

SECTION 1: VERSION CONTROL + ECLIPSE

cse331-staff@cs.washington.edu

slides borrowed and adapted from Alex Mariakis and CSE 390a

OUTLINE

- **Version control**
-

- **IDEs – Eclipse**
- **Debugging**

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

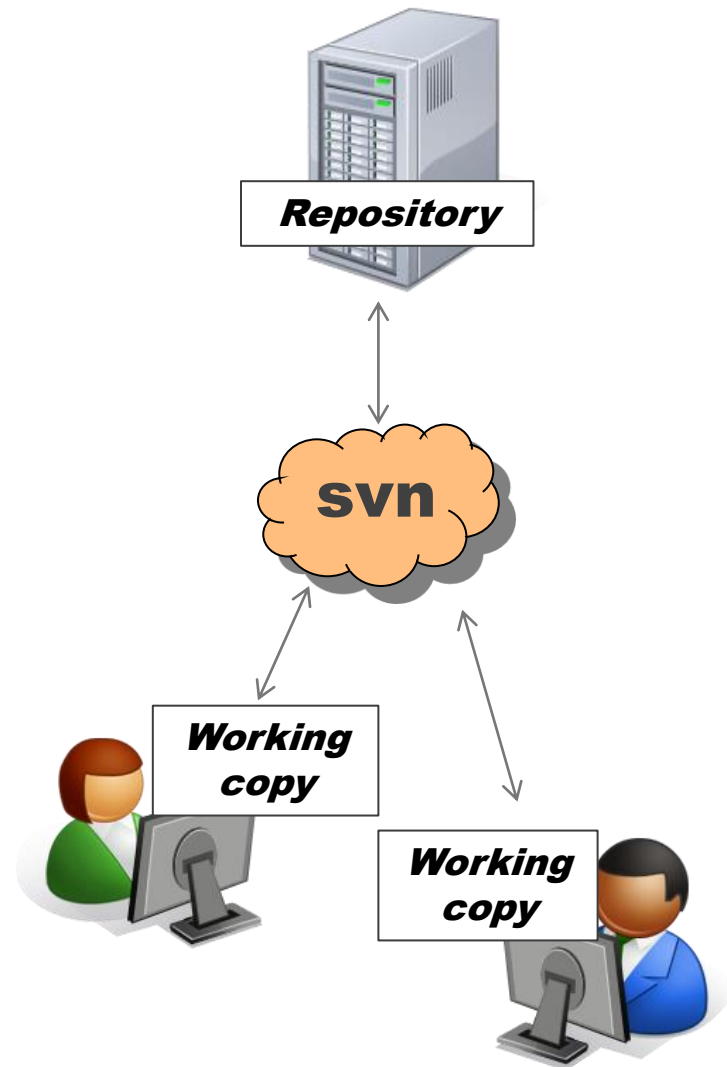
VERSION CONTROL ORGANIZATION

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)**

VERSION CONTROL

COMMON ACTIONS

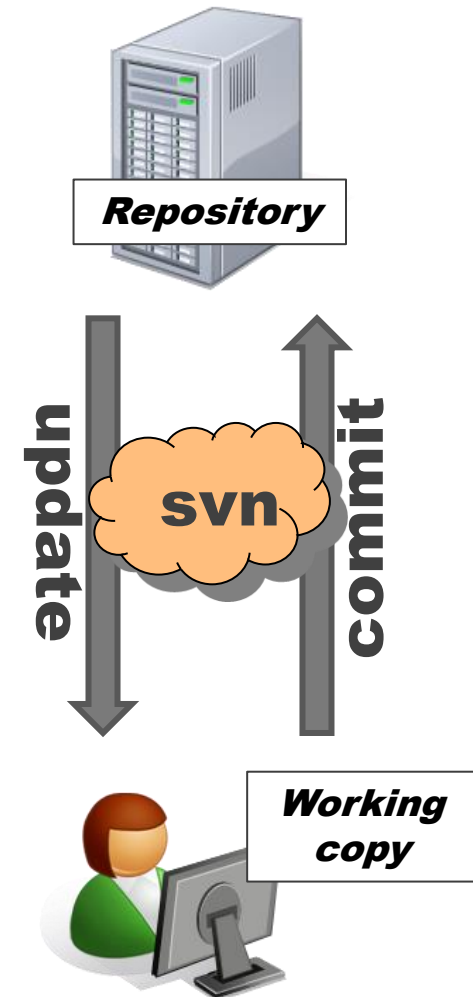
Most common commands:

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

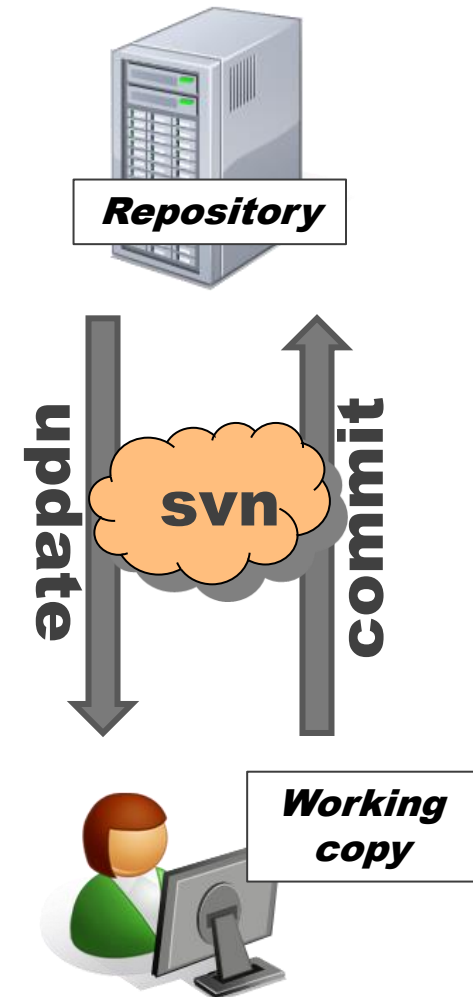
- add or delete a file in the repository
- just putting a new file in your working copy does not add it to the repo!

Revert

- wipe out your local changes to a file

Resolve, diff, merge

- handle a conflict – two users editing the same code



HOW TO USE SUBVERSION

- 1. Eclipse plugin: Subclipse**
- 2. GUI interface: TortoiseSVN, NautilusSVN**
- 3. Command line: PuTTY**

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

DEMO!

<http://www.cs.washington.edu/education/courses/cse331/14au/tools/versioncontrol.html>

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 debug session. The top toolbar includes standard development icons and a 'Quick Access' search bar. The main workspace is divided into several panels:

- Debug Console:** Shows the execution stack with the following entries:
 - 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
- Variables View:** A table showing the current state of variables:

Name	Value
this	RatPolyStackTest (id=33)
- Source Code Editor:** Displays the file `RatPolyStackTest.java`. The current line of execution is highlighted in green:

```
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 in the class, with `testDupWithOneVal()` selected:
 - testClear(): void
 - testCtor(): void
 - testDifferentiate(): v
 - testDivMultiElems():
 - testDivTwoElems():
 - testDupWithMultVal
 - testDupWithOneVal()
 - testDupWithTwoVal()
 - testIntegrate(): void

ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes standard development icons. Below it, the 'Quick Access' search bar is visible. The main workspace is divided into several panes:

- Debug Console:** Shows the current execution stack with the following entries:
 - 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
- Variables View:** Shows a table with the following data:

Name	Value
this	RatPolyStackTest (id=33)
- Code Editor:** Displays the source code for `RatPolyStackTest.java`. A green arrow icon in the left margin indicates a breakpoint is set on line 57. The code includes comments and assertions:

```
51 ////////////////////////////////////////////////////
52 //
53 //
54 //
55 @
56 p
57
58
59
60
61 stk1.dup();
62 assertStackTs(stk1, "1123");
```
- Outline View:** Shows the class structure on the right side of the editor.

A text box overlaid on the code editor provides instructions: "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

The screenshot shows the Eclipse IDE interface during a debug session. At the top, a toolbar contains several icons for debugging, with a green box highlighting the Run, Breakpoint, Step Over, Step Into, and Step Return buttons. Below the toolbar is the 'Quick Access' search bar. The main workspace is divided into three panes:

- Debug Console:** Located on the left, it displays a stack trace of the current thread, listing various Java classes and methods such as `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, and `ParentRunner$3.run`.
- Variables View:** Located on the right, it shows a table with a 'Name' column and a 'Value' column, currently empty.
- Code Editor:** Located at the bottom, it shows the source code for `RatPolyStackTest.java`. The current line is 157, which is highlighted in green: `stk1 = stack("3");`. The code includes comments, a `@Test` annotation, and several assertions.

At the bottom right, the **Outline** view shows a list of methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`. The `testDupWithOneVal()` method is currently selected.

Controlling your program while debugging is done with these buttons

ECLIPSE DEBUGGING

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

Name	Value
this	RatPolyStackTest (id=33)

