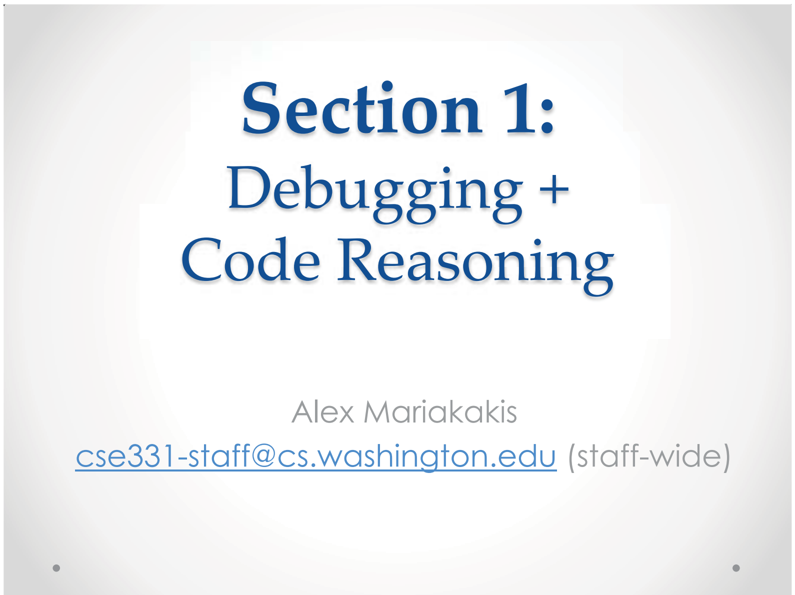
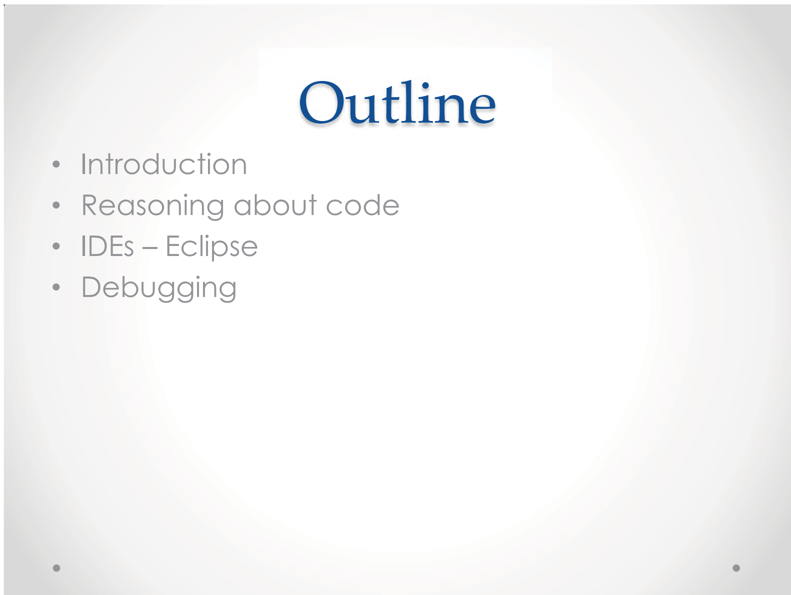


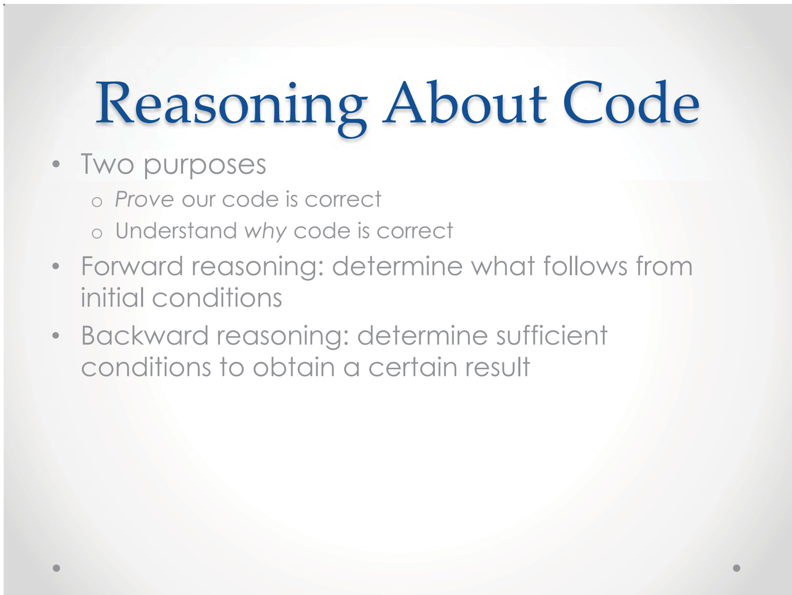
Section 1:
Debugging + Code Reasoning

Alex Mariakakis
cse331-staff@cs.washington.edu (staff-wide)



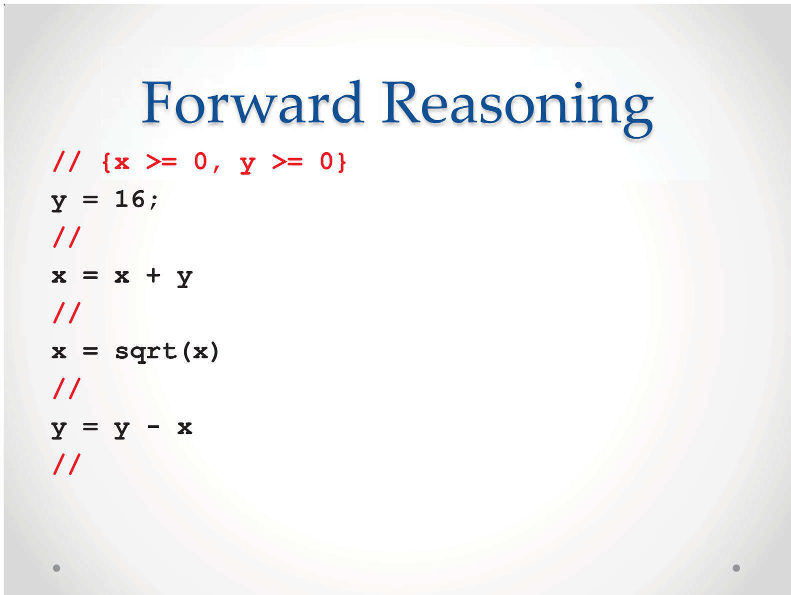
Outline

- Introduction
- Reasoning about code
- IDEs – Eclipse
- Debugging



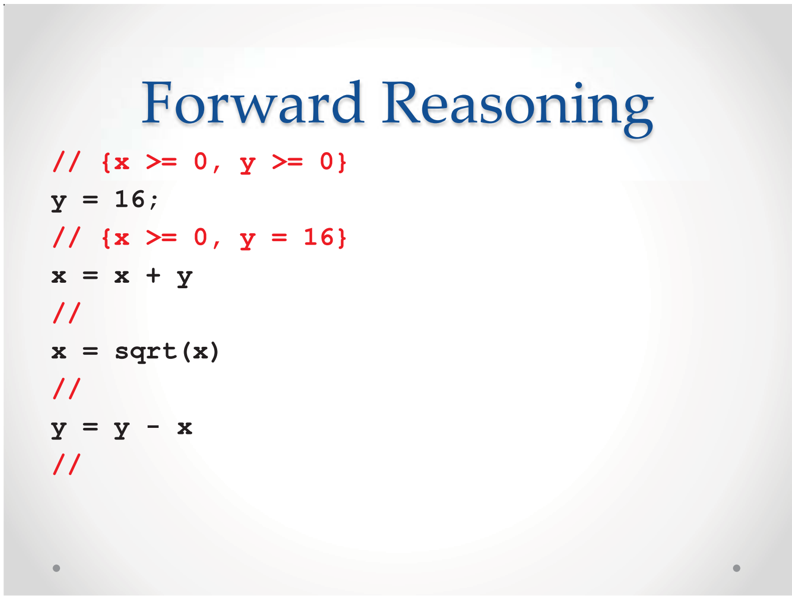
Reasoning About Code

- Two purposes
  - *Prove* our code is correct
  - Understand *why* code is correct
- Forward reasoning: determine what follows from initial conditions
- Backward reasoning: determine sufficient conditions to obtain a certain result



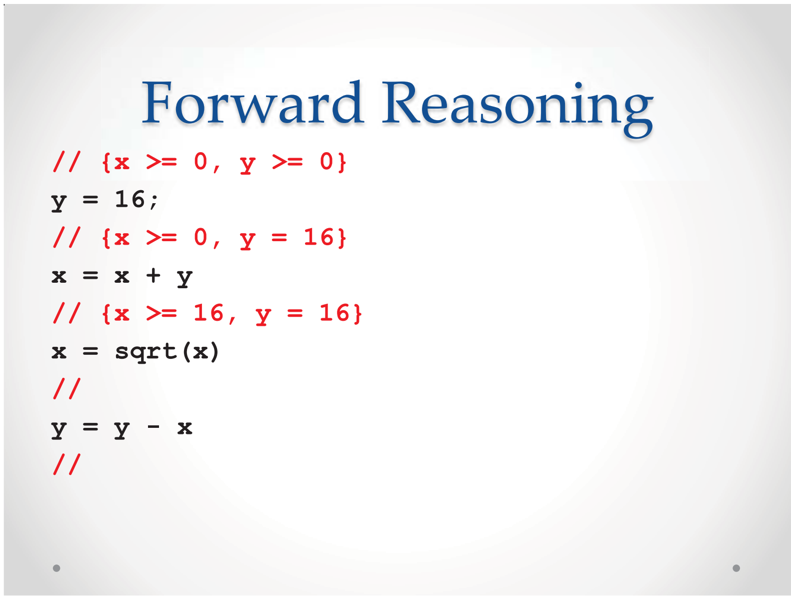
Forward Reasoning

```
// {x >= 0, y >= 0}
y = 16;
//
x = x + y
//
x = sqrt(x)
//
y = y - x
//
```



