

SECTION 2:

HW3 Setup

cse331-staff@cs.washington.edu

slides borrowed and adapted from Alex Mariakis and CSE 390a

DEVELOPER TOOLS

- Remote access
- Eclipse and Java versions
- Version Control

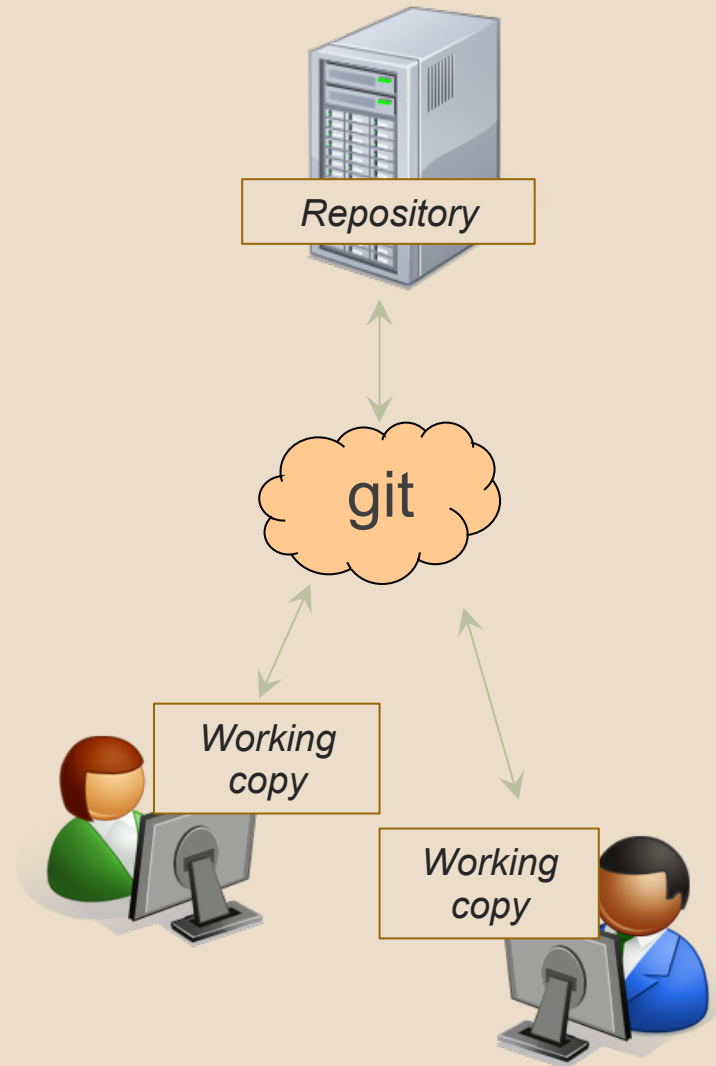
VERSION CONTROL

WHAT IS VERSION CONTROL?

- Also known as source control/revision control
- System for tracking changes to code
 - Software for developing software
- Essential for managing projects
 - See a history of changes
 - Revert back to an older version
 - Merge changes from multiple sources
- We'll be talking about git/GitLab, but there are alternatives
 - Subversion, Mercurial, CVS
 - Email, Dropbox, USB sticks (don't even think of doing this)

VERSION CONTROL ORGANIZATION

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



REPOSITORY

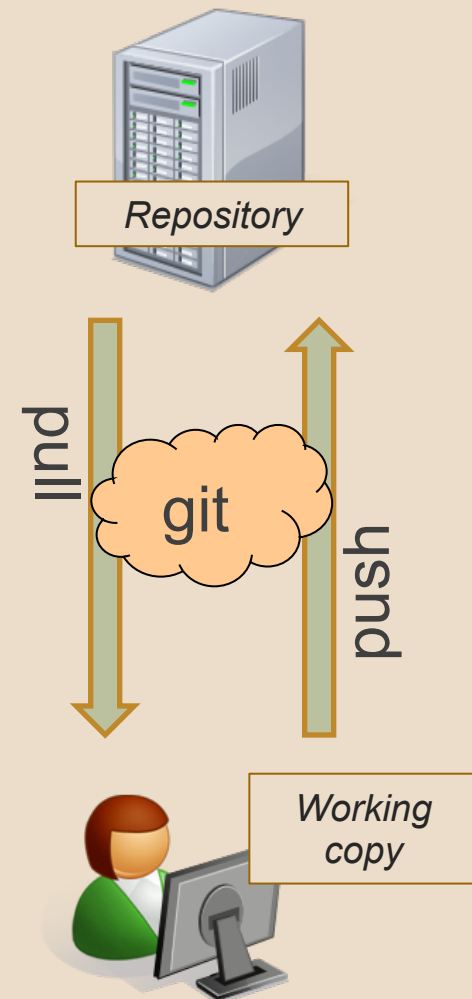
- Can create the repository anywhere
 - Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection
- But, usually you want the repository to be robust:
 - On a computer that's up and running 24/7
 - Everyone always has access to the project
 - On a computer that has a redundant file system
 - No more worries about that hard disk crash wiping away your project!
- We'll use CSE GitLab – very similar to GitHub but tied to CSE accounts and authentication

