Principles of Software Construction: Objects, Design, and Concurrency

Object-Oriented Programming in Java and Functional Correctness

Charlie Garrod **Chris Timperley**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- No smoking
- Office hours
- Homework 1 due Thursday 11:59 p.m.
  - Everyone must read and sign our collaboration policy
- Second reading assignment due next Tuesday
  - Effective Java Items 17 and 50

Key concepts from Thursday

- Bipartite type system – primitives & object refs
  - Single implementation inheritance
  - Multiple interface inheritance
- Easiest output – `println` , `printf`
- Easiest input – Command line args, `Scanner`
- Collections framework is powerful & easy to use

Outline

I. Object-oriented programming basics
II. Information hiding
III. Contracts

Recap: Objects

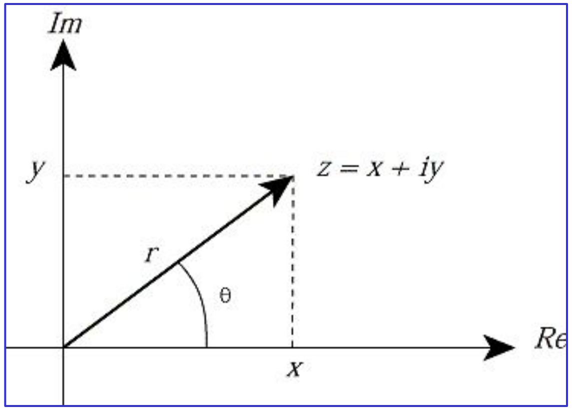
- An **object** is a bundle of state and behavior
- State – the data contained in the object
  - In Java, these are the **fields** of the object
- Behavior – the actions supported by the object
  - In Java, these are called **methods**
  - Method is just OO-speak for function
  - Invoke a method = call a function

Recap: Classes

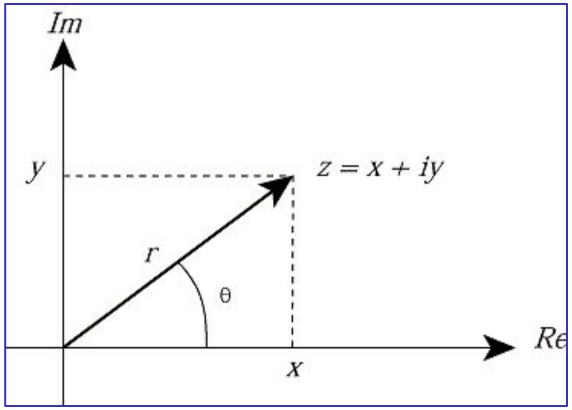
- Every object has a class
  - A class defines methods and fields
  - Methods and fields collectively known as **members**
- Class defines both type and implementation
  - Type ≈ where the object can be used
  - Implementation ≈ how the object does things
- Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
  - Defines how users interact with instances

A more complex example

```
class Complex {
  private final double re;  // Real Part
  private final double im;  // Imaginary Part

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```



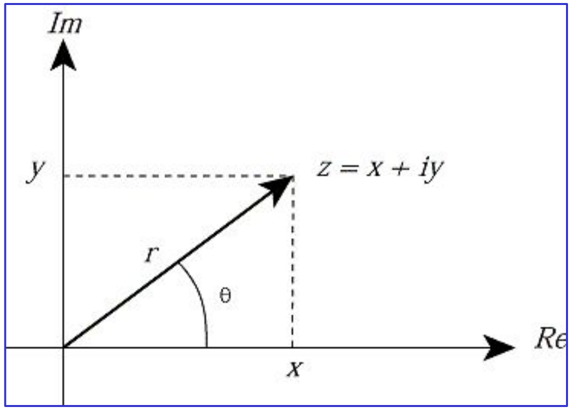
A more complex example



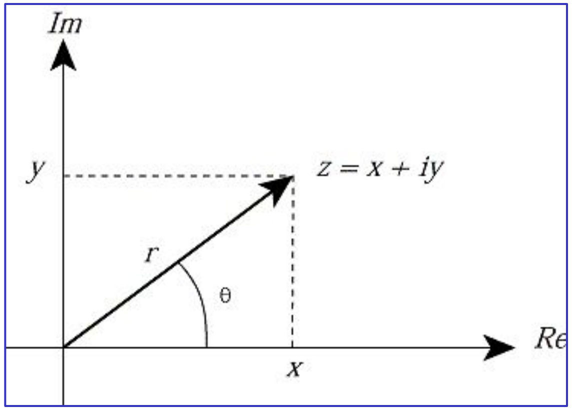

