toad

Fall 2014

Principles of Software Construction: Objects, Design, and Concurrency

Contracts and Exceptions

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 2 due at 11:59pm tonight

Review: In Tuesday's Lecture…

- What design principles did we discuss?
- What GRASP heuristics were described?

Today's Lecture

- 214: managing complexity, from programs to systems
  - **T**hreads and concurrency
  - **O**bject-oriented programming
  - **A**nalysis and modeling
  - **D**esign

Today's Lecture: Learning Goals

- Use exceptions to write robust programs
- Review formal pre- and post-condition specifications (from 15-122)
- Use class invariants to reason about object representation
- Apply behavioral subtyping to reason about the behavior of subclasses and subtypes
- Check programs with assertions and static analysis tools

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:

```
try {
    FileInputStream fileInput = new FileInputStream(filename);
    DataInput dataInput = new DataInputStream(fileInput);
    int i = dataInput.readInt();
    fileInput.close();
    return i;
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
    return -1;
} catch (IOException e) {
    System.out.println("Error reading binary data from file "
                                     + filename);
    return -1;
}
```

Exceptions

- Exceptions notify the caller of an exceptional circumstance (usually operation failure)
- Semantics
  - An exception propagates *up the function-call stack* until `main()` is reached or until the exception is caught
- Sources of exceptions:
  - Programmatically throwing an exception
  - Exceptions thrown by the Java Virtual Machine

Exceptional control-flow

```java
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[42] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
}
```

- Prints:

```
Top
Caught index out of bounds
```

Exceptional control-flow, part 2

```
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");
    }
}
```

- Prints:

```
Top
Caught index out of bounds
```

The `finally` keyword

- The finally block always runs after try/catch:

```
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[42] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```

- Prints:

```
Top
Caught index out of bounds
Finally got here
```

The `finally` keyword, part 2

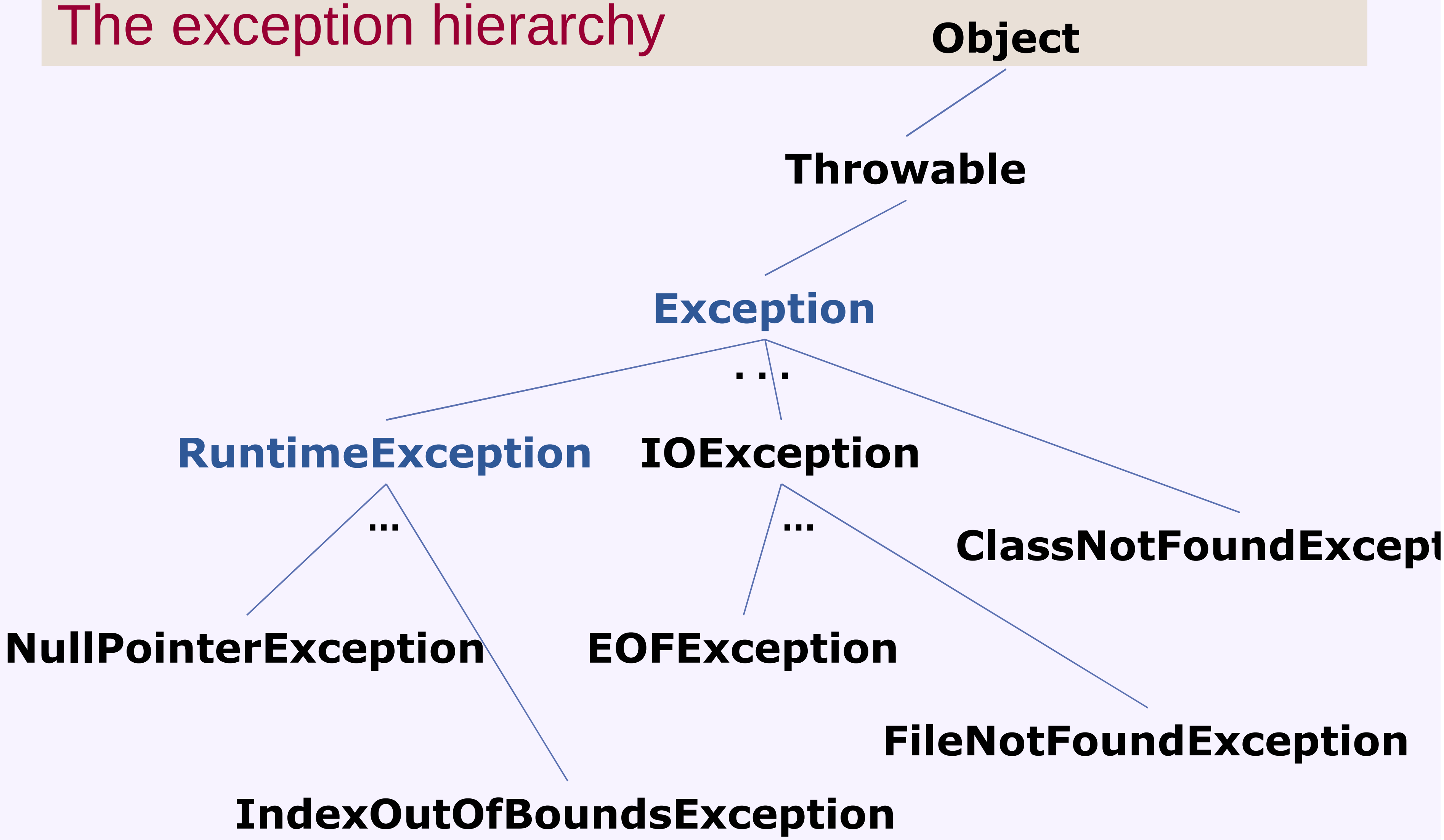
- The finally block always runs after try/catch:

```
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[2] = 2;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```

- Prints:

```
Top
Bottom
Finally got here
```

The exception hierarchy



Checked and unchecked exceptions

- Unchecked exception: any subclass of `RuntimeException`
  - Indicates an error which is highly unlikely and/or typically unrecoverable
- Checked exception: any subclass of `Exception` that is not a subclass of `RuntimeException`
  - Indicates an error that every caller should be aware of and explicitly decide to handle or pass on

Creating and throwing your own exceptions

- Methods must declare any checked exceptions they might throw
- If your class extends `java.lang.Throwable` you can throw it:

```
if (someErrorBlahBlahBlah) {
   throw new MyCustomException(“Blah blah blah”);
}
```

Benefits of exceptions

- Provide high-level summary of error and stack trace
  - Compare: core dumped in C
- Can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Can optionally recover from failure
  - Compare: calling `System.exit()`
- Improve code structure
  - Separate routine operations from error-handling
- Allow consistent clean-up in both normal and exceptional operation

Guidelines for using exceptions

- Catch and handle all checked exceptions
  - Unless there is no good way to do so…
- Use runtime exceptions for programming errors
- Other good practices
  - Do not catch an exception without (at least somewhat) handling the error
  - When you throw an exception, describe the error
  - If you re-throw an exception, always include the original exception as the cause

Testing for presence of an exception

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

public class Tests {

      @Test
      public void testSanityTest(){
            try {
                  openNonexistingFile();
                  fail("Expected exception");
            } catch(IOException e) { }
      }

      @Test(expected = IOException.class)
      public void testSanityTestAlternative() {
            openNonexistingFile();
      }
}
```

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 specification language in Java (inside comments)

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

Could write a function to use here, but it would be the same as `total`. In practice, not useful unless the assertion is simpler than the function..

```
java -ea Main
```

Runtime Checking with Exceptions

```
/*@ 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) {
    if (len < 0 || array == null || array.length != len)
        throw new IllegalArgumentException(…);
    int 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!

Example Java I/O Library Specification (abridged)

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

Example Java I/O Library Specification (abridged)

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
  1], leaving elements b[off+*k*]
- In every case, elements b[0] through b[off] and
  elements b[off+len] through b[b.length-1] are unaffected.

- Specification of return
- Timing behavior (blocks)
- 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

Textual Specifications

List:

boolean addAll(int index, Collection<? extends E> c)

Inserts all of the elements in the specified collection into this list at the specified position (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices). The new elements will appear in this list in the order that they are returned by the specified collection's iterator. The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (Note that this will occur if the specified collection is this list, and it's nonempty.)

