Debugging

CSE 2231 Supplement A
Annatala Wolf

Testing is not debugging!

- The purpose of testing is to detect the *existence* of errors, *not* to identify precisely where the errors came from.
- Error messages may be unhelpful or even deceptive.
- This is especially true with "kernel" implementations.

The only way to isolate and fix errors is by using a *systematic strategy* to locate mistakes in code. This process of "detective work" is known as **debugging**, and it is the vast majority of what we do in software engineering.

Curiosity is not a good debugging strategy!

- If you're new to debugging, you will be tempted to investigate bizarre output.

"Why did my Word Counter program just tell me that jet fuel can't melt steel beams?"

- This is a natural reaction, but it's a bad approach.
- It might seem strange, but it is 100x easier to *fix* a program than to figure out *why* it broke!

Before you begin...

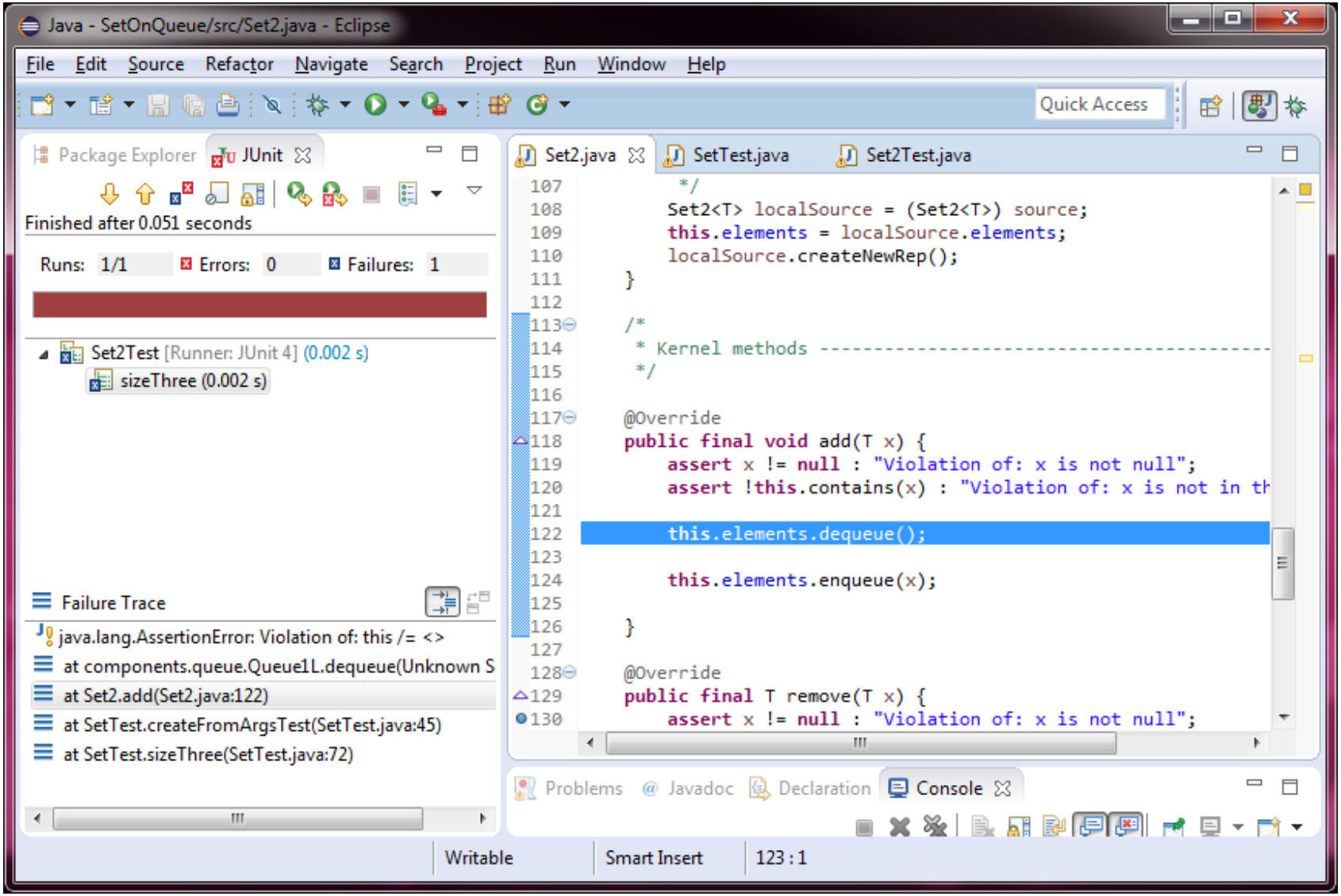
- Start with the **simplest-looking test case that fails**.
- Then, reduce that case to a **minimal example that fails**.
  - The simpler the thing that fails, the more luck you'll have finding the problem.
  - You can do this very quickly (logarithmic time) with a binary search. (As in, remove half the problem; if that removes the bug, add back half what you removed...)

Warning! Debugging in "kernel" implementations...

- It can be *very* tricky to debug components with a data rep.
- If `toString()` or `equals()` is called, what happens?
  - Those methods are written in the *abstract class*, remember?
  - Those methods call *your* methods: *the same code you're testing!*
  - Testing with a reference class uses `assertEquals()`, which calls `equals()`. This means it is *pointless* to test kernel code until you have written *every* method.
- It's also more *dangerous* to debug kernels: if you accidentally `toString()` `this` while debugging, it can ruin your test...

Assertions: The Programmer's Friend

- If you see a **violated assertion** that isn't from the JUnit test case itself (like, it isn't one of the `assertEquals()` calls), this is usually a *good* sign!
- It means that somewhere in your code, you called a method without satisfying the precondition first.
- It also tells you the *exact* place in code where the bad call was made (it will appear *one line down* in the stack trace.)
- These tend to be the easiest errors to locate, which is why we use assertions to check preconditions during testing.
- In some cases it won't help much, though. For example, the assertion might have failed because you did something to corrupt the data representation in a different method.

