Principles of Software Construction:
Objects, Design, and Concurrency

Part 1: Designing classes

Testing, exceptions, and behavioral subtyping

Charlie Garrod Chris Timperley

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due tonight 11:59 p.m.
  - Everyone must read and sign our collaboration policy
- Optional reading due today
- Reading due Tuesday: Effective Java, Items 17 and 50
- Homework 2 due next Thursday at 11:59 p.m.

Key concepts from Tuesday

Key concepts from Tuesday

- Information hiding: Design for change, design for reuse
  - Encapsulation: visibility modifiers in Java
  - Interface types vs. class types
- Functional correctness
  - JUnit and friends

Selecting test cases

- Write tests based on the specification, for:
  - Representative cases
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker
- Other tests: performance, security, system interactions, …

A testing example

```
/**
  * computes the sum of the first len values of the array
  *
  * @param array array of integers of at least length len
  * @param len number of elements to sum up
  * @return sum of the first len array values
  * @throws NullPointerException if array is null
  * @throws ArrayIndexOutOfBoundsException if len > array.length
  * @throws IllegalArgumentException if len < 0
  */
int partialSum(int array[], int len);
```

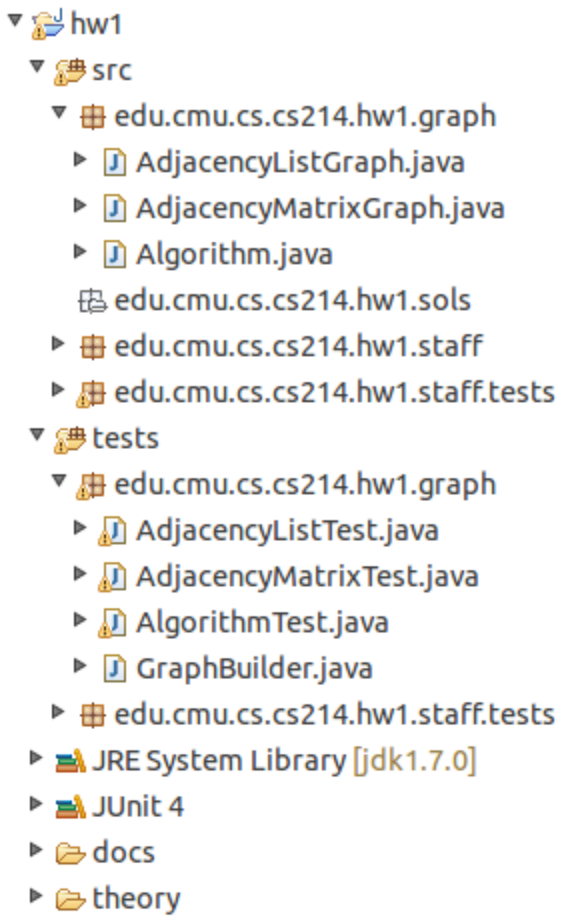
- Test null array
- Test length > array.length
- Test negative length
- Test small arrays of length 0, 1, 2
- Test long array
- Test length == array.length
- Stress test with randomly-generated arrays and lengths

Testable code

- Think about testing when writing code
  - Modularity and testability go hand in hand
- Same test can be used on all implementations of an interface!
- Test-driven development
  - Writing tests before you write the code
  - Tests can expose API weaknesses