Parameters:
- index - index at which to insert the first element from the specified collection
- c - collection containing elements to be added to this list

Returns:
- true if this list changed as a result of the call

Throws:
- UnsupportedOperationException - if the addAll operation is not supported by this list
- ClassCastException - if the class of an element of the specified collection prevents it from being added to this list
- NullPointerException - if the specified collection contains one or more null elements and this list does not permit null elements, or if the specified collection is null
- IllegalArgumentException - if some property of an element of the specified collection prevents it from being added to this list
- IndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())

Data Structure Invariants (*cf.* 122)

```
struct list {
        elem data;
        struct list* next;
};
struct queue {
        list front;
        list back;
};
```

Data Structure Invariants (*cf.* 122)

```
struct list {
        elem data;
        struct list* next;
};
struct queue {
        list front;
        list back;
};
bool is_queue(queue q) {
        if (q == NULL) return false;
        if (q->front == NULL || q->back == NULL) return false;
        return is_segment(q->front, q->back);
}
```

Data Structure Invariants (*cf.* 122)

```
struct list {
        elem data;
        struct list* next;
};
struct queue {
        list front;
        list back;
};
bool is_queue(queue q) {
        if (q == NULL) return false;
        if (q->front == NULL || q->back == NULL) return false;
        return is_segment(q->front, q->back);
}
void enq(queue q, elem s)
//@requires is_queue(q);
//@ensures is_queue(q);
{
        list l = alloc(struct list);
        q->back->data = s;
        q->back->next = l;
        q->back = l; }
```

Data Structure Invariants (*cf.* 122)

- Properties of the data structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```
void enq(queue q, elem s)
//@requires is_queue(q);
//@ensures is_queue(q);
{ … }
```

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- May be invalidated temporarily during method execution

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- May be invalidated temporarily during method execution

```
public class SimpleSet {

    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { … }
}
```

```
public class SimpleSet {

    int contents[];
    int size;

    //@invariant sorted(contents);

    SimpleSet(int capacity) { … }

    boolean add(int i) { … }

    boolean contains(int i) { … }
}
```

Behavioral Subtyping (Liskov Substitution Principle)

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

- An object of a subclass should be substitutable for an object of its superclass
- Known already from types:
  - May use subclass instead of superclass
  - Subclass can add, but not remove methods
  - Overriding method must return same or subtype
  - Overriding method may not throw additional exceptions
- Applies more generally to behavior:
  - A subclass must fulfill all contracts that the superclass does
  - Same or stronger invariants
  - Same or **stronger** postconditions for all methods
  - Same or **weaker** preconditions for all methods

Behavioral Subtyping (Liskov Substitution Principle)

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

	//@ invariant speed < limit;








	//@ requires speed != 0;
	//@ ensures speed < \old(speed)
	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

Behavioral Subtyping (Liskov Substitution Principle)

```
class Car extends Vehicle {
        int fuel;
        boolean engineOn;
        //@ 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;

        //@ 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 break has stronger postcondition

Behavioral Subtyping (Liskov Substitution Principle)

```
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 Square a behavioral subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

```
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==w;
        Square(int w) {
            super(w, w);
        }
}
```

Is Square a behavioral subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

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

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

        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 Square a behavioral subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

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

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

        void scale(int factor) {
            w=w*factor;
            h=h*factor;
        }

        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 Square a behavioral subtype of Rectangle?

Behavioral Subtyping (Liskov Substitution Principle)

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

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

        void scale(int factor) {
            w=w*factor;
            h=h*factor;
        }

        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 factor) {
      r.setWidth(r.getWidth() * factor);
   }
}
```