```
class Complex {
  private final double re;  // Real Part
  private final double im;  // Imaginary Part

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

A more complex example

```
class Complex {
  private final double re;  // Real Part
  private final double im;  // Imaginary Part

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```



A more complex example




```
class Complex {
  private final double re;  // Real Part
  private final double im;  // Imaginary Part

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Class usage example

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new Complex(-1, 0);
        Complex d = new Complex(0, 1);

        Complex e = c.add(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.multiply(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Interfaces and implementations

- Multiple implementations of API can coexist
  - Multiple classes can implement the same API
  - They can differ in performance and behavior
- In Java, an API is specified by *interface* or *class*
  - Interface provides only an API
  - Class provides an API and an implementation
  - A class can implement multiple interfaces

An interface to go with our class

```
public interface Complex {
    // No constructors, fields, or implementations!

    double realPart();
    double imaginaryPart();
    double r();
    double theta();

    Complex add(Complex c);
    Complex subtract(Complex c);
    Complex multiply(Complex c);
    Complex divide(Complex c);
}
```

An interface defines but does not implement API

Modifying class to use interface

```
class OrdinaryComplex implements Complex {
  final double re;  // Real Part
  final double im;  // Imaginary Part

  public OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Modifying client to use interface

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex(0, 1);

        Complex e = c.add(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.multiply(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it still prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Interface permits multiple implementations

```
class PolarComplex implements Complex {
  final double r;
  final double theta;

  public PolarComplex(double r, double theta) {
    this.r = r;
    this.theta = theta;
  }

  public double realPart()      { return r * Math.cos(theta) ; }
  public double imaginaryPart() { return r * Math.sin(theta) ; }
  public double r()             { return r; }
  public double theta()         { return theta; }

  public Complex add(Complex c)      { ... } // Completely different impls
  public Complex subtract(Complex c)     { ... }
  public Complex multiply(Complex c)     { ... }
  public Complex divide(Complex c) { ... }
}
```

Interface decouples client from implementation

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(Math.PI,   1);  // -1
        Complex d = new PolarComplex(Math.PI/2, 1);  //  i

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it STILL prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Why multiple implementations?

- Different performance
  - Choose implementation that works best for your use
- Different behavior
  - Choose implementation that does what you want
  - Behavior *must* comply with interface spec ("contract")
- Often performance and behavior *both* vary
  - Provides a functionality – performance tradeoff
  - Example: `HashSet`, `TreeSet`

Java interfaces and classes

- A type defines a family of objects
  - Each type offers a specific set of operations
  - Objects are otherwise opaque
- Interfaces vs. classes
  - Interface: specifies expectations
  - Class: delivers on expectations (the implementation)

Classes as types

- Classes *do* define types
  - Public class methods usable like interface methods
  - Public fields directly accessible from other classes
- But generally prefer the use of interfaces
  - Use interface types for variables and parameters unless you know a single implementation will suffice
    - Supports change of implementation
    - Prevents dependence on implementation details

```
Set<Criminal> senate = new HashSet<>();          // Do this…
HashSet<Criminal> senate = new HashSet<>();      // Not this
```

Check your understanding

```
interface Animal {
    void vocalize();
}
class Dog implements Animal {
    public void vocalize() { System.out.println("Woof!"); }
}
class Cow implements Animal {
    public void vocalize() { moo(); }
    public void moo() { System.out.println("Moo!"); }
}
```

What Happens?

1. ```
   Animal a = new Animal();
   a.vocalize();
   ```
2. ```
   Dog d = new Dog();
   d.vocalize();
   ```
3. ```
   Animal b = new Cow();
   b.vocalize();
   ```
4. ```
   b.moo();
   ```

Outline

I. Object-oriented programming basics
II. Information hiding
III. Contracts

Information hiding

- Single most important factor that distinguishes a well-designed module from a bad one is the degree to which it hides internal data and other implementation details from other modules
- Well-designed code hides *all* implementation details
  - Cleanly separates API from implementation
  - Modules communicate *only* through APIs
  - They are oblivious to each others' inner workings
- Known as *information hiding* or *encapsulation*
- Fundamental tenet of software design [Parnas, '72]

Benefits of information hiding

- **Decouples** the classes that comprise a system
  - Allows them to be developed, tested, optimized, used, understood, and modified in isolation
- **Speeds up system development**
  - Classes can be developed in parallel
- **Eases burden of maintenance**
  - Classes can be understood more quickly and debugged with little fear of harming other modules
- **Enables effective performance tuning**
  - "Hot" classes can be optimized in isolation
- **Increases software reuse**
  - Loosely-coupled classes often prove useful in other contexts

Information hiding with interfaces

- Declare variables using interface types
- Client can use only interface methods
- Fields not accessible from client code
- But this only takes us so far
  - Client can access non-interface members directly
  - In essence, it's voluntary information hiding

Mandatory Information hiding

visibility modifiers for members

- `private` – Accessible *only* from declaring class
- package-private – Accessible from any class in the package where it is declared
  - Technically known as default access
  - You get this if no access modifier is specified
- `protected` – Accessible from package and also from subclasses
- `public` – Accessible from anywhere

Hiding interior state in OrdinaryComplex