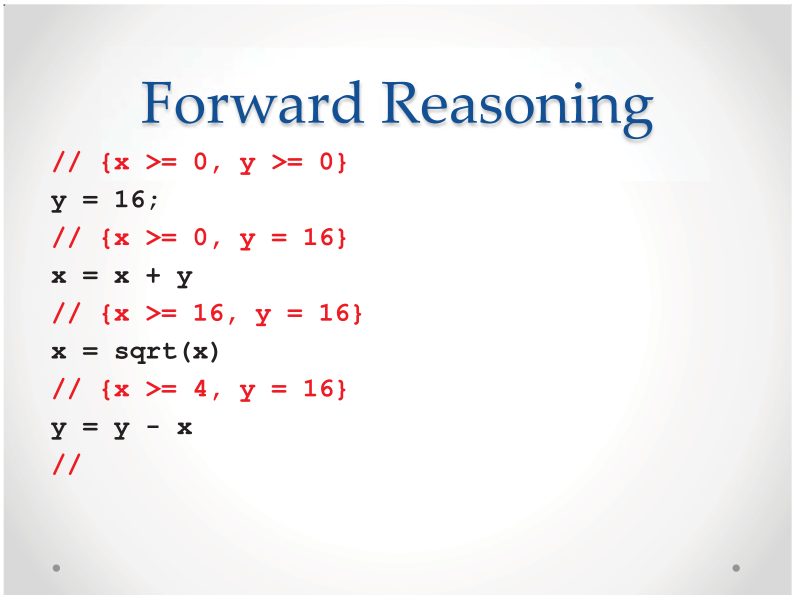
Forward Reasoning

```
// {x >= 0, y >= 0}
y = 16;
// {x >= 0, y = 16}
x = x + y
//
x = sqrt(x)
//
y = y - x
//
```



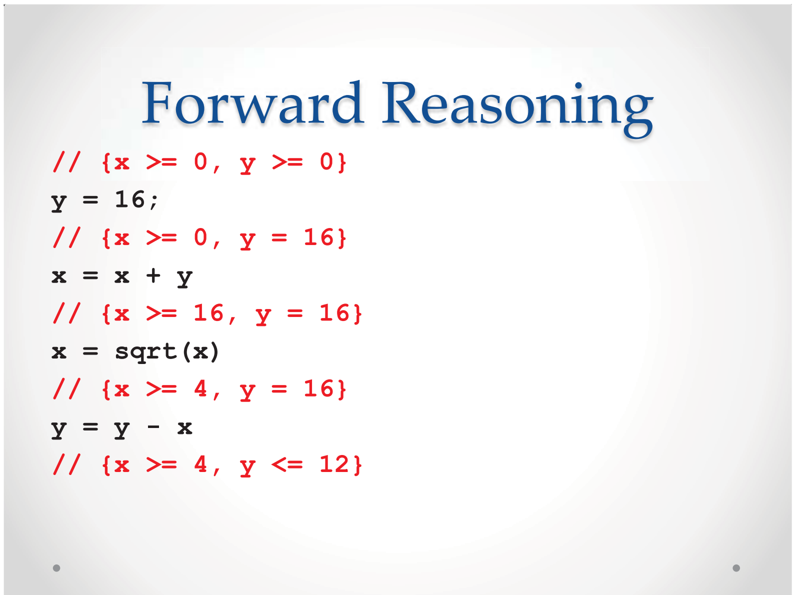
Forward Reasoning

```
// {x >= 0, y >= 0}
y = 16;
// {x >= 0, y = 16}
x = x + y
// {x >= 16, y = 16}
x = sqrt(x)
//
y = y - x
//
```



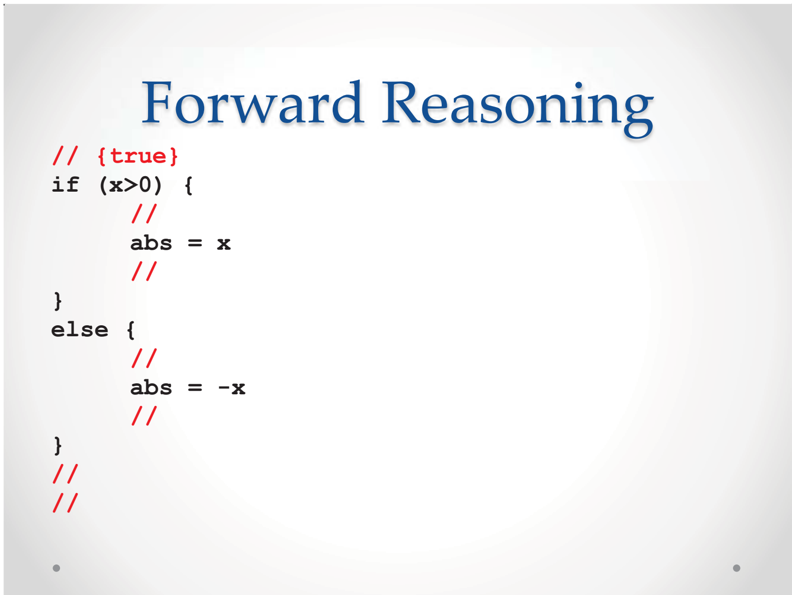
Forward Reasoning

```
// {x >= 0, y >= 0}
y = 16;
// {x >= 0, y = 16}
x = x + y
// {x >= 16, y = 16}
x = sqrt(x)
// {x >= 4, y = 16}
y = y - x
//
```



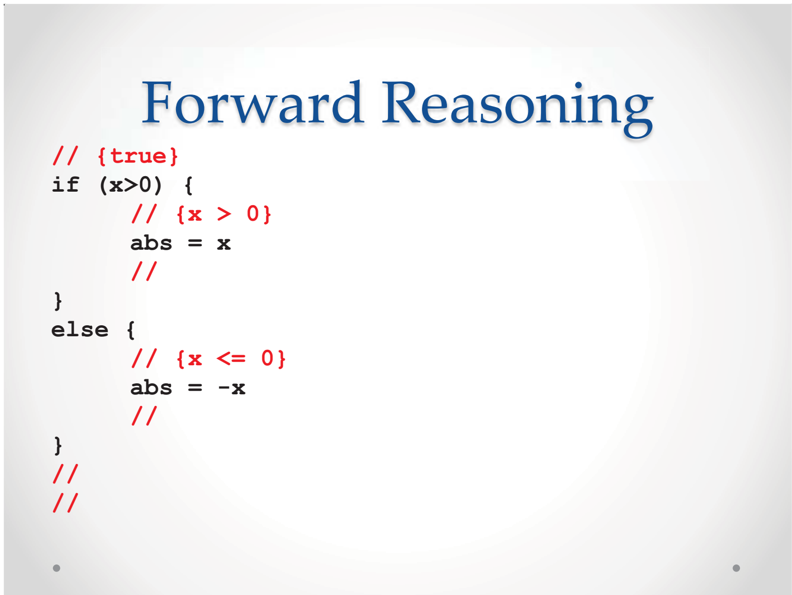
Forward Reasoning

```
// {x >= 0, y >= 0}
y = 16;
// {x >= 0, y = 16}
x = x + y
// {x >= 16, y = 16}
x = sqrt(x)
// {x >= 4, y = 16}
y = y - x
// {x >= 4, y <= 12}
```



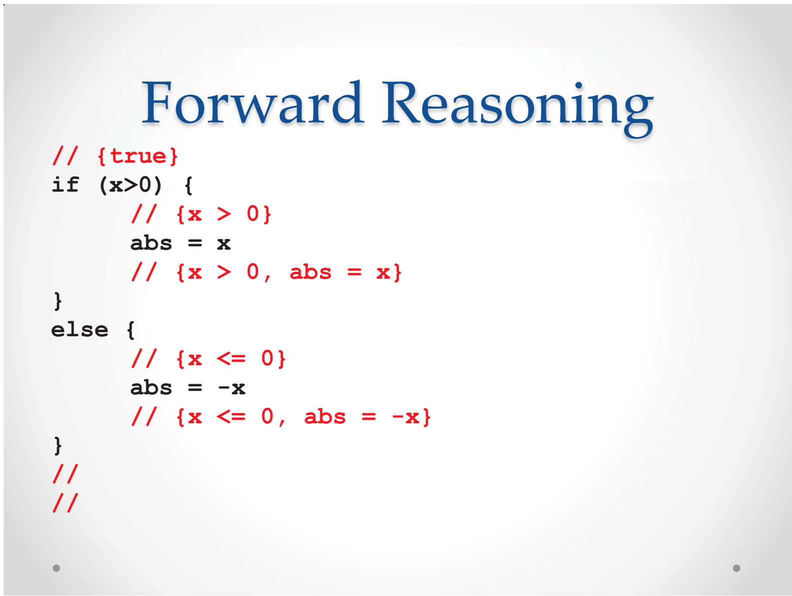
Forward Reasoning

```
// {true}
if (x>0) {
      //
      abs = x
      //
}
else {
      //
      abs = -x
      //
}
//
//
```