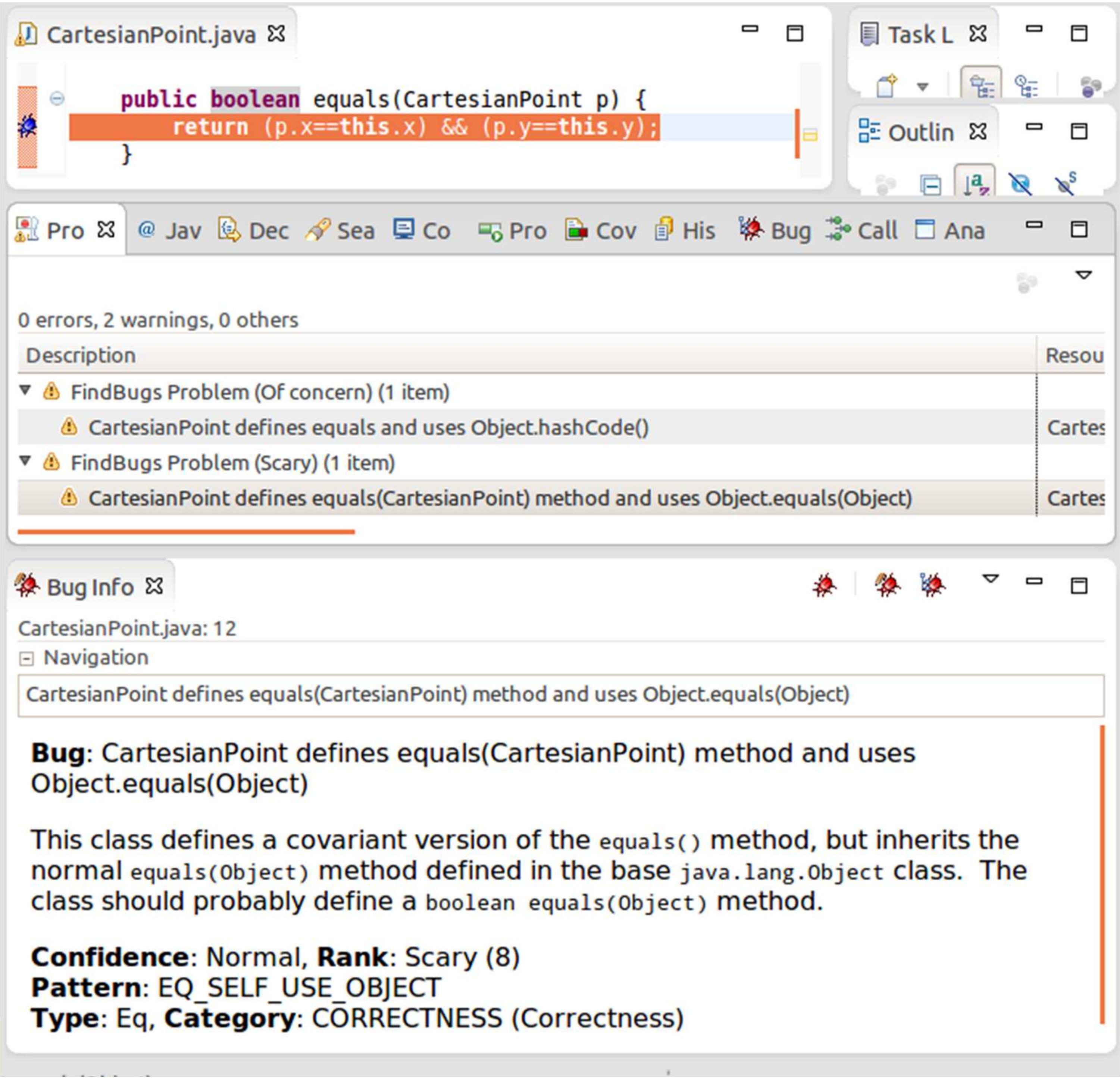
With these methods, Square is not a behavioral subtype of Rectangle

Static Analysis

Stupid Bugs

```
public class CartesianPoint {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    public boolean equals(CartesianPoint that) {
        return (this.getX()==that.getX()) &&
               (this.getY() == that.getY());
    }
}
```

FindBugs



```java
public boolean equals(CartesianPoint p) {
    return (p.x==this.x) && (p.y==this.y);
}
```

0 errors, 2 warnings, 0 others

Description	Resou
FindBugs Problem (Of concern) (1 item)	
CartesianPoint defines equals and uses Object.hashCode()	Cartes
FindBugs Problem (Scary) (1 item)	
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)	Cartes

CartesianPoint.java: 12

Navigation

CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)

Bug: CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)

This class defines a covariant version of the `equals()` method, but inherits the normal `equals(Object)` method defined in the base `java.lang.Object` class. The class should probably define a `boolean equals(Object)` method.