VERSION CONTROL

COMMON ACTIONS

Most common commands:

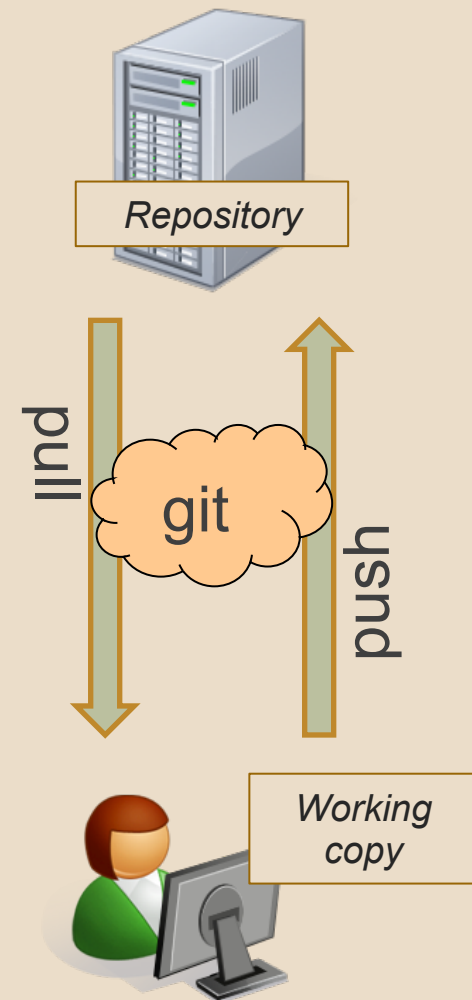
- **add / commit / push**
 - integrate changes *from* your working copy *into* the repository
- **pull**
 - integrate changes *into* your working copy *from* the repository



VERSION CONTROL UPDATING FILES

In a bit more detail:

- You make some local changes, test them, etc., then...
- `git add` – tell git which changed files you want to save in repo
- `git commit` – save all files you've "add"ed in the local repo copy as an identifiable update
- `git push` – synchronize with the GitLab repo by pushing local committed changes

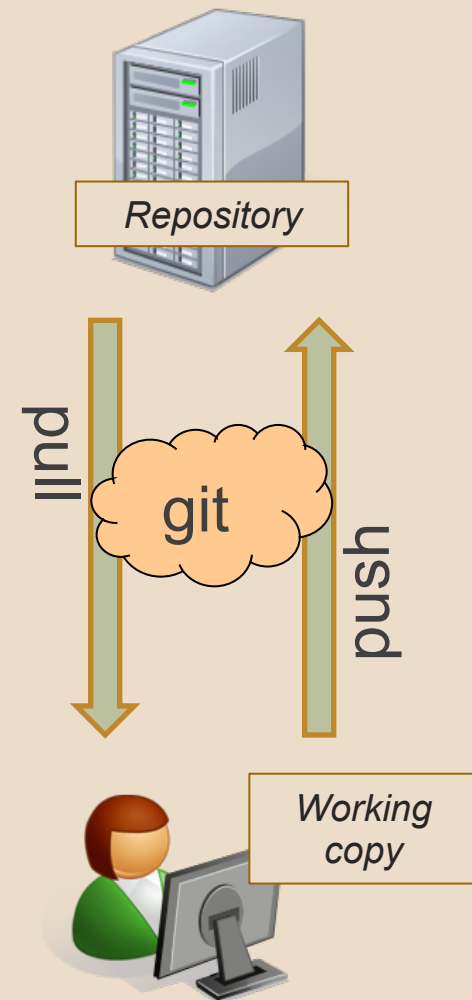


VERSION CONTROL

COMMON ACTIONS (CONT.)

