

# SECTION 1:

## CODE REASONING + VERSION CONTROL + ECLIPSE

[cse331-staff@cs.washington.edu](mailto:cse331-staff@cs.washington.edu)

# OUTLINE

- **Introductions**
- **Code Reasoning**
- **Version control**
- **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:  $\{P\} S \{Q\}$
  - $P \rightarrow Q$  after statement  $S$

P	Q	$P \rightarrow Q$
T	T	
T	F	
F	T	
F	F	

# IMPLICATION

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

P	Q	$P \rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

# WEAKER VS. STRONGER

- **If  $P1 \rightarrow P2$ , 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”

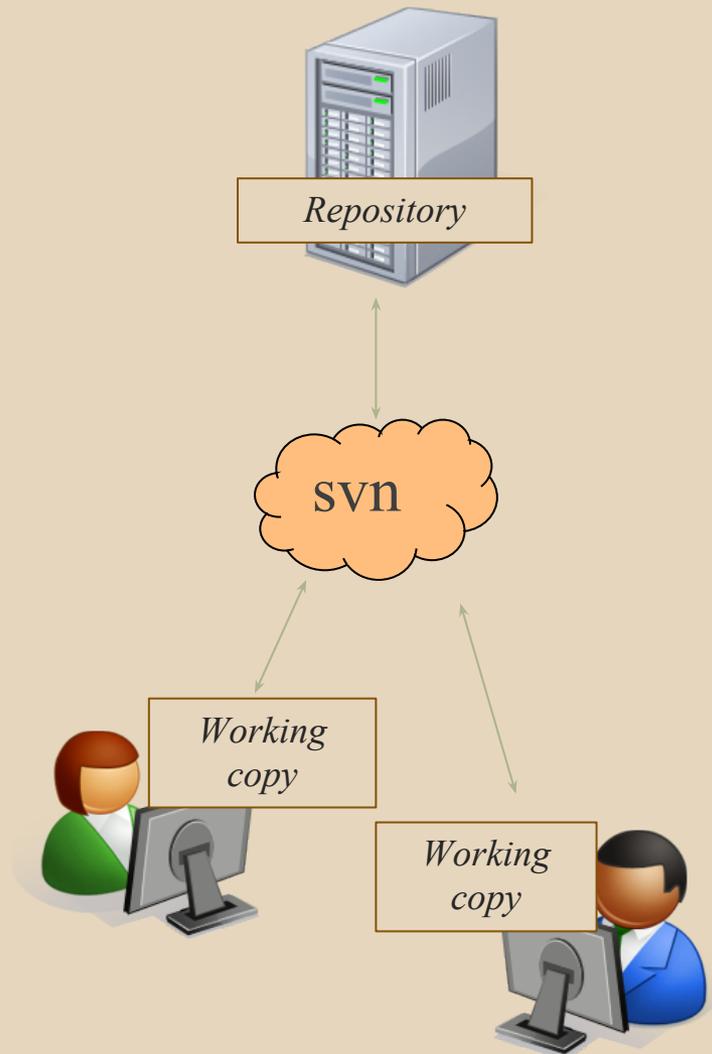
# **VERSION CONTROL**

# WHAT IS VERSION CONTROL?

- Also known as source control/revision control
- System for tracking changes to code
  - Software for developing software
- Essential for managing projects
  - See a history of changes
  - Revert back to an older version
  - Merge changes from multiple sources
- We'll be talking about Subversion, but there are alternatives
  - Git, Mercurial, CVS
  - Email, Dropbox, USB sticks

# VERSION CONTROL ORGANIZATION

- A *repository* stores the master copy of the project
  - Someone creates the repo for a new project
  - Then nobody touches this copy directly
  - Lives on a server everyone can access
- Each person *checks out* her own *working copy*
  - Makes a local copy of the repo
  - You'll always work off of this copy
  - The version control system syncs the repo and working copy (with your help)