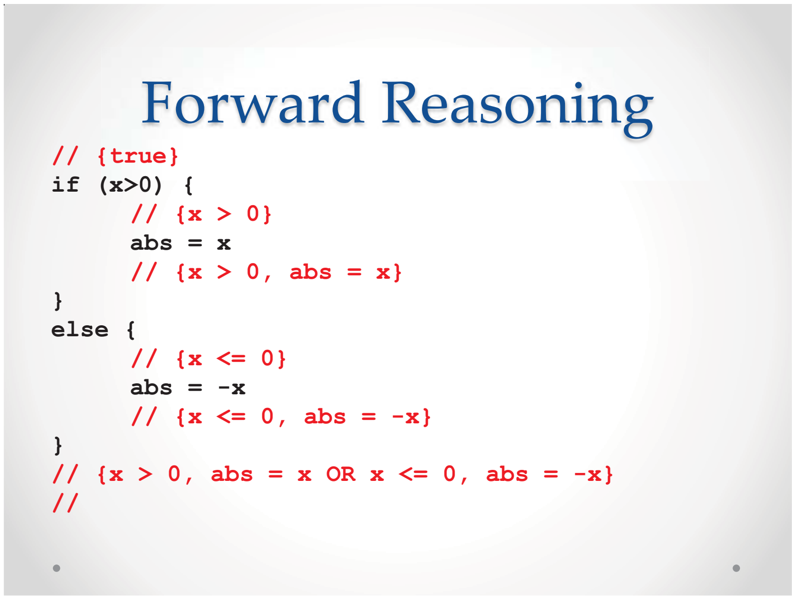
Forward Reasoning

```
// {true}
if (x>0) {
      // {x > 0}
      abs = x
      //
}
else {
      // {x <= 0}
      abs = -x
      //
}
//
//
```



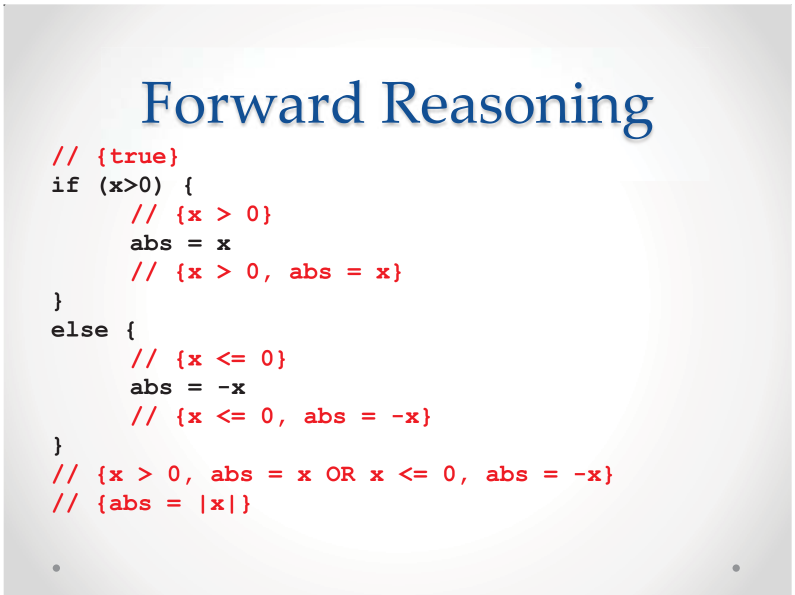
Forward Reasoning

```
// {true}
if (x>0) {
      // {x > 0}
      abs = x
      // {x > 0, abs = x}
}
else {
      // {x <= 0}
      abs = -x
      // {x <= 0, abs = -x}
}
//
//
```



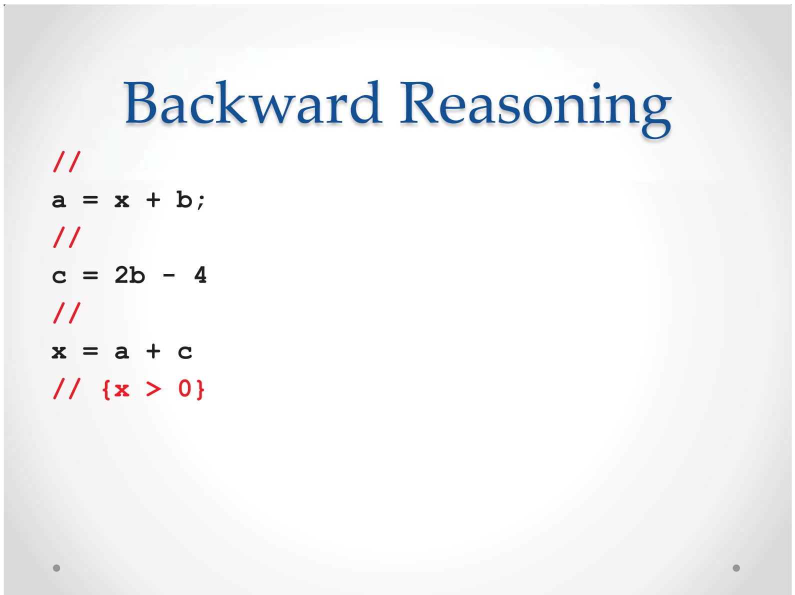
Forward Reasoning

```
// {true}
if (x>0) {
      // {x > 0}
      abs = x
      // {x > 0, abs = x}
}
else {
      // {x <= 0}
      abs = -x
      // {x <= 0, abs = -x}
}
// {x > 0, abs = x OR x <= 0, abs = -x}
//
```



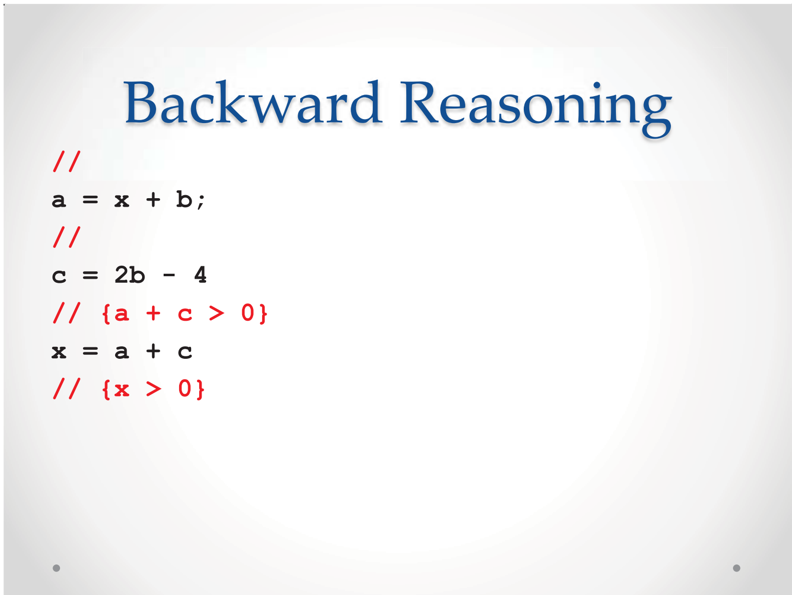
Forward Reasoning

```
// {true}
if (x>0) {
        // {x > 0}
        abs = x
        // {x > 0, abs = x}
}
else {
        // {x <= 0}
        abs = -x
        // {x <= 0, abs = -x}
}
// {x > 0, abs = x OR x <= 0, abs = -x}
// {abs = |x|}
```



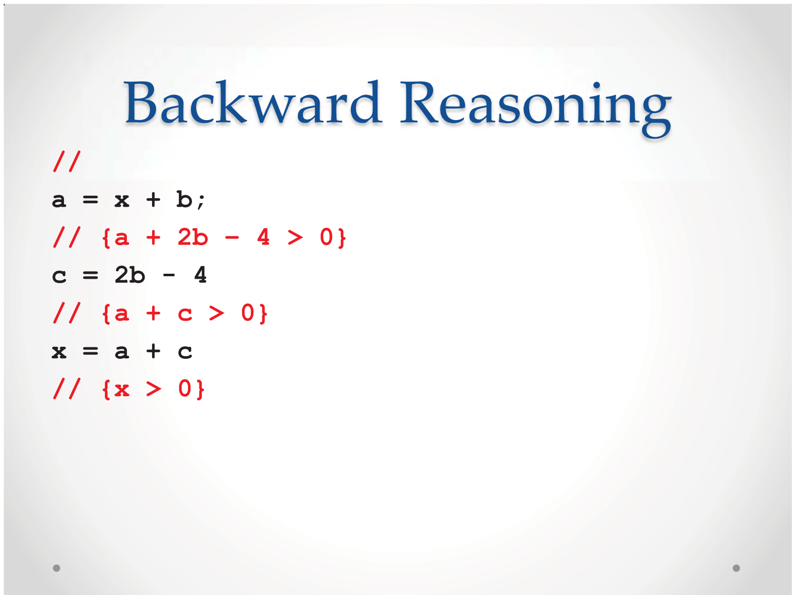
Backward Reasoning

```
//
a = x + b;
//
c = 2b - 4
//
x = a + c
// {x > 0}
```



