

Section 1:

Debugging + Code Reasoning

Alex Mariakakis

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

Outline

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

Reasoning About Code

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

Forward Reasoning

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
//
```

```
x = x + y
```

```
//
```

```
x = sqrt(x)
```

```
//
```

```
y = y - x
```

```
//
```

Forward Reasoning

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
//
```

```
x = sqrt(x)
```

```
//
```

```
y = y - x
```

```
//
```

Forward Reasoning

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
// {x >= 16, y = 16}
```

```
x = sqrt(x)
```

```
//
```

```
y = y - x
```

```
//
```

Forward Reasoning

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
// {x >= 16, y = 16}
```

```
x = sqrt(x)
```

```
// {x >= 4, y = 16}
```

```
y = y - x
```

```
//
```

Forward Reasoning

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


Forward Reasoning

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

Forward Reasoning

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

Forward Reasoning

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

Forward Reasoning

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

Forward Reasoning

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

Backward Reasoning

//

$a = x + b;$

//

$c = 2b - 4$

//

$x = a + c$

// { $x > 0$ }

Backward Reasoning

//

$a = x + b;$

//

$c = 2b - 4$

// $\{a + c > 0\}$

$x = a + c$

// $\{x > 0\}$

Backward Reasoning

//

$a = x + b;$

// $\{a + 2b - 4 > 0\}$

$c = 2b - 4$

// $\{a + c > 0\}$

$x = a + c$

// $\{x > 0\}$

Backward Reasoning

// { $x + 3b - 4 > 0$ }

$a = x + b;$

// { $a + 2b - 4 > 0$ }

$c = 2b - 4$

// { $a + c > 0$ }

$x = a + c$

// { $x > 0$ }

Implication

- Hoare triples are just an extension of logical implication
 - Hoare triple: $\{P\} S \{Q\}$
 - $P \rightarrow Q$ 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**"

Weakest Precondition

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

What is Eclipse?

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

Eclipse shortcuts

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

Eclipse Debugging

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

Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar contains various icons for file operations, running, and debugging. The menu bar includes options for Java, Debug, SVN Repository Exploring, and PyDev. The main workspace is divided into several panels:

- Debug Console:** Shows the current 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:** A table showing the current state of variables:

Name	Value
this	RatPolyStackTest (id=33)
- Source Editor:** Displays the source code for `RatPolyStackTest.java`. The current line is 157, which 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:** Shows a list of methods in the class, with `testDupWithOneVal()` selected:
 - testClear(): void
 - testCtor(): void
 - testDifferentiate(): void
 - testDivMultiElems(): void
 - testDivTwoElems(): void
 - testDupWithMultVal(): void
 - testDupWithOneVal(): void**
 - testDupWithTwoVal(): void
 - 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 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:** Shows a table with the following content:

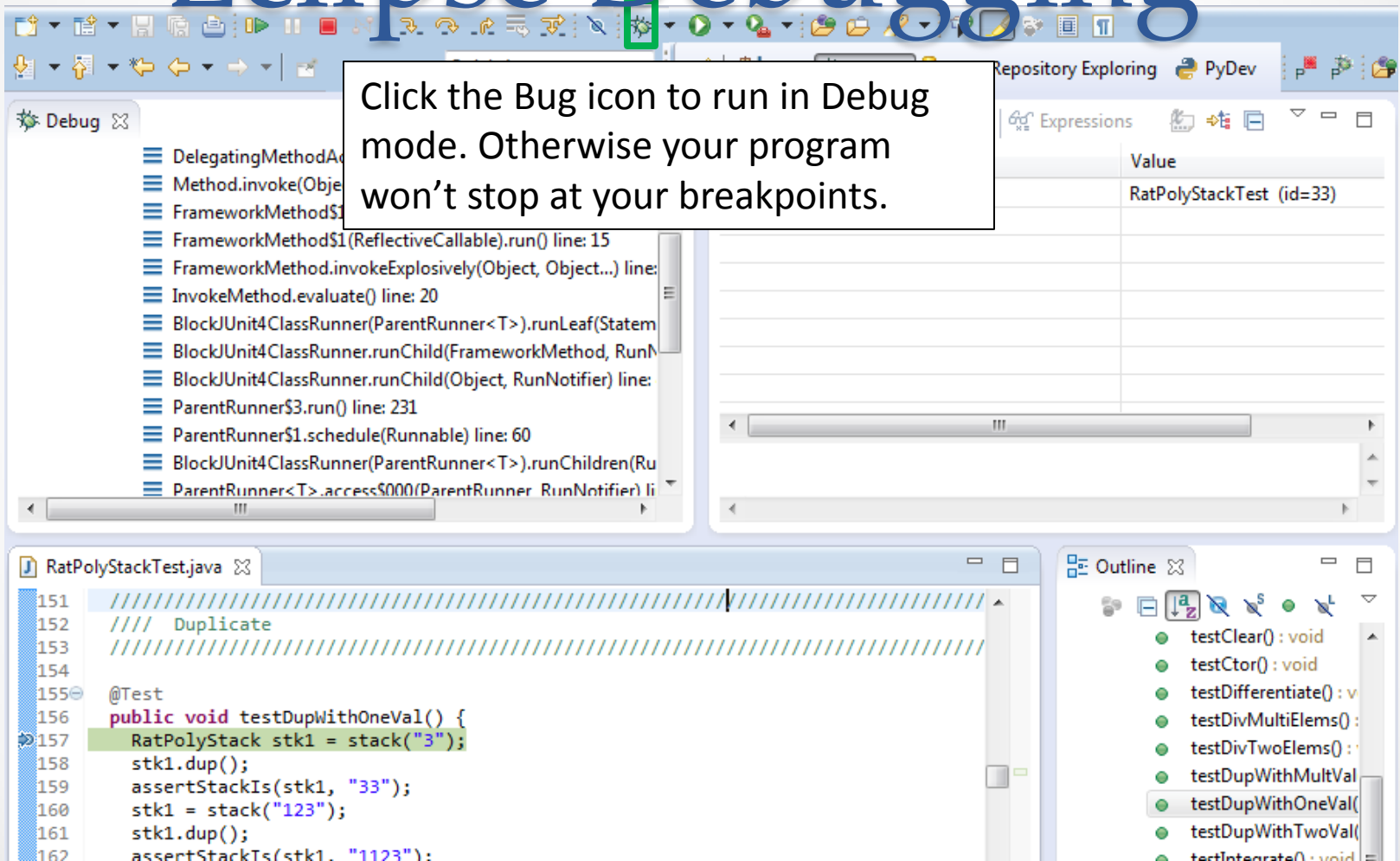
Name	Value
this	RatPolyStackTest (id=33)
- Code Editor:** Displays the source code for `RatPolyStackTest.java`. A breakpoint is set on line 57, indicated by a blue arrow icon in the left margin. The code includes comments and assertions.
- Outline View:** Shows the class structure on the right side of the editor.

A text box is overlaid on the code editor, providing instructions on how to set a breakpoint:

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.



