



# Section 2:

## Developer tools and you

Alex Mariakakis

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

# What is an SSH client?

- Uses the secure shell protocol (SSH) to connect to a remote computer
  - Enables you to work on a lab machine from home
  - Similar to remote desktop
- Windows and Linux users: PuTTY and WinSCP
  - Windows and Linux users
- Mac users: Terminal application
  - Go to Applications/Utilities/Terminal
  - Type in “ssh -XY cseNetID@attu.cs.washington.edu”

# What is UNIX?

- Command-line based operating system
  - Like Windows or Mac OS without the mouse

Command	What it does
pwd	<u>p</u> rints the name of the <u>w</u> orking <u>d</u> irectory
ls	lists the files in a directory (i.e., <u>l</u> ists <u>s</u> uff)
cd	<u>c</u> hanges a <u>d</u> irectory
cp	<u>c</u> opies a file or directory
mv	<u>m</u> ove/rename a file or directory
rm	<u>r</u> emoves a file
mkdir	<u>m</u> ake a new <u>d</u> irectory
rmdir	<u>r</u> emove an empty <u>d</u> irectory
man	pulls up the <u>m</u> anual pages

# DEMO #1

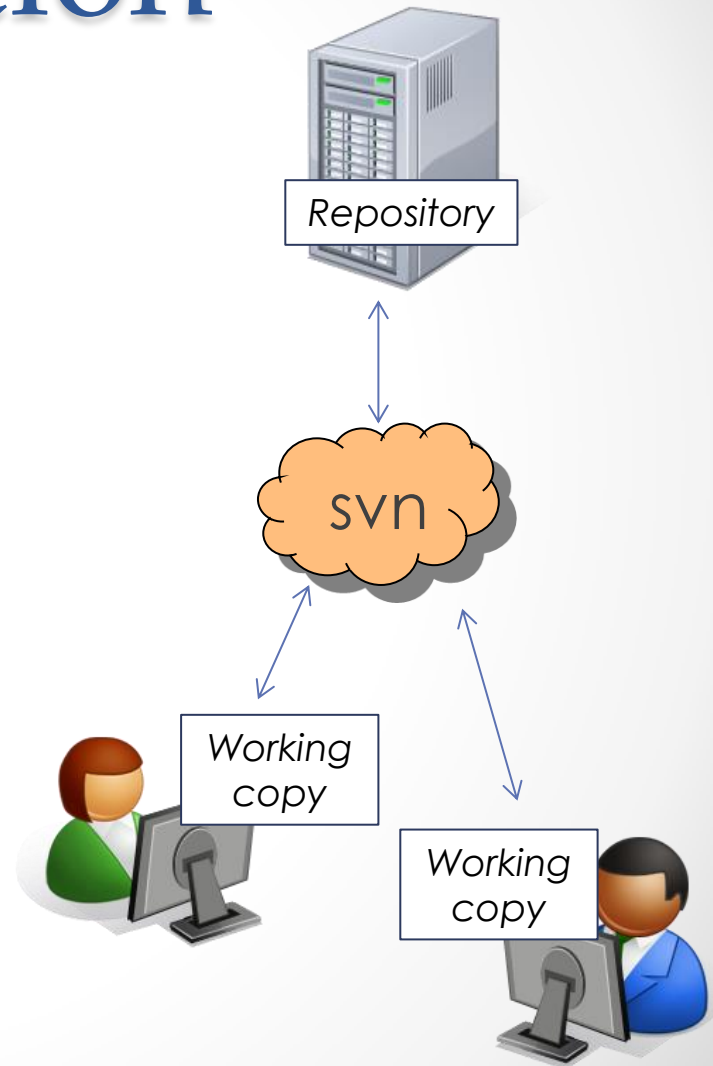
<http://courses.cs.washington.edu/courses/cse331/14sp/tools/WorkingAtHome.html>

# What is source control?

- Also known as version 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
  - Back up your work
  - Merge changes from multiple sources
- We'll be talking about Subversion, but there are alternatives
  - ✓ Git, Mercurial, CVS
  - × Email, Dropbox, USB sticks

# Source control organization

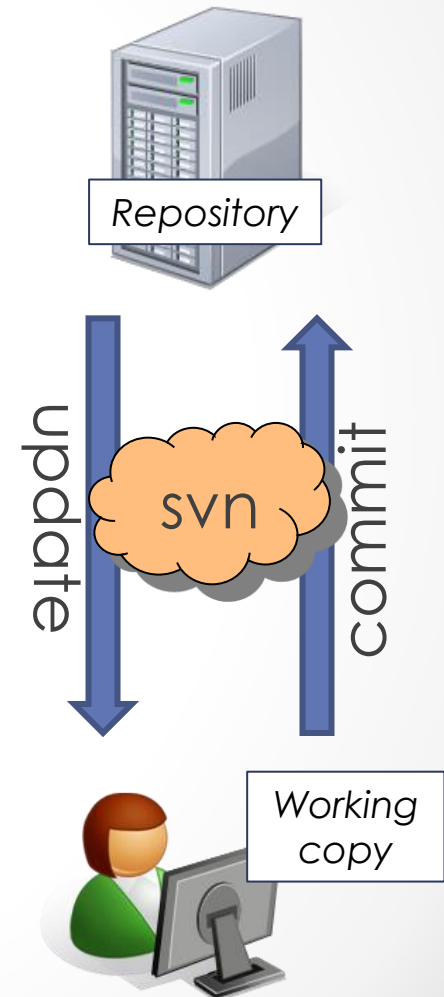
- A *repository* stores the master copy of the project
  - Someone creates the repo for a new project
  - Then nobody touches this copy directly
  - Lives on a server everyone can access
- Each person *checks out* her own *working copy*
  - Makes a local copy of the repo
  - You'll always work off of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



# Source control common actions

Most common commands:

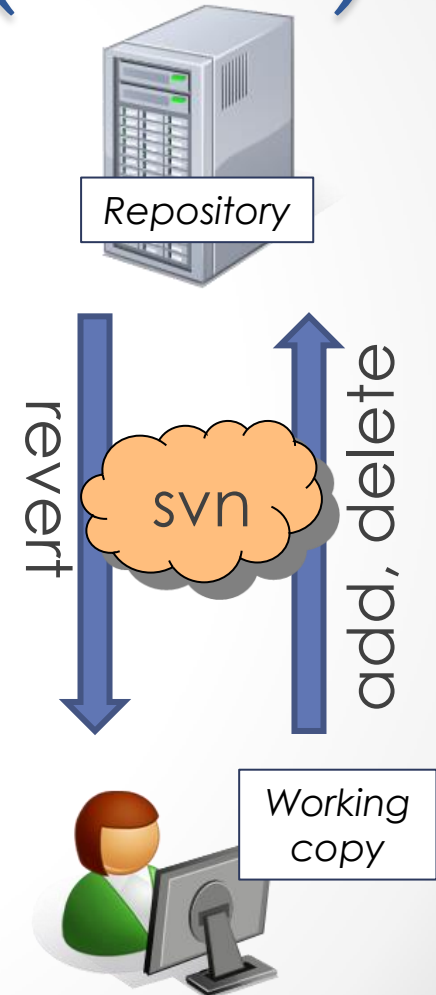
- **Commit / checkin**
  - integrate changes *from* your working copy *into* the repository
- **Update**
  - integrate changes *into* your working copy *from* the repository



# Source 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 just as you would have in your previous classes
- 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 #2

[http://www.cs.washington.edu/education/courses/  
cse331/14sp/tools/versioncontrol.html](http://www.cs.washington.edu/education/courses/cse331/14sp/tools/versioncontrol.html)

# Theoretical Scenario

You are working on a computer in the lab and feel like you are at a good stopping point. You hit “Save” on your computer and start driving to Canada, only to realize that you forgot to commit your changes to your repo. Do you have to turn around and drive back?