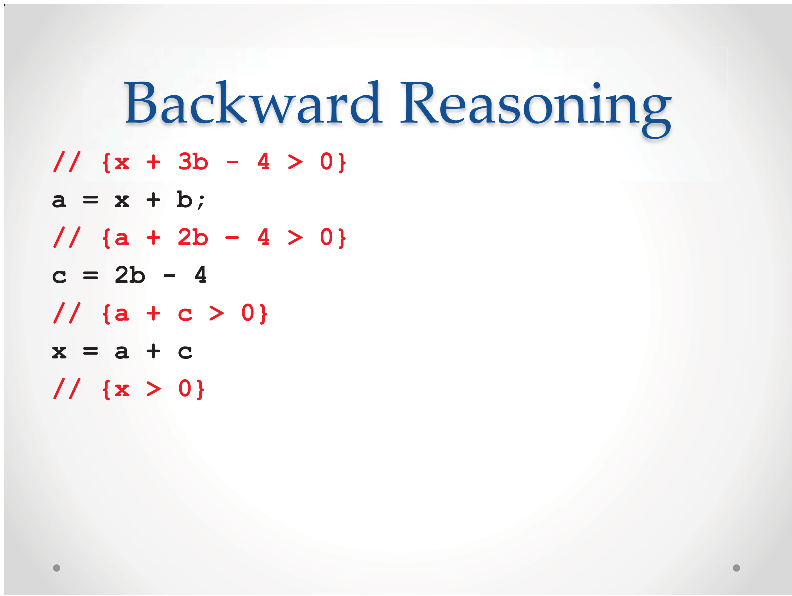
Backward Reasoning

```
//
a = x + b;
//
c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```



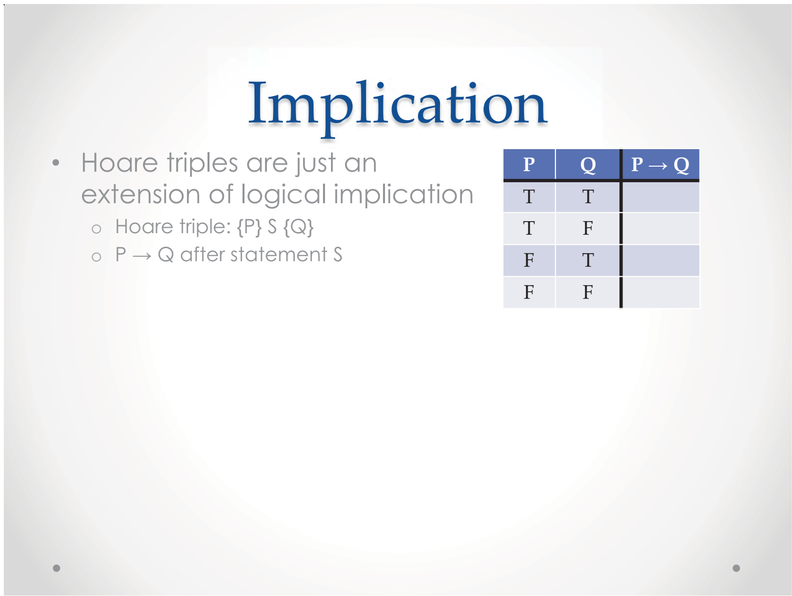
Backward Reasoning

```
//
a = x + b;
// {a + 2b – 4 > 0}
c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```