Other common commands:

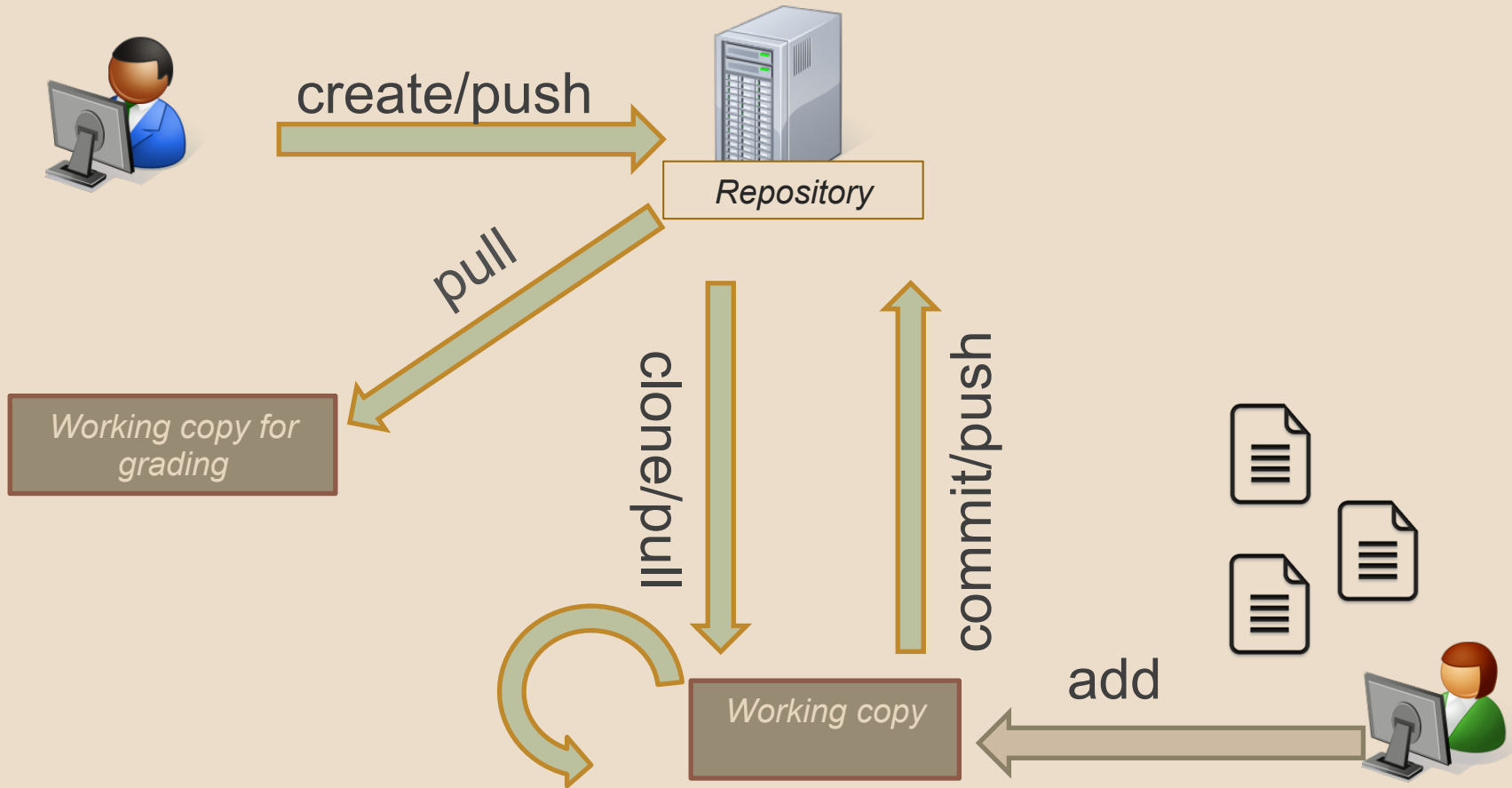
- **add, rm**
 - add or delete a file in the working copy
 - just putting a new file in your working copy does not add it to the repo!
 - still need to commit to make permanent



THIS QUARTER

- We distribute starter code by adding it to your GitLab **repo**. You retrieve it with **git clone** the first time then **git pull** for later assignments
- You will write **code** using Eclipse
- You turn in your files by **adding** them to the repo, **committing** your changes, and eventually **pushing** accumulated changes to GitLab
- You “turn in” an assignment by **tagging** your repo and pushing the tag to GitLab
- You will **validate** your homework by **SSHing** onto attu, cloning your repo, and running an Ant build file

331 VERSION CONTROL



ECLIPSE

WHAT IS ECLIPSE?