**No, SSH into the attu, change to your eclipse workspace directory, and then call “svn commit”**

# 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. The 'Quick Access' search bar is visible. The 'Debug' tab is active, showing the 'Debug console' with a list of stack frames. The 'Variables' view on the right shows the current scope with a single variable 'this' of type 'RatPolyStackTest (id=33)'. The 'Outline' view on the bottom right lists the methods of the current class. The 'RatPolyStackTest.java' editor shows the source code with line 157 highlighted, indicating the current execution point.

**Debug Console Stack Frames:**

- 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, Runnable) line: not available
- ParentRunner\$3.run() line: 231
- ParentRunner\$1.schedule(Runnable) line: 60
- BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Runnable) line: not available
- ParentRunner<T>.access\$000(ParentRunner, Runnable) line: not available

**Variables View:**

Name	Value
this	RatPolyStackTest (id=33)

**Outline View:**

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

**RatPolyStackTest.java Source Code:**

```
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 }
```

# Eclipse Debugging

The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar includes icons for file operations, running, and debugging. The 'Quick Access' search bar is visible. The 'Debug' tab is active, showing a list of stack frames in the 'Debug' console. The 'Variables' view on the right shows a single variable 'this' with the value 'RatPolyStackTest (id=33)'. The 'RatPolyStackTest.java' editor is open, showing lines 51 to 62. A green vertical bar on the left margin indicates a breakpoint is set at line 57. A text box is overlaid on the editor, explaining how to set a breakpoint.

Debug console stack frames:

- 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, Runnable) line: not available
- ParentRunner\$3.run() line: 231
- ParentRunner\$1.schedule(Runnable) line: 60
- BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Runnable) line: not available
- ParentRunner<T>.access\$000(ParentRunner, Runnable) line: not available

Variables view:

Name	Value
this	RatPolyStackTest (id=33)

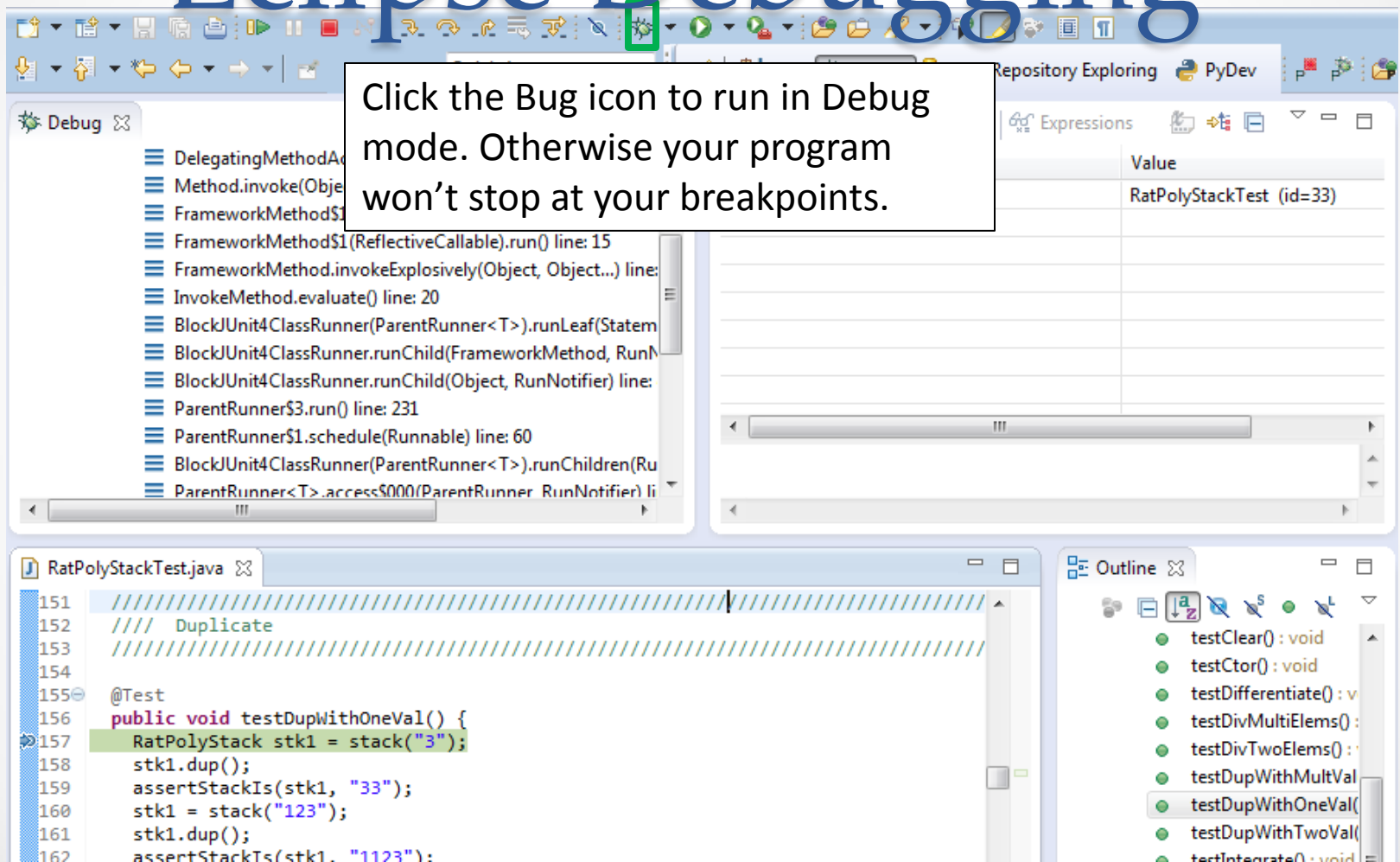
Code editor (RatPolyStackTest.java):

```
51 ////////////////////////////////////////////////////
52 /
53 /
54 @
55 p
56
57
58
59
60
61 stk1.dup();
62 assertStackIs(stk1, "1123");
```

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

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar contains several icons, with a green box highlighting the Run, Breakpoint, and Step-through icons. The Debug console on the left shows a stack of method calls, including `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate`, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `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`, and `ParentRunner<T>.access$000(ParentRunner RunNotifier) li`. The Variables view on the right shows a table with columns for Name and Value. The main editor window displays the source code of `RatPolyStackTest.java`, with line 157 highlighted: `@Test`, `public void testDupWithOneVal() {`, `RatPolyStack stk1 = stack("3");`, `stk1.dup();`, `assertStackIs(stk1, "33");`, `stk1 = stack("123");`, `stk1.dup();`, and `assertStackIs(stk1, "1123");`. The Outline view on the bottom right shows a list of methods, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : v`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate() : void`.

Controlling your program while debugging is done with these buttons

# Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar features a green play button, a pause button, and a red stop button, all of which are highlighted with a green rectangular box. A text box with a black border is positioned over the pause and stop buttons, containing the text: "Play, pause, stop work just like you'd expect".

The Debug console on the left shows a stack trace of the current execution state, listing various method calls and their corresponding line numbers. The Variables view on the right shows the current state of the program's variables, including a table with columns for Name and Value. The table contains one entry: "this" with the value "RatPolyStackTest (id=33)".