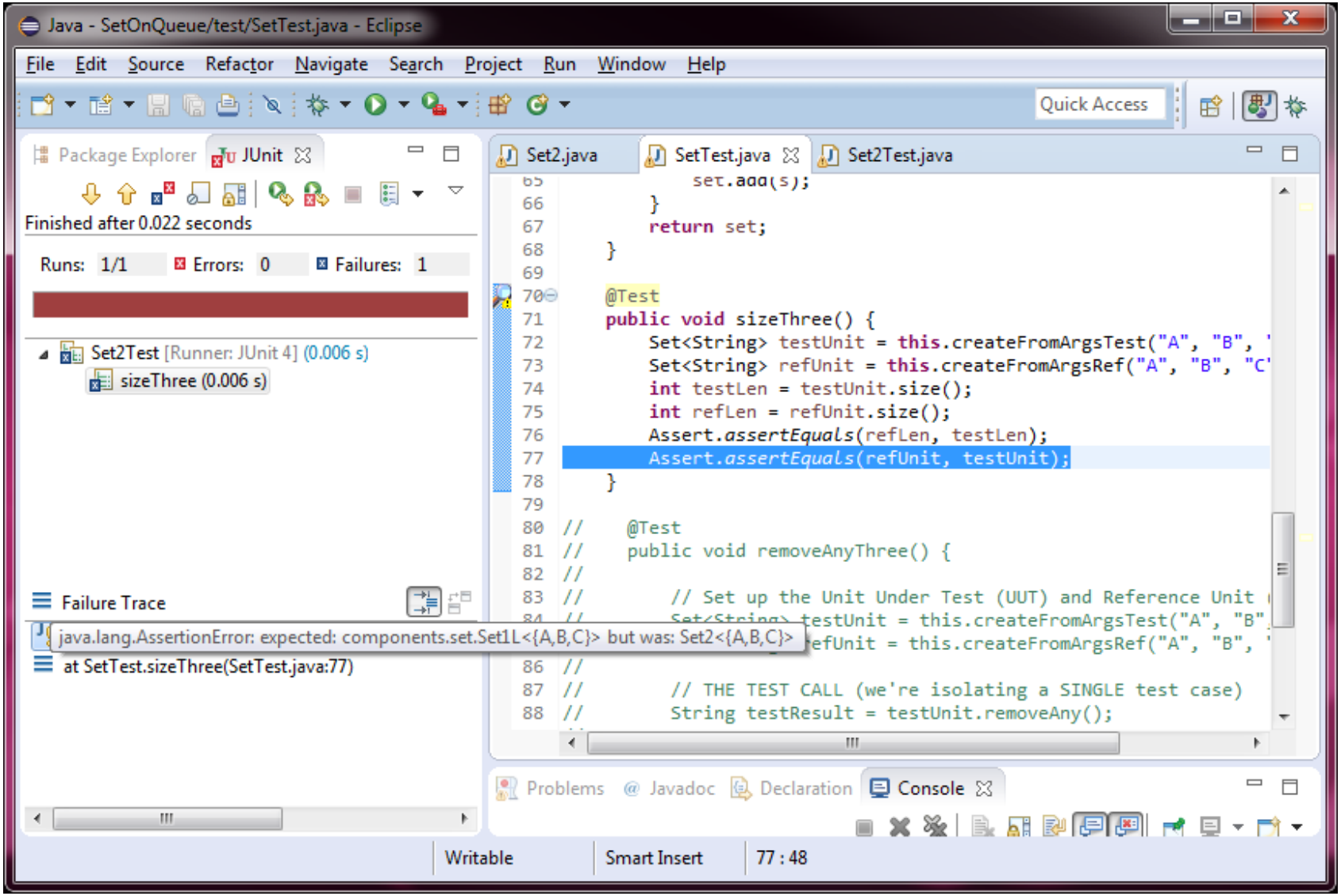


The failure **stack trace** is on the left. It shows us **`dequeue()`** was called incorrectly by **`set.add()`** on line **122** (highlighted). The assertion was "**`this /= <>`**", meaning the Queue was *supposed to* not-equal empty string. **In other words, line 122 calls dequeue on an empty Queue.**

What if `assertEquals()` messages make no sense?

Example on next slide: equals() detected an error (it returned false), but toString() afterwards doesn't see the error.

Several possible ways this can happen but it means you have an error and why it's an error is not revealed by testing: now use *debugging* strategies



If you see: expected: Set1L<{A, B, C}> but was: Set2<{B, A, C}>, that's the same problem. Those *should* return equal. toString() is not an equality test and is *never* a substitute for equals()!

Setting up a test properly for reference unit testing

Set up test object

Set up ref object identically

MAKE SINGLE CALL TO METHOD BEING TESTED IN UUT
(exception: exhaustive tests, like you'll need to do for Project 5)

Make identical call to RU (exception: next slide)

Test return values for equality (if applicable)

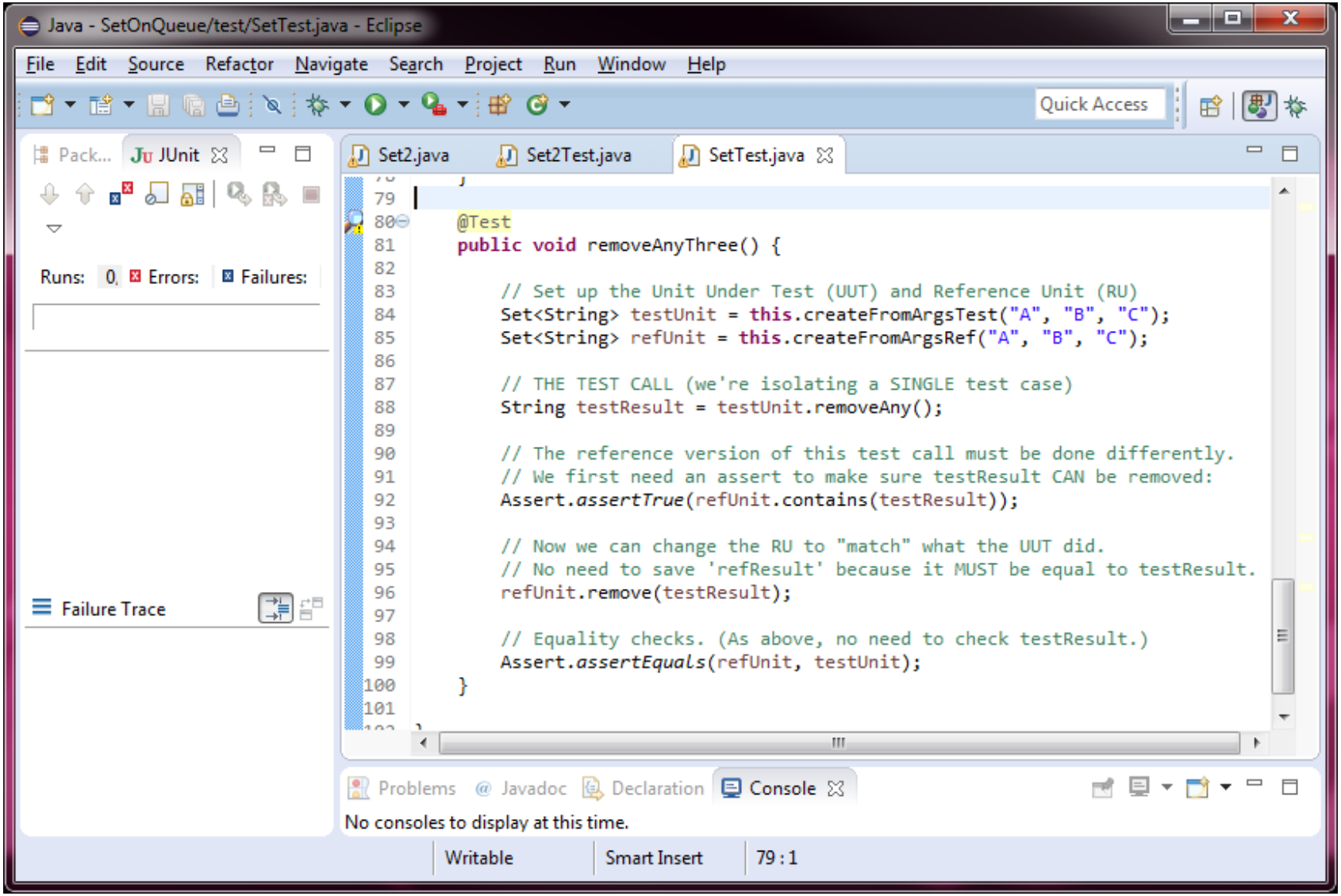
Test UUT and RU for equality (always do this!) (important to do in right order)

Exceptions

exhaustive unit tests

NOTE: you're testing a lot of stuff, but the reason you only make a single call to the test method (except with exhaustive unit tests) is you want to be certain you have isolated that specific test; it's fine to overtest, remember testing /= debugging

tests where results aren't fully deterministic (next slide)



Testing methods with indefinite behavior can be a little challenging. For `removeAny()`, we need an intermediate check to safely "match" the call we made on the UUT to the RU (by calling `remove()` *instead*).

