

Section 5: Parsing and Debugging

Slides by Alex Mariakakis

with material Kellen Donohue



Agenda

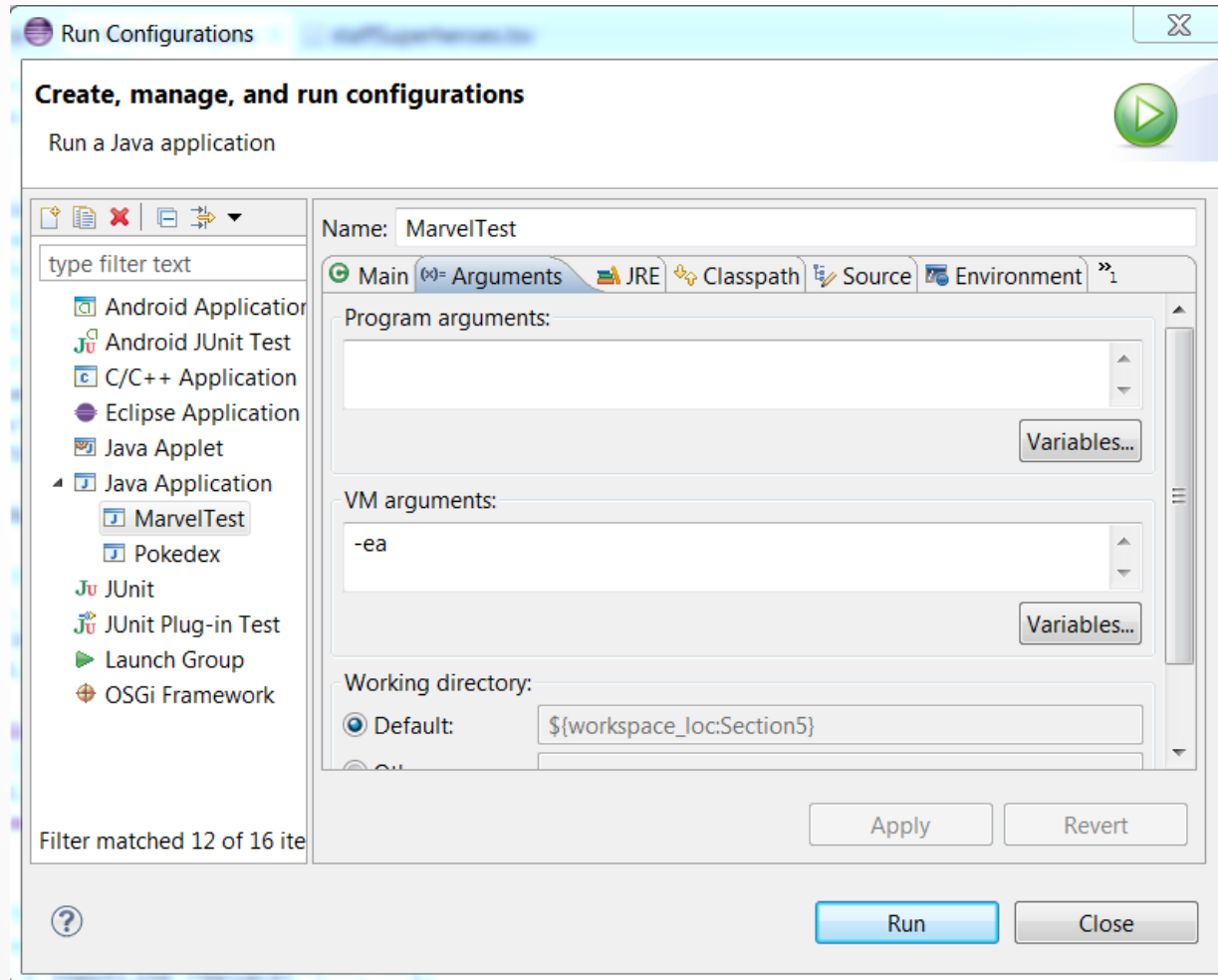
- HW 5 questions?
- HW 5 test driver
- Quick note on asserts
- Parsing the Marvel data
- Debugging



Asserts

- You must manually turn on assert statements for them to be run in your code.
- The command line flag is "-ea"
- To set command line flags in Eclipse:
 - Select the .java file you are running -> Run As -> Run Configurations
 - Arguments tab
 - Enter "-ea" under 'VM arguments'

Asserts



Demo 1:

Parsing the Marvel data



Eclipse Debugging

- `System.out.println()` works for debugging, but there are better methods
- Eclipse's debugger is powerful...if you know how to use it



Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar includes standard development icons 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[]) 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)
- Code Editor:** Displays the source code for `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:** Lists the 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 debugging 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 a list of stack frames from the current thread, including `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate`, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `BlockJUnit4ClassRunner.runChild`, `ParentRunner$3.run`, `ParentRunner$1.schedule`, and `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren`.
- Variables View:** Shows a table with two columns: 'Name' and 'Value'. The only entry is 'this' with the value 'RatPolyStackTest (id=33)'. Below the table is a scrollable area for expressions.
- Code Editor:** Displays the source code for `RatPolyStackTest.java`. Line 57 is highlighted in green, and a blue arrow icon indicates a breakpoint has been set at that line. 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 shows the Eclipse IDE interface. The top toolbar contains various icons, with the Bug icon (a green bug) highlighted by a red box. Below the toolbar, the Debug console is visible, showing a list of methods being executed. The main editor window displays the source code for `RatPolyStackTest.java`, with a breakpoint set at line 157: `RatPolyStack stk1 = stack("3");`. The Outline view on the right shows a list of methods, including `testDupWithOneVal()`, which is highlighted. The Expressions view at the top right shows the value of `RatPolyStackTest (id=33)`.

Eclipse Debugging

The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar contains several icons for program control, with a green box highlighting the Run (green play button), Stop (red square), and Step Over (orange arrow) buttons. A text box points to these buttons with the text: "Controlling your program while debugging is done with these buttons".

The Debug console on the left shows a stack of frames from the current thread, including:

- 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...

The Variable view on the right shows a table with columns for Name and Value, currently empty.

The source code editor at the bottom shows the file `RatPolyStackTest.java`. The following code is visible:

```
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 on the bottom right shows a list of methods in the class, with `testDupWithOneVal()` selected:

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

Eclipse Debugging

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

Debug

- DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 17
- Method.invoke(Object, Object...) line: not available
- FrameworkMethod\$1.runReflectiveCall() line: 45
- FrameworkMethod\$1(ReflectiveCallable).run() line: 15
- FrameworkMethod.invokeExplosively(Object, Object...) line: 15
- InvokeMethod.evaluate() line: 20
- BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf(Statement) line: 100
- BlockJUnit4ClassRunner.runChild(FrameworkMethod, Runnable) line: 115
- BlockJUnit4ClassRunner.runChild(Object, Runnable) line: 130
- ParentRunner\$3.run() line: 231
- ParentRunner\$1.schedule(Runnable) line: 60
- BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Runnable) line: 120
- ParentRunner<T>.access\$000(ParentRunner, Runnable) line: 100

Variables

Name	Value
this	RatPolyStackTest (id=33)

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 }
```

Outline

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

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 blue square with a white arrow pointing down). Below the toolbar, the 'Debug' console is open, displaying a stack trace of the current execution. The stack trace includes methods like `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `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)`.

In the center, the 'Variables' view is visible, showing a table with a 'Name' column and a single entry 't'.

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

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

A green highlight is placed over line 157, and a blue arrow points to the line number '157' in the left margin, indicating a breakpoint. To the right of the code editor, a list of test methods is shown, with `testDupWithOneVal()` selected and highlighted in green.

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.

```
Debug Console:  
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");
```

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

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:

- Debug Console:** A green-bordered window showing a stack trace. The top of the stack trace is highlighted in blue. The stack trace includes:
 - 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
- Code Editor:** Shows the source code for `RatPolyStackTest.java`. The current line is 157, which is highlighted in green:

```
157 RatPolyStack stk1 = stack("3");
```
- Stack Trace Panel:** A panel on the right side of the IDE showing a list of method names with a scroll bar. The current method is `testDupWithOneVal()`.

Stack Trace

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

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

Eclipse Debugging

Variables Window

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

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

Name	Value
• this	RatPolyStackTest (id=33)

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

Eclipse Debugging

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

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

The screenshot shows the Eclipse IDE in a debug state. The top toolbar includes icons for Run, Debug, and Breakpoints. The Variables window is open, displaying a table of variables:

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

The 'expt' variable is highlighted in yellow. The main editor shows the source code for `RatPolyStackTest.java`, with line 157 highlighted:

```
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 window on the right shows a list of methods, with `testDupWithOneVal()` selected.

Eclipse Debugging

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

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

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

- Variables Window:** A table with columns 'Name' and 'Value'. The 'expt' variable is highlighted in yellow. The table contains the following data:

Name	Value
this	RatTermTest (
t	RatTerm (id=4
coeff	RatNum (id=4
expt	5
- Code Editor:** Shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted, containing the code: `RatPolyStack stk1 = stack("3");`
- 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)