Test organization conventions

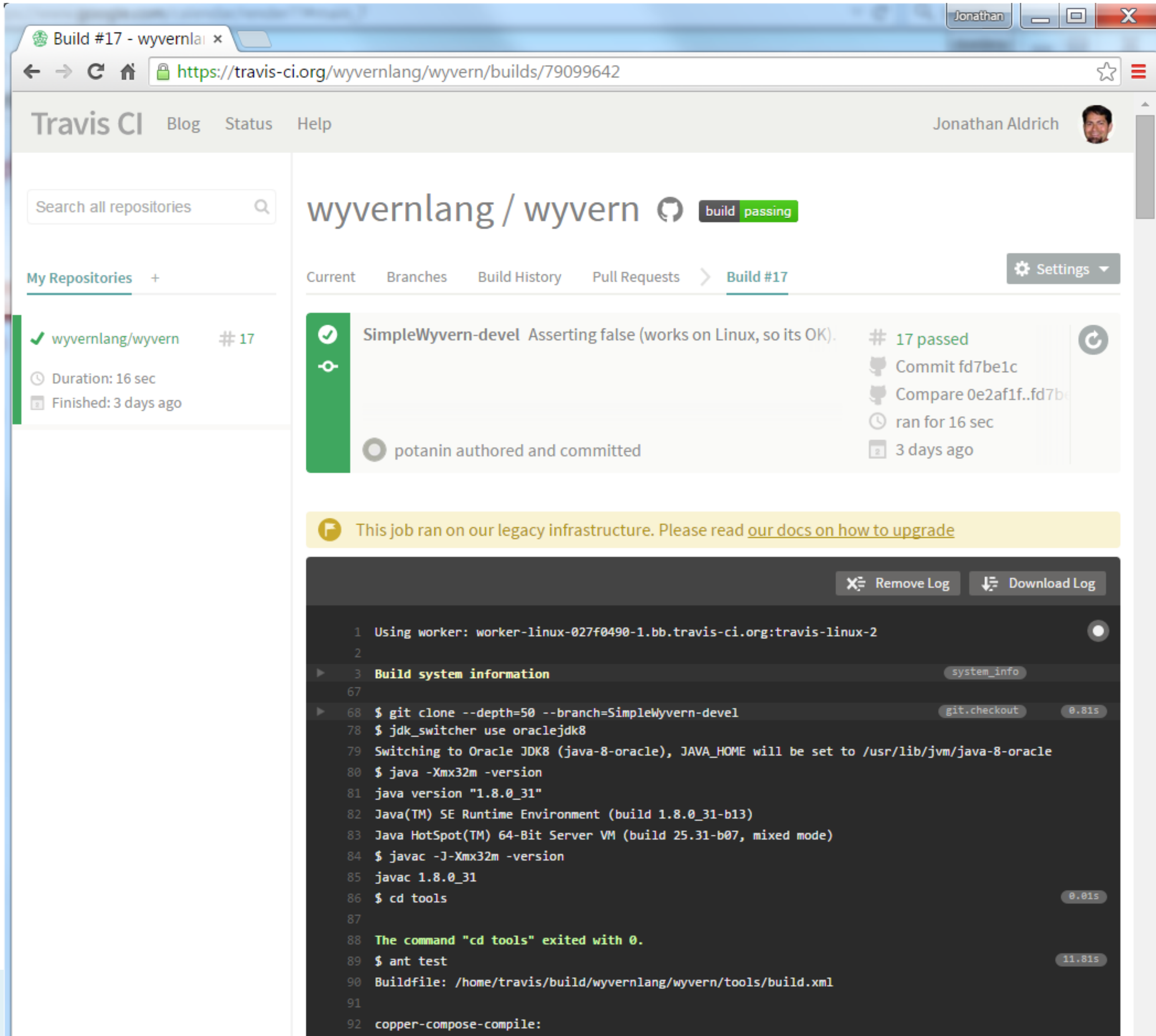
- Have a test class `FooTest` for each public class `Foo`
- Separate source and test directories
  - `FooTest` and `Foo` in the same package