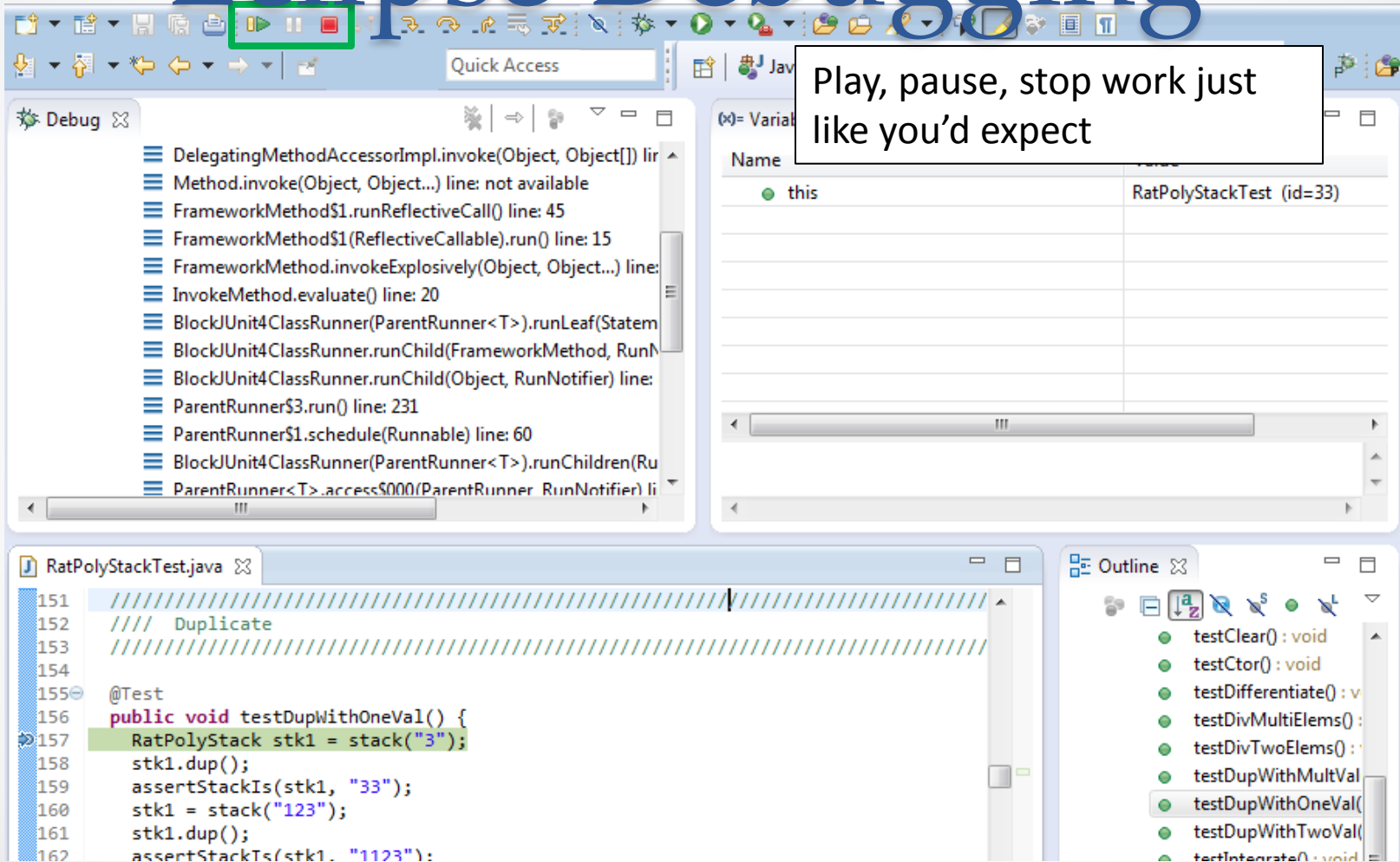
The screenshot displays the Eclipse IDE interface with several key components:

- Top Toolbar:** The Bug icon (a green bug) is highlighted with a green box, indicating the action to run the program in debug mode.
- Debug Console:** The left pane shows a stack trace of the current execution, including methods like `DelegatingMethodAdapter.invoke`, `FrameworkMethod.invokeExplosively`, and `BlockJUnit4ClassRunner.runLeaf`.
- Code Editor:** The `RatPolyStackTest.java` file is open, showing a test method `testDupWithOneVal()`. A breakpoint is set at line 157, which contains the statement `RatPolyStack stk1 = stack("3");`.
- Outline View:** The right pane shows the class outline with various test methods listed, such as `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.
- Expressions View:** The top right pane shows the current value of the selected expression, which is `RatPolyStackTest (id=33)`.

Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. At the top, the toolbar contains several icons for controlling the program's execution, with a green box highlighting the Run, Break, and Step Over buttons. Below the toolbar, the Debug console shows a call 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`, `BlockJUnit4ClassRunner.runChild`, `ParentRunner$3.run`, `ParentRunner$1.schedule`, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren`, and `ParentRunner<T>.access$000`. To the right of the call stack, a variable declaration window is visible with the text "Controlling your program while debugging is done with these buttons". The main editor window shows the source code for `RatPolyStackTest.java`, with line 157 highlighted: `@Test public void testDupWithOneVal() { RatPolyStack stk1 = stack("3");`. The Outline view on the right lists several test methods, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate() : void`.