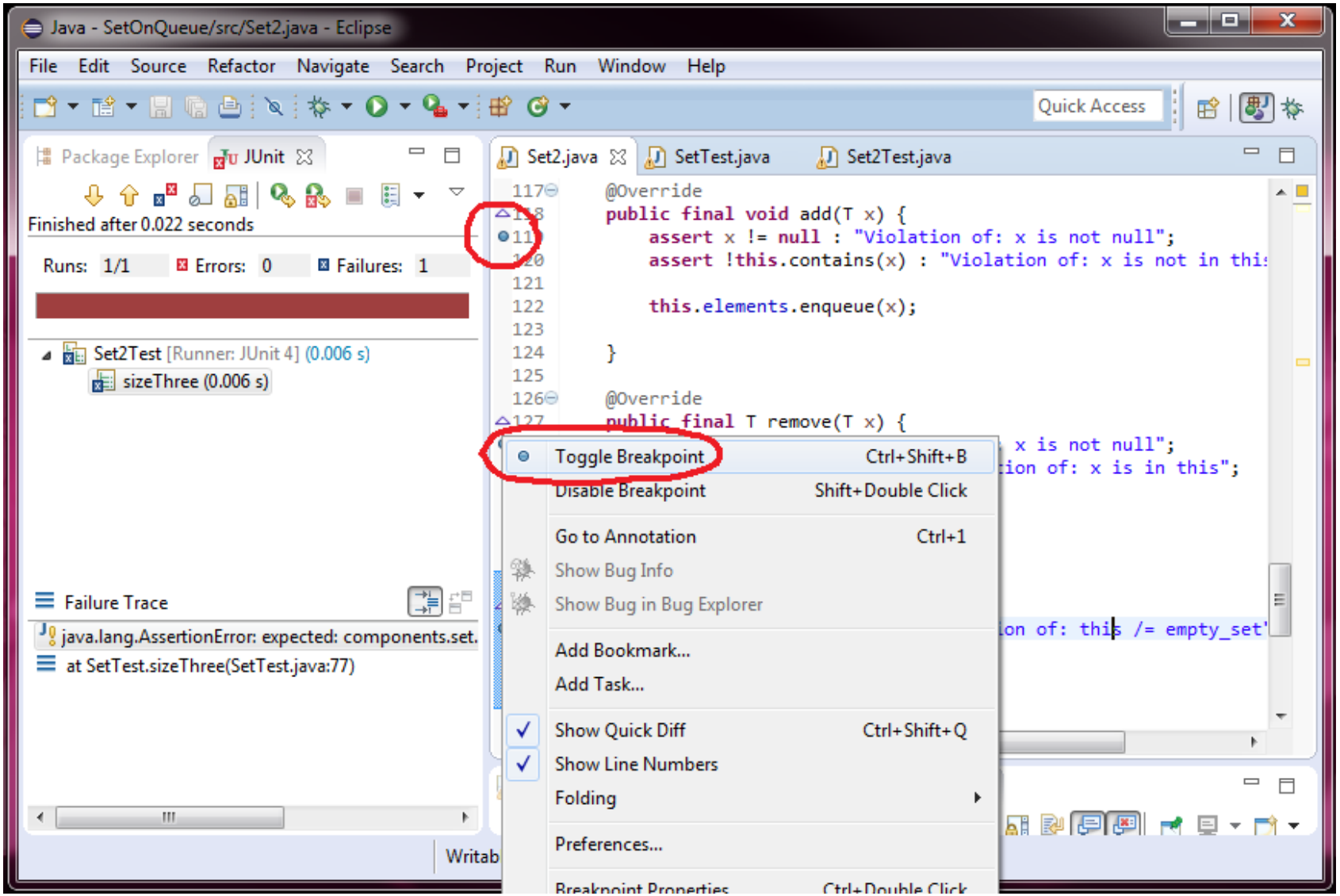
Where to begin?

- You can't test your kernel until you've written all of the methods because it can fail when you try to **`toString()`** or **`equals()`** the test object, and all JUnit tests do these
- You don't want to start tracing into your code at the beginning of your test cases unless that's where it broke
  - You should trace right before the error was *detected*, which often means tracing through **`equals()`**
- You don't have code for **`equals()`**, but can trace in by:
  - Putting breakpoints into the kernel methods
  - Cleverly turning breakpoints on/off at the right time

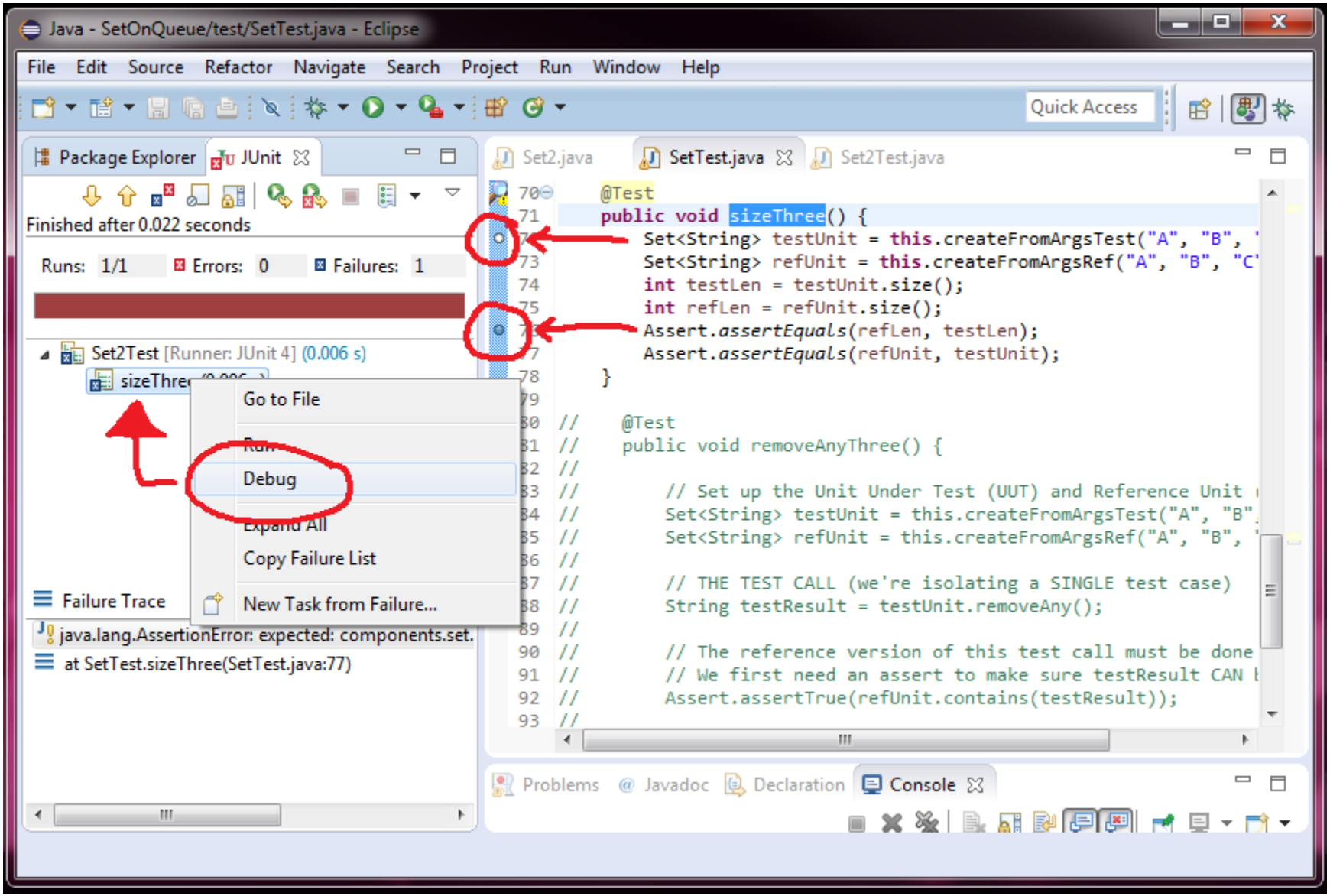
Identify where things first might have gone wrong...

Three potential places: object setup, test call, assertEquals()

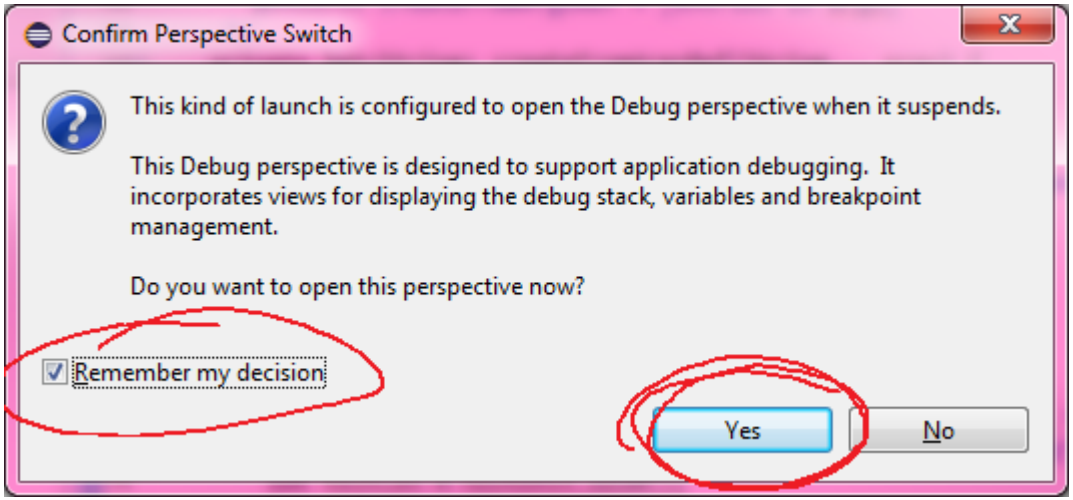
Start with the last one and work your way backwards. Let the test run up to assertEquals() before you pause. Check to see if the object is already messed up. If it isn't, this is where you start. If it is, stop the test and back up to the test call.



Before doing any debugging in a kernel, breakpoint the beginning of *each* kernel method



Right-click to run a single test case in debug mode



You need to use the debugger to have any hope of writing code that works. Running a debug should automatically open the Debug perspective (a bunch of debug windows). If you ever screw up Debug perspective you can reset the perspective from the menu.