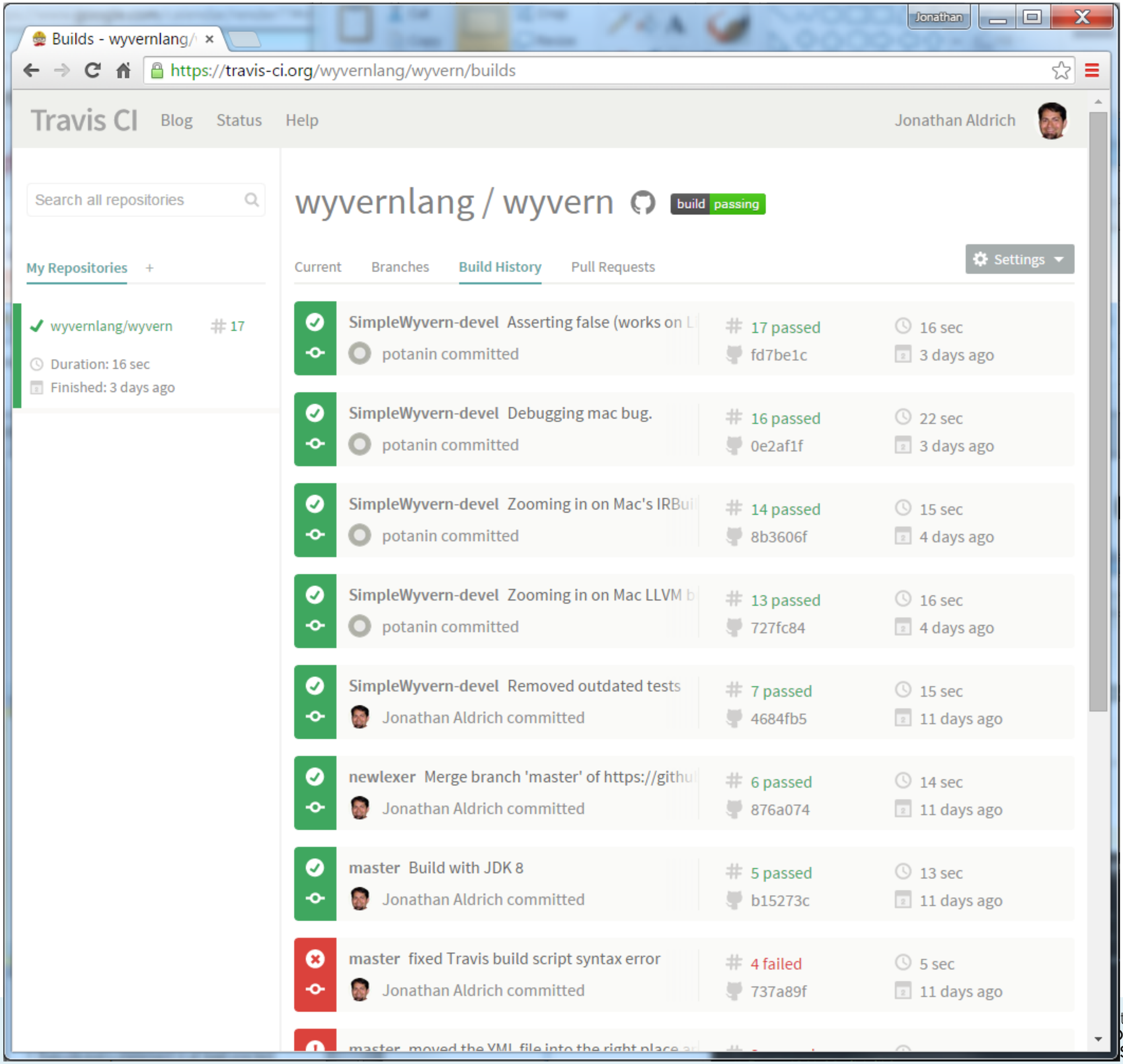
Run tests frequently

- Run tests before every commit
  - Do not commit code that fails a test
- If entire test suite becomes too large and slow:
  - Run local package-level tests ("smoke tests“) frequently
  - Run all tests nightly
  - Medium sized projects easily have 1000s of test cases
- Continuous integration servers scale testing

Continuous integration: Travis CI



Continuous integration: Travis CI build history



When should you stop writing tests?

When should you stop writing tests?