The source code editor at the bottom shows the file "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");
163 }
```

The Outline view on the right shows the structure of the project, listing various methods and their locations. The methods listed are: testClear(), testCtor(), testDifferentiate(), testDivMultiElems(), testDivTwoElems(), testDupWithMultVal(), testDupWithOneVal(), testDupWithTwoVal(), and testIntegrate(). The method testDupWithOneVal() is currently selected.

# Eclipse Debugging

The screenshot shows the Eclipse IDE interface. At the top, the toolbar contains various icons, with a green box highlighting the 'Step Into' icon (a green arrow pointing into a box). Below the toolbar, the 'Debug' console is open, displaying a call stack of method calls. The 'RatPolyStackTest.java' editor is open at the bottom, showing a Java class with a test method. A breakpoint is set at line 157, which is highlighted in green. The 'Variables' view on the right shows a list of variables, with 'testDupWithOneVal()' selected.

**Step Into**

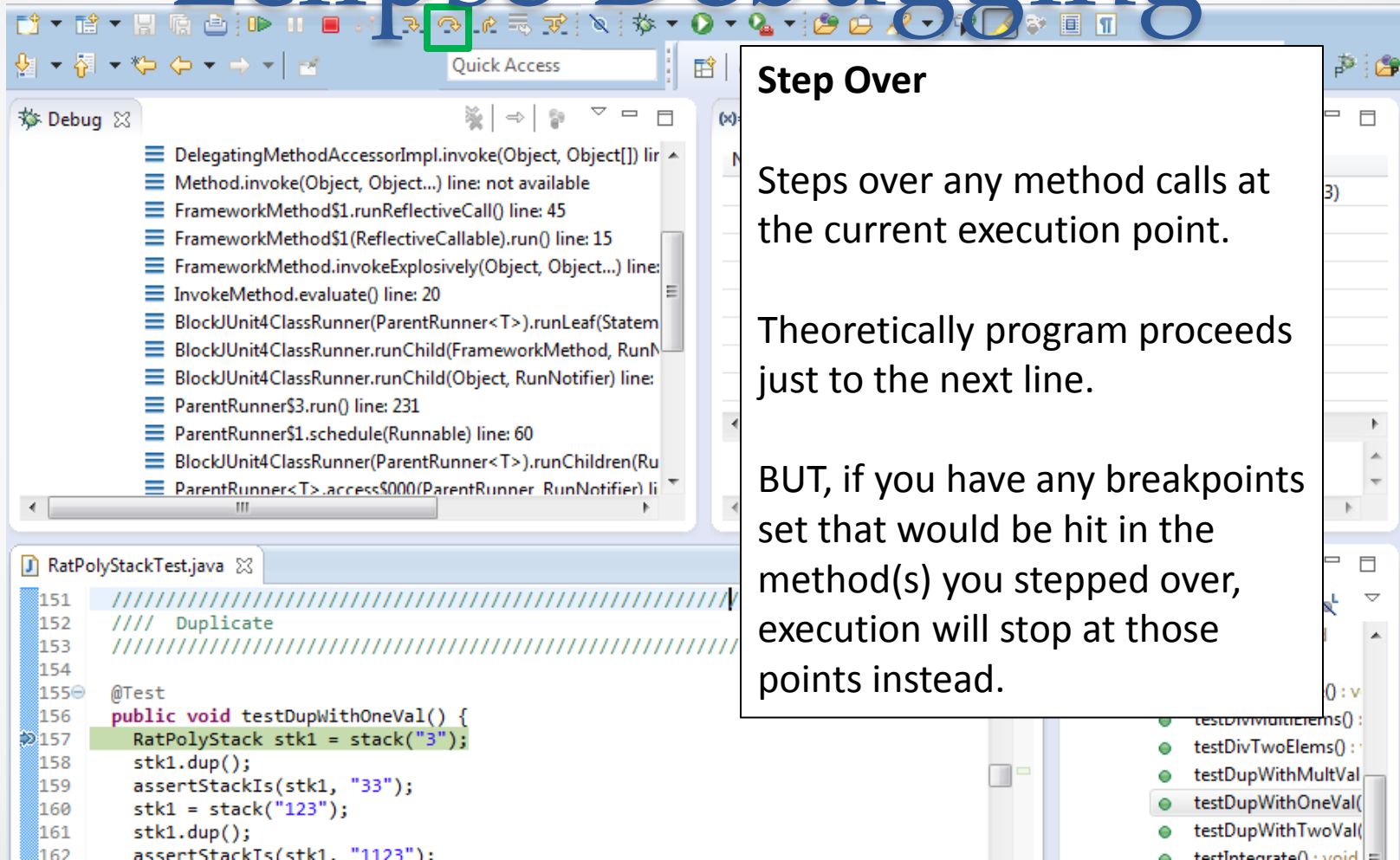
Steps into the method at the current execution point – if possible. If not possible then just proceeds to the next execution point.

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

```
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");
}
```

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

# Eclipse Debugging



The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons, with the 'Step Over' icon (a right-pointing arrow with a small circle above it) highlighted by a green square. Below the toolbar, the 'Debug' console is open, displaying a stack trace of method calls. The bottom editor shows the source code of 'RatPolyStackTest.java'. The code includes a test method 'testDupWithOneVal()' which creates a stack, duplicates it, and asserts its contents. The 'Step Over' icon is highlighted in the toolbar, and the 'testDupWithOneVal()' method is highlighted in the source code.

**Step Over**

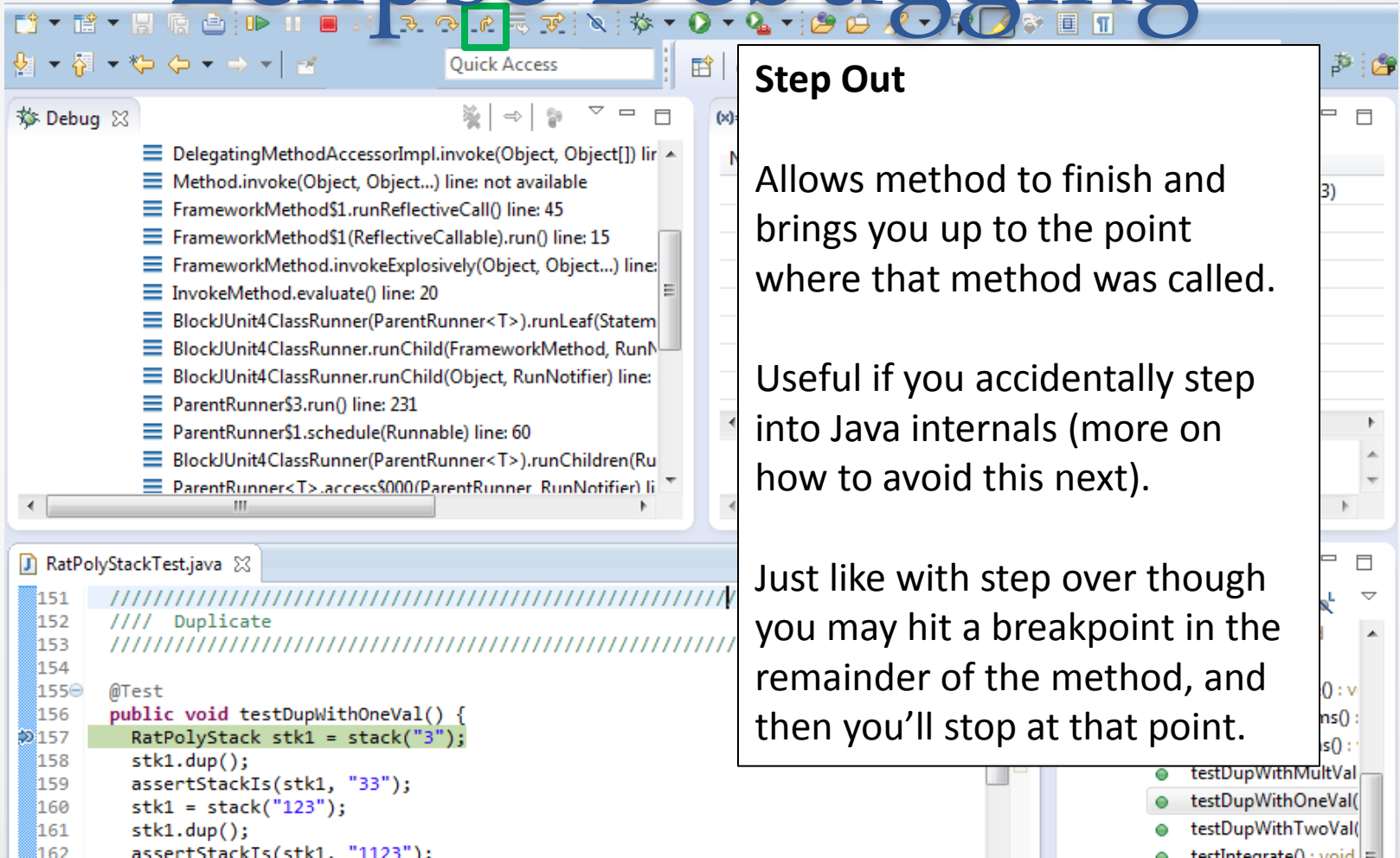
Steps over any method calls at the current execution point.

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");
163 }
```