Backward Reasoning

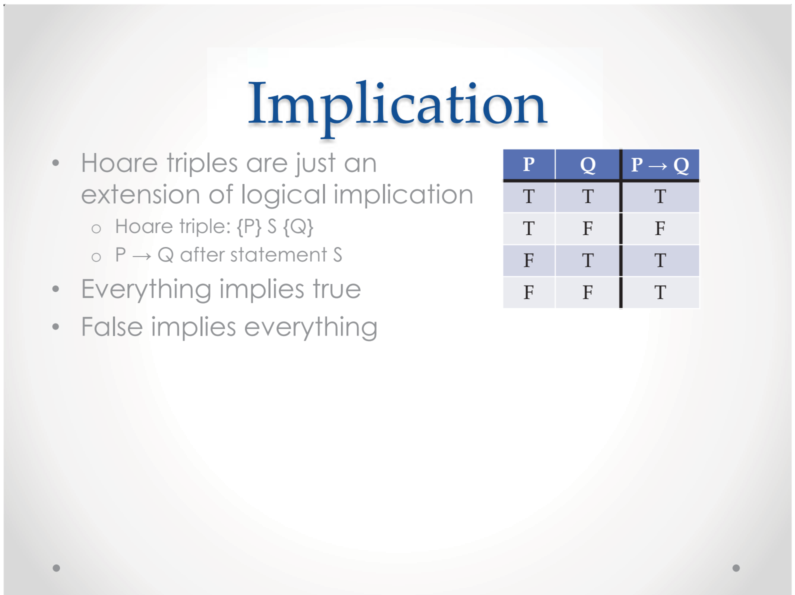
```
// {x + 3b - 4 > 0}
a = x + b;
// {a + 2b - 4 > 0}
c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```



Implication

- Hoare triples are just an extension of logical implication
  - Hoare triple: {P} S {Q}
  - P → Q after statement S

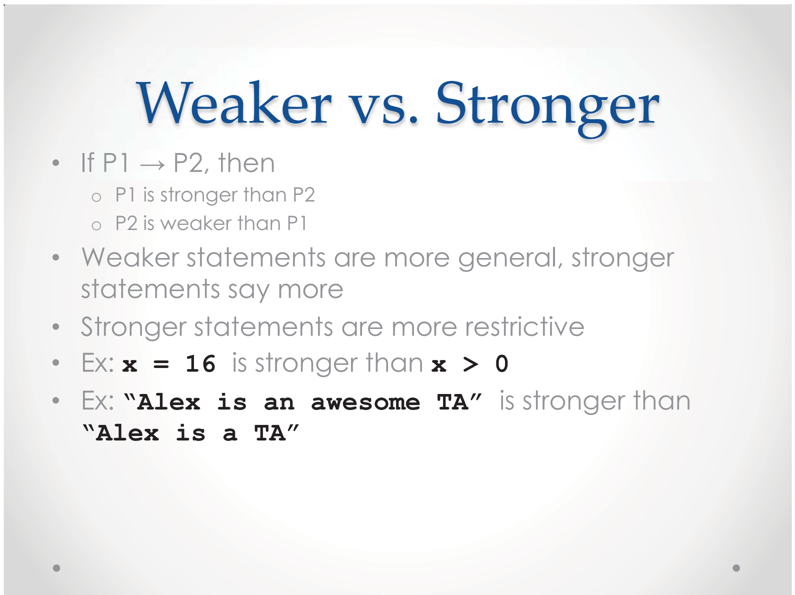
P	Q	P → Q
T	T	
T	F	
F	T	
F	F	



Implication

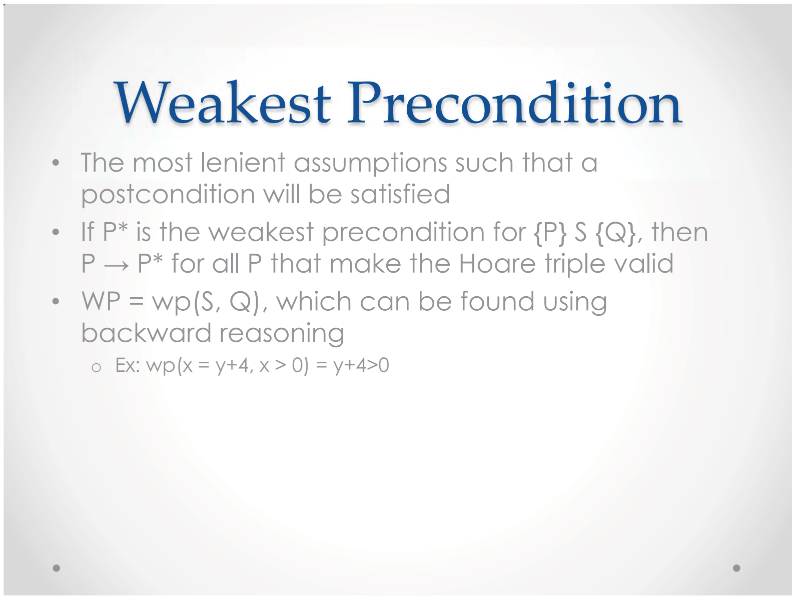
- Hoare triples are just an extension of logical implication
  - Hoare triple: {P} S {Q}
  - P → Q after statement S
- Everything implies true
- False implies everything

P	Q	P → Q
T	T	T
T	F	F
F	T	T
F	F	T



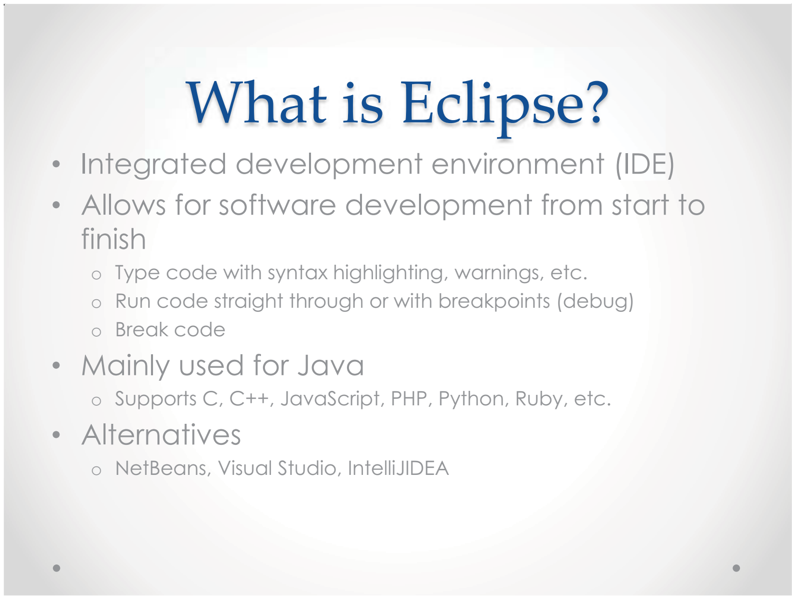
Weaker vs. Stronger

- If P1 → P2, then
  - P1 is stronger than P2
  - P2 is weaker than P1
- Weaker statements are more general, stronger statements say more
- Stronger statements are more restrictive
- Ex: `x = 16` is stronger than `x > 0`
- Ex: **"Alex is an awesome TA"** is stronger than **"Alex is a TA"**



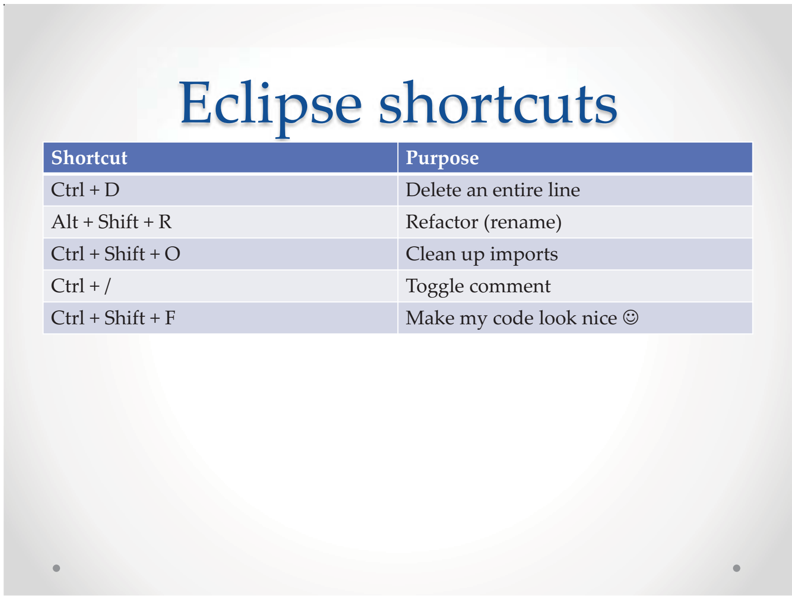
Weakest Precondition

- The most lenient assumptions such that a postcondition will be satisfied
- If P* is the weakest precondition for {P} S {Q}, then P → P* for all P that make the Hoare triple valid
- WP = wp(S, Q), which can be found using backward reasoning
  - Ex: wp(x = y+4, x > 0) = y+4>0



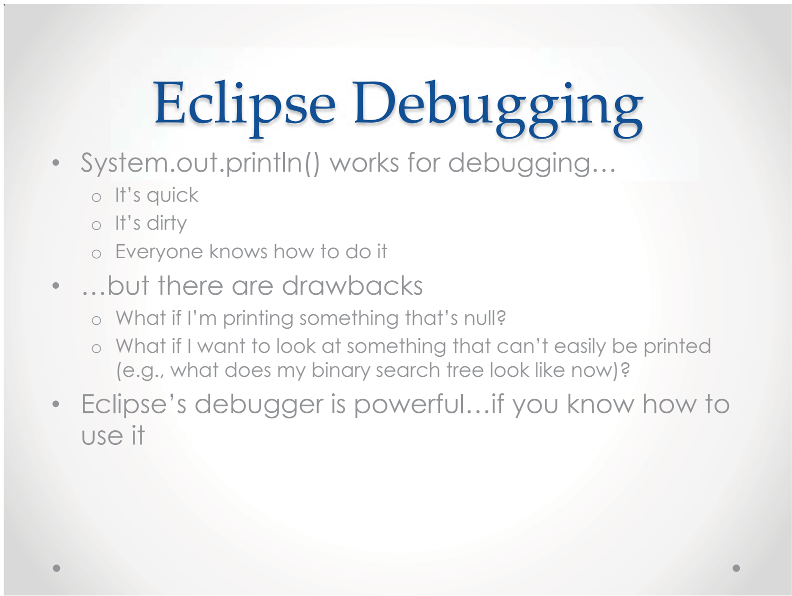
What is Eclipse?

- Integrated development environment (IDE)
- Allows for software development from start to finish
  - Type code with syntax highlighting, warnings, etc.
  - Run code straight through or with breakpoints (debug)
  - Break code
- Mainly used for Java
  - Supports C, C++, JavaScript, PHP, Python, Ruby, etc.
- Alternatives
  - NetBeans, Visual Studio, IntelliJIDEA



Eclipse shortcuts

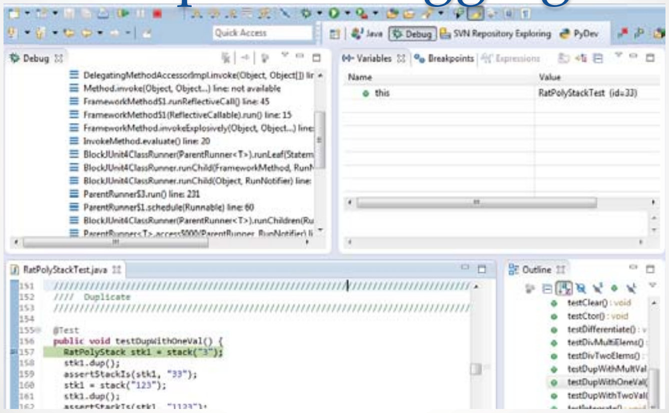
Shortcut	Purpose
Ctrl + D	Delete an entire line
Alt + Shift + R	Refactor (rename)
Ctrl + Shift + O	Clean up imports
Ctrl + /	Toggle comment
Ctrl + Shift + F	Make my code look nice ☺



Eclipse Debugging

- System.out.println() works for debugging…
  - It's quick
  - It's dirty
  - Everyone knows how to do it
- …but there are drawbacks
  - What if I'm printing something that's null?
  - What if I want to look at something that can't easily be printed (e.g., what does my binary search tree look like now)?
- Eclipse's debugger is powerful…if you know how to use it

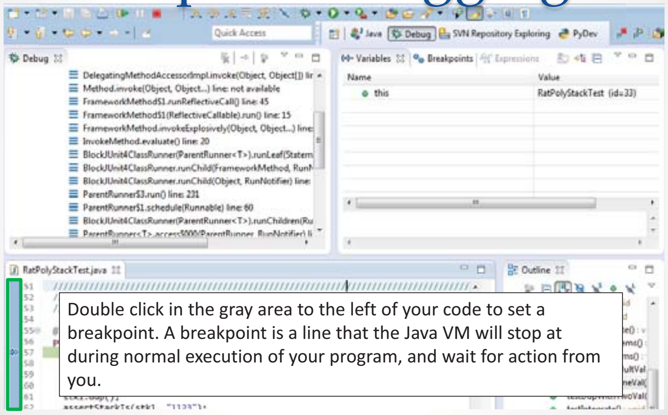
Eclipse Debugging