# REPOSITORY

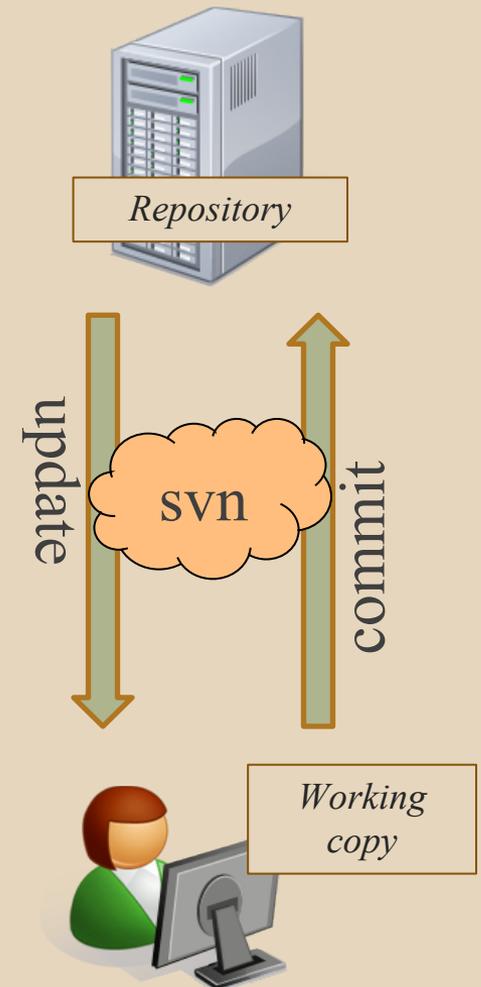
- Can create the repository anywhere
  - Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection
- But, usually you want the repository to be robust:
  - On a computer that's up and running 24/7
    - Everyone always has access to the project
  - On a computer that has a redundant file system
    - No more worries about that hard disk crash wiping away your project!
- We'll use attu! ([attu.cs.washington.edu](http://attu.cs.washington.edu))

# VERSION CONTROL

## COMMON ACTIONS

Most common commands:

- **Commit / checkin**
  - integrate changes *from* your working copy *into* the repository
- **Update**
  - integrate changes *into* your working copy *from* the repository