When I see this pop-up it usually tells me you have no clue what you're doing because you never use the debugger! :V

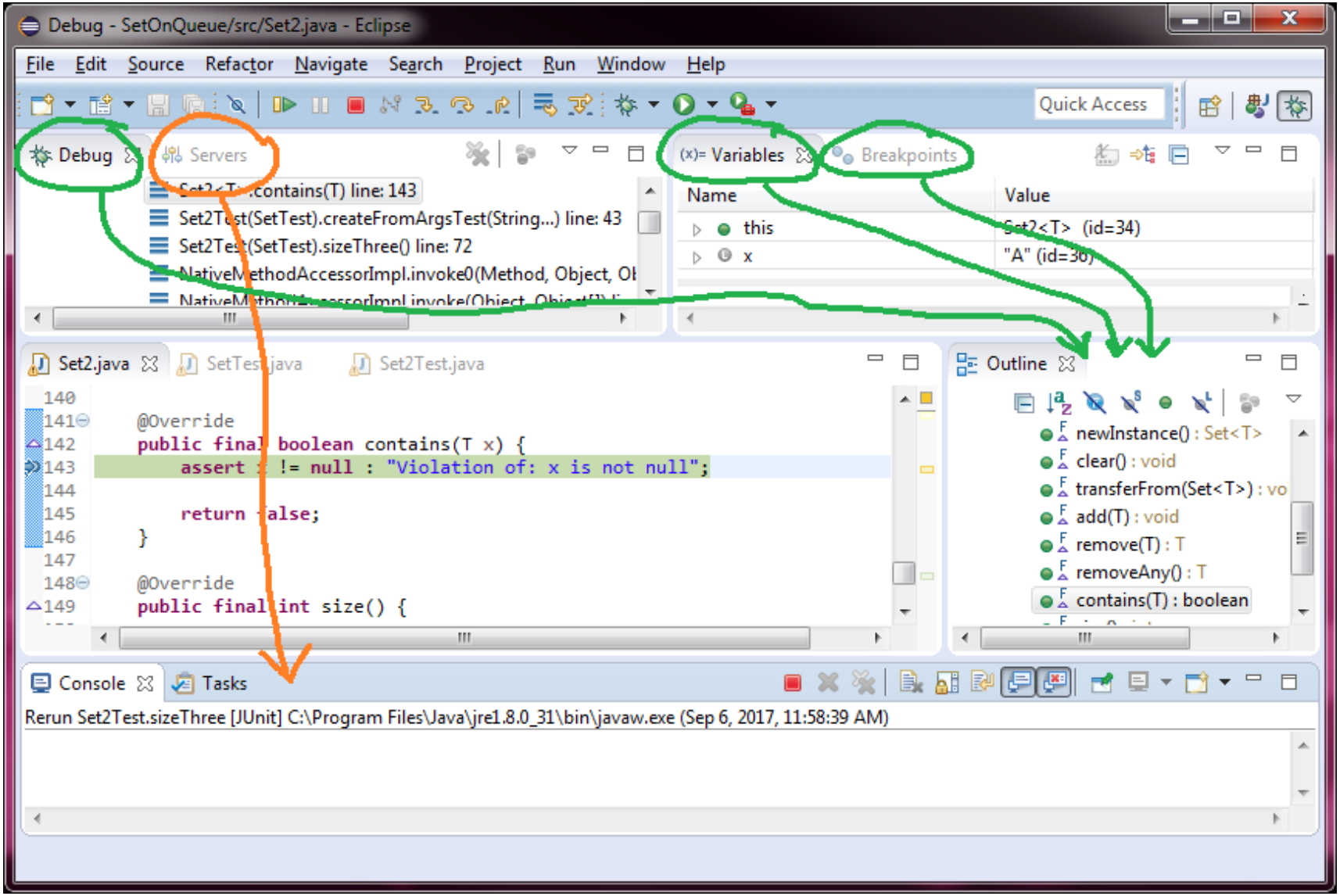
Working with Eclipse Perspectives

You'll mainly be using Java and Debug perspectives (well, also version control perspectives)

View → Perspective → Reset Perspective when you need to fix things (like if you accidentally remove a window)

You can modify perspectives there too

Play around, use Google, etc.

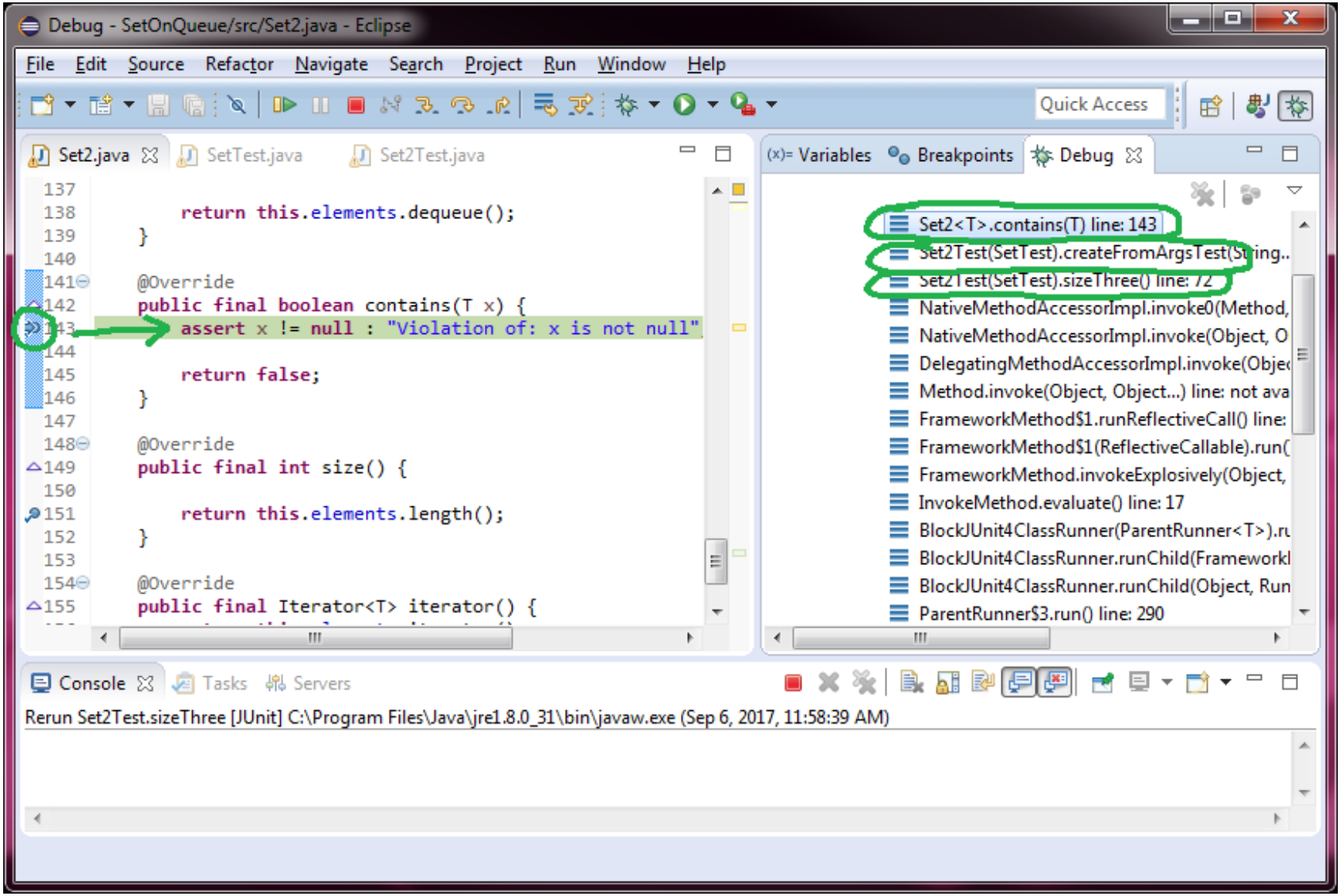


I've never used an editor whose default setup is such a mess. My suggestions are above.