```
DelegatingMethodAccessorImpl.invoke(Object, Object[]) lir
Method.invoke(Object, Object...) line: not available
FrameworkMethod$1.runReflectiveCall() line: 45
FrameworkMethod$1(ReflectiveCallable).run() line: 15
FrameworkMethod.invokeExplosively(Object, Object...) line:
InvokeMethod.evaluate() line: 20
BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf(Statem
BlockJUnit4ClassRunner.runChild(FrameworkMethod, RunN
BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line:
ParentRunner$3.run() line: 231
ParentRunner$1.schedule(Runnable) line: 60
BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Ru
ParentRunner<T>.access$000(ParentRunner, RunNotifier) li
```

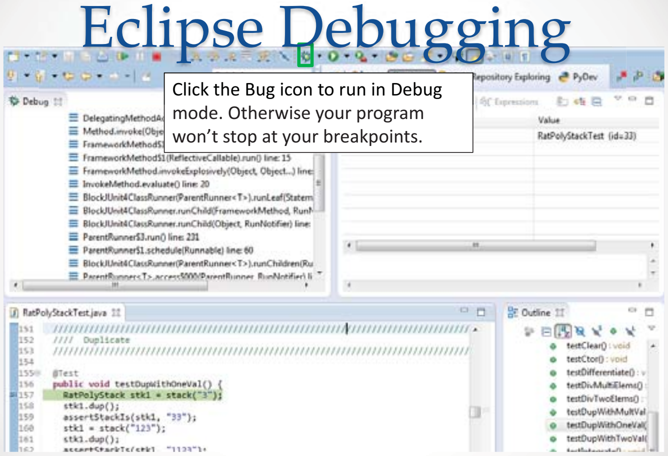
```
////////////////////////////////////////////////////////////////////
////  Duplicate
////////////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
  RatPolyStack stk1 = stack("3");
  stk1.dup();
  assertStackIs(stk1, "33");
  stk1 = stack("123");
  stk1.dup();
  assertStackIs(stk1, "1123");
```

Eclipse Debugging



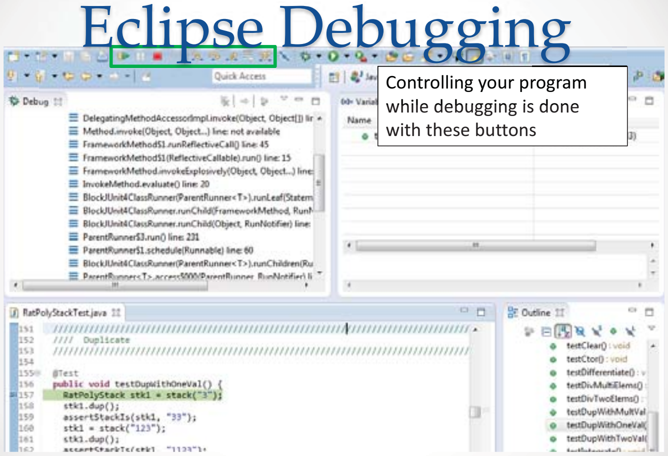
Double click in the gray area to the left of your code to set a breakpoint. A breakpoint is a line that the Java VM will stop at during normal execution of your program, and wait for action from you.

Eclipse Debugging



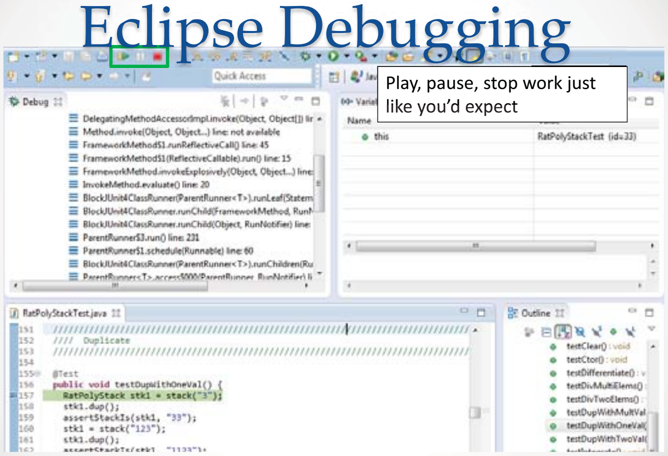
Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints.

Eclipse Debugging



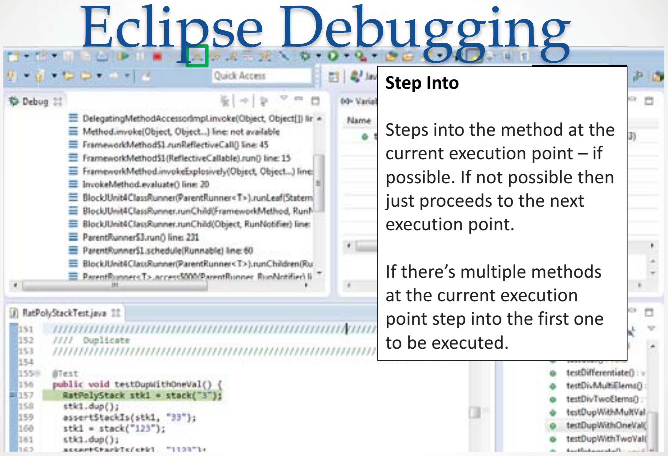
Controlling your program while debugging is done with these buttons

Eclipse Debugging



Play, pause, stop work just like you'd expect

Eclipse Debugging

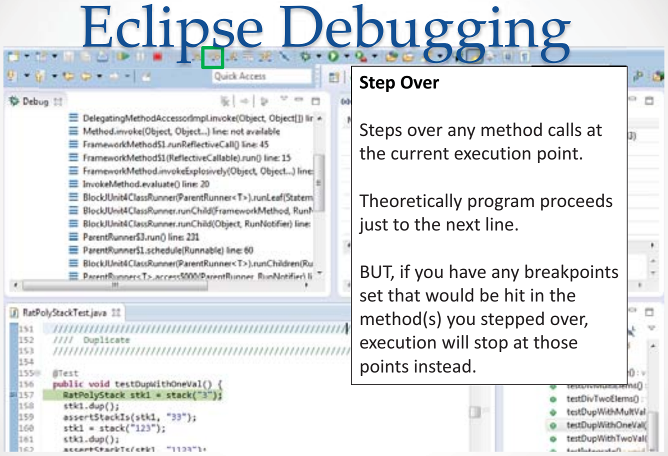


Step Into

Steps into the method at the current execution point – if possible. If not possible then just proceeds to the next execution point.

If there's multiple methods at the current execution point step into the first one to be executed.

Eclipse Debugging



Step Over

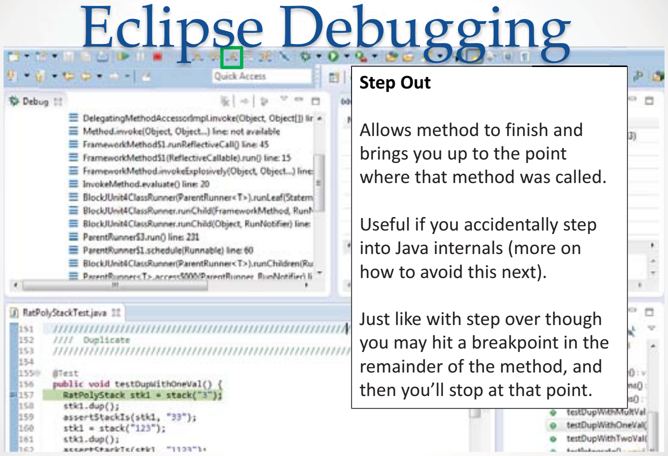
Steps over any method calls at the current execution point.

Theoretically program proceeds just to the next line.

BUT, if you have any breakpoints set that would be hit in the method(s) you stepped over, execution will stop at those points instead.