```
class OrdinaryComplex implements Complex {
  private double re;  // Real Part
  private double im;  // Imaginary Part

  public OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Discussion

- You know the benefits of private fields
- What are the benefits of private methods?

Best practices for information hiding

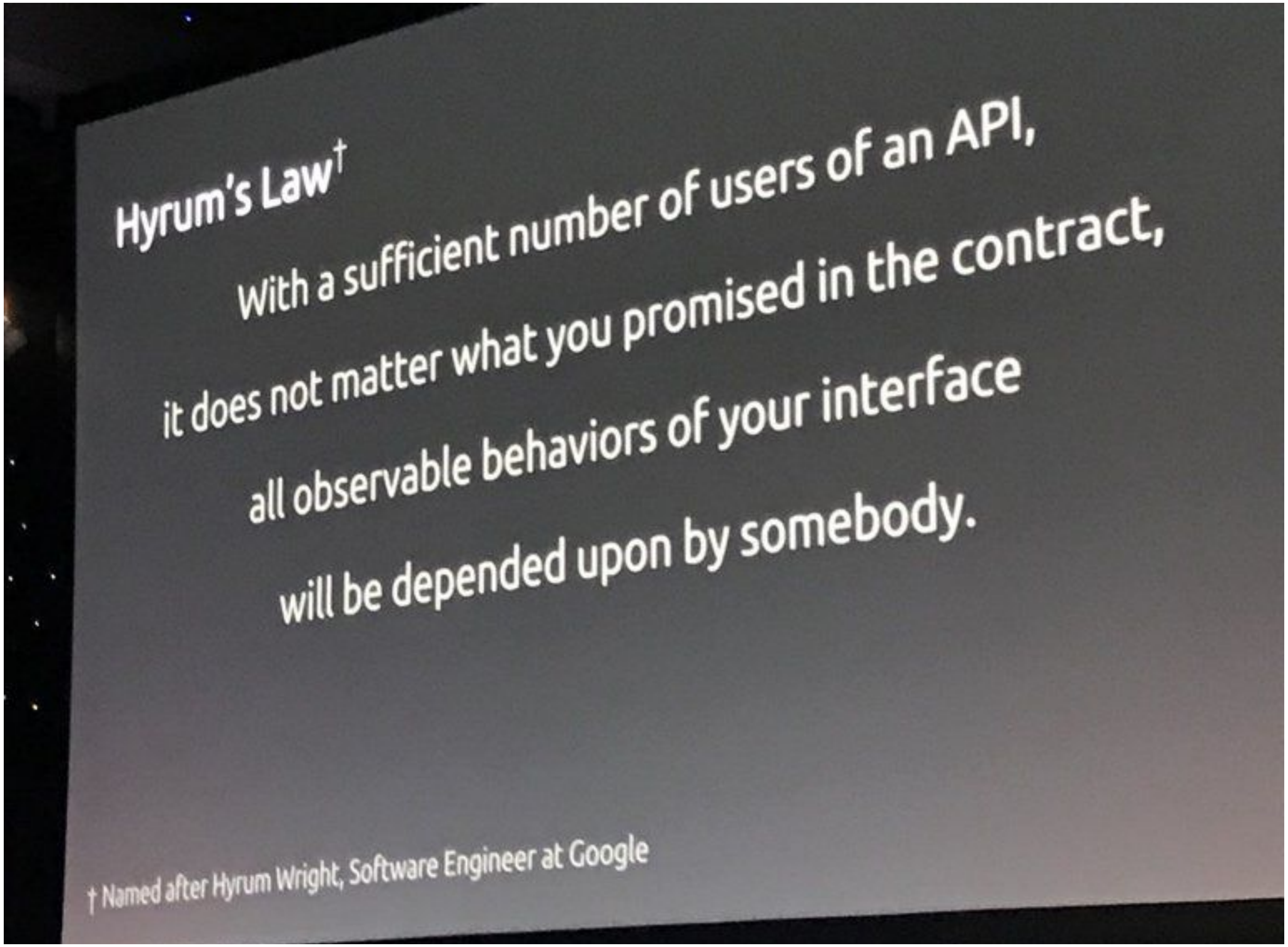
- Carefully design your API
- Provide *only* functionality required by clients
  - *All* other members should be private
- You can always make a private member public later without breaking clients
  - But not vice-versa!