```
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");
```

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

ECLIPSE DEBUGGING

The screenshot shows the Eclipse IDE interface. The top toolbar has a green box around the 'Step Into' icon. The Debug console on the left shows a stack trace of method calls. The code editor at the bottom shows a Java file named 'RatPolyStackTest.java' with a breakpoint at line 157. The code at line 157 is `RatPolyStack stk1 = stack("3");`. The right side of the IDE shows a list of test methods, with 'testDupWithOneVal()' selected.

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.

```
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 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

The image shows the Eclipse IDE interface. At the top, the toolbar has a green box around the 'Step Through Filters' icon. Below it, the 'Preferences' dialog is open, with the 'Step Filtering' section selected in the left-hand tree. The 'Step Filtering' section has a green box around its title. The 'Use Step Filters' checkbox is checked. Under 'Defined step filters', several packages are listed with checkboxes. The 'java.lang.ClassLoader' checkbox is checked and has a green circle next to it. At the bottom of the dialog, the 'OK' button is highlighted with a green box. To the right, a code editor shows a list of methods, each with a green circle next to it, indicating they are being skipped during 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

The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons for file operations 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 displays a stack trace with the following entries (from top to bottom):

- 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

The stack trace is highlighted with a green border. To the right of the stack trace is the 'Variables' pane, which shows a table with a 'Name' column and a single entry 't'. Below the stack trace is the 'Code Editor' pane, which displays the source code for 'RatPolyStackTest.java'. 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 is highlighted in green. On the right side of the IDE, there is a 'Stack Trace' pane with the title 'Stack Trace'. It contains the following text:

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.

At the bottom right, there is a list of test methods with green circular icons next to them:

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

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 shows the 'Debug' button is active. The main window is divided into several panes:

- Variables Window:** A table showing the current state of variables. The 'expt' variable is highlighted in yellow, indicating it has changed since the last breakpoint. Its value is 5.
- Code Editor:** Shows the source code for `RatPolyStackTest.java`. Line 157 is highlighted, corresponding to the current execution point.
- Outline Window:** Lists the methods in the class, with `testDupWithOneVal()` selected.