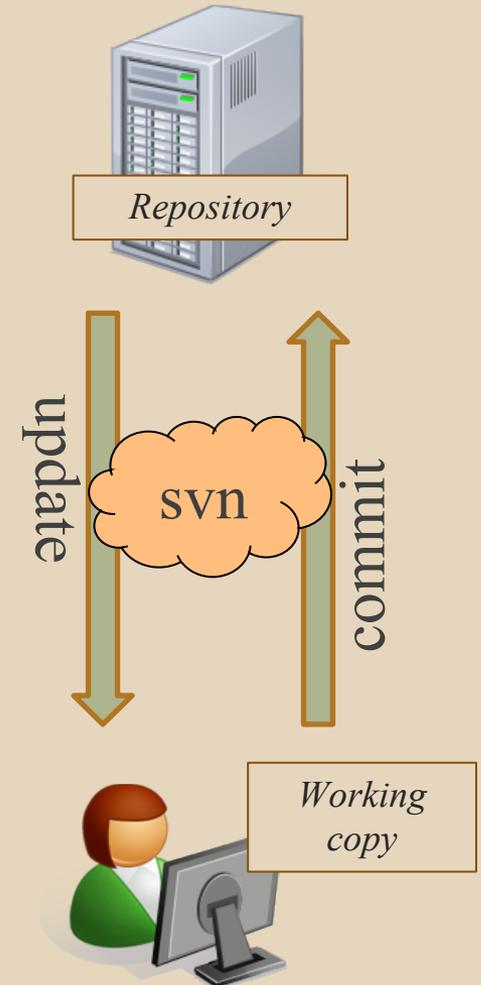


# VERSION CONTROL

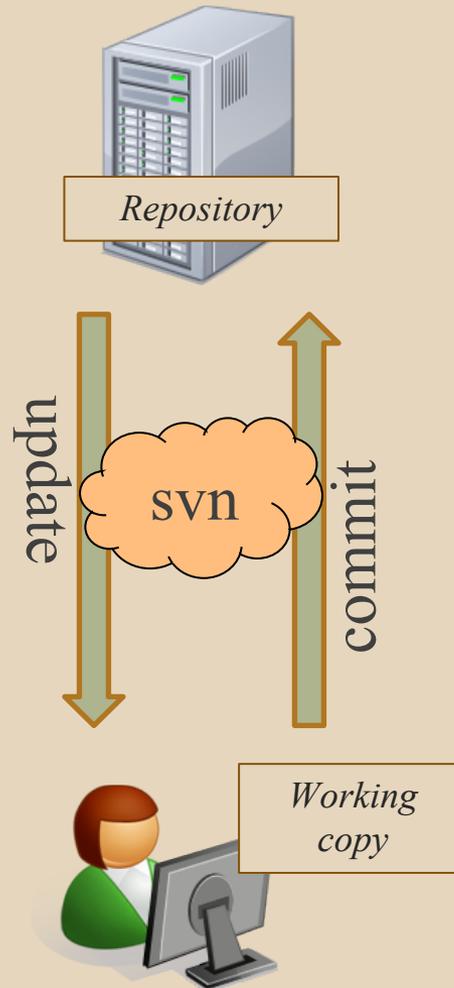
## COMMON ACTIONS (CONT.)

More common commands:

- **Add, delete**
  - add or delete a file in the repository
  - just putting a new file in your working copy does not add it to the repo!
- **Revert**
  - wipe out your local changes to a file
- **Resolve, diff, merge**
  - handle a conflict – two users editing the same code



# VERSION CONTROL



# THIS QUARTER

- We distribute starter code by adding it to your **repo**
- You will **code** in Eclipse
- You turn in your files by **adding** them to the repo and **committing** your changes
- You will **validate** your homework by **SSHing** onto attu and running an Ant build file

**More on this next section!**

**ECLIPSE**

# 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

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar includes standard development icons. The main workspace is divided into several panels:

- Debug Console:** Shows the execution stack with the following entries:
  - DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: not available
  - Method.invoke(Object, Object...) line: not available
  - FrameworkMethod\$1.runReflectiveCall() line: 45
  - FrameworkMethod\$1(ReflectiveCallable).run() line: 15
  - FrameworkMethod.invokeExplosively(Object, Object...) line: not available
  - InvokeMethod.evaluate() line: 20
  - BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf(Statement) line: not available
  - BlockJUnit4ClassRunner.runChild(FrameworkMethod, Runnable) line: not available
  - BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line: not available
  - ParentRunner\$3.run() line: 231
  - ParentRunner\$1.schedule(Runnable) line: 60
  - BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(RunNotifier) line: not available
  - ParentRunner<T>.access\$000(ParentRunner, RunNotifier) line: not available
- Variables View:** A table showing the current state of variables:

Name	Value
this	RatPolyStackTest (id=33)
- Source Editor:** Displays the file `RatPolyStackTest.java`. A breakpoint is set at line 157, which is highlighted in green. The code snippet is:

```
151 ////////////////////////////////////////////////////  
152 /// Duplicate  
153 ////////////////////////////////////////////////////  
154  
155 @Test  
156 public void testDupWithOneVal() {  
157     RatPolyStack stk1 = stack("3");  
158     stk1.dup();  
159     assertStackIs(stk1, "33");  
160     stk1 = stack("123");  
161     stk1.dup();  
162     assertStackIs(stk1, "1123");  
}
```
- Outline View:** Lists the methods of the class, with `testDupWithOneVal()` selected and marked with a green circle, indicating it is the current method being debugged.

# ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes icons for file operations, running, and debugging. The 'Debug' toolbar is active, showing a play button and a red stop button. The 'Quick Access' search bar is visible. The 'Debug' console on the left shows a stack of method calls, with the current execution point at 'ParentRunner\$1.schedule(Runnable) line: 60'. The 'Variables' view on the right shows a table with one entry: 'this' with the value 'RatPolyStackTest (id=33)'. The 'Code Editor' at the bottom shows the source code for 'RatPolyStackTest.java'. A green vertical bar highlights the left margin, and a red arrow points to a breakpoint set on line 57. A text box is overlaid on the code editor, providing instructions on how to set a breakpoint.