Confidence: Normal, **Rank**: Scary (8)
Pattern: EQ_SELF_USE_OBJECT
Type: Eq, **Category**: CORRECTNESS (Correctness)

~~Stupid~~ Subtle Bugs

```
public class Object {
    public boolean equals(Object other) { … }

    // other methods…
}

public class CartesianPoint extends Object {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    public boolean equals(CartesianPoint that) {
        return (this.getX()==that.getX()) &&
               (this.getY() == that.getY());
    }
}
```

classes with no explicit superclass implicitly extend `Object`

can't change argument type when overriding

This defines a different `equals` method, rather than overriding `Object.equals()`

Method dispatch

Example:

```
Object o1 = new CartesianPoint(1,2);
Object o2 = new CartesianPoint(1,2);
boolean iAmFalse = o1.equals(o2);
```

- Step 1 (compile time): determine what type to look in
  - Look at the static type (Object) of the receiver (o1)
- Step 2 (compile time): find the method in that type
  - Find the methods in the interface/class with the right name

    ```
    boolean equals(Object other);
    ```

  - Keep the method only if it is *accessible*
    - e.g. remove private methods
  - Keep the method only if the arguments are *applicable* to the actual argument types
  - Keep the method whose argument types are *most specific*
  - Error if there is no such method

Method dispatch

Example:

```
Object o1 = new CartesianPoint(1,2);
Object o2 = new CartesianPoint(1,2);
boolean iAmFalse = o1.equals(o2);
```

- Step 3 (run time): Determine the run-time type of the receiver
  - Look at the object in the heap and get its class
- Step 4 (run time): Locate the method implementation to invoke
  - Look in the class for an implementation of the method we found statically (step 2)
    - In this example we get the `equals` method inherited from `Object`
      - This method checks if they are the identical object: **false**
    - If we had overridden `equals` properly, we would get the version in `CartesianPoint` (and it would hopefully return **`true`**)
  - Invoke the method

Method dispatch

Example:

```
CartesianPoint o1 = new CartesianPoint(1,2);
CartesianPoint o2 = new CartesianPoint(1,2);
boolean iAmTrue = o1.equals(o2);
```

- Step 1 (compile time): determine what type to look in
  - Look at the static type (CartesianPoint) of the receiver (o1)
- Step 2 (compile time): find the method in that type
  - Find the methods in the interface/class with the right name

```
boolean equals(Object other);                // inherited
boolean equals(CartesianPoint other);        // most specific
```

  - Keep the method only if it is *accessible*
    - e.g. remove private methods
  - Keep the method only if the arguments are *applicable* to the actual argument types
  - Keep the method whose argument types are *most specific*
  - Error if there is no such method

Method dispatch

Example:

```
CartesianPoint o1 = new CartesianPoint(1,2);
CartesianPoint o2 = new CartesianPoint(1,2);
boolean iAmFalse = o1.equals(o2);
```

- Step 3 (run time): Determine the run-time type of the receiver
  - Look at the object in the heap and get its class
- Step 4 (run time): Locate the method implementation to invoke
  - Look in the class for an implementation of the method we found statically (step 2)
    - In this example we get the `equals` method from `CartesianPoint`
      - But this is fragile; it depends on the receiver and argument static types being right!
  - Invoke the method

Fixing the Bug

```
public class CartesianPoint {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof CartesianPoint)
            return false;

        CartesianPoint that = (CartesianPoint) o;

        return (this.getX()==that.getX()) &&
            (this.getY() == that.getY());
    }
}
```