Debugging Panes

Outline: it's bookmarks that jump you to places in the program; I rarely use this in debug because I almost always want to stay where the action is (the line being run)

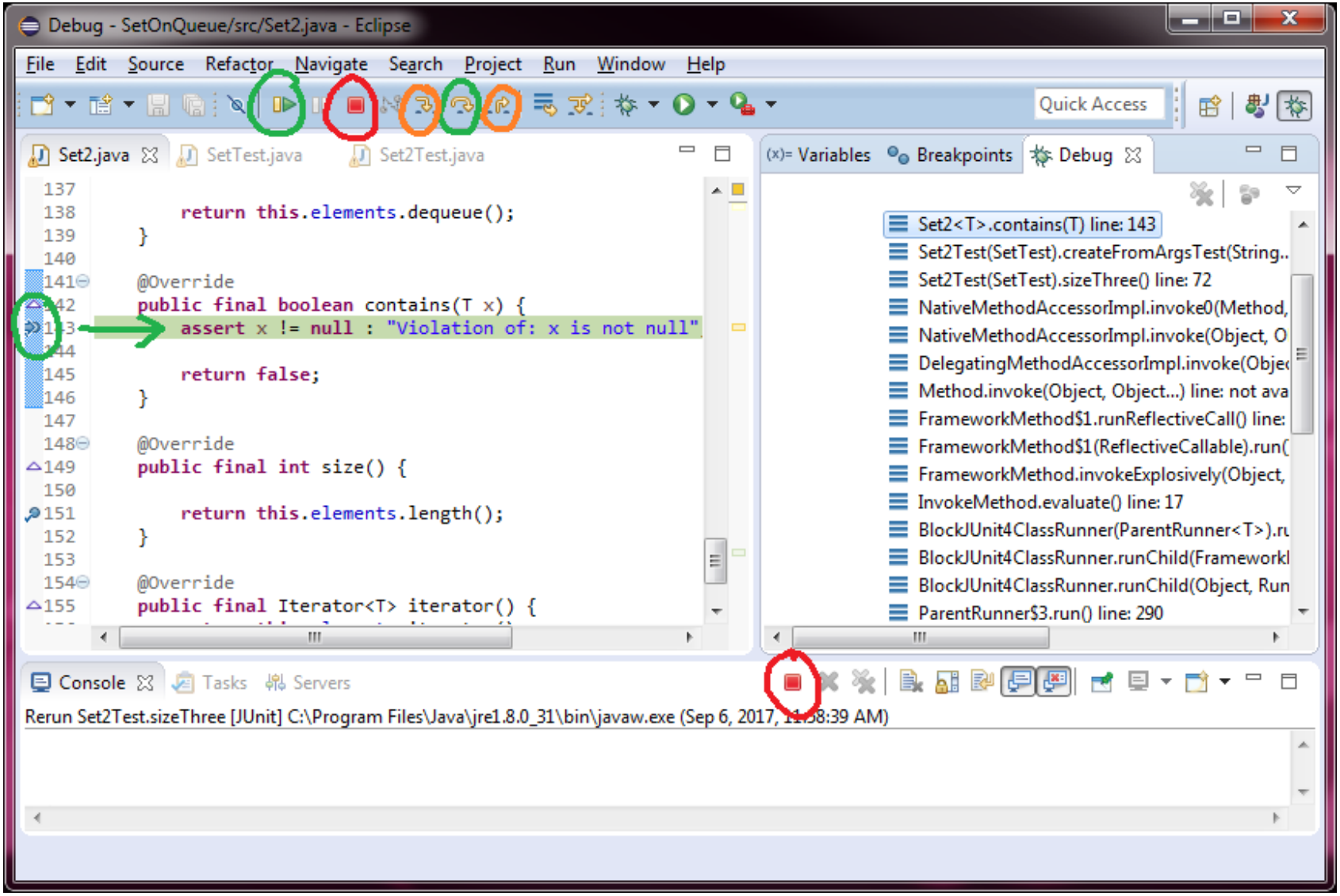
add stuff about other panes (Debug, Breakpoints, Variables), kind of obvious from upcoming slides though

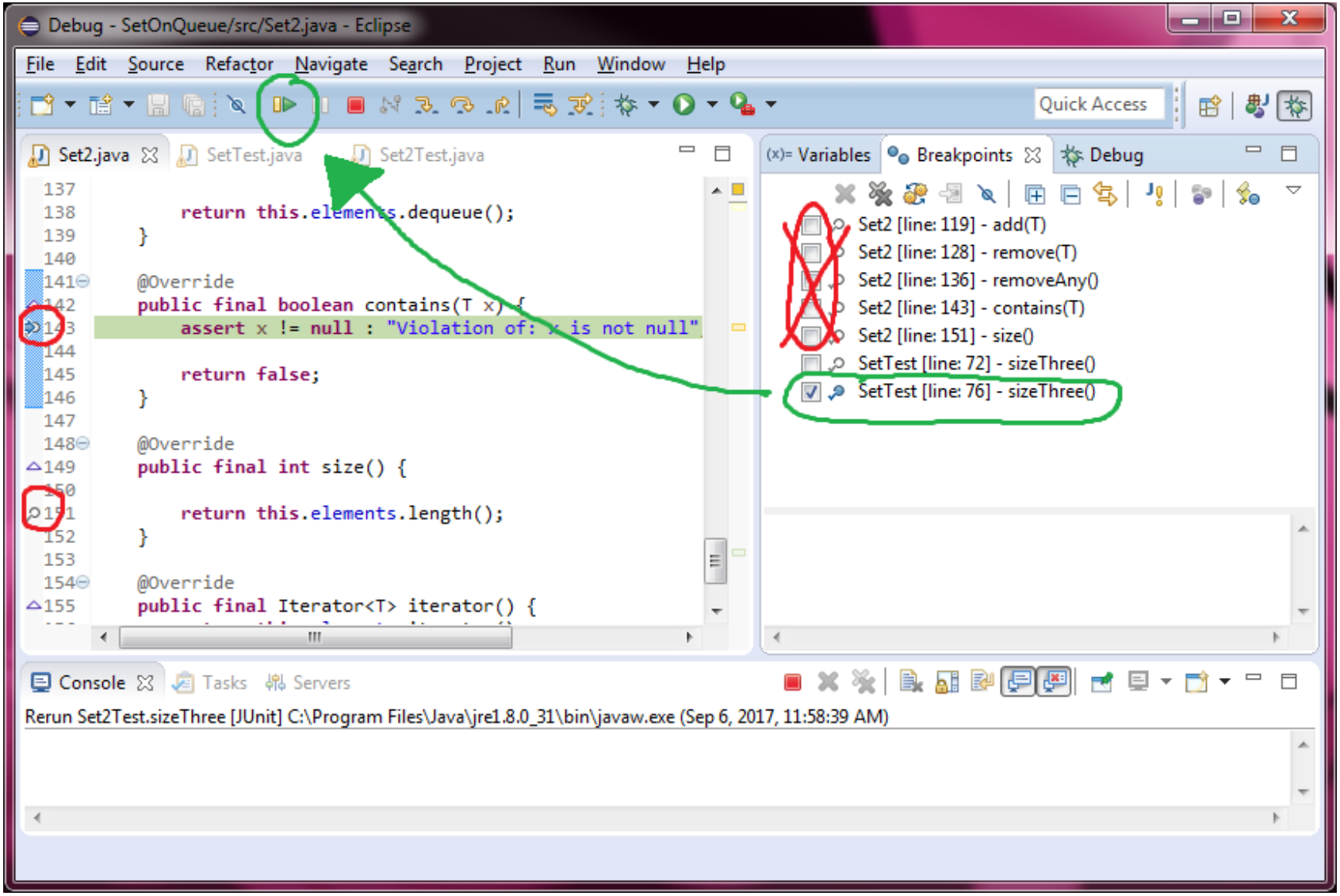


Much better! To the right is the stack trace. Note we're currently in **`createFromArgsTest()`**, which is earlier than we wanted to start debugging.