```
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");
```

The screenshot shows the Eclipse IDE interface during a debug session. The 'Variables' view is open, displaying a tree structure of variables. The variable 't' is expanded, showing its sub-variables 'coeff' and 'expt'. The 'expt' variable is selected, and a right-click context menu is displayed over it. The menu includes options 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 highlighted, indicating it is the current selection.

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

Eclipse Debugging

Show Logical Structure

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

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

- Left Pane:** Shows the stack trace for the current thread. The current frame is `ParentRunner$1.schedule(Runnable) line: 60`.
- Bottom Left Pane:** Shows the source code for `RatPolyStackTest.java`. The current line is 157: `RatPolyStack stk1 = stack("3");`.
- Bottom Right Pane:** Shows the list of methods in the current class, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate() : void`.
- Top Right Pane:** Shows the variable view. The variables are listed in a table:

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

A green box highlights the `coeff` field of the `RatTerm` object, which is a `RatNum` object with the value 0. This demonstrates the 'Show Logical Structure' feature, which expands list items into a tree-like structure.

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 Breakpoints window is open, showing 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 Breakpoints window also shows options for hit count, suspend thread, suspend VM, and conditional execution. The conditional execution is set to "Suspend when 'true'", and the condition is `x == 6`.

The code editor shows the following 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");
```

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 during a debug session. The Breakpoints view is open, displaying a list of breakpoints for the current project. The breakpoint for line 162 is disabled, as indicated by the unchecked checkbox next to it. The breakpoint is for the method `testDupWithOneVal()` in the class `RatPolyStackTest`. The breakpoint is conditional, with the condition `x == 6`. The breakpoint is set to suspend the thread when the condition is true. The breakpoint is also set to suspend when the value changes. The breakpoint is also set to suspend when the value changes. The breakpoint is also set to suspend when the value changes.

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 displays the Eclipse IDE's breakpoint configuration dialog. The dialog is titled "Hit count" and shows a list of breakpoints for the method `RatPolyStackTest.testDupWithOneVal()`. The "Hit count" field is highlighted with a green box. The "Suspend thread" radio button is selected. The "Conditional" checkbox is checked, and the condition `x == 6` is entered in the text field below. The background shows the Eclipse IDE interface with a code editor and a variable viewer.

```
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 }
```

Variables:

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

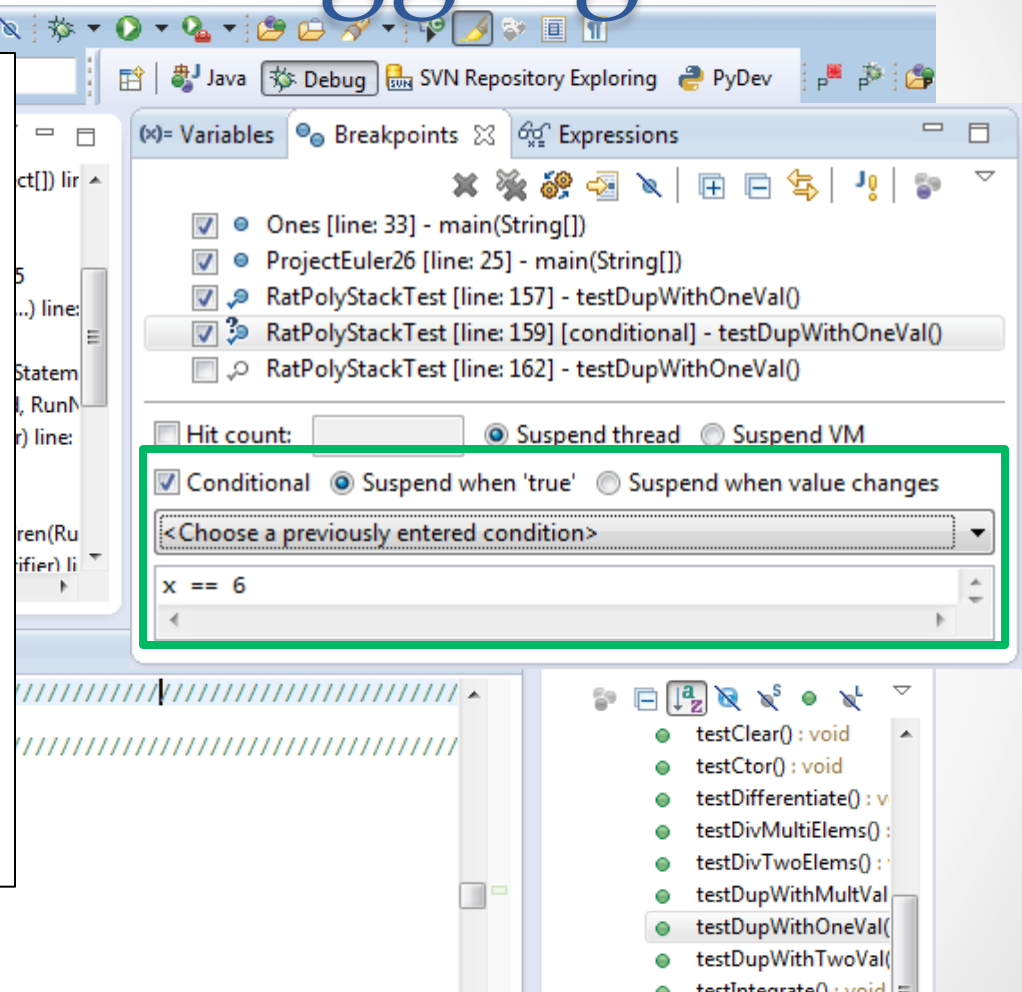
Eclipse Debugging

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");
```

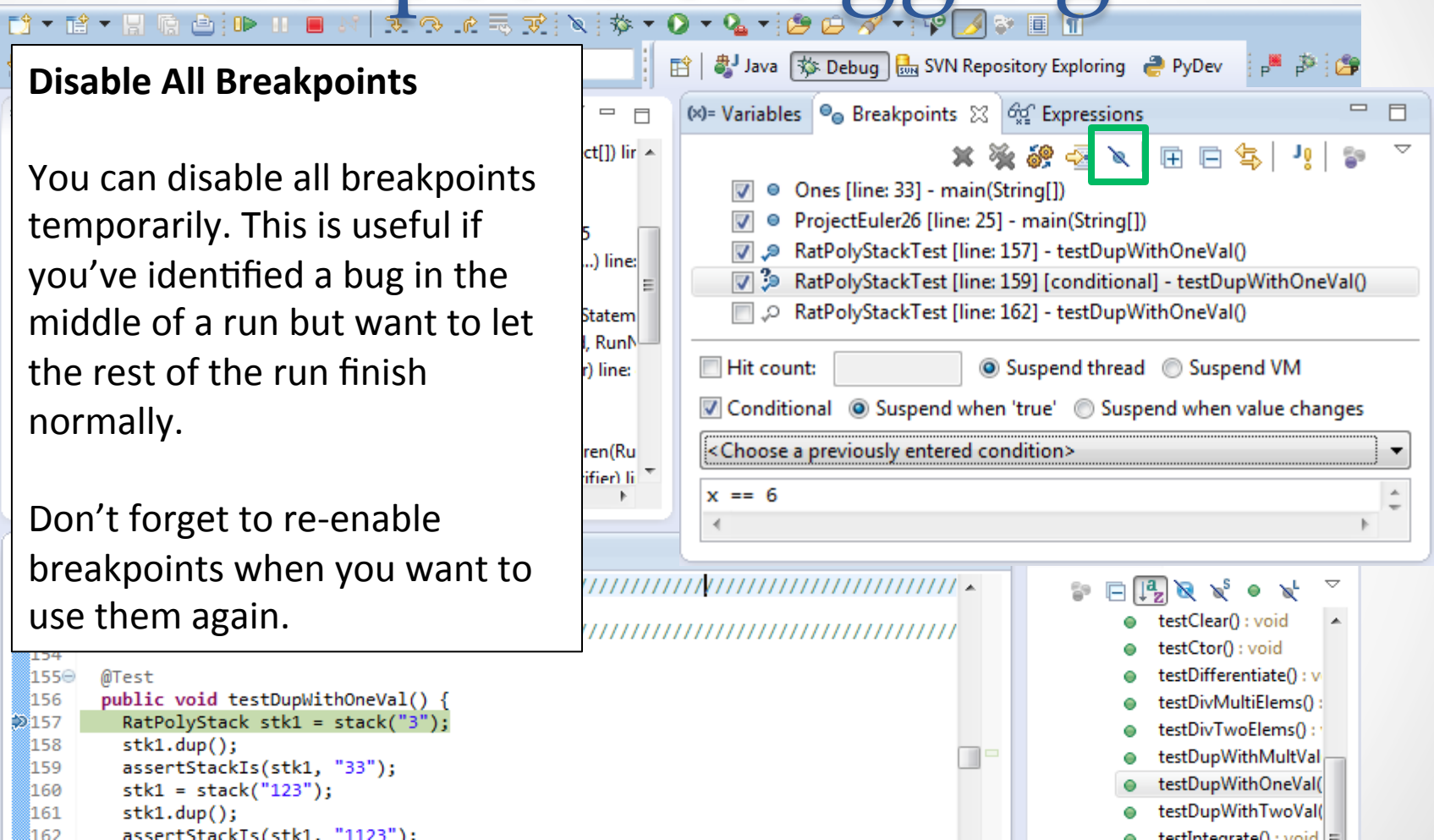


Eclipse Debugging

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 current project. A green box highlights the 'Disable All Breakpoints' icon (a crossed-out pencil) 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' options are 'Suspend thread' and 'Suspend when value changes'. The 'Conditional' checkbox is checked, and the condition is set to 'x == 6'. The 'Expressions' view is also visible, showing a list of expressions for the current method.