Hyrum's Law

Hyrum Wright

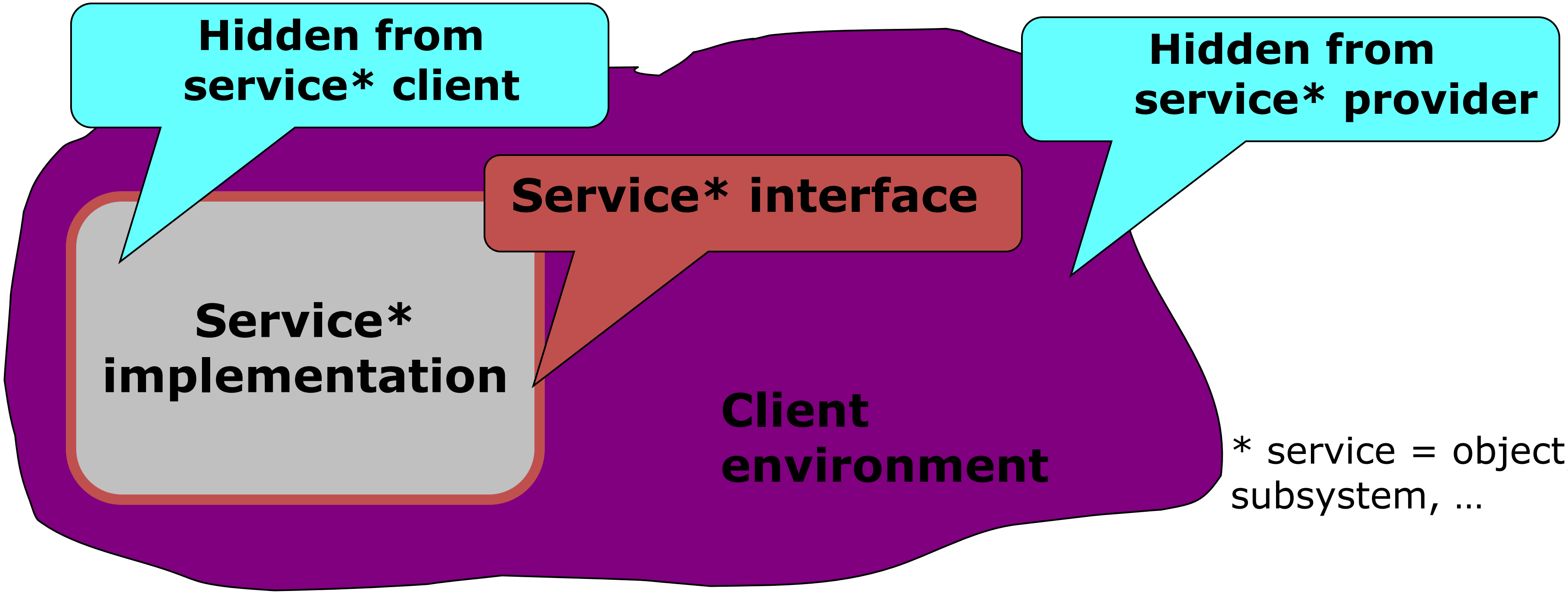
@hyrumwright

Infrastructure software engineer. Googler. Father. Occasional professor.



institute for SOFTWARE RESEARCH

The best API is the thinnest



* service = object, subsystem, …

institute for SOFTWARE RESEARCH

Outline

I. Object-oriented programming basics
II. Information hiding
III. Contracts

Contracts

- Agreement between provider and users of an object
- Includes
  - Interface specification (types)
  - Functionality and correctness expectations
  - Performance expectations
- What the method does, not how it does it
  - Interface (API), not implementation

Who’s to blame?

```
Algorithms.shortestDistance(g, “Tom”, “Anne”);
```

What should happen if there is no path between Tom or Anne?