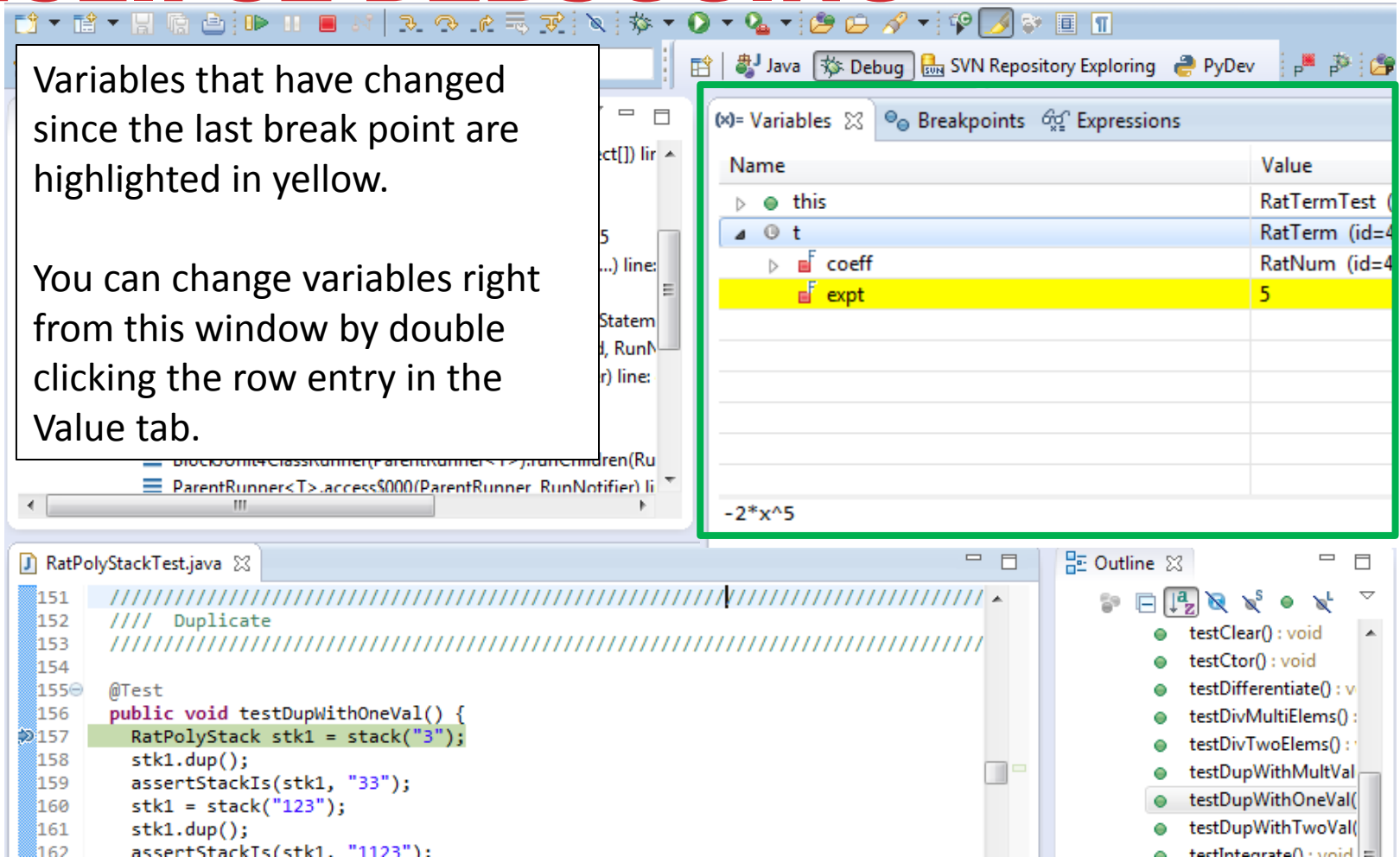
Name	Value
this	RatTermTest (
t	RatTerm (id=4
coeff	RatNum (id=4
expt	5

```
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

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 shows the Eclipse IDE interface during a debug session. The top toolbar includes icons for running, debugging, and other IDE functions. The main window is divided into several panes:

- Variables Window:** A table showing the current state of variables. The 'expt' variable is highlighted in yellow, indicating it has changed since the last breakpoint. The table has two columns: 'Name' and 'Value'.

Name	Value
this	RatTermTest (
t	RatTerm (id=4
coeff	RatNum (id=4
expt	5
- Code Editor:** Shows the source code for 'RatPolyStackTest.java'. The method `testDupWithOneVal()` is visible, and the line `RatPolyStack stk1 = stack("3");` is highlighted in green, corresponding to the current line of execution in the debug view.
- Outline Window:** Shows a list of methods in the current class, including `testClear(): void`, `testCtor(): void`, `testDifferentiate(): v`, `testDivMultiElems(): :`, `testDivTwoElems(): :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate(): void`.

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 Variables view is open, showing a tree structure with 'this' expanded to 't', which contains 'coeff' and 'expt'. The 'expt' variable is selected, and a right-click context menu is displayed over it. The menu items include 'Select All', 'Copy Variables', 'Find...', 'Change Value...', 'All References...', 'All Instances...' (highlighted), 'Instance Count...', 'New Detail Formatter...', 'Open Declared Type', 'Open Declared Type Hierarchy', 'Instance Breakpoints...', 'Watch', and 'Inspect'. The background shows a code editor with a Java class 'Runner.class' containing a test method 'testDupWithOneVal()' with several lines of code. The bottom of the screen shows a list of test cases.