- When you run out of money…
- When your homework is due…
- When you can't think of any new test cases...
- Preview: The *coverage* of a test suite
  - Trying to test all parts of the implementation
  - Statement coverage: percentage of program statements executed
    - Compare to: method coverage, branch coverage, path coverage

Today

- Functional correctness, continued
- Exceptions
- Behavioral subtyping
  - Liskov Substitution Principle
  - The `java.lang.Object` contracts

What does this code do?

```
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == 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(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:

```
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

- Notify the caller of an exceptional condition by automatic transfer of control
- Semantics:
  - Propagates up stack until `main` method is reached (terminates program), or exception is caught
- Can be thrown by:
  - The program: e.g., `IllegalArgumentException`
  - The JVM: e.g., `StackOverflowError`

Control-flow of exceptions

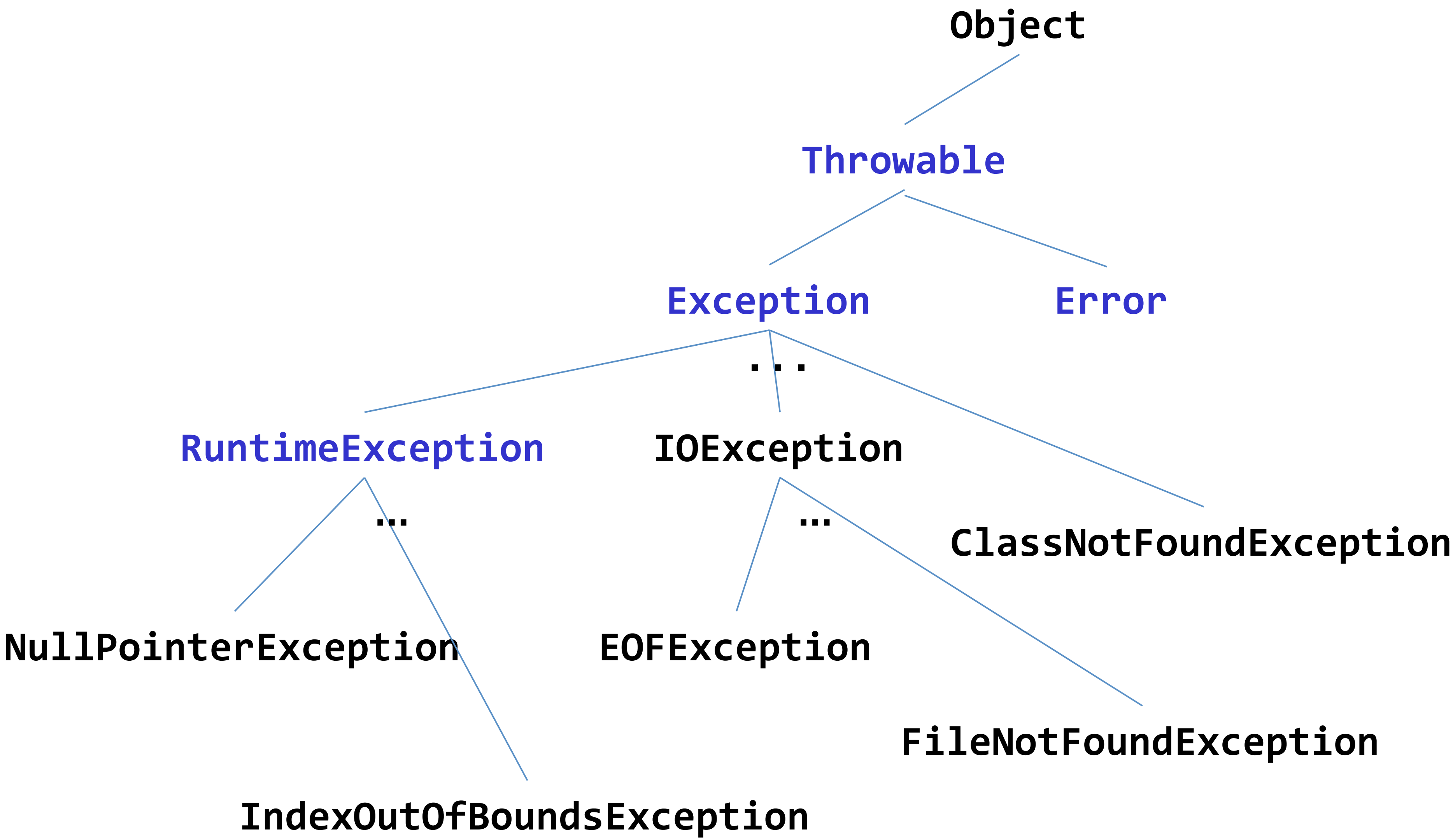
```
public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println"("Caught index out of bounds");
    }
}

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");
    }
}
```

Checked vs. unchecked exceptions

- Checked exception
  - Must be caught or propagated, or program won't compile
- Unchecked exception
  - No action is required for program to compile
  - But uncaught exception will cause program to fail!