Name	Value
this	RatPolyStackTest (id=33)

Double click in the grey 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

**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.

```
Debug Console:  
DelegatingMethodAccessorImpl.invoke(Object, Object[]) line:  
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  
RatPolyStackTest.java:  
151 ///////////////////////////////////////////////////////////////////  
152 /// Duplicate  
153 ///////////////////////////////////////////////////////////////////  
154  
155 @Test  
156 public void testDupWithOneVal() {  
157 RatPolyStack stk1 = stack("3");  
158 stk1.dup();  
159 assertStackIs(stk1, "33");  
160 stk1 = stack("123");  
161 stk1.dup();  
162 assertStackIs(stk1, "1123");
```

# 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.

```
Debug Console:  
DelegatingMethodAccessorImpl.invoke(Object, Object[]) line:  
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  
RatPolyStackTest.java:  
151 ///////////////////////////////////////////////////  
152 /// Duplicate  
153 ///////////////////////////////////////////////////  
154  
155 @Test  
156 public void testDupWithOneVal() {  
157 RatPolyStack stk1 = stack("3");  
158 stk1.dup();  
159 assertStackIs(stk1, "33");  
160 stk1 = stack("123");  
161 stk1.dup();  
162 assertStackIs(stk1, "1123");
```

# ECLIPSE DEBUGGING

The screenshot shows the Eclipse IDE interface. The top toolbar has a green box around the 'Step Out' icon. The Debug console on the left shows a stack trace of method calls. The code editor at the bottom shows the source code for `RatPolyStackTest.java`, with line 157 highlighted. A callout box on the right explains the 'Step Out' action.

**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.

```
151 ////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
}
```

# 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.

The screenshot shows the Eclipse IDE interface. The top toolbar has a 'Run' button highlighted with a green box. The 'Preferences' dialog is open, with the 'Java' category selected in the left sidebar. The 'Debug' sub-category is also selected. The 'Step Filtering' section is expanded, showing a list of defined filters. The 'Use Step Filters' checkbox is checked. The 'java.lang.ClassLoader' filter is checked and highlighted with a green circle. The 'Run/Debug' preference category is also highlighted with a green box. A text box on the right explains that checked items are skipped.

# ECLIPSE DEBUGGING

The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons for file operations, running, and debugging. Below the toolbar is the 'Quick Access' search bar. The main workspace is divided into several panes. On the left, the 'Debug' console is open, displaying a stack trace of method calls. The stack trace is highlighted with a green border. The stack trace includes the following entries (from top to bottom):

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

On the right side of the IDE, the 'Variables' pane is visible, showing a table with columns for 'Name' and 'Value'. The table is currently empty.

At the bottom of the IDE, the 'RatPolyStackTest.java' editor is open. The code is as follows:

```
151 ////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
}
```

The line `157 RatPolyStack stk1 = stack("3");` is highlighted in green. A mouse cursor is pointing at the end of this line.

On the far right, a list of test methods is visible, including `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate(): void`.

## 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.

```
159   assertStackIs(stk1, "33");
160   stk1 = stack("123");
161   stk1.dup();
162   assertStackIs(stk1, "1123");
```

Name	Value
this	RatPolyStackTest (id=33)

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

# 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.

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes icons for file operations, running, and debugging. The main window is divided into several panes:

- Variables Window (top right, green border):** A table showing the current state of variables. The 'Value' tab is active. The variable 'expt' is highlighted in yellow, indicating it has changed since the last breakpoint. The table contains the following data:

Name	Value
this	RatTermTest (
t	RatTerm (id=4
coeff	RatNum (id=4
expt	5
- Code Editor (bottom left):** Shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted in green, corresponding to the current execution point: `RatPolyStack stk1 = stack("3");`
- Outline (bottom right):** Lists the methods of the class, with 'testDupWithOneVal()' selected.

# 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.

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes icons for file operations, running, and debugging. The main window is divided into several panes:

- Variables Window (top right, green border):** A table showing the current state of variables. The 'Value' tab is active. The variable 'expt' is highlighted in yellow, indicating it has changed since the last breakpoint. The table is as follows:

Name	Value
this	RatTermTest (
t	RatTerm (id=4
coeff	RatNum (id=4
expt	5
- Code Editor (bottom left):** Shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted in green, corresponding to the current execution point: `RatPolyStack stk1 = stack("3");`
- Outline (bottom right):** Lists the methods of the class, with 'testDupWithOneVal()' selected.