# Eclipse Debugging



The screenshot shows the Eclipse IDE interface. At the top, the toolbar contains various icons, with the 'Step Out' icon (a green square with a white arrow pointing right) highlighted by a green box. Below the toolbar, the 'Debug' console is open, displaying a stack trace of method calls. The stack trace includes methods like `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate`, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `BlockJUnit4ClassRunner.runChild(FrameworkMethod, RunNotifier)`, `BlockJUnit4ClassRunner.runChild(Object, RunNotifier)`, `ParentRunner$3.run`, `ParentRunner$1.schedule(Runnable)`, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(RunNotifier)`, and `ParentRunner<T>.access$000(ParentRunner, RunNotifier)`. Below the console, the source code of `RatPolyStackTest.java` is visible. The code includes a `@Test` annotation and a `testDupWithOneVal()` method. The line `RatPolyStack stk1 = stack("3");` is highlighted in green. On the right side, a list of test methods is shown, including `testDupWithMultVal`, `testDupWithOneVal`, `testDupWithTwoVal`, and `testIntegrate0: void`.

## 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.

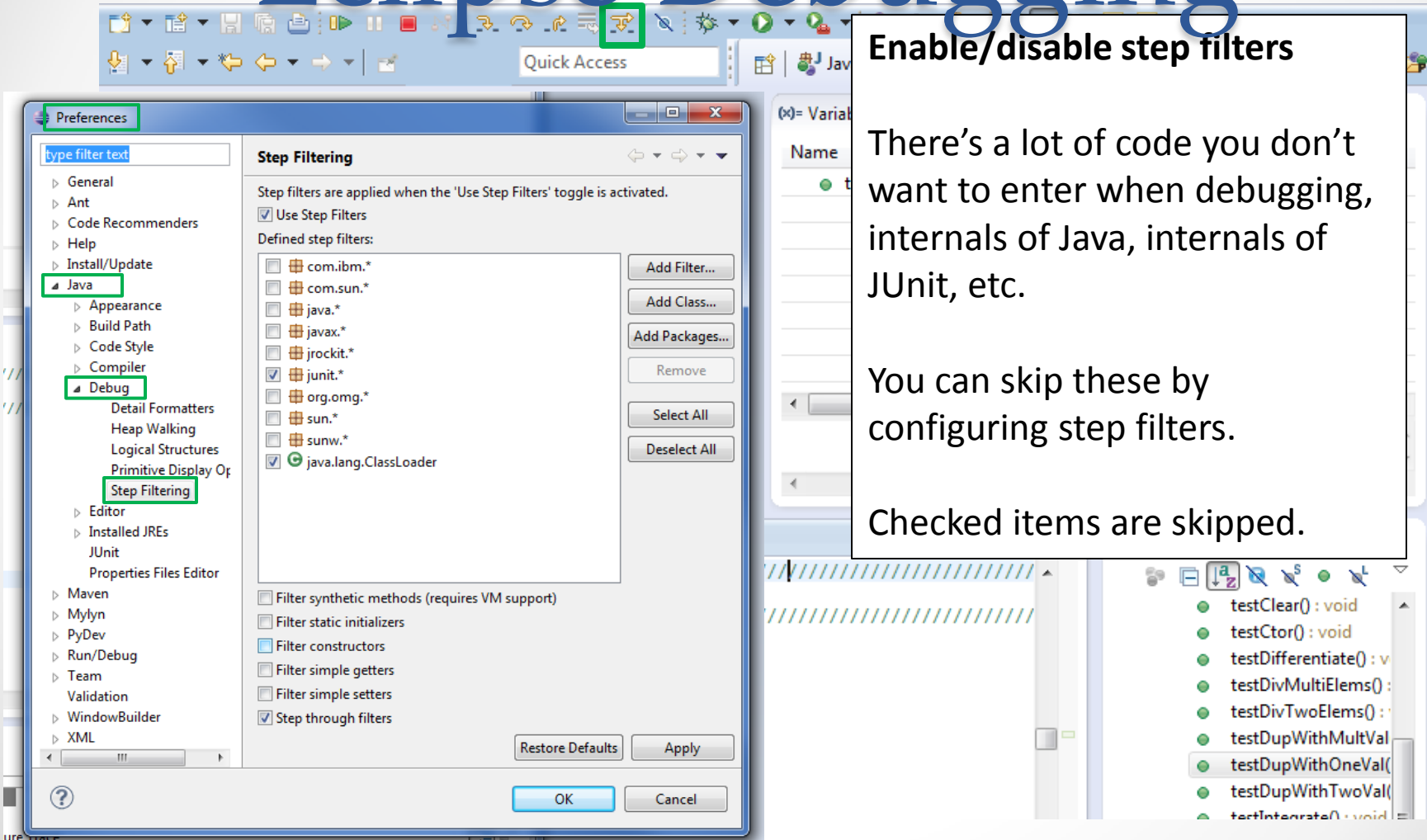
# Eclipse Debugging

## 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

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 editor area is divided into three panes. The left pane, titled 'Debug', shows a stack trace 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. The middle pane shows the source code of 'RatPolyStackTest.java' with line numbers 151 to 162. Line 157 is highlighted, showing the code: 'RatPolyStack stk1 = stack("3");'. The right pane shows a list of methods, with 'testDupWithOneVal()' selected.