Eclipse Debugging



The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar has a green box around the Play, Pause, and Stop buttons. A tooltip over the Play button reads: "Play, pause, stop work just like you'd expect".

The Debug console (top left) shows a stack trace for `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate`, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `BlockJUnit4ClassRunner.runChild(FrameworkMethod, Runnable)`, `BlockJUnit4ClassRunner.runChild(Object, Runnable)`, `ParentRunner$3.run`, `ParentRunner$1.schedule(Runnable)`, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Runnable)`, and `ParentRunner<T>.access$000(ParentRunner, Runnable)`.

The Variable Inspector (top right) shows a table with one variable: `this` of type `RatPolyStackTest (id=33)`.

The Source Editor (bottom left) shows the code for `RatPolyStackTest.java`. Line 157 is highlighted with a green background, indicating the current execution point. 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 Outline view (bottom right) shows a list of methods: `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() : void`, `testDivTwoElems() : void`, `testDupWithMultVal() : void`, `testDupWithOneVal() : void`, `testDupWithTwoVal() : void`, and `testIntegrate() : void`.

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
```

```
Variable View:  
Name  
Value
```

```
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");
```

```
Outline View:  
testDifferentiate(): void  
testDivMultiElems(): void  
testDivTwoElems(): void  
testDupWithMultVal(): void  
testDupWithOneVal(): void  
testDupWithTwoVal(): void  
testIntegrate(): void
```

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");
```

- testDivWithTwoElems():
- testDivTwoElems():
- testDupWithMultVal():
- testDupWithOneVal():
- testDupWithTwoVal():
- testIntegrate(): void

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.

```
Debug
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

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");

testDupWithMultVal
testDupWithOneVal(
testDupWithTwoVal(
testIntegrate0: void
```