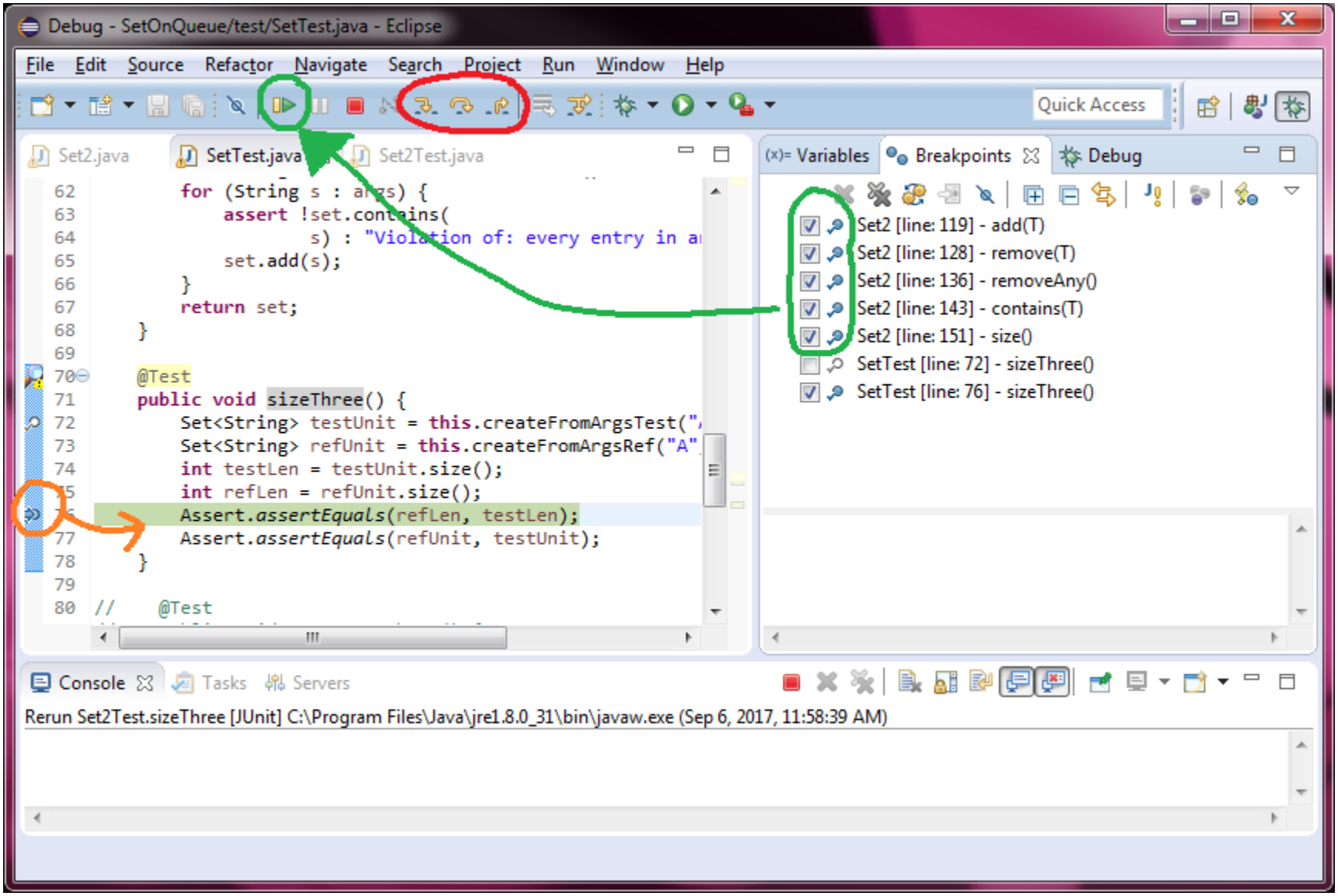
Debugging Controls

- When you run debug mode *nothing will happen unless you remembered to turn on at least one breakpoint*
- **Play** skips to the next breakpoint
- **Stop** quits (this helps if you're in an infinite loop)
- **Step into** follows the next method call into method you're calling; **won't work if you don't have access to code**
- **Step over** is your friend!
    - It's usually what you usually want
    - Runs one line of code so you can see variable changes
- **Step out** skips to the end of the method (use this if you accidentally step into code that you don't have)

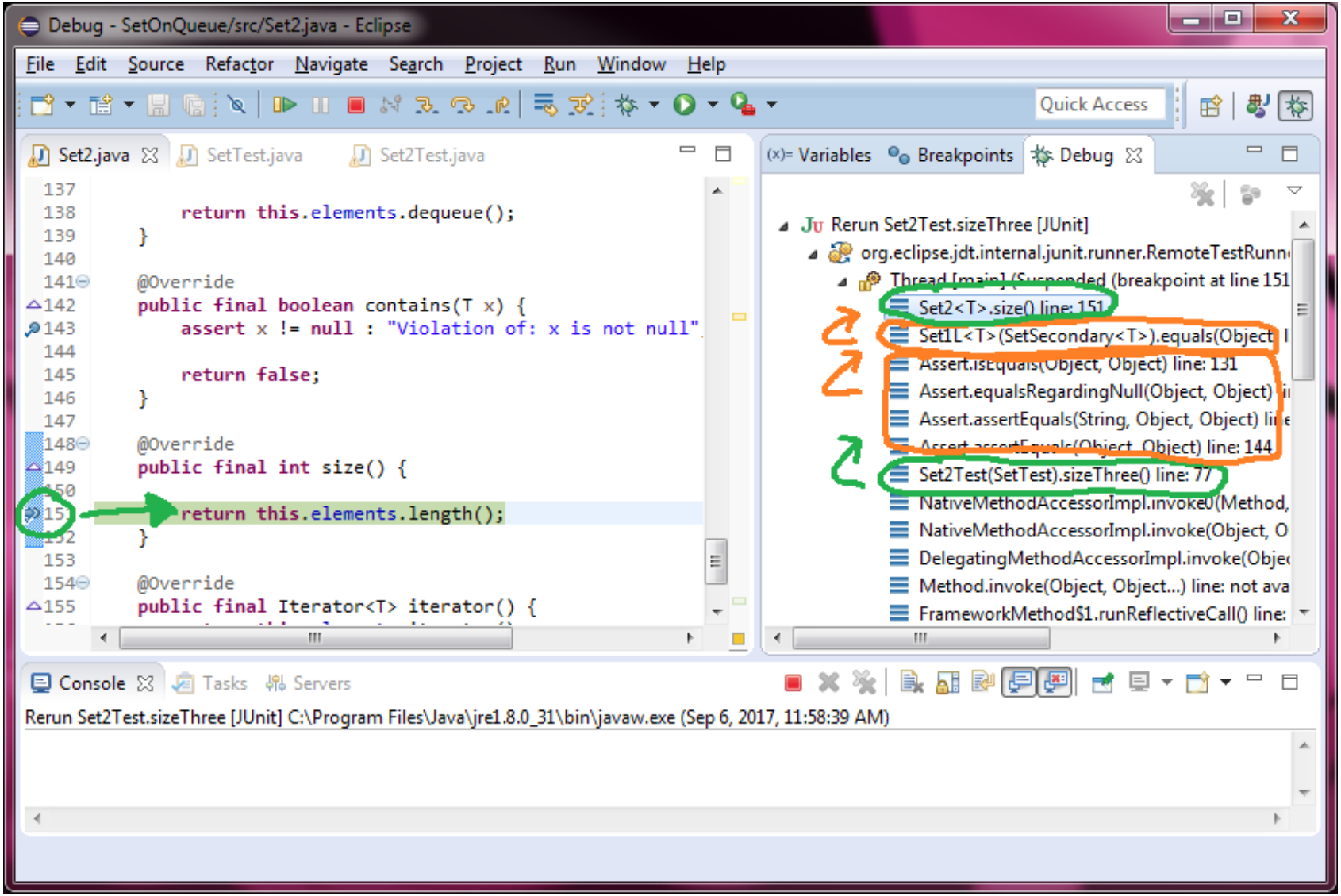

Debug - SetOnQueue/src/Set2.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Set2.java
SetTest.java
Set2Test.java
137
138 return this.elements.dequeue();
139 }
140
141 @Override
142 public final boolean contains(T x) {
143 assert x != null : "Violation of: x is not null";
144
145 return false;
146 }
147
148 @Override
149 public final int size() {
150
151 return this.elements.length();
152 }
153
154 @Override
155 public final Iterator<T> iterator() {
(x)= Variables
Breakpoints
Debug
Set2<T>.contains(T) line: 143
Set2Test(SetTest).createFromArgsTest(String..
Set2Test(SetTest).sizeThree() line: 72
NativeMethodAccessorImpl.invoke0(Method,
NativeMethodAccessorImpl.invoke(Object, O
DelegatingMethodAccessorImpl.invoke(Objec
Method.invoke(Object, Object...) line: not ava
FrameworkMethod$1.runReflectiveCall() line:
FrameworkMethod$1(ReflectiveCallable).run(
FrameworkMethod.invokeExplosively(Object,
InvokeMethod.evaluate() line: 17
BlockJUnit4ClassRunner(ParentRunner<T>).ru
BlockJUnit4ClassRunner.runChild(Frameworkl
BlockJUnit4ClassRunner.runChild(Object, Run
ParentRunner$3.run() line: 290
Console
Tasks
Servers
Rerun Set2Test.sizeThree [JUnit] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (Sep 6, 2017, 11:58:39 AM)



Let's skip to the breakpoint in our JUnit test by turning off the kernel breakpoints for now



We can't step into assertEquals, but we can turn the kernel breakpoints back on and hit play



Okay, we're currently in our test object, on which equals() in SetSecondary is calling size()

In what order are your methods called...?