```
////  Duplicate

@Test
public void testDupWithOneVal() {
  RatPolyStack stk1 = stack("3");
  stk1.dup();
  assertStackIs(stk1, "33");
  stk1 = stack("123");
  stk1.dup();
```

Eclipse Debugging



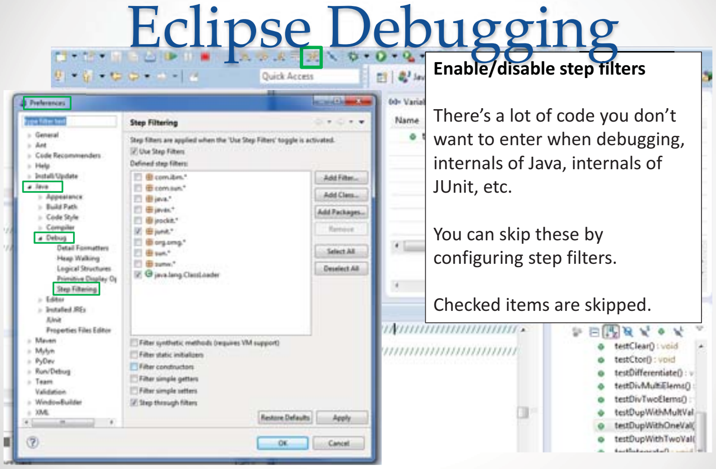
Step Out

Allows method to finish and brings you up to the point where that method was called.

Useful if you accidentally step into Java internals (more on how to avoid this next).

Just like with step over though you may hit a breakpoint in the remainder of the method, and then you'll stop at that point.

Eclipse Debugging



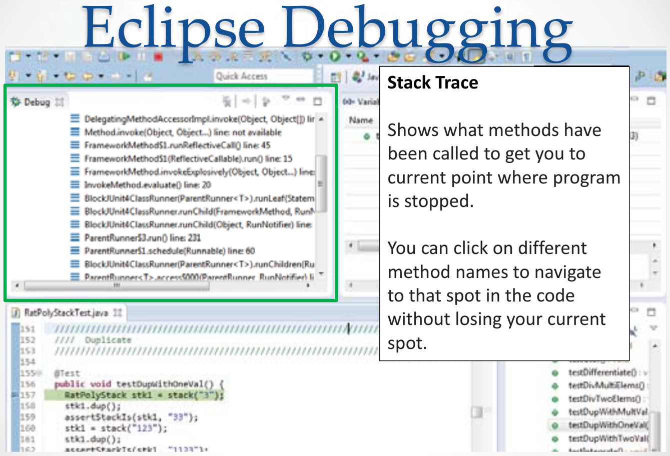
Enable/disable step filters

There's a lot of code you don't want to enter when debugging, internals of Java, internals of JUnit, etc.

You can skip these by configuring step filters.

Checked items are skipped.

Eclipse Debugging

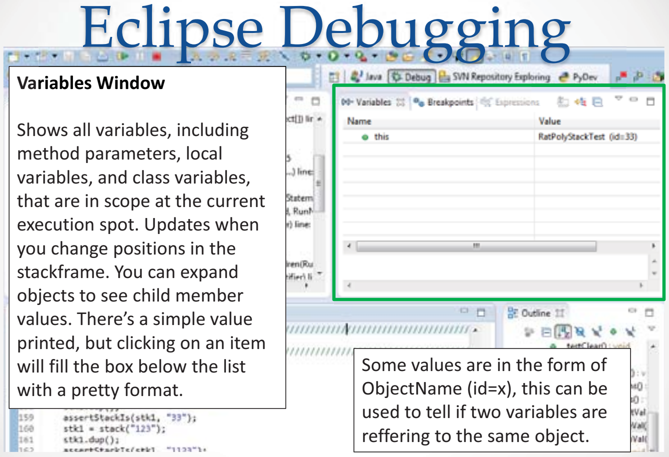


Stack Trace

Shows what methods have been called to get you to current point where program is stopped.

You can click on different method names to navigate to that spot in the code without losing your current spot.

Eclipse Debugging



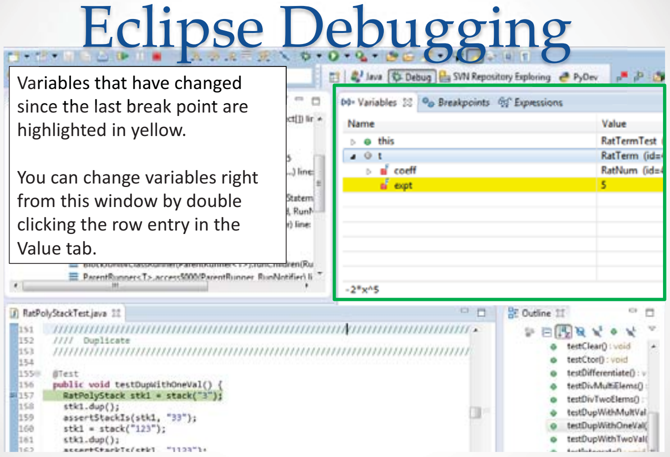
Variables Window

Shows all variables, including method parameters, local variables, and class variables, that are in scope at the current execution spot. Updates when you change positions in the stackframe. You can expand objects to see child member values. There's a simple value printed, but clicking on an item will fill the box below the list with a pretty format.

Some values are in the form of ObjectName (id=x), this can be used to tell if two variables are reffering to the same object.

Name	Value
this	RatPolyStackTest (id=33)

Eclipse Debugging



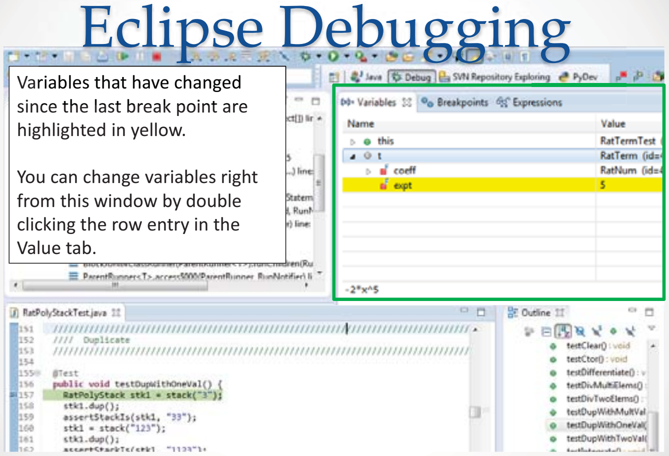
Variables that have changed since the last break point are highlighted in yellow.

You can change variables right from this window by double clicking the row entry in the Value tab.

Name	Value
this	RatTermTest
t	RatTerm (id=
coeff	RatNum (id=
expt	5

-2*x^5

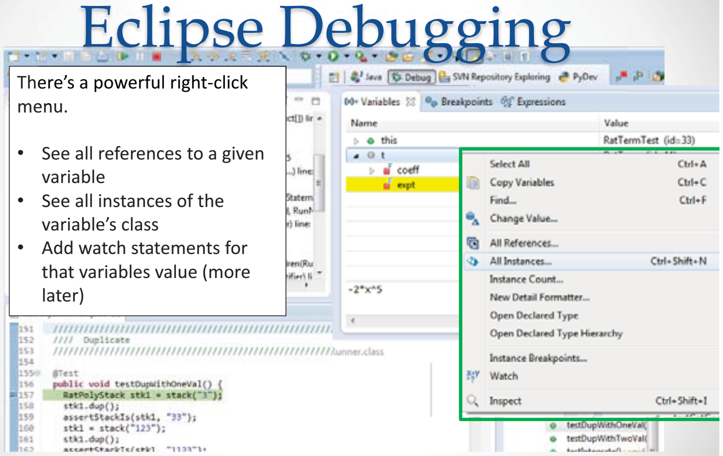
Eclipse Debugging



Variables that have changed since the last break point are highlighted in yellow.

You can change variables right from this window by double clicking the row entry in the Value tab.

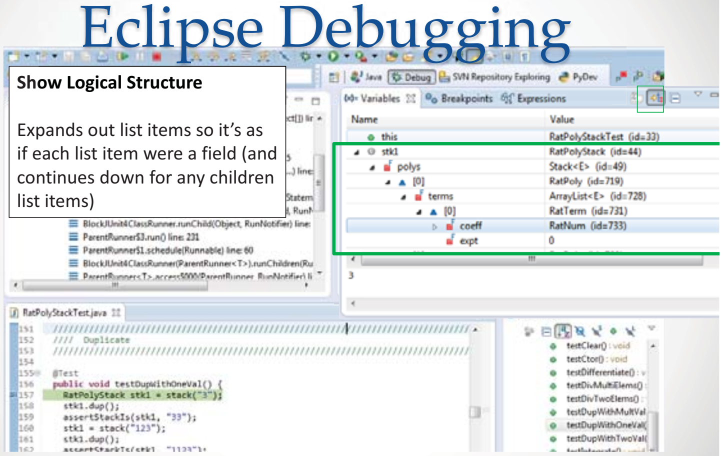
Eclipse Debugging



There's a powerful right-click menu.

- See all references to a given variable
- See all instances of the variable's class
- Add watch statements for that variables value (more later)

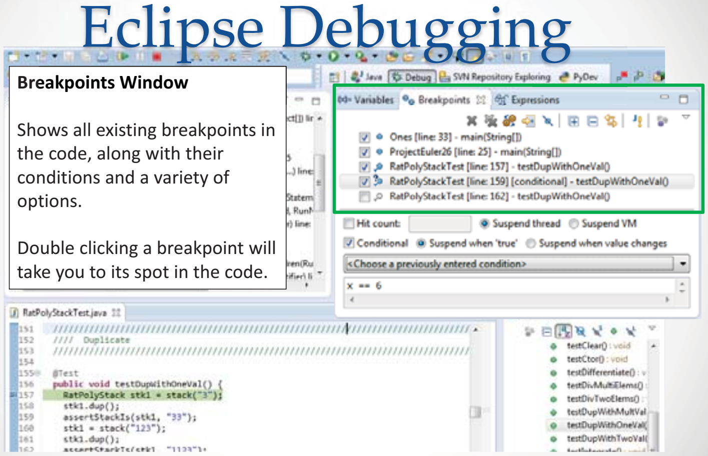
Eclipse Debugging



Show Logical Structure

Expands out list items so it's as if each list item were a field (and continues down for any children list items)

Eclipse Debugging



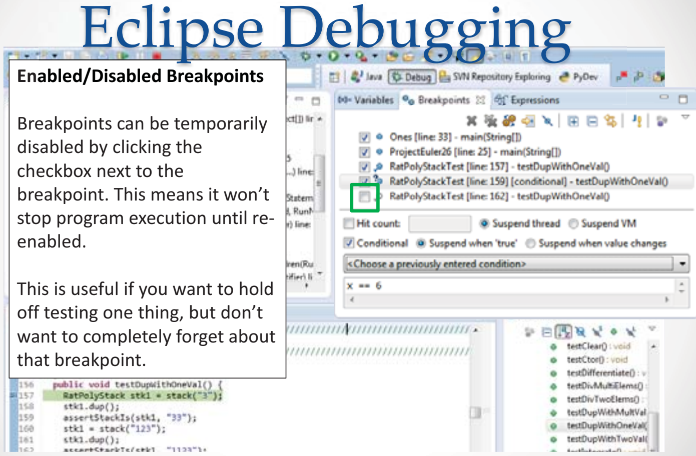
Breakpoints Window

Shows all existing breakpoints in the code, along with their conditions and a variety of options.

Double clicking a breakpoint will take you to its spot in the code.

Ones [line: 33] - main(String[])
ProjectEuler26 [line: 25] - main(String[])
RatPolyStackTest [line: 157] - testDupWithOneVal()
RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()
RatPolyStackTest [line: 162] - testDupWithOneVal()
Hit count:
Suspend thread
Suspend VM
Conditional
Suspend when 'true'
Suspend when value changes
<Choose a previously entered condition>
x == 6

Eclipse Debugging

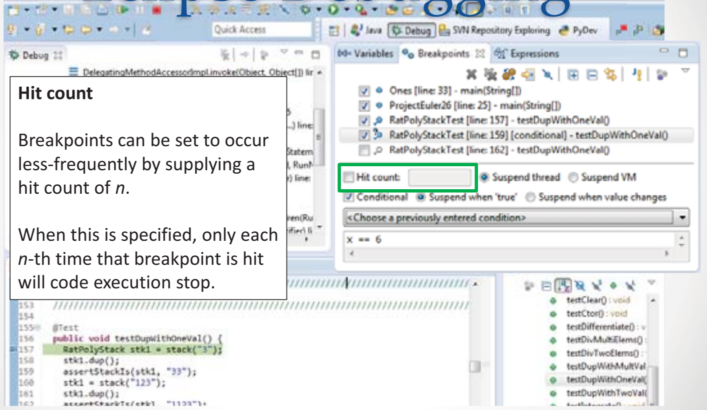


Enabled/Disabled Breakpoints

Breakpoints can be temporarily disabled by clicking the checkbox next to the breakpoint. This means it won't stop program execution until re-enabled.

This is useful if you want to hold off testing one thing, but don't want to completely forget about that breakpoint.

Eclipse Debugging

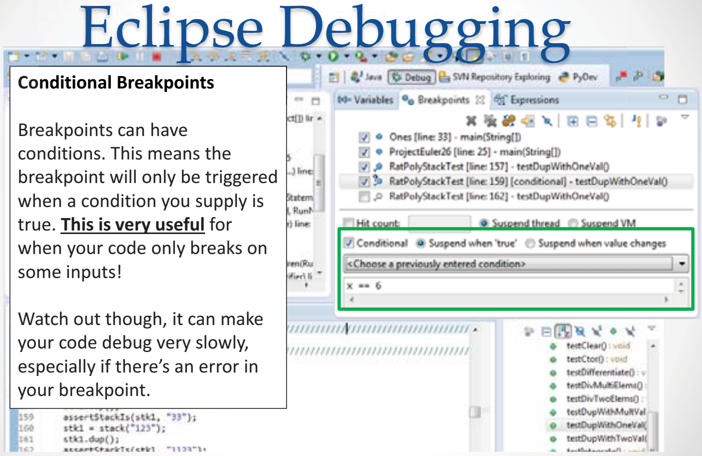


Hit count

Breakpoints can be set to occur less-frequently by supplying a hit count of *n*.

When this is specified, only each *n*-th time that breakpoint is hit will code execution stop.

Eclipse Debugging



Conditional Breakpoints

Breakpoints can have conditions. This means the breakpoint will only be triggered when a condition you supply is true. **<u>This is very useful</u>** for when your code only breaks on some inputs!

Watch out though, it can make your code debug very slowly, especially if there's an error in your breakpoint.

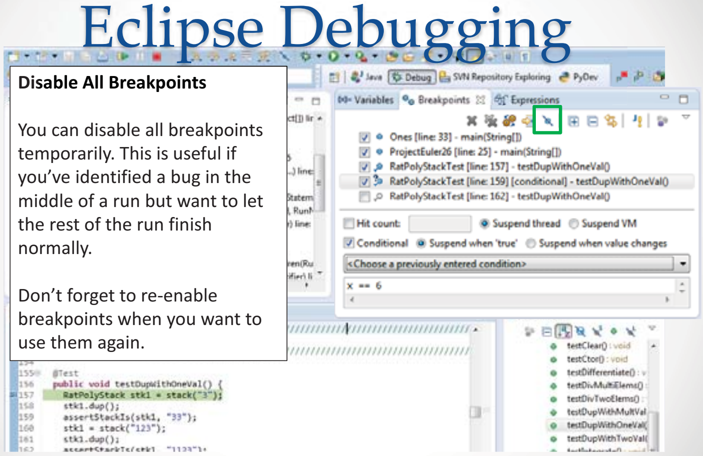
☑ Ones [line: 33] - main(String[])
☑ ProjectEuler26 [line: 25] - main(String[])
☑ RatPolyStackTest [line: 157] - testDupWithOneVal()
☑ RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()
□ RatPolyStackTest [line: 162] - testDupWithOneVal()

□ Hit count: ● Suspend thread ○ Suspend VM
☑ Conditional ● Suspend when 'true' ○ Suspend when value changes
<Choose a previously entered condition>