**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.

```
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");
}
```

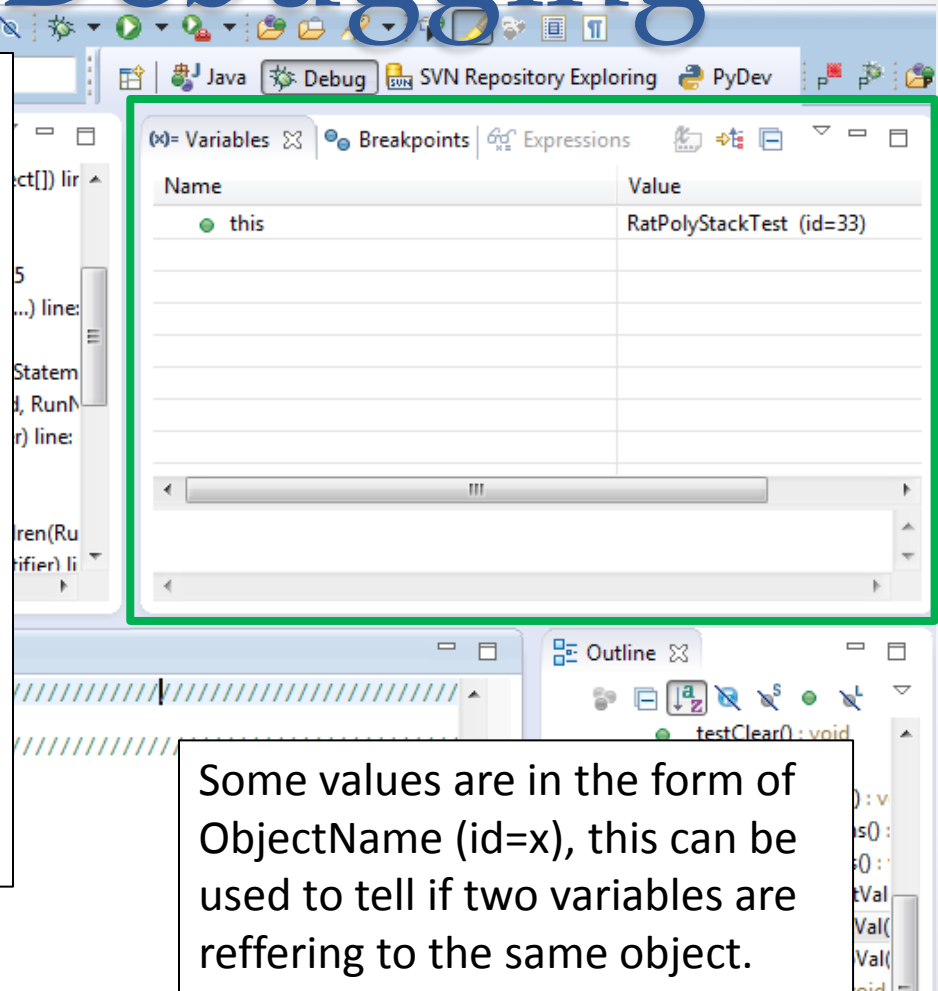
testDifferentiate(): void  
testDivMultiElems(): void  
testDivTwoElems(): void  
testDupWithMultVal(): void  
testDupWithOneVal(): void  
testDupWithTwoVal(): void  
testIntegrate(): void

# 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 includes icons for file operations, running, and debugging. The 'Variables' window is open, showing a table of current variables. The 'this' variable is of type 'RatTermTest', 't' is of type 'RatTerm (id=4)', 'coeff' is of type 'RatNum (id=4)', and 'expt' is of type 'int' with a value of 5. The 'expt' row is highlighted in yellow. The source code editor shows the file 'RatPolyStackTest.java' with lines 151 through 162. Line 157 is highlighted, showing the code: `RatPolyStack stk1 = stack("3");`. The 'Outline' window on the right lists the methods of the class, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() : void`, `testDivTwoElems() : void`, `testDupWithMultVal() : void`, `testDupWithOneVal() : void`, `testDupWithTwoVal() : void`, and `testIntegrate() : void`.

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");
163 }
```

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

# 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. The top toolbar includes icons for file operations, running, and debugging. The 'Variables' view is open, showing a table of variables. The 'Value' tab is selected, and the variable 'expt' is highlighted in yellow, indicating it has changed since the last breakpoint. The 'Outline' view on the right shows the method list for the current class, with 'testDupWithOneVal()' selected. The source code editor at the bottom shows the implementation of this method, with line 157 highlighted.

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");
163 }
```

# 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. On the left, a Java file named `Runner.class` is open, showing a method `testDupWithOneVal()` at line 157. The method contains several lines of code, including `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 `t` pointing to `-2*x^5`. A right-click context menu is open over the `expt` variable in the `Variables` view, showing options like `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`.

```
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 }
```

Name	Value
this	RatTermTest (id=33)
t	-2*x^5
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)

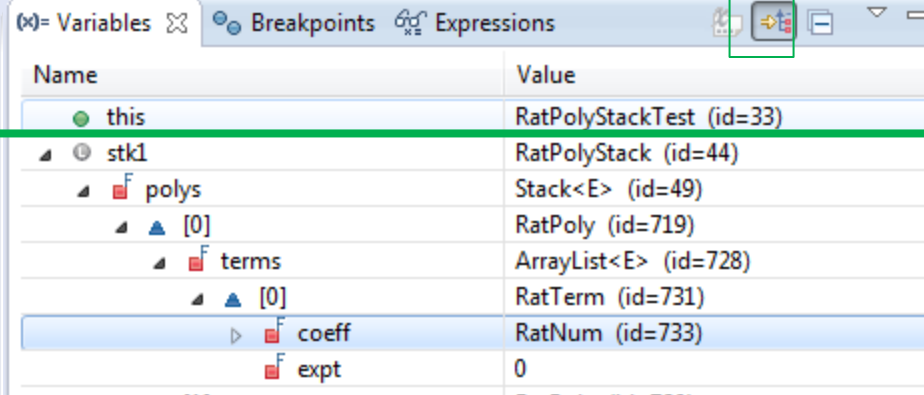
# Eclipse Debugging

## 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)

```
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");
```



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

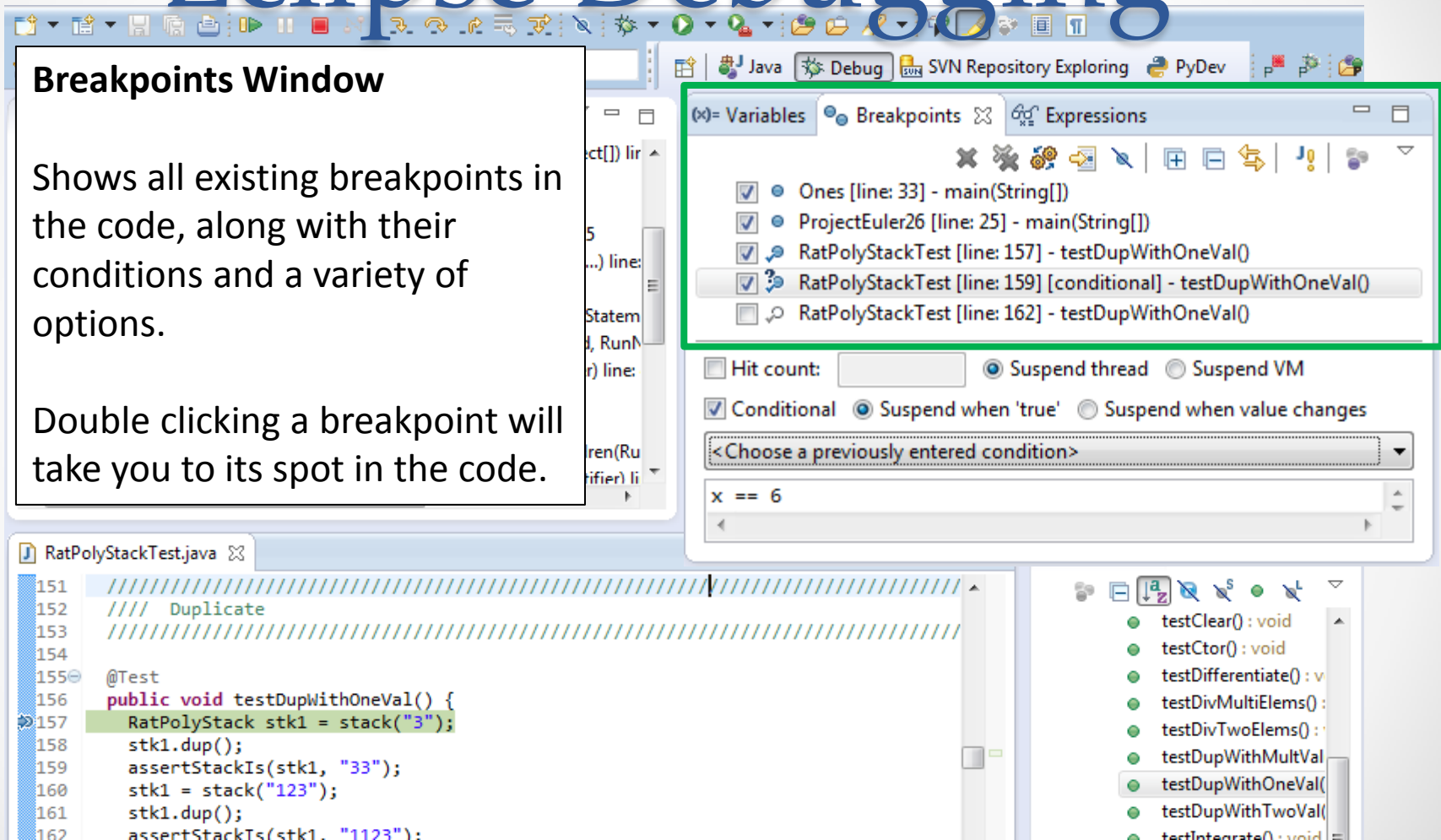
```
testClear() : void  
testCtor() : void  
testDifferentiate() : void  
testDivMultiElems() : void  
testDivTwoElems() : void  
testDupWithMultVal() : void  
testDupWithOneVal() : void  
testDupWithTwoVal() : void  
testIntegrate() : void
```