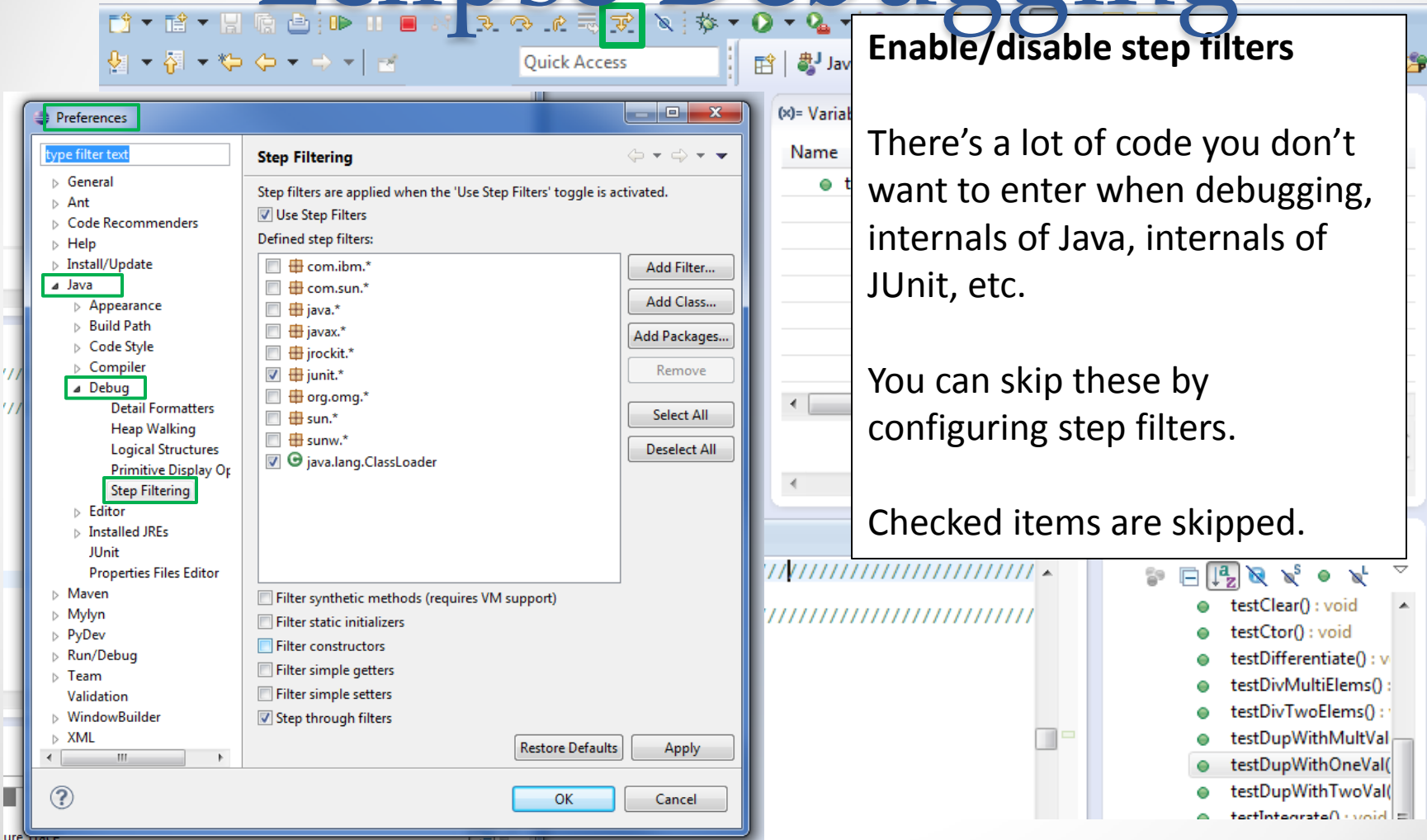

Eclipse Debugging

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 and debugging. Below the toolbar is the 'Quick Access' bar. The main workspace is divided into several panes:

- Debug Console:** A window titled 'Debug' showing a stack trace. The stack trace lists several methods, with the current method being `ParentRunner<T>.access$000(ParentRunner, RunNotifier) li`. The stack trace is highlighted with a green border.
- Code Editor:** A window titled 'RatPolyStackTest.java' showing the source code. The current line of code is `stk1 = stack("3");`, which is highlighted in green. The code includes a `@Test` annotation and a `testDupWithOneVal()` method.
- Stack Trace Panel:** A panel on the right side of the IDE showing a list of methods in the stack trace. The methods are listed with their names and line numbers. The current method is highlighted in green.

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");
```

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

- Variables View (top right, highlighted with a green border):** A table showing the current state of variables in memory. The 'Value' column for 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`. The current line of execution is highlighted in blue. The code includes a test method `testDupWithOneVal()` that creates a `RatPolyStack` object and performs operations on it.

```
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 (bottom right):** Shows a list of methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

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 Run, Break, and other debugging actions. The main window is divided into several panes:

- Variables Window (top right):** A table showing the current state of variables. The 'Value' tab is active. The variable 't' is expanded to show its fields: 'coeff' and 'expt'. The 'expt' field has a yellow background, indicating it has changed since the last breakpoint. The value of 'expt' is 5. The expression $-2*x^5$ is visible at the bottom of this window.
- 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");`. Other lines include `stk1.dup();`, `assertStackIs(stk1, "33");`, `stk1 = stack("123");`, `stk1.dup();`, and `assertStackIs(stk1, "1123");`.
- Outline (bottom right):** Lists the methods in the class, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `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 main editor displays the source code for `Runner.class`, with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The Variables view on the right shows the current stack frame, with the variable `expt` selected. A right-click context menu is open over `expt`, 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 `All Instances...` option is currently selected.

```
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)

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)
 - RatPolyStackTest [line: 162] - testDupWithOneVal()Below the list are options for 'Hit count', 'Suspend thread', 'Suspend VM', 'Conditional', and 'Suspend when 'true' / 'Suspend when value changes'. A text field contains the condition `x == 6`.
- Code Editor:** Shows the source code for `RatPolyStackTest.java`. Line 159 is highlighted, 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:** Located in the bottom right, it 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");  
}
```

The screenshot shows the Eclipse IDE interface. The Breakpoints view is open, displaying a list of breakpoints for the testDupWithOneVal() method. The breakpoint at line 162 is disabled, indicated by a green box around the unchecked checkbox. The Breakpoints view shows several breakpoints for the testDupWithOneVal() method. The 'Hit count' is set to 1, and the 'Suspend thread' option is selected. The conditional expression is 'x == 6'. The code editor shows the testDupWithOneVal() method with line 157 highlighted.