```
x == 6
```

```
assertStackIs(stk1, "33");
stk1 = stack("123");
stk1.dup();
```

Eclipse Debugging



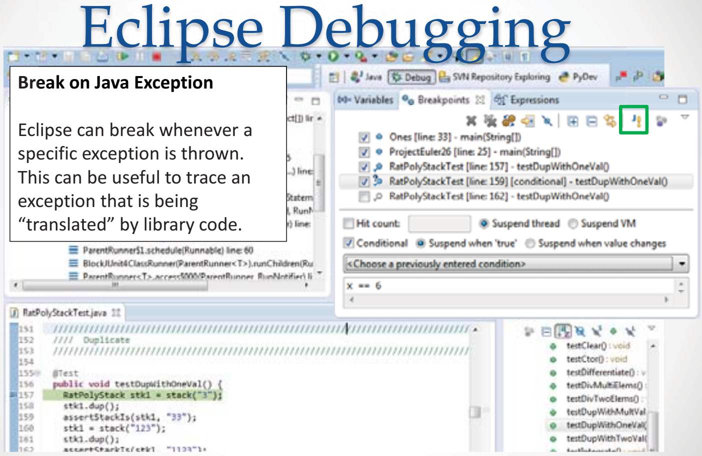
Disable All Breakpoints

You can disable all breakpoints temporarily. This is useful if you've identified a bug in the middle of a run but want to let the rest of the run finish normally.

Don't forget to re-enable breakpoints when you want to use them again.

```
@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
```

Eclipse Debugging

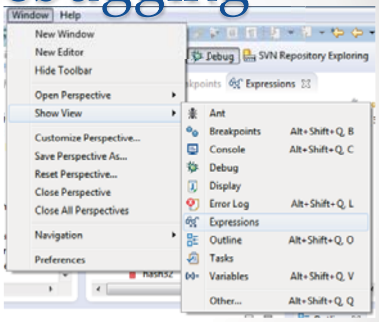


Break on Java Exception

Eclipse can break whenever a specific exception is thrown. This can be useful to trace an exception that is being "translated" by library code.

```
////////////////////////////////////////////
////  Duplicate
////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
```

Eclipse Debugging

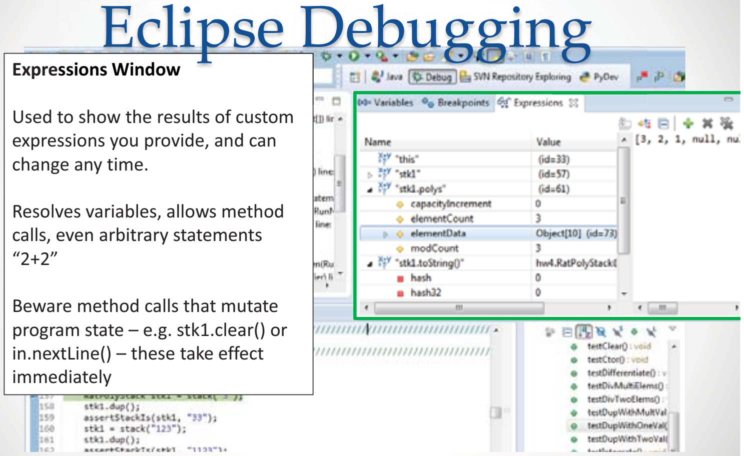


Expressions Window

Used to show the results of custom expressions you provide, and can change any time.

Not shown by default but highly recommended.

Eclipse Debugging



Expressions Window

Used to show the results of custom expressions you provide, and can change any time.

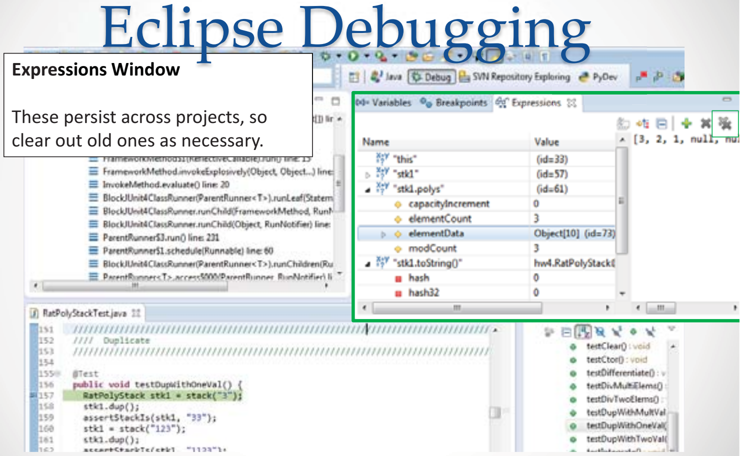
Resolves variables, allows method calls, even arbitrary statements "2+2"

Beware method calls that mutate program state – e.g. stk1.clear() or in.nextLine() – these take effect immediately

Name	Value
"this"	(id=33)
"stk1"	(id=57)
"stk1.polys"	(id=61)
capacityIncrement	0
elementCount	3
elementData	Object[10] (id=73)
modCount	3
"stk1.toString()"	hw4.RatPolyStack
hash	0
hash32	0

```
[3, 2, 1, null, nu
```

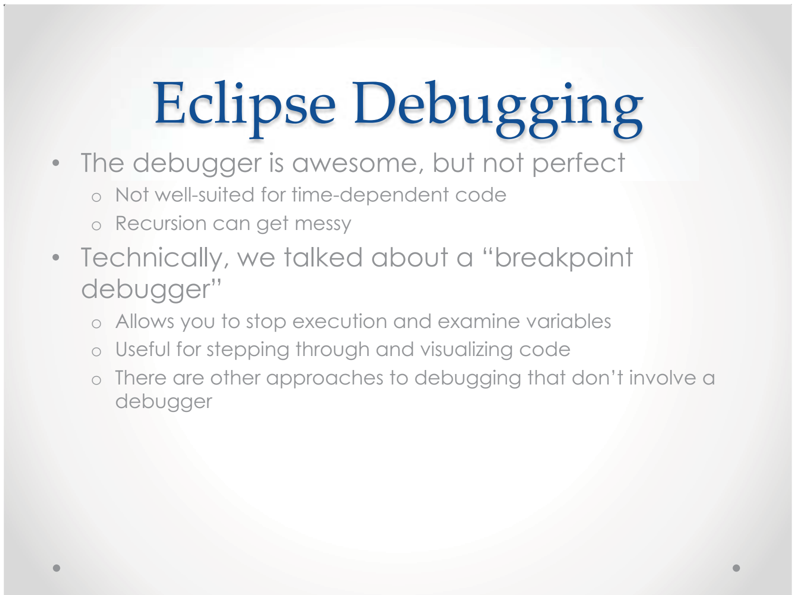
Eclipse Debugging



Expressions Window

These persist across projects, so clear out old ones as necessary.

Demo!!!



Eclipse Debugging

- The debugger is awesome, but not perfect
  - Not well-suited for time-dependent code
  - Recursion can get messy
- Technically, we talked about a "breakpoint debugger"
  - Allows you to stop execution and examine variables
  - Useful for stepping through and visualizing code
  - There are other approaches to debugging that don't involve a debugger