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 to
     * vertices. It returns -1 if
     * the two nodes are not
     * connected. */
    int shortestDistance(…) {…}
}
```

Who’s to blame?

```
Math.sqrt(-5);
```

```
> 0
```

Who's to blame?

```
/**
  * Returns the correctly rounded positive square root of a
  * {@code double} value.
  * Special cases:
  * <ul><li>If the argument is NaN or less than zero, then the
  * result is NaN.
  * <li>If the argument is positive infinity, then the result
  * is positive infinity.
  * <li>If the argument is positive zero or negative zero, then
  * the result is the same as the argument.</ul>
  * Otherwise, the result is the {@code double} value closest to
  * the true mathematical square root of the argument value.
  *
  * @param a a value.
  * @return the positive square root of {@code a}.
  * If the argument is NaN or less than zero, the result is NaN.
  */
public static double sqrt(double a) { …}
```

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data f
  attempt is made to read as ma
  The number of bytes actually r
  until input data is available, en
- If len is zero, then no bytes ar
  attempt to read at least one by
  end of file, the value -1 is retu
  into b.
- The first byte read is stored in
  on. The number of bytes read
  bytes actually read; these byte
  elements b[off] throughb[off+
  elements b[off+*k*] through b[o
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Specification of return
- Case-by-case spec
  - len=0 return 0
  - len>0 && eof return -1
  - len>0 && !eof return >0
- Exactly where the data is stored
- What parts of the array are not affected

- Throws:
  - IOException - If the first byte
    or if the input stream has been
  - NullPointerException - If b is n
  - IndexOutOfBoundsException -
    than b.length - off

- **Multiple error cases, each with a precondition**
- **Includes "runtime exceptions" not in throws clause**

Specifications

- Contains
  - Functional behavior
  - Erroneous behavior
  - Quality attributes (performance, scalability, security, …)
- Desirable attributes
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need

Functional Specification

- States method’s and caller’s responsibilities
- Analogy: legal contract
  - If you pay me this amount on this schedule…
  - I will build a with the following detailed specification
  - 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 what it means for implementation to be correct

Functional Specification

What does the implementation have to fulfill if the client violates the precondition?

- States
- Analog
  - If yo
  - I wil ...pecification
  - Som... ...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 what it means for implementation to be correct

Formal Specifications

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

Advantage of formal specifications:

* runtime checks (almost) for free
* basis for formal verification
* assisting automatic analysis tools

JML (Java Modelling Language) as specifications language in Java (inside comments)

Disadvantages ?

Runtime Checking of Specifications with Assertions

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @             (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
float sum(int array[], int len) {
       assert len >= 0;
       assert array.length == len;
       float sum = 0.0;
       int i = 0;
       while (i < len) {
           sum = sum + array[i]; i = i + 1;
       }
       assert sum ...;
       return sum;
}
```

Enable assertions with -ea flag, e.g.,

`> java -ea Main`

Runtime Checking of Specifications with Exceptions

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @            (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
float sum(int array[], int len) {
       if (len < 0 || array.length != len)
           throw IllegalArgumentException(...);
       float sum = 0.0;
       int i = 0;
       while (i < len) {
           sum = sum + array[i]; i = i + 1;
       }
       return sum;
}
```

Check arguments even when assertions are disabled.
Good for robust libraries!

Specifications in the real world

Javadoc

```
/**
 * 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

Precondition

Exceptional behavior

Javadoc contents

- Document
  - Every parameter
  - Return value
  - Every exception (checked and unchecked)
  - What the method does, including
    - Purpose
    - Side effects
    - Any thread safety issues
    - Any performance issues
- Do **not** document implementation details

Contracts and Interfaces

- All objects implementing an interface must adhere to the interface's contracts
  - Objects may provide different implementations for the same specification
  - Subtype polymorphism: Client only cares about interface, not about the implementation