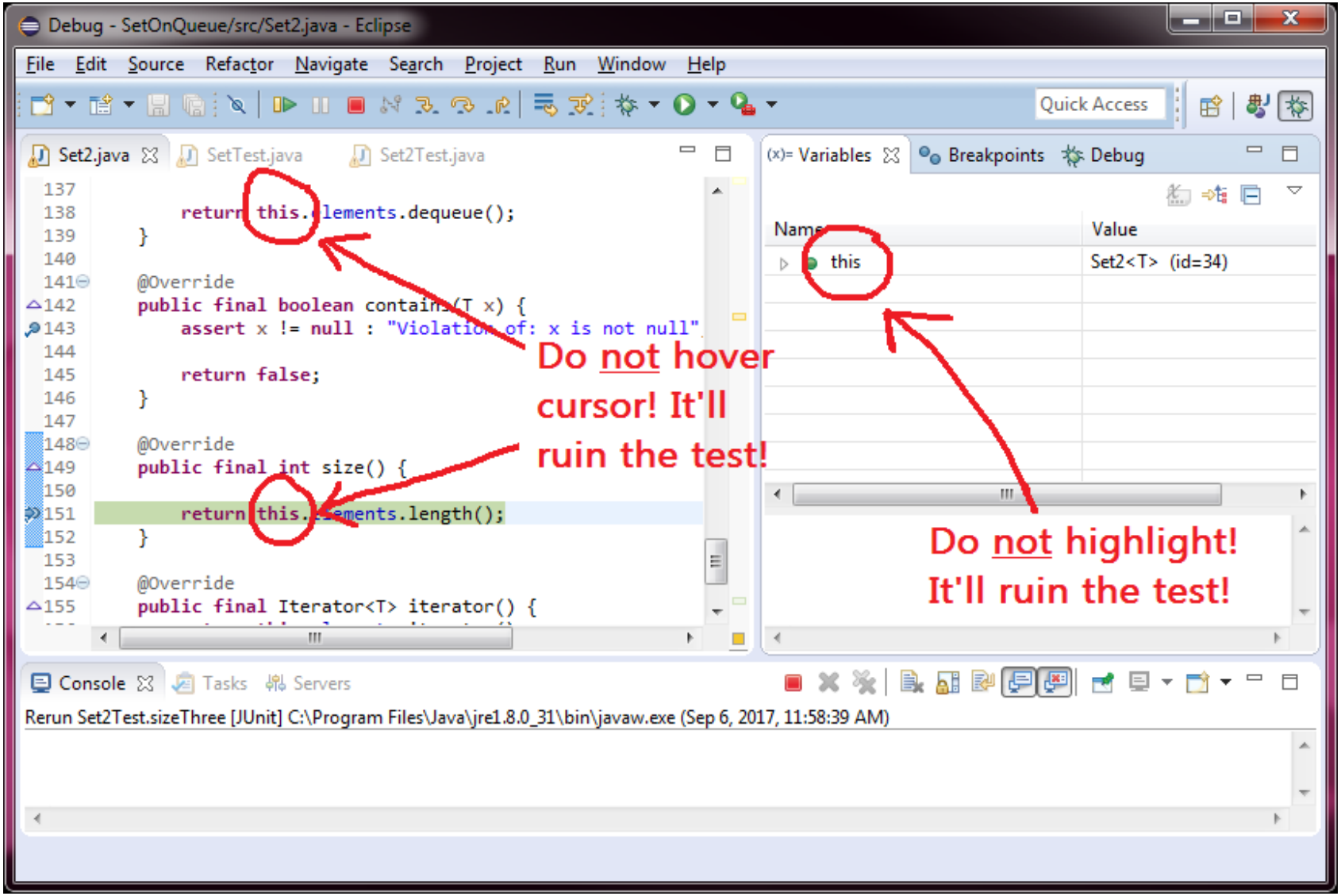
You *don't* know the order the methods will be called in, but you can examine the initial/final states of variables each time a method runs to find the error)

The asserts inside kernel methods can call additional kernel methods (not a violation of KPR, because it's an assert, and it happens before method begins running)

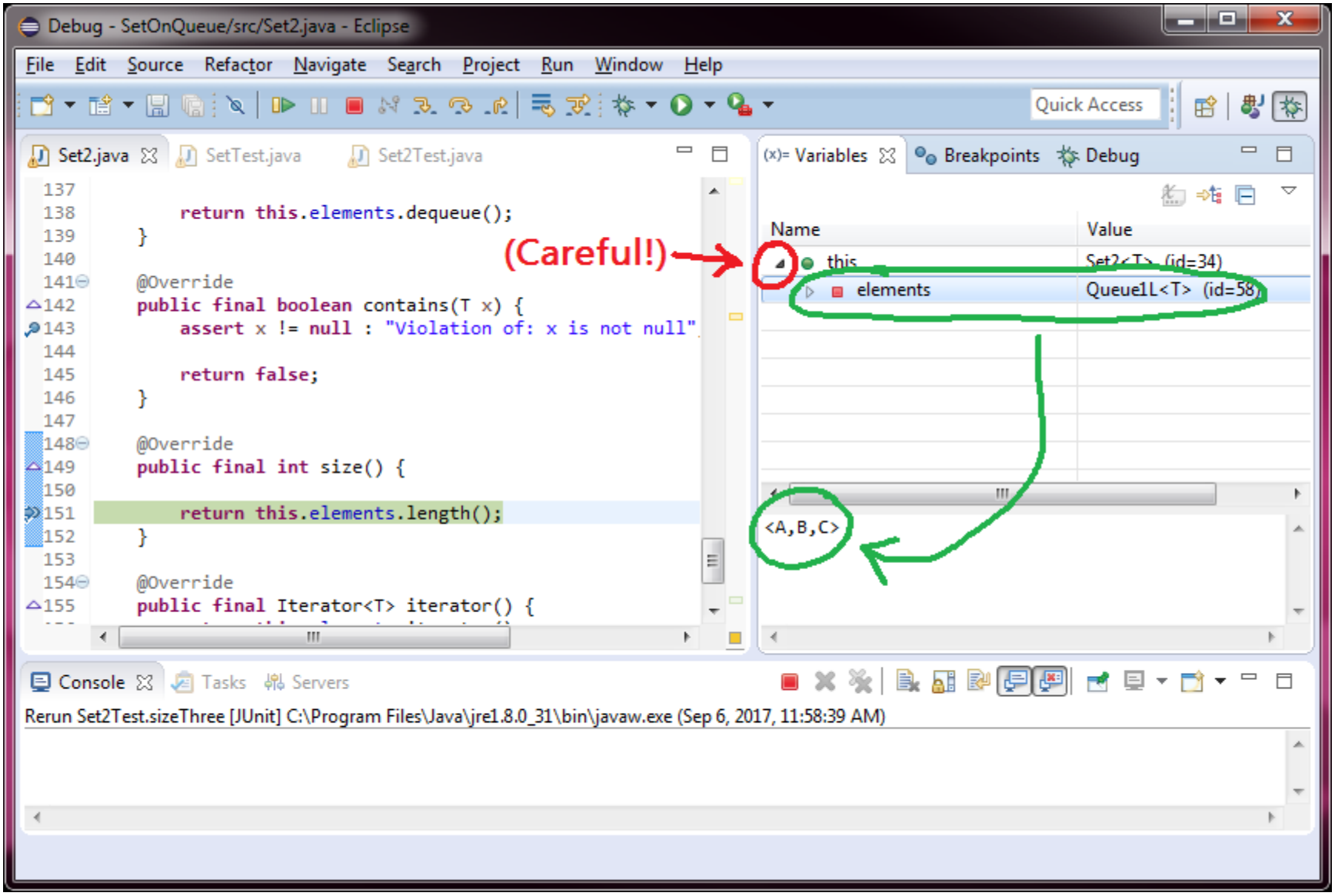
Why debugging in a kernel is dangerous

- Anytime `equals()` or `toString()` is called, it's going to automatically test many of the test unit's methods
  - `toString()` gets called **DURING** debugging if you:
    - **hover your cursor over `this`** (in the kernel)
    - **highlight `this` in the variables window**