# 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 variable's value (more later)

The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar includes icons for file operations, running, and debugging. The main editor displays Java code for a class named `Runner.class`. The code includes a `@Test` annotation and a `testDupWithOneVal()` method. Line 157 is highlighted, showing the assignment `RatPolyStack stk1 = stack("3");`. The `Variables` view on the right shows the current state of the program, with `this` pointing to `RatTermTest (id=33)` and a local variable `t` containing a `coeff` and `expt` field. The `expt` field is selected, and a right-click context menu is open, listing various actions such as `Select All`, `Copy Variables`, `Find...`, `Change Value...`, `All References...`, `All Instances...`, `Instance Count...`, `New Detail Formatter...`, `Open Declared Type`, `Open Declared Type Hierarchy`, `Instance Breakpoints...`, `Watch`, and `Inspect`. The `Watch` option is highlighted, indicating the user's intention to add a watch statement for the selected variable.

```
151 ////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////Runner.class
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
```

Name	Value
this	RatTermTest (id=33)
t	
coeff	
expt	

- Select All (Ctrl+A)
- Copy Variables (Ctrl+C)
- Find... (Ctrl+F)
- Change Value...
- All References...
- All Instances... (Ctrl+Shift+N)
- Instance Count...
- New Detail Formatter...
- Open Declared Type
- Open Declared Type Hierarchy
- Instance Breakpoints...
- Watch
- Inspect (Ctrl+Shift+I)

# 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)

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes icons for running, stepping through code, and other debugging actions. The main window is divided into several panes:

- Stack Trace:** Shows the current execution stack, including `BlockJUnit4ClassRunner.runChild`, `ParentRunner.run`, and `ParentRunner.schedule`.
- Variables View:** Displays the current state of variables. The `coeff` variable is highlighted in blue. The structure is as follows:

Name	Value
this	RatPolyStackTest (id=33)
stk1	RatPolyStack (id=44)
polys	Stack<E> (id=49)
[0]	RatPoly (id=719)
terms	ArrayList<E> (id=728)
[0]	RatTerm (id=731)
coeff	RatNum (id=733)
expt	0
- Code Editor:** Shows the source code for `RatPolyStackTest.java`. Line 157, `RatPolyStack stk1 = stack("3");`, is highlighted in green, corresponding to the current stack frame.
- Outline View:** Shows the project structure, including the `testDupWithOneVal()` method.