```
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");
163 }
```

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)

testDupWithOneVal() [●]
testDupWithTwoVal() [●]
testIntegrate0: void [●]

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)

```
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");
```

Variables Breakpoints Expressions

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

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

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.

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

- Breakpoints Window:** Located in the upper right, it lists several breakpoints:
 - Ones [line: 33] - main(String[])
 - ProjectEuler26 [line: 25] - main(String[])
 - RatPolyStackTest [line: 157] - testDupWithOneVal()
 - RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()** (highlighted with a green border)
 - RatPolyStackTest [line: 162] - testDupWithOneVal()Below the list, there are options for "Hit count", "Suspend thread", "Suspend VM", "Conditional", "Suspend when 'true'", and "Suspend when value changes". A dropdown menu shows "<Choose a previously entered condition>" and a text field contains the condition "x == 6".
- Code Editor:** The bottom left pane shows the source code for `RatPolyStackTest.java`. Line 159 is highlighted in green, corresponding to the selected breakpoint:

```
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:** The bottom right pane shows a list of methods in the `RatPolyStackTest` class, with `testDupWithOneVal()` selected.

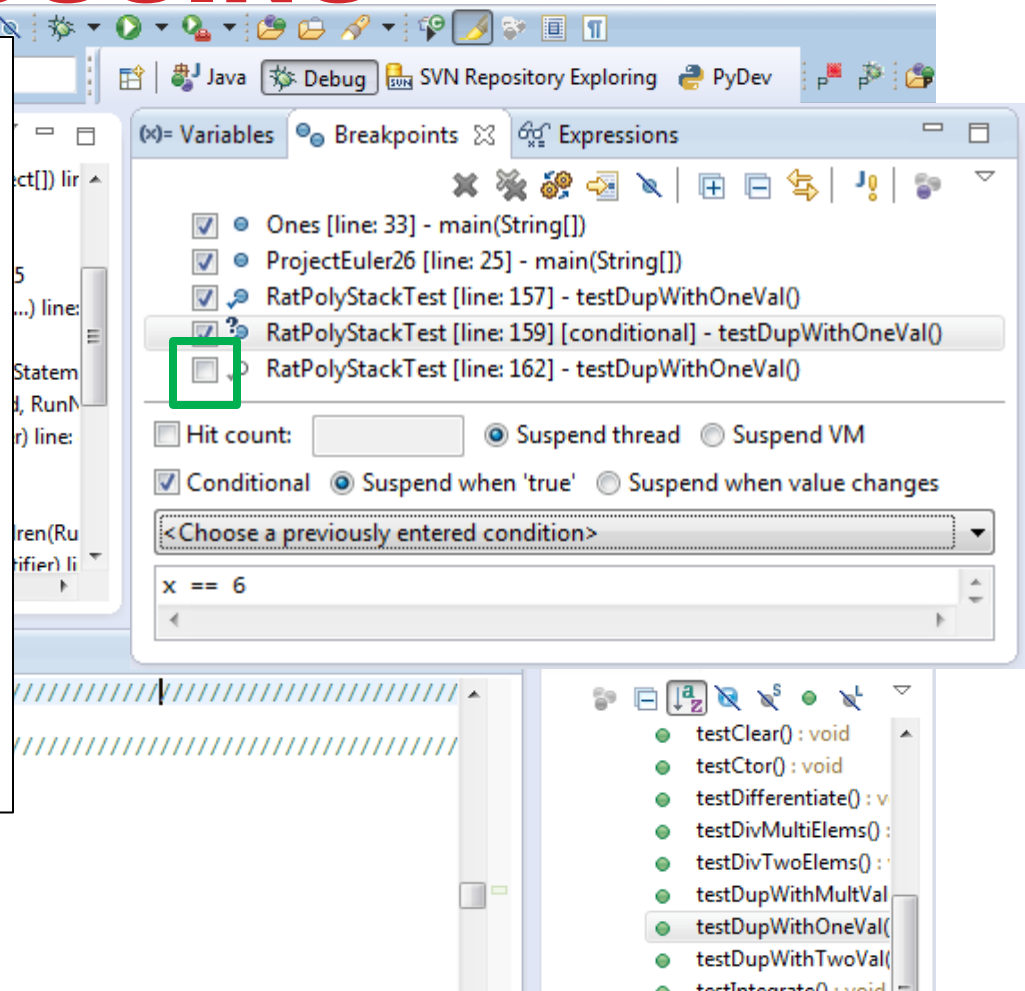
ECLIPSE DEBUGGING

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 following Java code:

```
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 breakpoint configuration dialog is open, showing the following settings:

- Hit count:
- Suspend thread Suspend VM
- Conditional Suspend when 'true' Suspend when value changes
- Condition:

The background shows the Eclipse IDE interface with a code editor displaying a Java method 'testDupWithOneVal()' and a list of test methods on the right.

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.

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

The screenshot shows the Eclipse IDE interface. The Breakpoints view is open, displaying a list of breakpoints. A conditional breakpoint is highlighted for line 159 in the file `RatPolyStackTest`, with the condition `x == 6`. The breakpoint is set to "Suspend when 'true'". The code editor shows the following code:

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

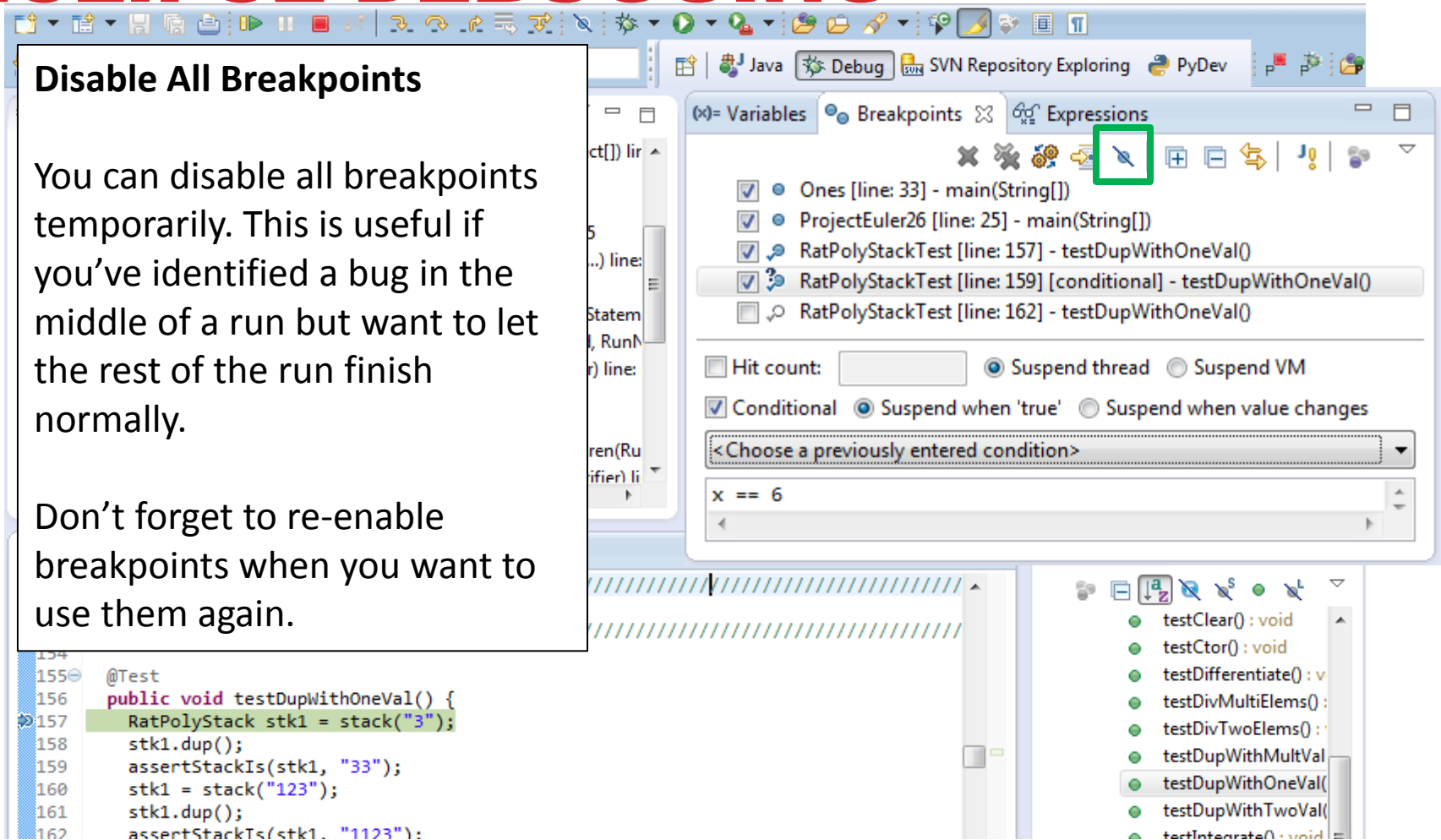
The Breakpoints view also shows other breakpoints for lines 33, 25, 157, and 162. The Hit count is set to 1. The breakpoint is set to "Suspend thread". The condition is `x == 6`.

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 Breakpoints view is open, displaying a list of breakpoints for the file 'RatPolyStackTest.java'. A green box highlights the 'Disable All Breakpoints' icon (a blue circle with a diagonal slash) in the toolbar of the Breakpoints view. The list of breakpoints includes:

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