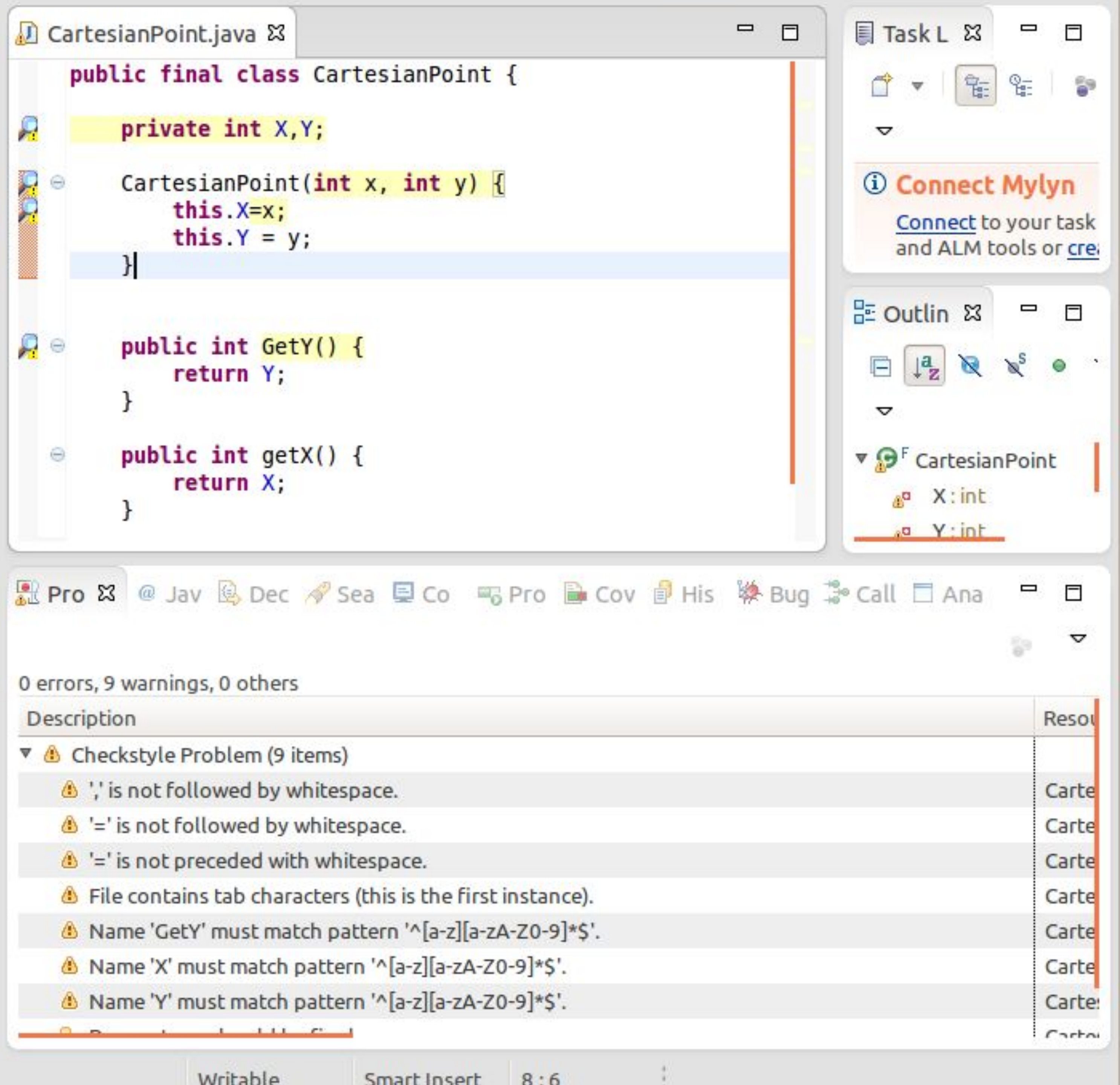
    p.getX() s.read()