# 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.



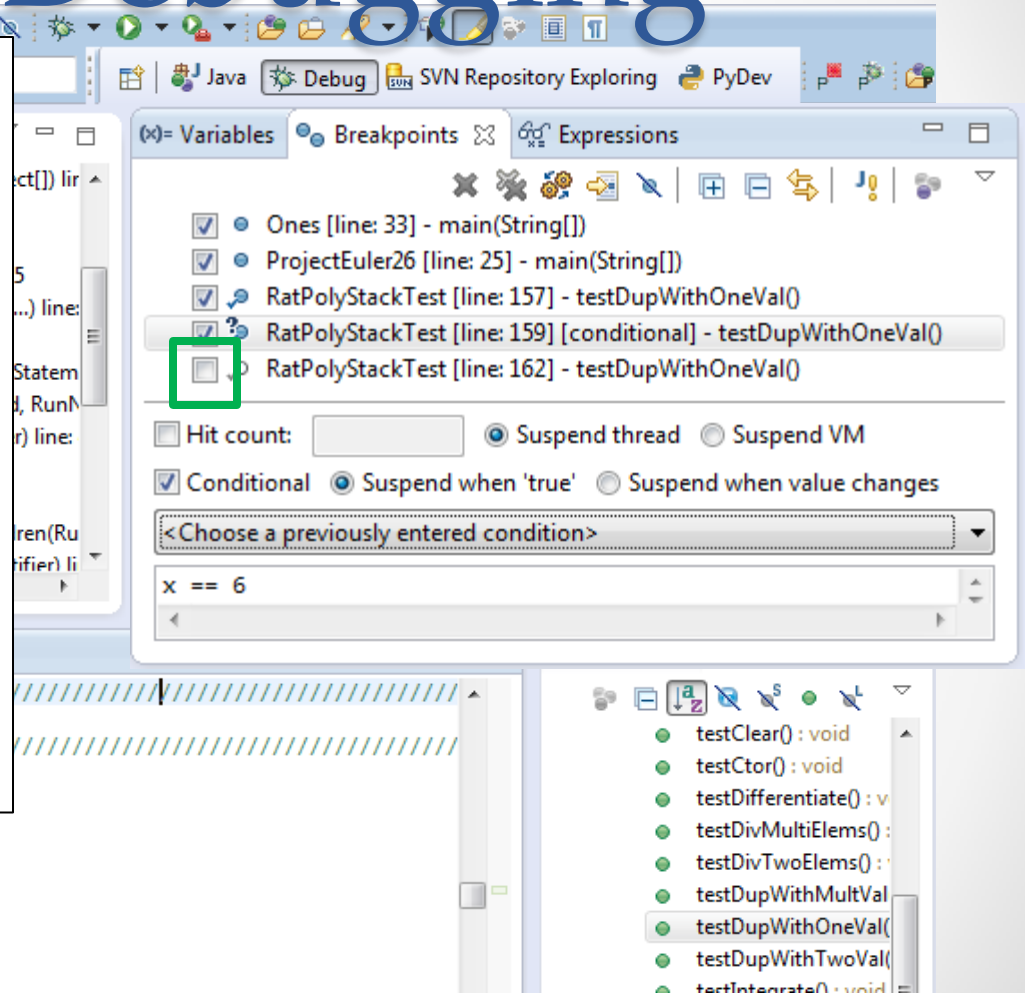
# Eclipse Debugging

## 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");  
}
```

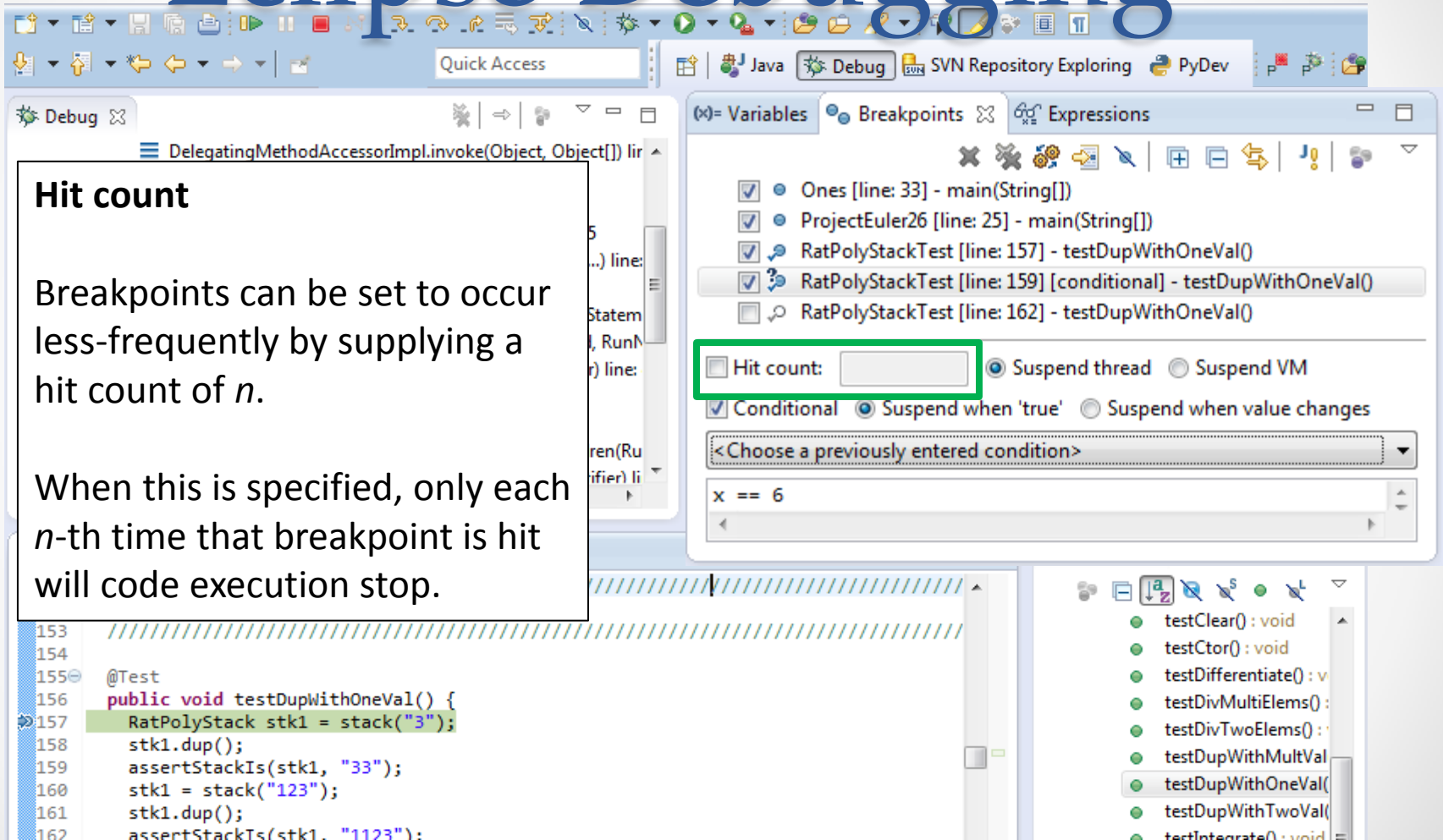