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.

DelegatingMethodAccessorImpl.invoke(Object, Object[]) lir

Quick Access

Java Debug SVN Repository Exploring PyDev

Debug

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

```
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() : void
- testDivMultiElems() :
- testDivTwoElems() :
- testDupWithMultVal
- testDupWithOneVal()
- testDupWithTwoVal()
- testIntegrate() : void

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 during a debug session. The 'Breakpoints' view is open, displaying a list of breakpoints for the file 'RatPolyStackTest.java'. The breakpoint at line 159 is selected and highlighted with a green box. The configuration for this breakpoint is as follows:

- Conditional
- Suspend when 'true'
- Suspend when value changes
- Condition: `x == 6`

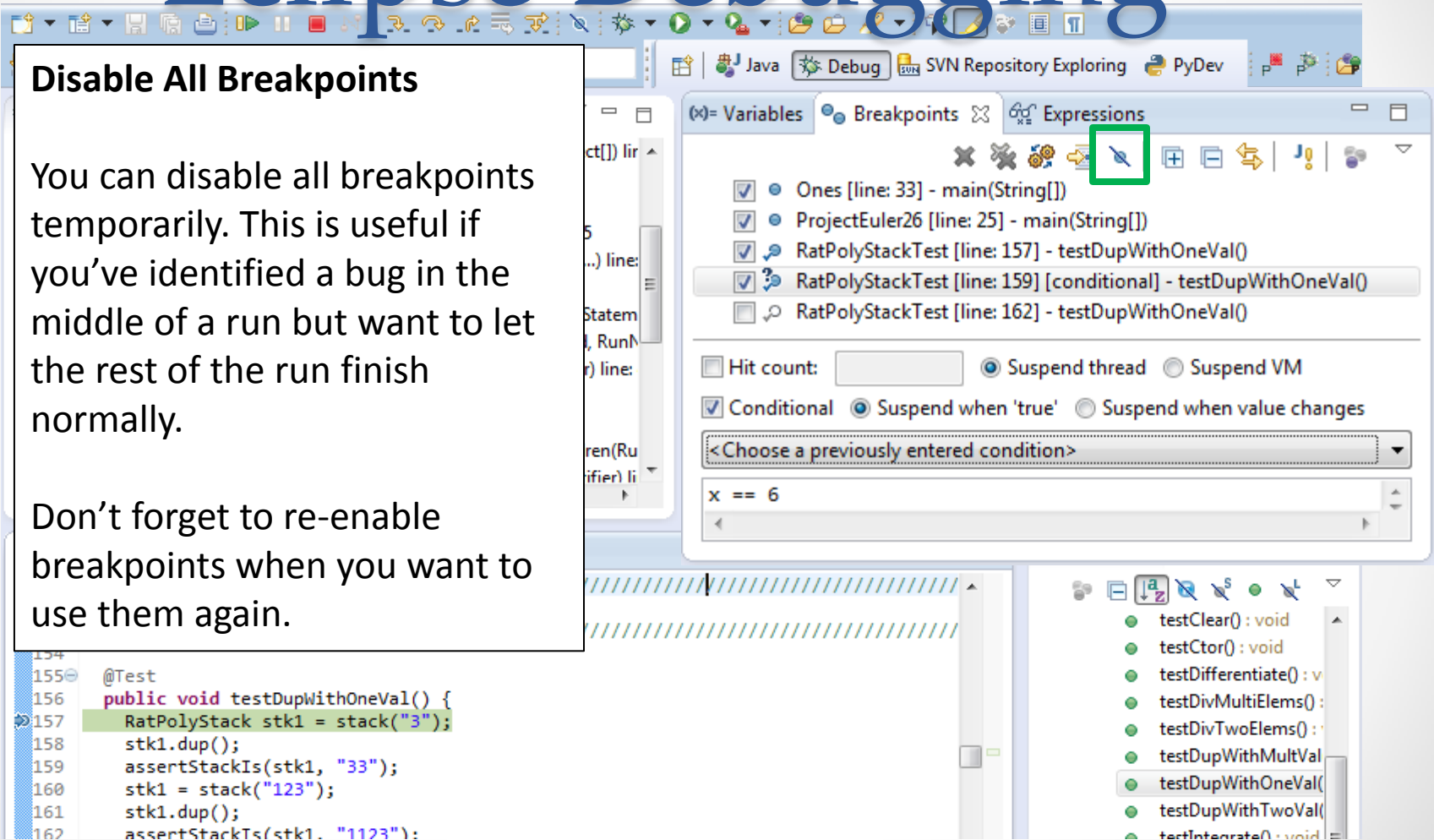
The 'Hit count' is set to 1. The 'Suspend thread' option is selected. The 'Expressions' view shows the current state of the program, including the stack and the current method being executed.

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'. 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 is selected. The 'Hit count' is set to 1. The 'Suspend thread' option is selected. The 'Conditional' checkbox is checked, and the 'Suspend when 'true'' option is selected. The condition is 'x == 6'. A red box highlights the 'Disable All Breakpoints' icon (a crossed-out blue circle) in the Breakpoints view toolbar.