=> **Design for Change**

Functional correctness

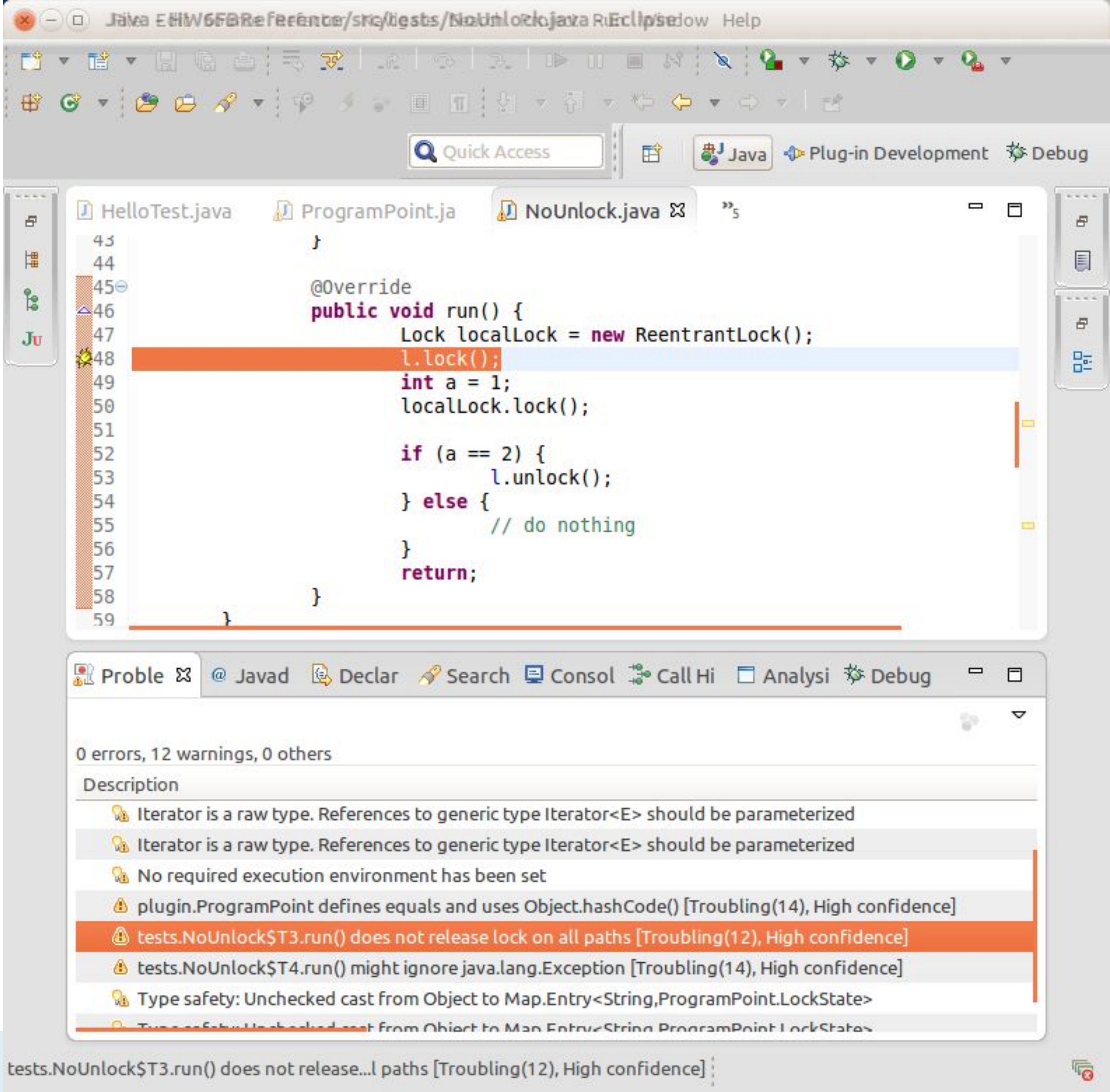
- Compiler ensures types are correct
- Static analysis tools recognize common problems ("bug patterns")
- …

CheckStyle



CartesianPoint.java
public final class CartesianPoint {
private int X,Y;
CartesianPoint(int x, int y) {
this.X=x;
this.Y = y;
}
public int GetY() {
return Y;
}
public int getX() {
return X;
}
Task L
Connect Mylyn
Connect to your task and ALM tools or
Outlin
CartesianPoint
X : int
Pro Jav Dec Sea Co Pro Cov His Bug Call Ana
0 errors, 9 warnings, 0 others
Description
Checkstyle Problem (9 items)
',' is not followed by whitespace.
'=' is not followed by whitespace.
'=' is not preceded with whitespace.
File contains tab characters (this is the first instance).
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Writable
Smart Insert
8 : 6

SpotBugs



HelloTest.java
ProgramPoint.ja
NoUnlock.java
@Override
public void run() {
Lock localLock = new ReentrantLock();
l.lock();
int a = 1;
localLock.lock();
if (a == 2) {
l.unlock();
} else {
// do nothing
}
return;
}
}
Proble
Javad
Declar
Search
Consol
Call Hi
Analysi
Debug
0 errors, 12 warnings, 0 others
Description
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
No required execution environment has been set
plugin.ProgramPoint defines equals and uses Object.hashCode() [Troubling(14), High confidence]
tests.NoUnlock$T3.run() does not release lock on all paths [Troubling(12), High confidence]
tests.NoUnlock$T4.run() might ignore java.lang.Exception [Troubling(14), High confidence]
Type safety: Unchecked cast from Object to Map.Entry<String,ProgramPoint.LockState>
tests.NoUnlock$T3.run() does not release...l paths [Troubling(12), High confidence]
Quick Access
Java
Plug-in Development
Debug

Functional correctness

- Compiler ensures types are correct
- Static analysis tools recognize common problems ("bug patterns")
- Formal verification
  - Mathematically prove code matches its specification
- Testing
  - Execute program with select inputs in a controlled environment
- …

Formal verification vs. 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

Formal verification vs. testing?