# Eclipse Debugging

## 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.



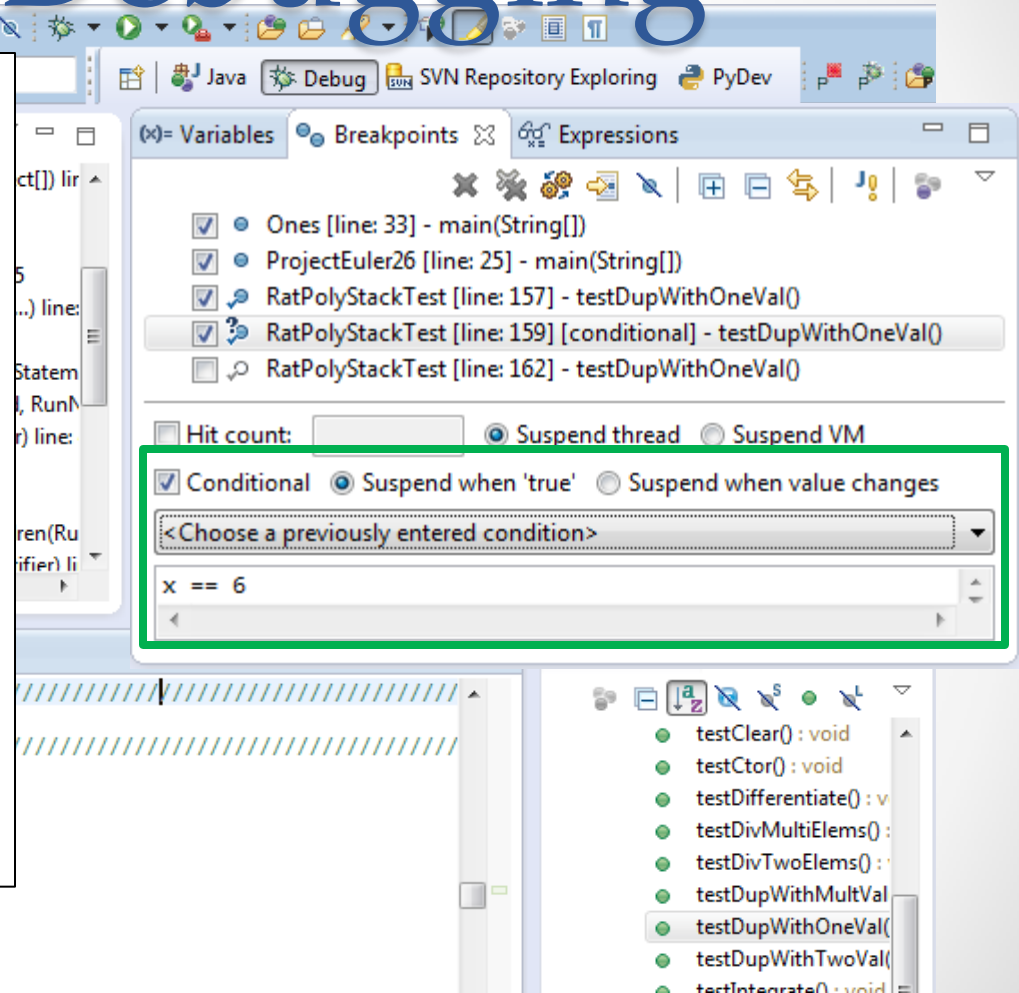
# Eclipse Debugging

## 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");
```