The exception hierarchy in Java



Exceptional design choices

- 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
- Do NOT use return codes
- NEVER return `null` to indicate a zero-length result
  - Use a zero-length list or array instead

Creating and throwing your own exceptions

```
public class SpanishInquisitionException extends RuntimeException {
    public SpanishInquisitionException() {
    }
}

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

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, and stack trace
  - Compare: core dump in C
- Improve code structure
  - Separate normal code path from exceptional
  - Ease task of recovering from failure
- Ease task of writing robust, maintainable code

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
  - Checked and unchecked (EJ Item 74)
  - But don't declare unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IlegalArgumentException(
      "Modulus must be prime: " + modulus);
```

- Don't ignore exceptions (EJ Item 77)

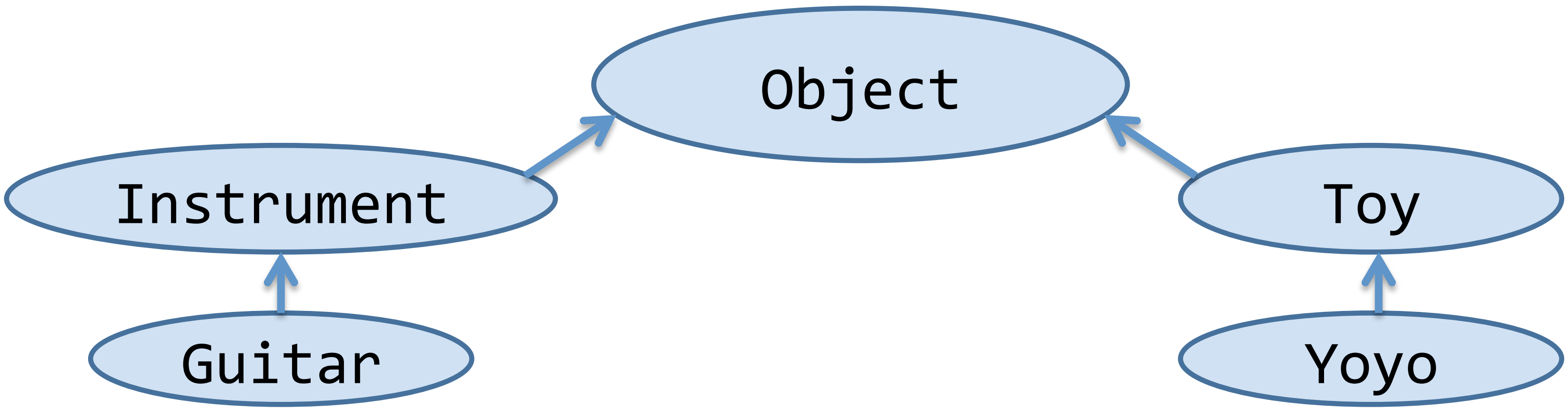
```
// Empty catch block IGNORES exception, Bad smell in code!
try {
   …
} catch (SomeException e) { }
```

Today

- Functional correctness, continued
- Exceptions
- Behavioral subtyping
  - Liskov Substitution Principle
  - The `java.lang.Object` contracts

The class hierarchy

- The root is `Object` (all non-primitives are `Object`s)
- All classes except `Object` have one parent class
  - Specified with an `extends` clause:
    ```
    class Guitar extends Instrument { ... }
    ```
  - If `extends` clause is omitted, defaults to `Object`
- A class is an instance of all its superclasses



Behavioral subtyping

> Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
>
> Barbara Liskov

- e.g., Compiler-enforced rules in Java:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions

This is called the *Liskov Substitution Principle*.