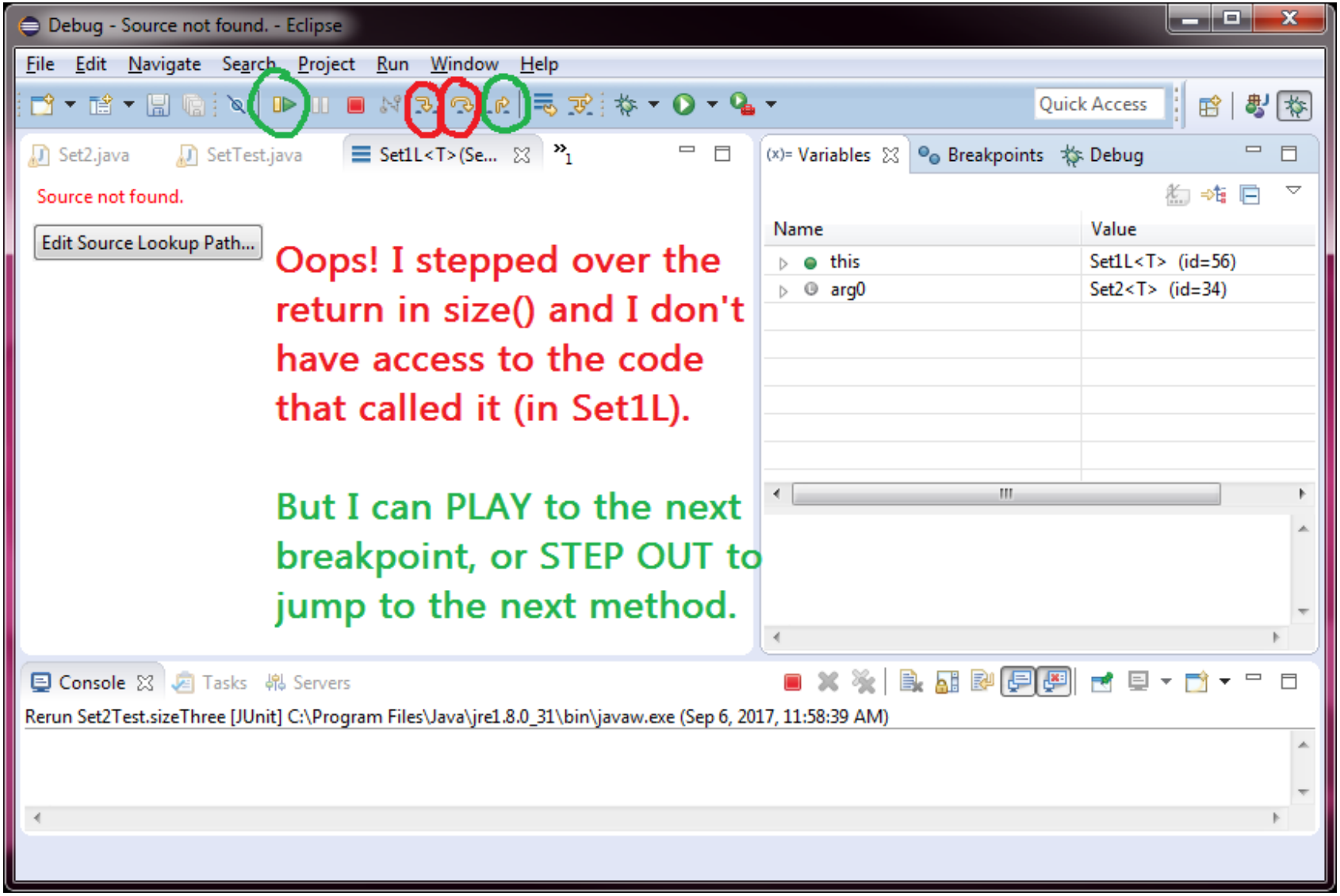
- The way to prevent this is by *very* carefully clicking the arrow to open up the object in the variables window *without* highlighting it.
  - It's fine to look at the *fields* of the object.



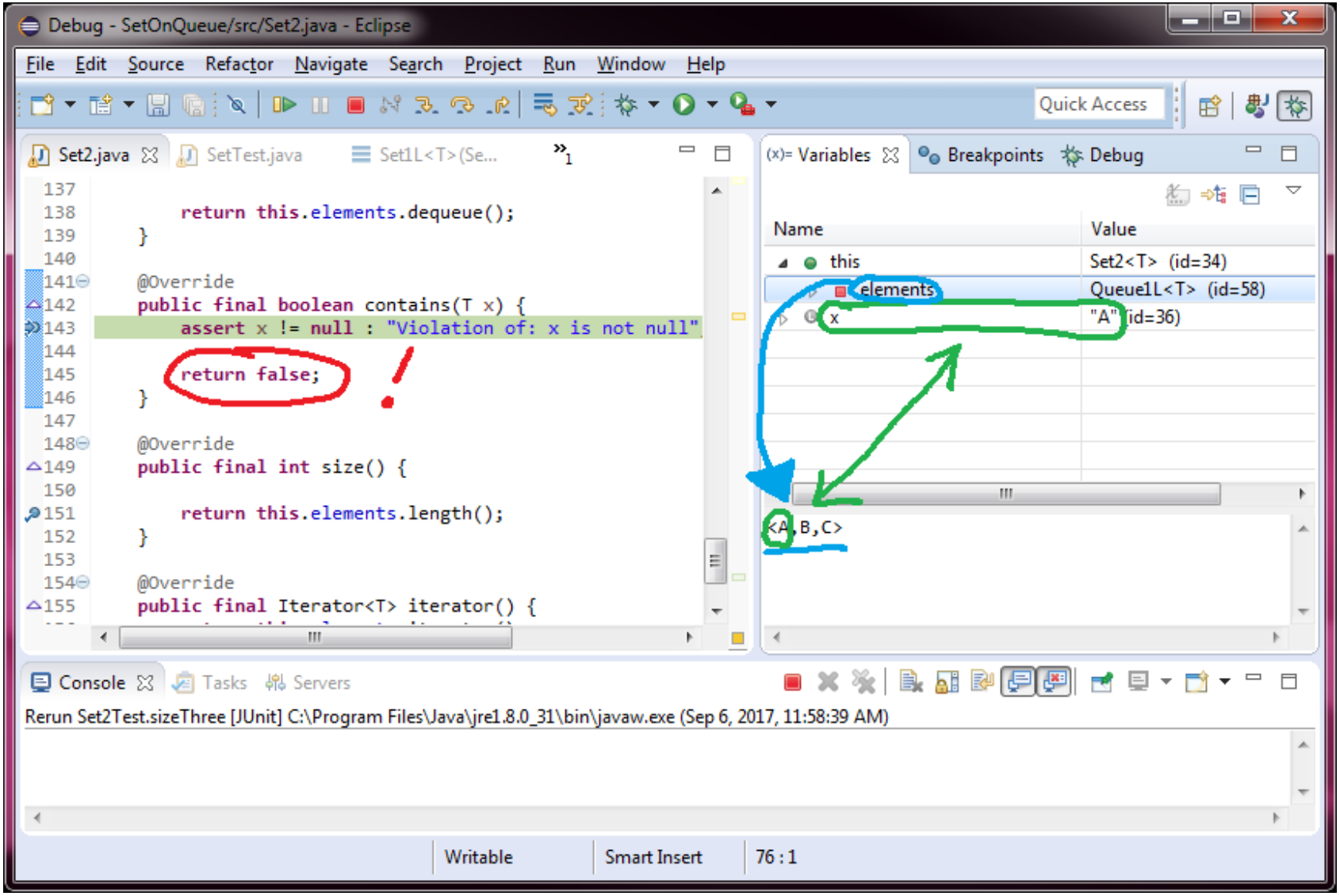
If you hover over variable **this** or highlight it in the vars window, you'll toString() it which will call all your kernel methods; OOPS



But if you click the arrow, you can safely look at the fields of **this**



Here's what happens if you accidentally go somewhere where you don't have code (I stepped over the return in size())



Find the first error, fix, then rerun tests

Baby Steps

- In debugging you want to find and correct the **first error that you see**. As soon as you've found an error, correct that error *only*, then run that same test case again.
- Don't try to *find* more errors until you've rerun your tests (but you can correct any errors you see).

> Even if the error you found couldn't possibly fix the test case you're debugging, the case could run differently without that error. Simple errors can *mask* more complicated errors.