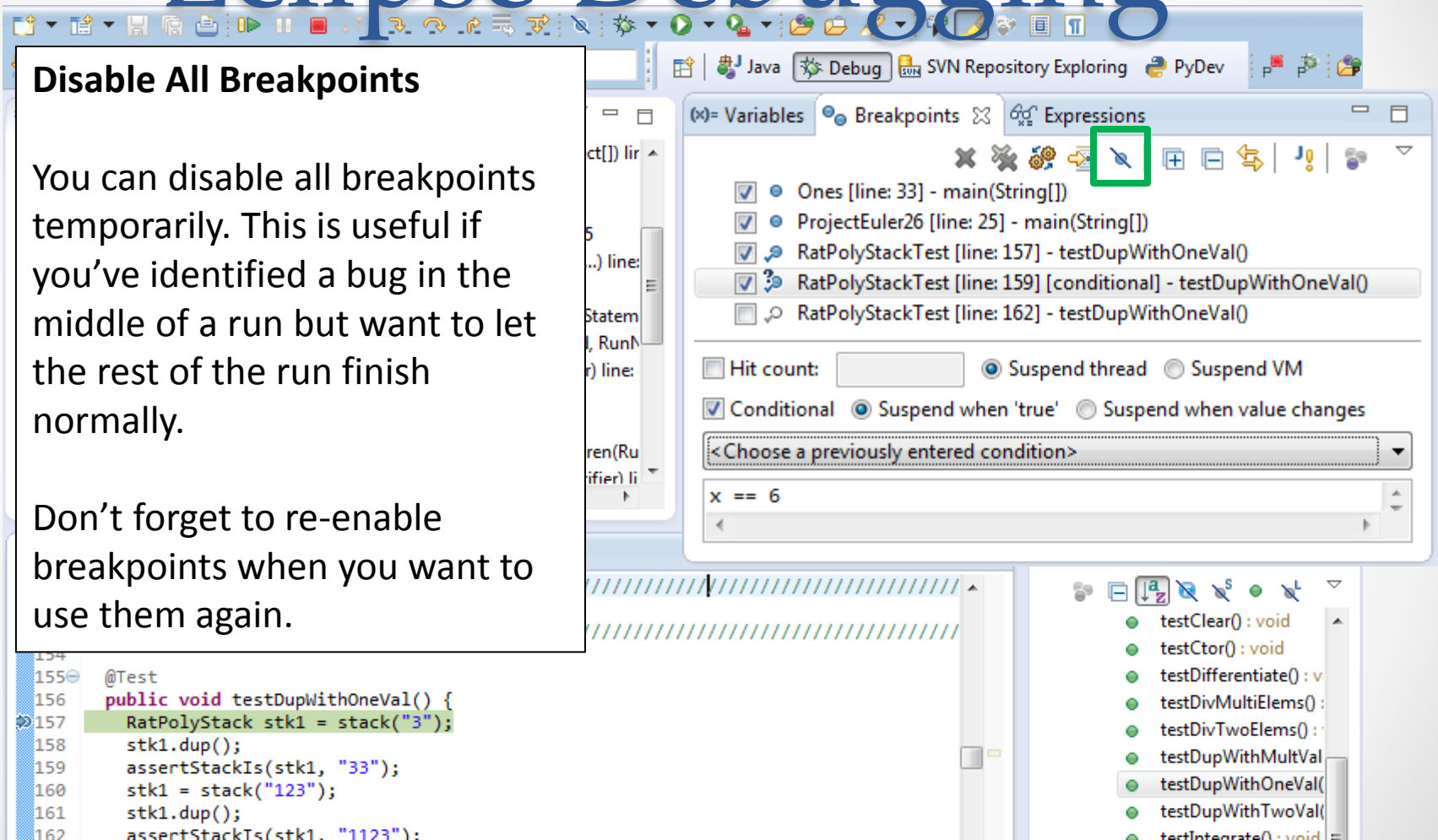


# Eclipse Debugging

## 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.



# 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
```

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

- ☒ 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

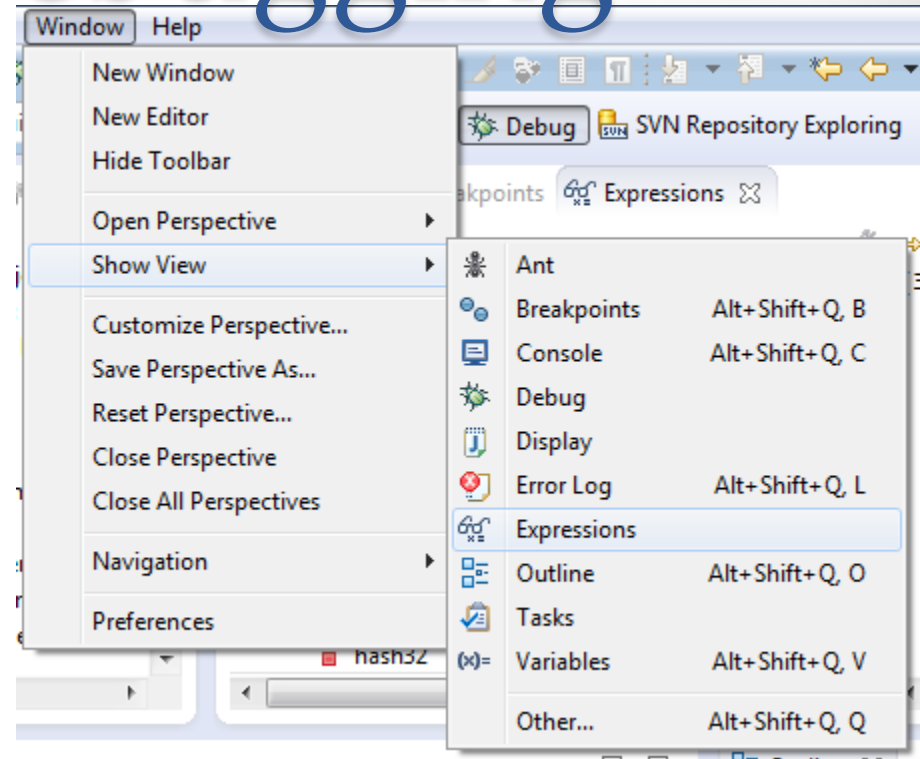
- testClear() : void
- testCtor() : void
- testDifferentiate() : void
- testDivMultiElems() :
- testDivTwoElems() :
- testDupWithMultVal
- testDupWithOneVal()
- testDupWithTwoVal()
- testIntegrate() : void

# 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.



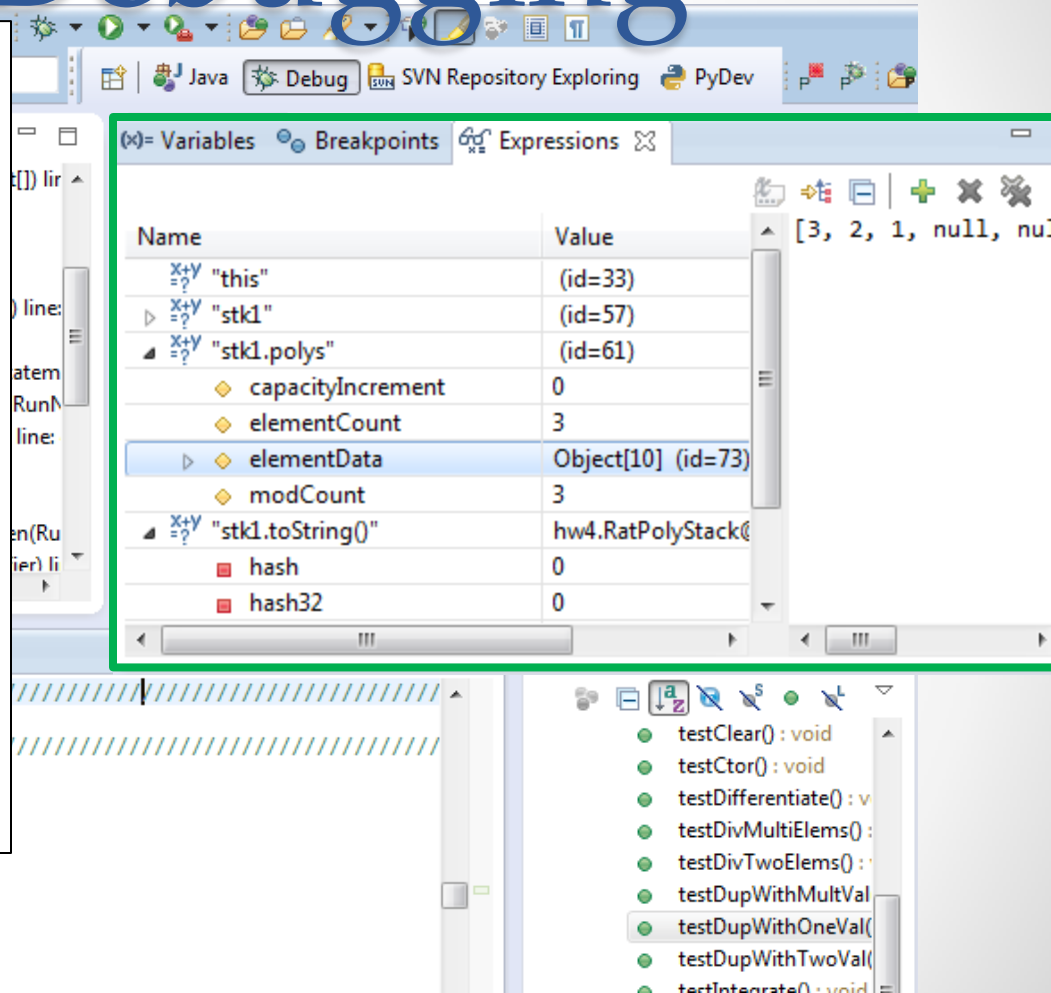
# Eclipse Debugging

## 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



# Eclipse Debugging

## Expressions Window

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

```
FrameworkMethod$1(ReflectiveCallable).run() line: 13  
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");
```

Variables Breakpoints Expressions

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

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

# 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

# DEMO #3