Consider `java.util.Arrays.binarySearch`:

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
             low = mid + 1
         else if (midVal > key)
             high = mid - 1;
         else
             return mid; // key found
     }
     return -(low + 1);  // key not found.
 }
```

Formal verification vs. testing?

Consider `java.util.Arrays.binarySearch`:

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
             low = mid + 1
         else if (midVal > key)
             high = mid - 1;
         else
             return mid; // key found
     }
     return -(low + 1);  // key not found.
 }
```

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

Comparing strategies for correctness

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)
- Static Analysis
  - Analysis of all possible executions
  - Specific issues only with conservative approx. and bug patterns
  - Tools available, useful for bug finding
  - Automated, but unsound and/or incomplete
- Proofs (formal verification)
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

Which strategies to use in your project?

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 or a testing system?
- How to check output / assertions?
- What are the costs?
- Are bugs reproducible?

Automate testing

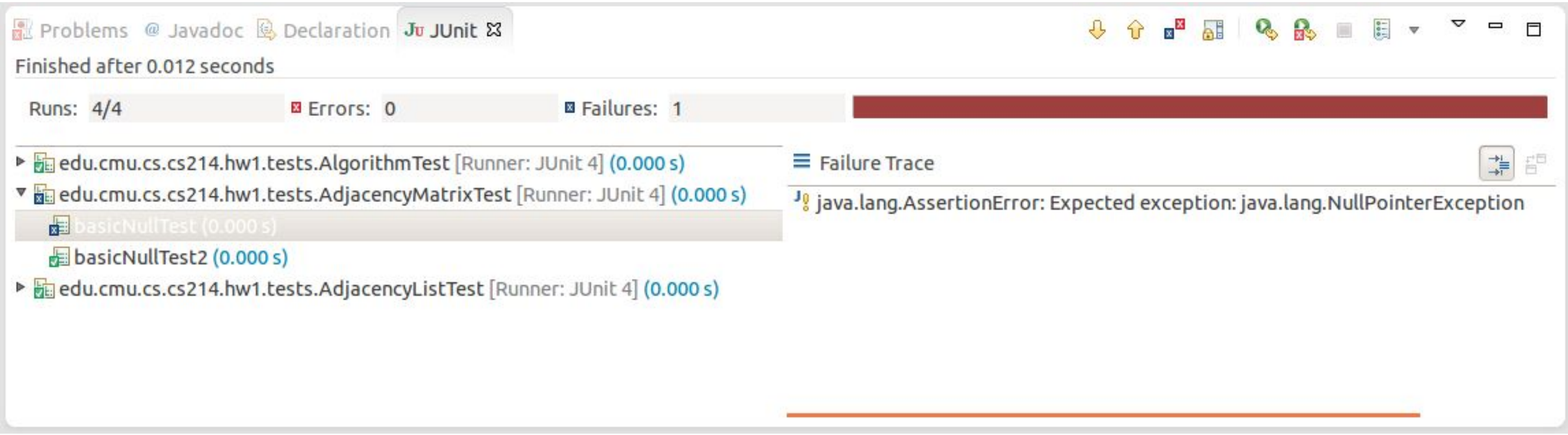
- Execute a program with specific inputs, check output for expected values
- Set up testing infrastructure
- Execute tests regularly
  - After *every* change

Unit testing

- Tests for small units: methods, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment

JUnit

- A popular, easy-to-use, unit-testing framework for Java



A JUnit example

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    @Test
    public void testSanityTest(){
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g1.addVertex(s1));
        assertEquals(true, g1.addVertex(s2));
        assertEquals(true, g1.addEdge(s1, s2));
        assertEquals(s2, g1.getNeighbors(s1)[0]);
    }

    @Test
    public void test….

    private int helperMethod…
}
```

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

```java
/**
 * 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 IndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */
int partialSum(int array[], int len);
```

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 IndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */
int partialSum(int array[], int len);
```

- Test negative length

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 IndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */
int partialSum(int array[], int len);
```

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

Summary

- Interface-based designs handle change well
- Information hiding is crucial to good design
- Keep your API as thin as possible
- Test early and test often