```
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

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
```

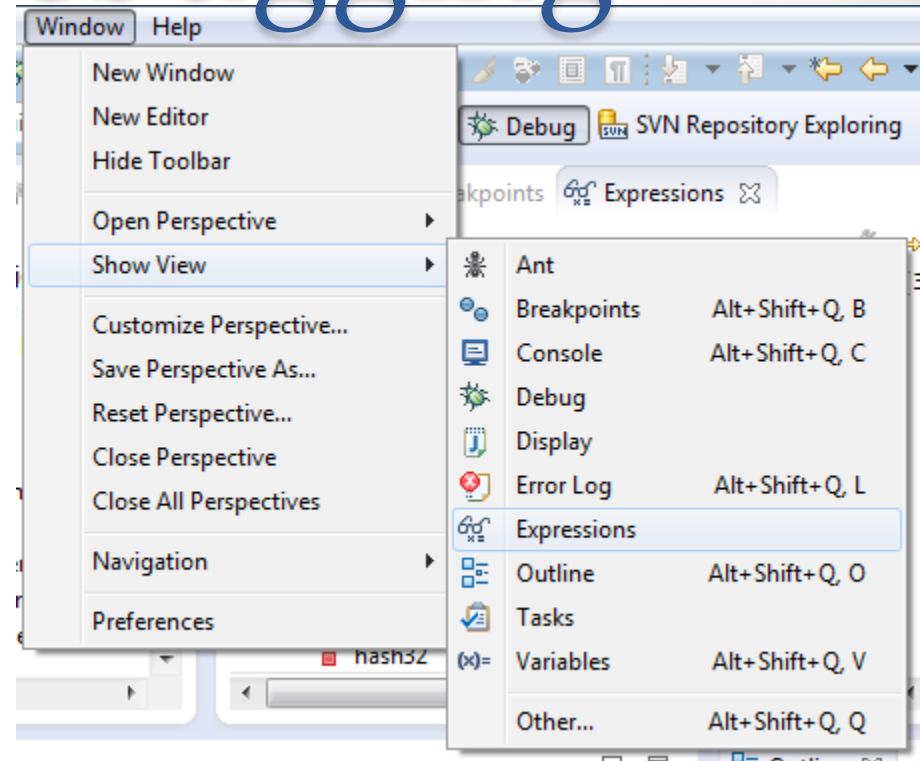
The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for Run, Debug, and Breakpoints. The Breakpoints view is open, showing a list of breakpoints for the current project. A breakpoint is set on line 159 of RatPolyStackTest.java, with the condition `x == 6`. The Breakpoints view also shows options for Hit count, Suspend thread, Suspend VM, and Conditional (Suspend when 'true'). 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 the Outline view, listing the methods of the class: `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `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

The screenshot shows the Eclipse IDE interface during a debug session. The Expressions Window is open, displaying a table of variables and their values. The table has two columns: 'Name' and 'Value'. The variables shown are:

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 Expressions Window also shows a list of methods in the bottom right corner, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate() : void`.

In the background, the source code editor shows 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");
```

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 open, displaying a table of variables and their values. The table has two columns: 'Name' and 'Value'. The variables shown are:

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 the source code of `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");
163 }
```


Demo!!!

Eclipse Debugging

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