Below the list, the 'Hit count' is set to 0, and the 'Suspend' mode is set to 'Suspend thread'. The 'Conditional' checkbox is checked, and the 'Suspend when' mode is set to 'true'. The condition field contains the expression 'x == 6'. The main editor shows the source code for 'RatPolyStackTest.java' with line 157 highlighted: 'RatPolyStack stk1 = stack("3");'. The Package Explorer on the right shows the project structure with 'testDupWithOneVal()' selected.

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 contains various icons for file operations, running, and debugging. The main window is divided into several panes:

- Breakpoints:** A list of breakpoints is shown, including:
 - Ones [line: 33] - main(String[])
 - ProjectEuler26 [line: 25] - main(String[])
 - RatPolyStackTest [line: 157] - testDupWithOneVal()
 - RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()** (highlighted with a green box)
 - RatPolyStackTest [line: 162] - testDupWithOneVal()
- Conditional:** The configuration for the selected breakpoint is shown:
 - Hit count:
 - Suspend thread Suspend VM
 - Conditional Suspend when 'true' Suspend when value changes
 - Condition:
- Code Editor:** The source code for `RatPolyStackTest.java` is displayed, with line 159 highlighted:

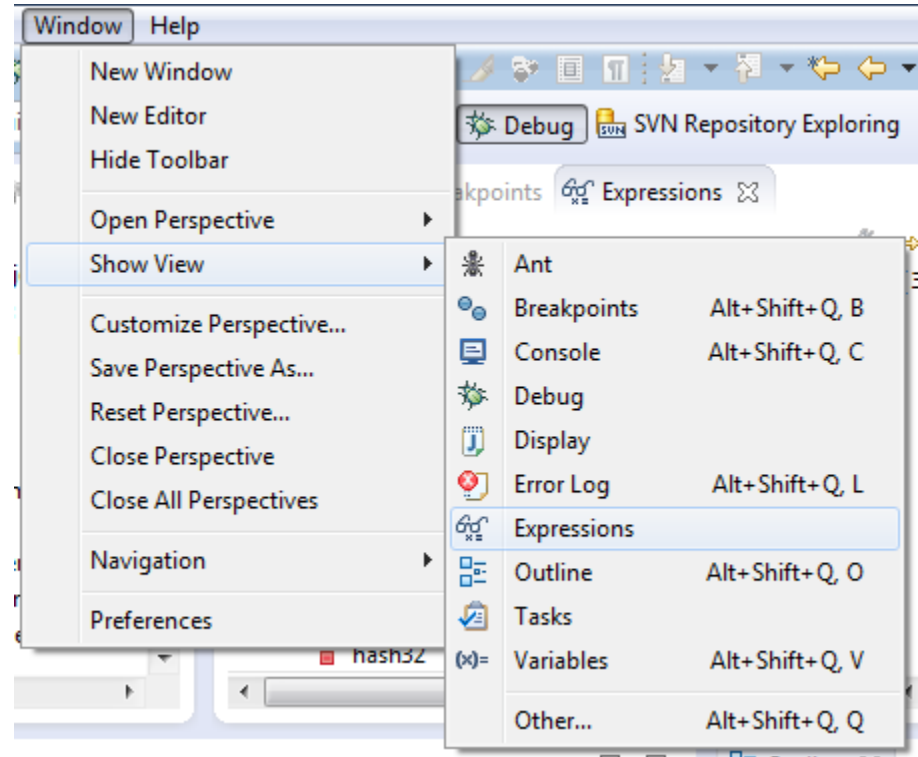
```
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
```
- Outline:** A list of methods in the class is shown, including `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

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

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

- 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 editor with the following code snippet from `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");
163 }
```

Demo 2!!

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