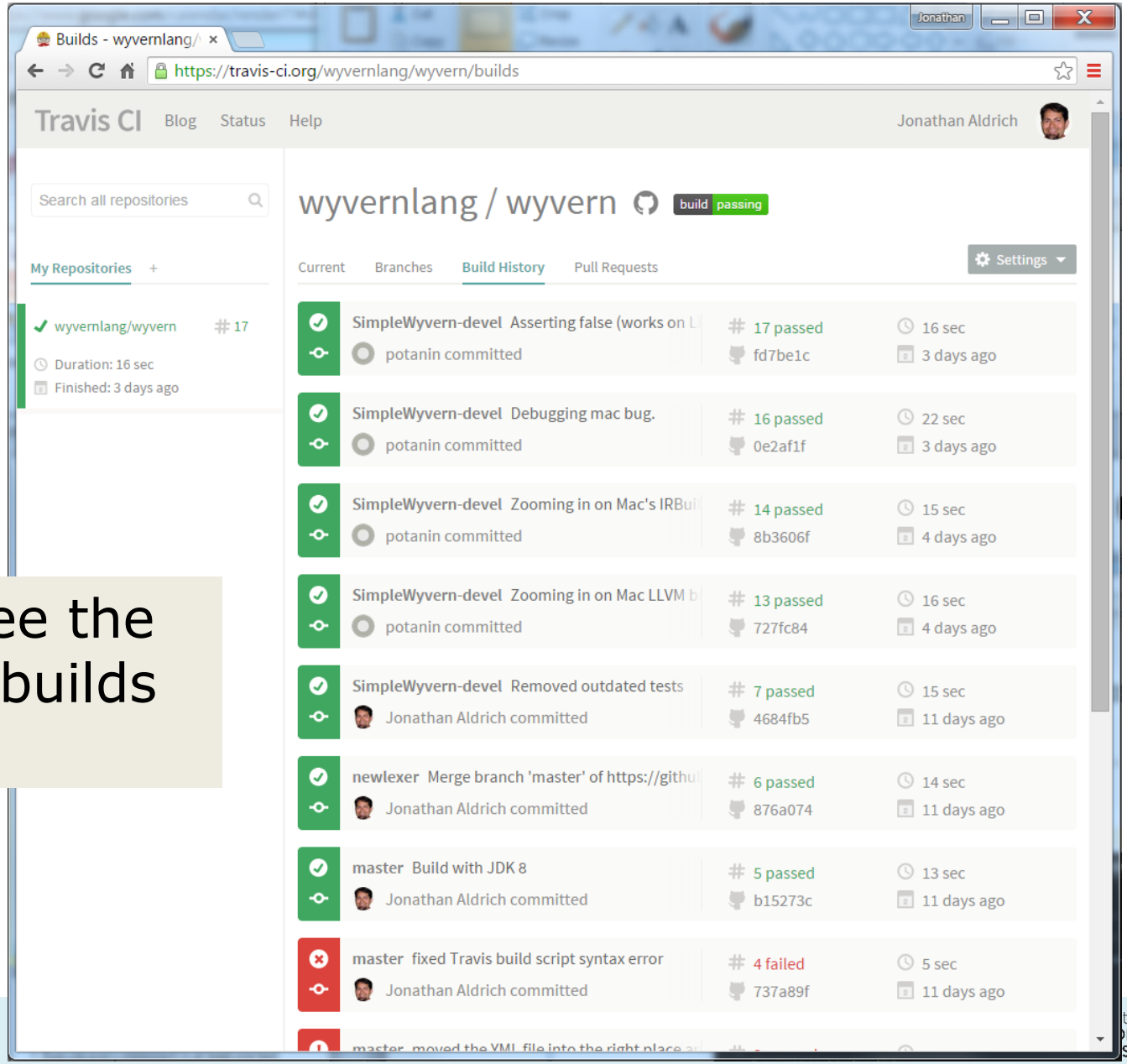
- You should only commit code that passses all tests…
- So run tests before every commit
- If test suite becomes too large & slow for rapid feedback
  - Run local package-level tests (“smoke tests”) frequently
  - Run all tests nightly
  - Medium sized projects often have thousands of test cases
- Continuous integration (CI) servers help to scale testing

Continuous integration – Travis CI



Automatically builds, tests, and displays the result

Continuous integration – Travis CI



You can see the results of builds over time

Outlook: **statement coverage**

- Trying to test all parts of the implementation
- Execute every statement, ideally

```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                                && amount() == aMoney.amount();
47        }
48        return false;
49    }
```

- Does 100% coverage guarantee correctness?