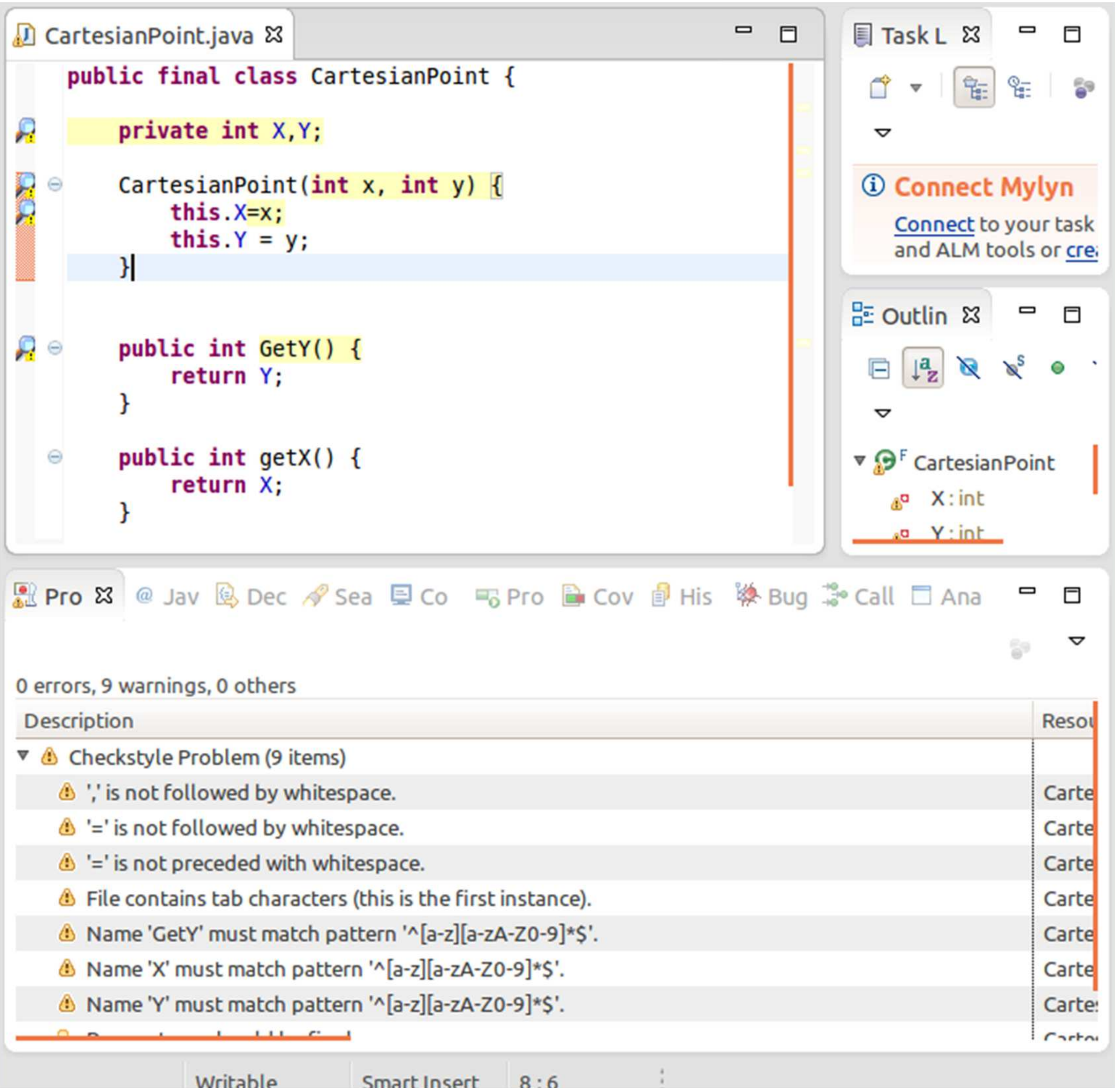
Declare our intent to override; Eclipse checks that we did it

Use the same argument type as the method we are overriding

Check if the argument is a CartesianPoint

Create a variable of the right type, initializing it with a cast

CheckStyle



```
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;
    }
```

0 errors, 9 warnings, 0 others

Description	Resou
Checkstyle Problem (9 items)	
',' is not followed by whitespace.	Carte
'=' is not followed by whitespace.	Carte
'=' is not preceded with whitespace.	Carte
File contains tab characters (this is the first instance).	Carte
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.	Carte
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.	Carte
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.	Carte