Behavioral subtyping

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

Barbara Liskov

- e.g., Compiler-enforced rules in Java:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions
- Also applies to specified behavior. Subtypes must have:
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

This is called the *Liskov Substitution Principle*.

LSP example: `Car` is a behavioral subtype of `Vehicle`

```
abstract class Vehicle {
  int speed, limit;

  //@ invariant speed < limit;

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  abstract void brake();
}
```

```
class Car extends Vehicle {
  int fuel;
  boolean engineOn;
  //@ invariant speed < limit;
  //@ invariant fuel >= 0;

  //@ requires fuel > 0
        && !engineOn;
  //@ ensures engineOn;
  void start() { … }

  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake() { … }
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions

LSP example: Hybrid is a behavioral subtype of Car

```
class Car extends Vehicle {
  int fuel;
  boolean engineOn;
  //@ invariant speed < limit;
  //@ invariant fuel >= 0;

  //@ requires fuel > 0
        && !engineOn;
  //@ ensures engineOn;
  void start() { … }

  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake() { … }
}
```

```
class Hybrid extends Car {
  int charge;
  //@ invariant charge >= 0;
  //@ invariant …
  //@ requires (charge > 0
                 || fuel > 0)
              && !engineOn;
  //@ ensures engineOn;
  void start() { … }

  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  //@ ensures charge > \old(charge)
  void brake() { … }
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method start has weaker precondition
Overridden method brake has stronger postcondition

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

(Yes.)

institute for SOFTWARE RESEARCH

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

```
class GraphicProgram {
    void scaleW(Rectangle r, int f) {
        r.setWidth(r.getWidth() * f);
    }
}
```

← **Invalidates stronger invariant (h==w) in subclass**

(Yes! But the Square is not a square…)

This Square is *not* a behavioral subtype of Rectangle

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
   //@ ensures w==neww
             && h==old.h;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
   //@ invariant h>0 && w>0;
   //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }

   //@ requires neww > 0;
   //@ ensures w==neww
             && h==neww;
   @Override
   void setWidth(int neww) {
      w=neww;
      h=neww;
   }
}
```

Today

- Functional correctness, continued
- Exceptions
- Behavioral subtyping
  - Liskov Substitution Principle
  - The `java.lang.Object` contracts

Recall: Methods common to all `Objects`

- `equals`: returns true if the two objects are "equal"
- `hashCode`: returns an `int` that must be equal for equal objects, and is likely to differ for unequal objects
- `toString`: returns a printable string representation

The built-in `java.lang.Object` implementations

- Provide identity semantics:
  - `equals(Object o)`: returns `true` if o refers to this object
  - `hashCode()`: returns a near-random `int` that never changes
  - `toString()`: returns a string consisting of the type and hash code
    - For example: `java.lang.Object@659e0bfd`

The `toString()` specification

- Returns a concise, but informative textual representation
- Advice: Always override `toString()`, e.g.:

```
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);
    }
}

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

Prints: `(707) 867-5309`

The `equals(Object)` specification

- Must define an equivalence relation:
  - Reflexive: For every object `x`, `x.equals(x)` is always true
  - Symmetric: If `x.equals(y)`, then `y.equals(x)`
  - Transitive: If `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`
- Consistent: Equal objects stay equal, unless mutated
- "Non-null": `x.equals(null)` is always false

An `equals(Object)` 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()` specification

- Equal objects must have equal hash codes
  - If you override `equals` you must override `hashCode`
- Unequal objects should usually have different hash codes
  - Take all value fields into account when constructing it
- Hash code must not change unless object is mutated

A hashCode() 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;
    }

    ...
}
```

Summary

- Please complete the course reading assignments
- Test early, test often!
- Subtypes must fulfill behavioral contracts
- Always override `hashCode` if you override `equals`
- Always use `@Override` if you intend to override a method
  - Or let your IDE generate these methods for you…