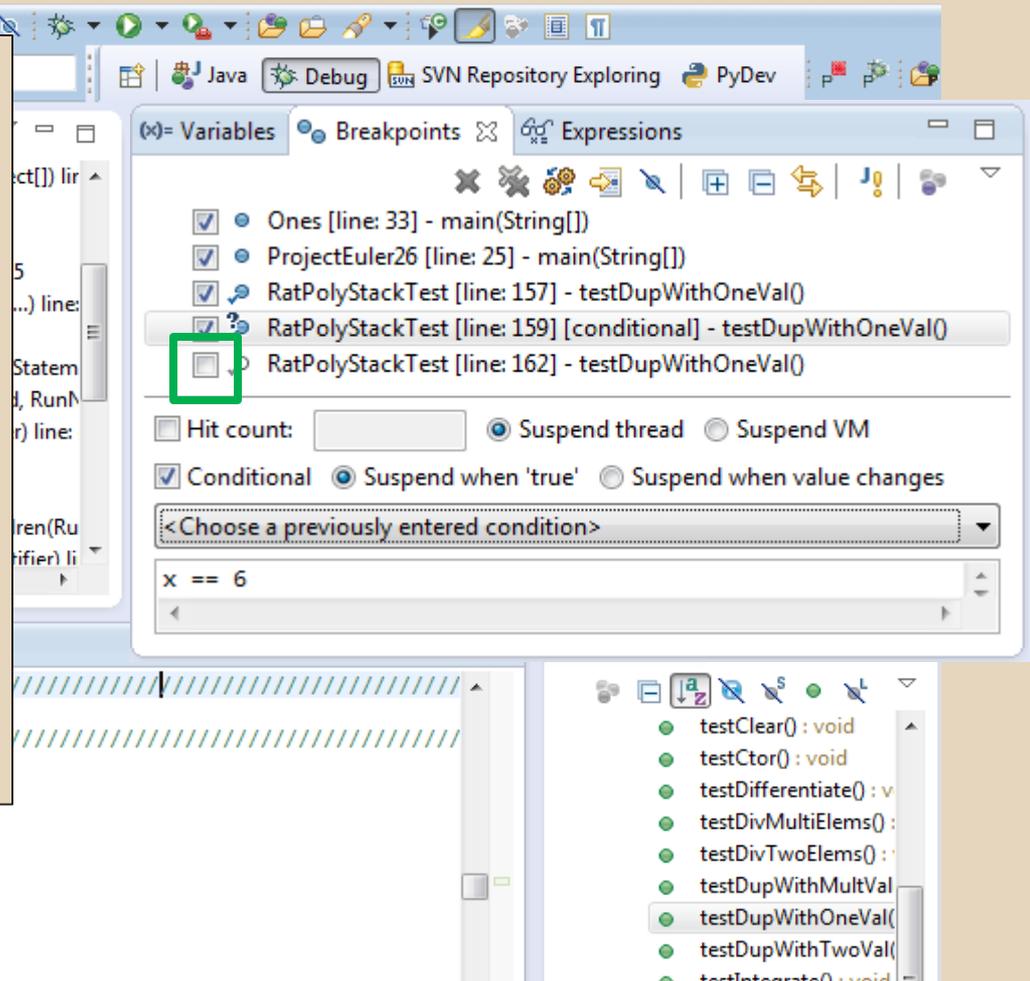
# ECLIPSE DEBUGGING

## 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.

```
156 public void testDupWithOneVal() {  
157     RatPolyStack stk1 = stack("3");  
158     stk1.dup();  
159     assertStackIs(stk1, "33");  
160     stk1 = stack("123");  
161     stk1.dup();  
162     assertStackIs(stk1, "1123");
```



# 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.

The screenshot shows the Eclipse IDE interface during a debugging session. The main editor displays the source code for `DelegatingMethodAccessorImpl.invoke(Object, Object[])`. A breakpoint is set at line 157, which is highlighted in green. The breakpoint configuration dialog is open, showing the following settings:

- Hit count:
- Suspend thread  Suspend VM
- Conditional  Suspend when 'true'  Suspend when value changes
- <Choose a previously entered condition>
- `x == 6`

The background shows the code editor with the following code snippet:

```
153  
154  
155 @Test  
156 public void testDupWithOneVal() {  
157     RatPolyStack stk1 = stack("3");  
158     stk1.dup();  
159     assertStackIs(stk1, "33");  
160     stk1 = stack("123");  
161     stk1.dup();  
162     assertStackIs(stk1, "1123");
```

# ECLIPSE DEBUGGING

## Conditional Breakpoints

Breakpoints can have conditions. This means the breakpoint will only be triggered when a condition you supply is true. **This is very useful** 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.

The screenshot shows the Eclipse IDE interface. The main editor displays Java code with line numbers 159 to 162. A breakpoint is set at line 159. The Breakpoints view on the right shows a list of breakpoints, with the one at line 159 highlighted. The configuration for this breakpoint is shown in a dialog box, where the 'Conditional' checkbox is checked, and the condition 'x == 6' is entered in the text field. The 'Suspend when 'true'' radio button is selected.

```
159  assertStackIs(stk1, "33");
160  stk1 = stack("123");
161  stk1.dup();
162  assertStackIs(stk1, "1123");
```

Breakpoints view:

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

Breakpoint configuration:

- Conditional  Suspend when 'true'  Suspend when value changes
- <Choose a previously entered condition>
- x == 6

Method list:

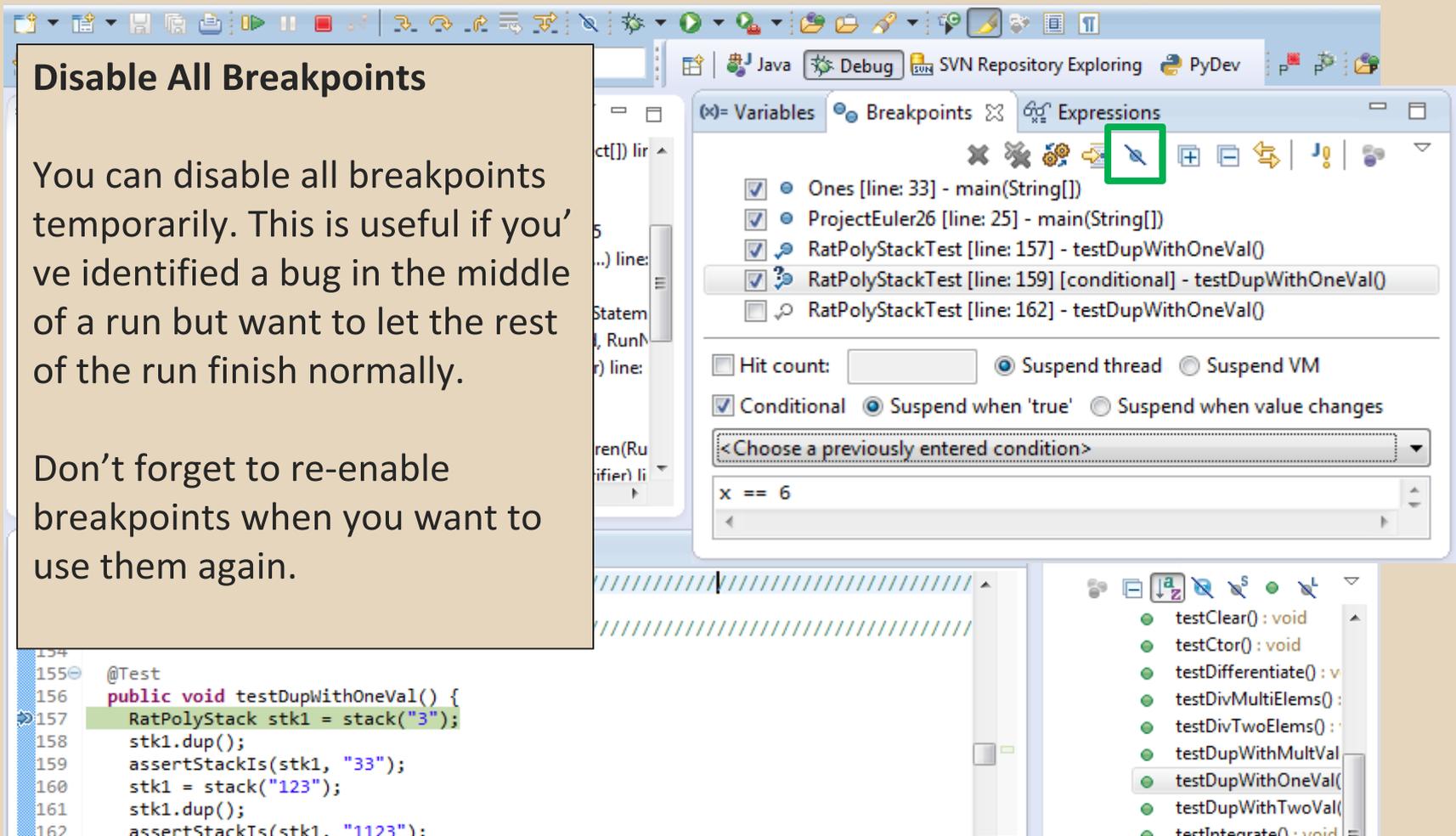
- testClear(): void
- testCtor(): void
- testDifferentiate(): void
- testDivMultiElems(): void
- testDivTwoElems(): void
- testDupWithMultiVal(): void
- testDupWithOneVal(): void**
- testDupWithTwoVal(): void
- testIntegrate(): void

# 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.



The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar contains various icons, including a green box around the 'Disable All' icon (a crossed-out bug). The Breakpoints view is open, showing a list of breakpoints for the current project. The breakpoints are:

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

The conditional breakpoint at line 159 has a condition of `x == 6`. The Breakpoints view also shows options for Hit count, Suspend thread, Suspend VM, and Conditional (Suspend when 'true' or Suspend when value changes). The main editor shows the source code for `testDupWithOneVal()` with line 157 highlighted. The bottom right pane shows a list of methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