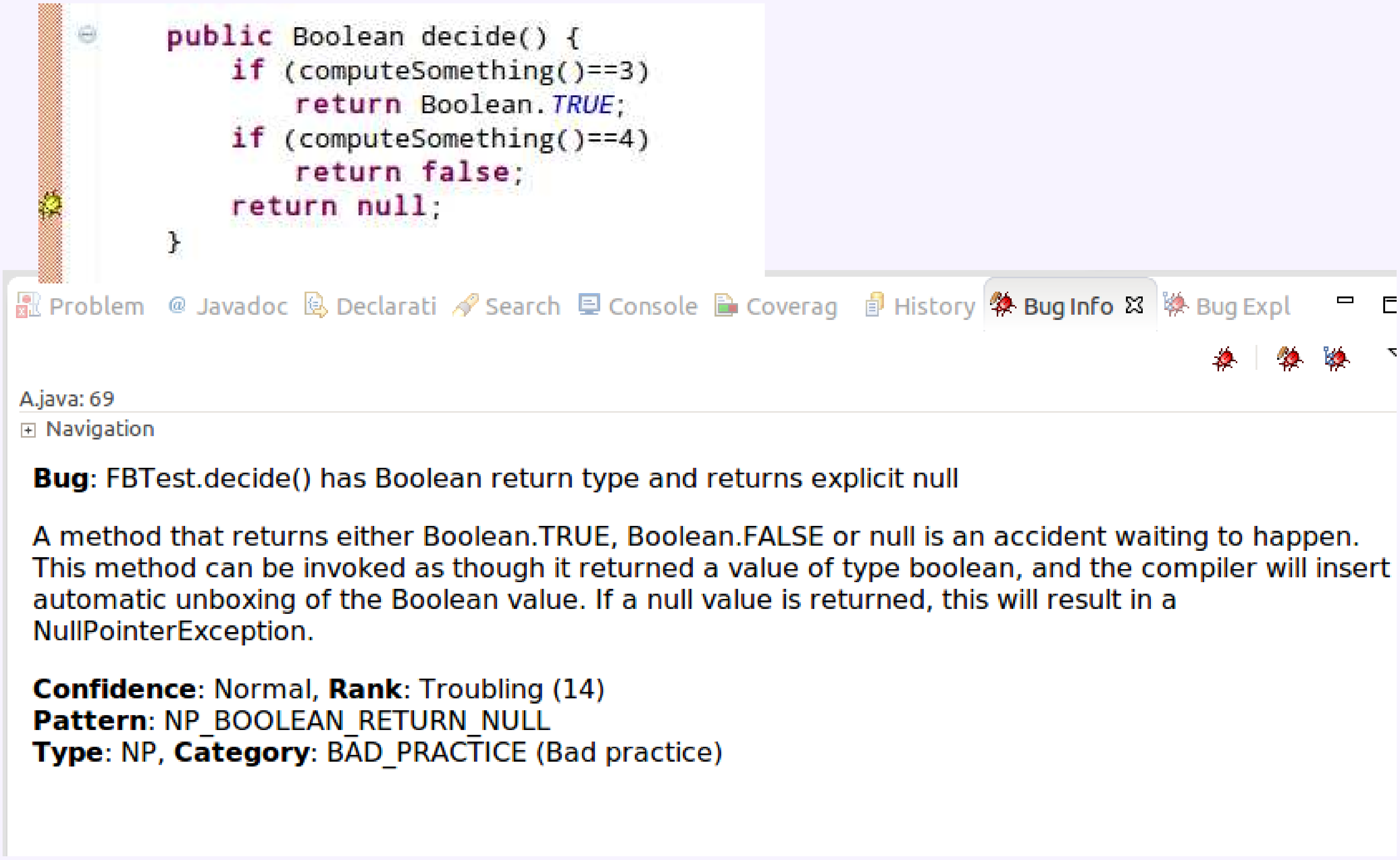
Static Analysis

- Analyzing code without executing it (automated inspection)
- Looks for bug patterns
- Attempts to formally verify specific aspects
- Point out typical bugs or style violations
  - NullPointerExceptions
  - Incorrect API use
  - Forgetting to close a file/connection
  - Concurrency issues
  - And many, many more (over 250 in FindBugs)
- Integrated into IDE or build process
- FindBugs and CheckStyle open source, many commercial products exist

Example FindBugs Bug Patterns

- Correct equals()
- Use of ==
- Closing streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation problems
- Inconsistent synchronization
- Inefficient String use
- Dead store to variable

Bug finding



```
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}
```

A.java: 69

Navigation

Bug: FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

Confidence: Normal, **Rank**: Troubling (14)
Pattern: NP_BOOLEAN_RETURN_NULL
Type: NP, **Category**: BAD_PRACTICE (Bad practice)

Abstract Interpretation

- Static program analysis is the **systematic examination** of an **abstraction of a program's state space**
- Abstraction
  - Don't track everything! (That's normal interpretation)
  - Track an important abstraction
- Systematic
  - Ensure everything is checked in the same way

Details on how this works in 15-313

Toad’s Take-Home Messages

- Exceptions help with remembering errors, signaling them across function boundaries
- Class invariants check the representation of a class
- Behavioral subtyping relates the specification of a subclass to its superclass
- Static analysis tools can help find bugs, check code style