```
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.

The screenshot shows the Eclipse IDE interface during a debug session. The main editor displays the source code of `RatPolyStackTest.java`, with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The Breakpoints view on the right shows a list of breakpoints, with a conditional breakpoint on line 159 selected. The condition for this breakpoint is `x == 6`. The Breakpoints view also shows options for hit count, suspend thread, suspend VM, and conditional settings. The Expressions view is also visible, showing a list of expressions.

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

```
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");
```

Breakpoints view:

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

Conditional settings:

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

Expressions view:

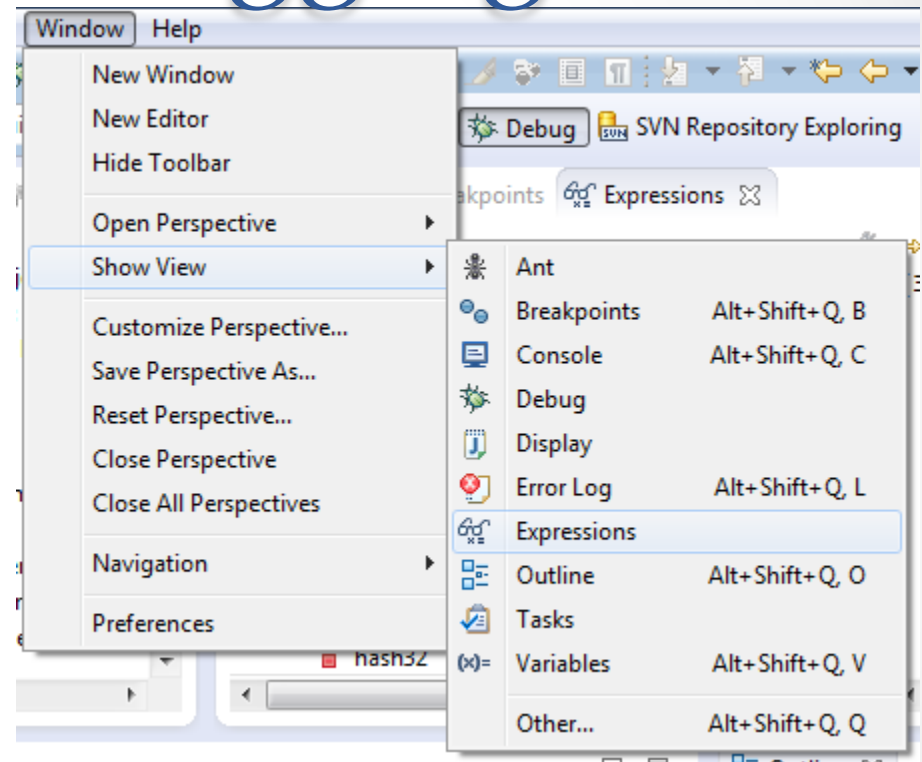
- testClear(): void
- testCtor(): void
- testDifferentiate(): void
- testDivMultiElems(): void
- testDivTwoElems(): void
- testDupWithMultVal(): void
- testDupWithOneVal(): void
- testDupWithTwoVal(): void
- testIntegrate(): void

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`.

```
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 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 source code for `RatPolyStackTest.java` with the following content:

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

Demo 2: Debugging