# 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.

```
ParentRunner$1.schedule(Runnable) line: 60
BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Ru
ParentRunner<T>.access$000(ParentRunner RunNotifier) li
```

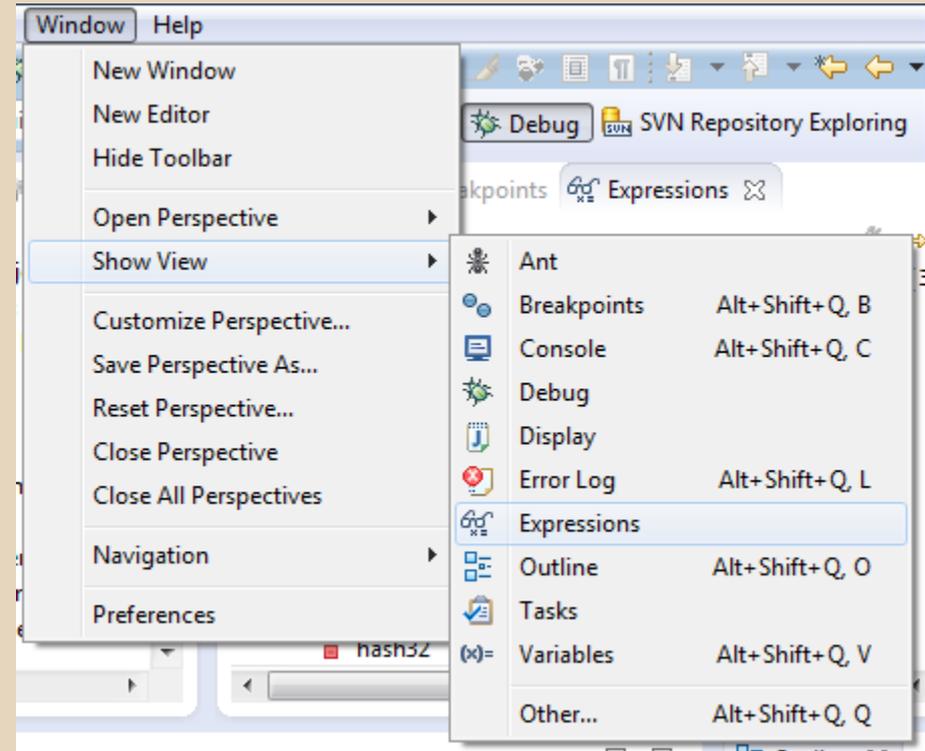
The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for Run, Debug, and Breakpoints. The Breakpoints view on the right shows a list of breakpoints for the project. A conditional breakpoint is set on the method `testDupWithOneVal()` at line 159 of `RatPolyStackTest.java`. The condition is `x == 6`. The Breakpoints view also shows options for Hit count, Suspend thread, Suspend VM, and Conditional (Suspend when 'true' or Suspend when value changes). The main editor shows the source code of `RatPolyStackTest.java` with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The right-hand side of the IDE shows a list of methods in the `RatPolyStack` class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

# 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

The screenshot shows the Eclipse IDE's Expressions Window. The window is titled "Expressions" and contains a table with the following data:

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

The background shows a Java code editor with the following code:

```
157 RatPolyStack stk1 = stack( 3 );
158 stk1.dup();
159 assertStackIs(stk1, "33");
160 stk1 = stack("123");
161 stk1.dup();
162 assertStackIs(stk1, "1123");
```

Below the code editor, a list of test methods is visible:

- testClear(): void
- testCtor(): void
- testDifferentiate(): void
- testDivMultiElems(): void
- testDivTwoElems(): void
- testDupWithMultVal(): void
- testDupWithOneVal(): void
- testDupWithTwoVal(): void
- testIntegrate(): void

# ECLIPSE DEBUGGING

## Expressions Window

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

The screenshot shows the Eclipse IDE interface during a debug session. The Expressions window is highlighted with a green border and contains the following data:

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

The background shows the Java source code for `RatPolyStackTest.java` with the following content:

```
151 ////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
```

# 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