Outline

I. Exceptions
II. Specifying program behavior – contracts
III. Testing correctness – Junit and friends
IV. Overriding `Object` methods

Methods common to all objects

- How do collections know how to test objects for equality?
- How do they know how to hash and print them?
- The relevant methods are all present on `Object`
  - **`equals`** - returns true if the two objects are "equal"
  - **`hashCode`** - returns an `int` hash value that *must* be equal for equal objects, and is likely to differ on unequal objects
  - **`toString`** - returns a printable string representation

`Object` implementations

- Provide *identity semantics*
  - `equals(Object o)` - returns `true` if `o` refers to this object
  - `hashCode()` - returns an unspecified `int` that never changes over the object's lifetime
  - `toString()` - returns a nasty looking string consisting of the type and hash code
    - For example: `java.lang.Object@659e0bfd`

Overriding `Object` implementations

- **(nearly) Always override `toString`**
  - `println` invokes it automatically
  - Why settle for ugly?
- **No need to override `equals` and `hashCode` if you want identity semantics**
  - **When in doubt, don't override them**
  - Identity semantics are often what you want
  - It's easy to get the overrides wrong

Overriding `toString` is easy & beneficial

```
final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;
      ...
    @Override public String toString() {
        return String.format("(%03d) %03d-%04d",
            areaCode, prefix, lineNumber);
    }
}

PhoneNumber jenny = ...;
System.out.println(jenny);
```

Prints: `(707) 867-5309`

The `equals` contract

The equals method implements an **equivalence relation**. It is:

- **Reflexive**: For any non-null reference value x, x.equals(x) must return true.
- **Symmetric**: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true.
- **Transitive**: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
- **Consistent**: For any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) must return false.

The `equals` contract in English

- **Reflexive** – every object is equal to itself
- **Symmetric** – `if a.equals(b)` then `b.equals(a)`
- **Transitive** – if `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
- **Consistent** – equal objects stay equal unless mutated
- **"Non-null"** – `a.equals(null)` returns false
- Taken together these ensure that equals is a global **equivalence relation** over all objects

`equals` Override Example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public boolean equals(Object o) {
        if (!(o instanceof PhoneNumber))  // Does null check
            return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNumber == lineNumber
               && pn.prefix == prefix
               && pn.areaCode == areaCode;
    }

    ...
}
```

The hashCode contract

Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

The `hashCode` contract in English

- Equal objects **must** have equal hash codes
  - If you override `equals` you must override `hashCode`
- Unequal objects **should** have different hash codes
  - Take all value fields into account when calculating it
- Hash code must not change unless object mutated
  - Use a deterministic function of the field values

hashCode override example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        int result = 17;  // Nonzero is good
        result = 31 * result + areaCode;   // Constant must be odd
        result = 31 * result + prefix;     //      "      "   "   "
        result = 31 * result + lineNumber; //      "      "   "   "
        return result;
    }

    ...
}
```

Alternative hashCode override

Less efficient, but otherwise equally good!

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        return Objects.hash(areaCode, prefix, lineNumber);
    }

    ...
}
```

A one liner. No excuse for failing to override hashCode!

For more than you want to know about overriding object methods, see *Effective Java* Chapter 2

Summary

- Exceptions are way better than error codes
- Use try-with-resources; not manual cleanup
- Contracts specify method behavior
  - Document the contract of every method
- Testing is critical if you want program to work
- Always override `toString` (except for enums)
- Override `equals` when you need value semantics
- Override `hashCode` when your override `equals`