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

ECLIPSE SHORTCUTS

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

ECLIPSE and Java

- Get Java **8**
- Important: Java separates compile and execution, eg:
 - javac Example.java $\xrightarrow{\text{produces}}$ Example.class
 - Both compile and execute have to be the same Java!
- Please use **Eclipse 4.5 (Mars)**, “Eclipse for Java Developers”
- Instructions:
http://courses.cs.washington.edu/courses/cse331/15au/tools/WorkingAtHome.html#Step_1

ECLIPSE and Java

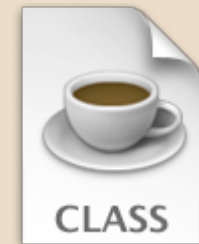
.java files

- Human readable 'code' file



.class files

- Compiled version of .java files. Typically represented as Byte code to run on the Java Virtual Machine (JVM)



.jar files

- Packaged aggregate of .class files and metadata



331 VERSION CONTROL

- Your main repository is on GitLab
- Only clone once (unless you're working in a lot of places)
- Don't forget to add/commit/push files!
- Check in your work!

HW 3

- Many small exercises to get you used to version control and tools and a Java refresher
- More information on homework instructions: <http://courses.cs.washington.edu/courses/cse331/15au/hws/hw3/hw3.html>
- Committing changes: [Instructions](#)
 - How you turn in your assignments
- Updating changes: [Instructions](#)
 - How you retrieve new assignments

Turning in HW3

- [Instructions](#)
- Done by simply committing your changes
 - Good to do this early and often
 - Then when you're done, create a **hw3-final tag** on the last commit and push the tag to the repo
- After the final commit and tag pushed, remember to log on to attu and run [ant validate](#)

Ant Validate

- **What will this do?**

- You start with a freshly cloned copy of your repo and do “git checkout hw3-final” to switch to the files you intend for us to grade, then run ant validate
- Makes sure you have all the **required** files
- Make sure your homework builds without errors
- Passes specification and implementation tests in the repository
 - **Note:** this does not include the additional tests we will use when grading
 - This is just a sanity check that your current tests pass

Ant Validate

- **How do you run ant validate?**
 - Has to be done on attu from the command line since that is the environment your grading will be done on
 - Do not use the Eclipse ant validate build tool!
 - Be *sure* to use a fresh copy of your repo, and discard that copy when you're done
 - If you need to fix things, do it in your primary working copy (eclipse)

Ant Validate

- How do you run ant validate?
 - Steps
 - Log into attu via [SSH](#)
 - In attu, checkout a brand new local copy (clone) of your repository through the [command-line](#)
 - **Note:** Now, you have two local copies of your repository, one on your computer through Eclipse and one in attu
 - Go to the hw folder which you want to validate through the 'cd' command, then switch to the hw3 tag
 - For example: `cd ~/cse331/src/hw3`
`git checkout hw3-final`
 - Run ant validate

Ant Validate

- **How do you know it works?**
 - If successful, will output **Build Successful** at the bottom
 - If unsuccessful, will output **Build Failed** at the bottom with information on why
 - If ant validate failed, discard the validate copy of the repo on attu, fix and commit changes through eclipse, go back to attu, clone a fresh copy of the repo, and try ant validate again

ECLIPSE DEBUGGING (if time)

- `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 tools like Run, Stop, and Step Over. Below the toolbar, 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:** Displays a table with the following content:

Name	Value
this	RatPolyStackTest (id=33)
- Source Code Editor:** Shows the file `RatPolyStackTest.java` with 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");
```
- Outline View:** Lists the methods in the class:
 - 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. The main workspace is divided into several panels:

- Debug Console:** Shows a stack trace of the current execution, with the following methods listed:
 - 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 green vertical bar on the left side of the editor indicates a breakpoint is set on line 57. The code includes comments and assertions.
- Outline View:** Shows the class structure of the project.

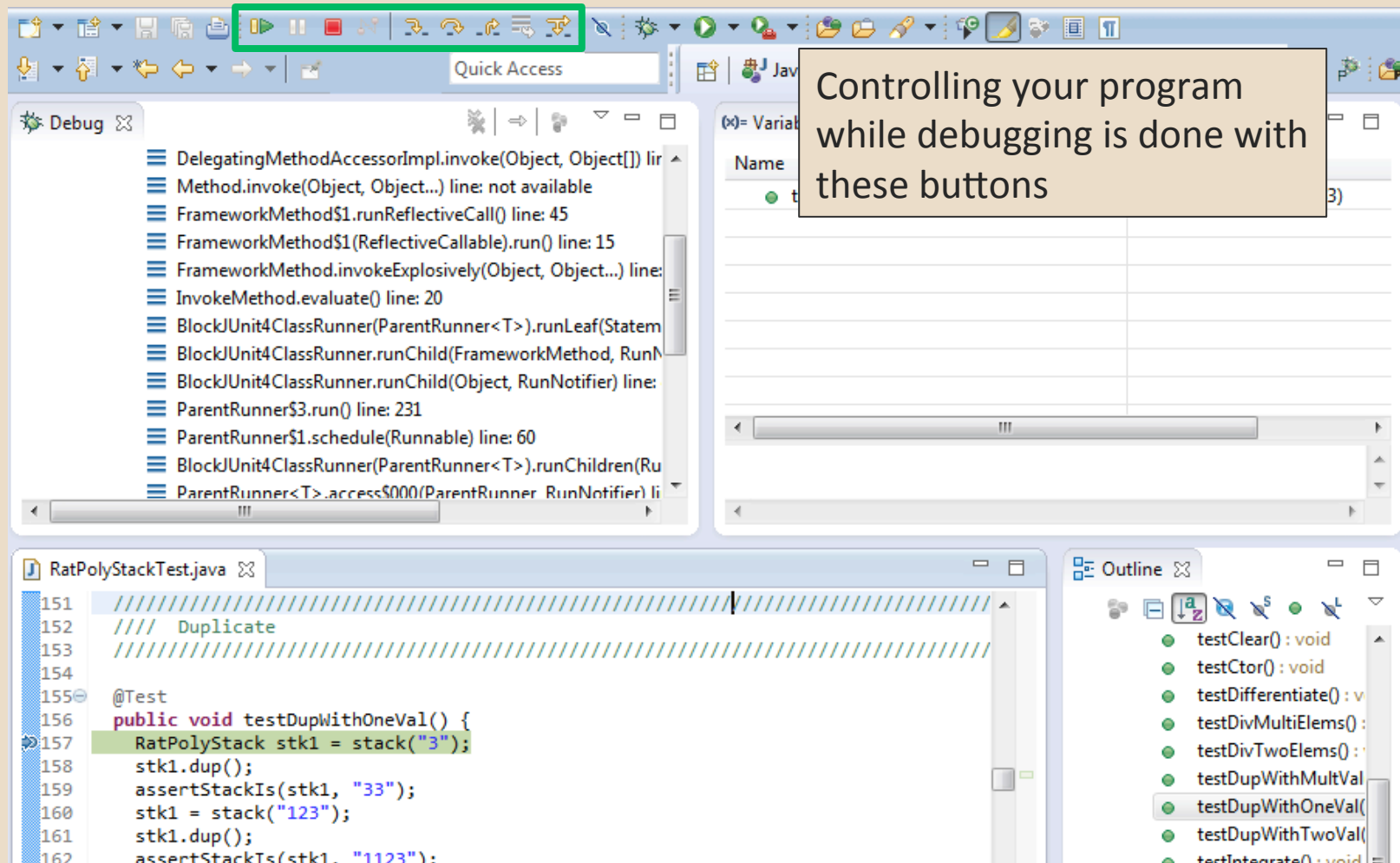
A text box overlaid on the code editor provides instructions: "Double click in the grey area to the left of your code to set a breakpoint. A breakpoint is a line that the Java VM will stop at during normal execution of your program, and wait for action from you."

ECLIPSE DEBUGGING

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, a text box contains the instruction: "Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints." The IDE shows a Java project named "RatPolyStackTest" with a class "RatPolyStackTest.java" open in the editor. The code in the editor includes a test method "testDupWithOneVal()". The Outline view on the right shows a list of methods in the class, including "testClear()", "testCtor()", "testDifferentiate()", "testDivMultiElems()", "testDivTwoElems()", "testDupWithMultVal()", "testDupWithOneVal()", "testDupWithTwoVal()", and "testIntegrate()".

ECLIPSE DEBUGGING



ECLIPSE DEBUGGING

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

Debug Console:

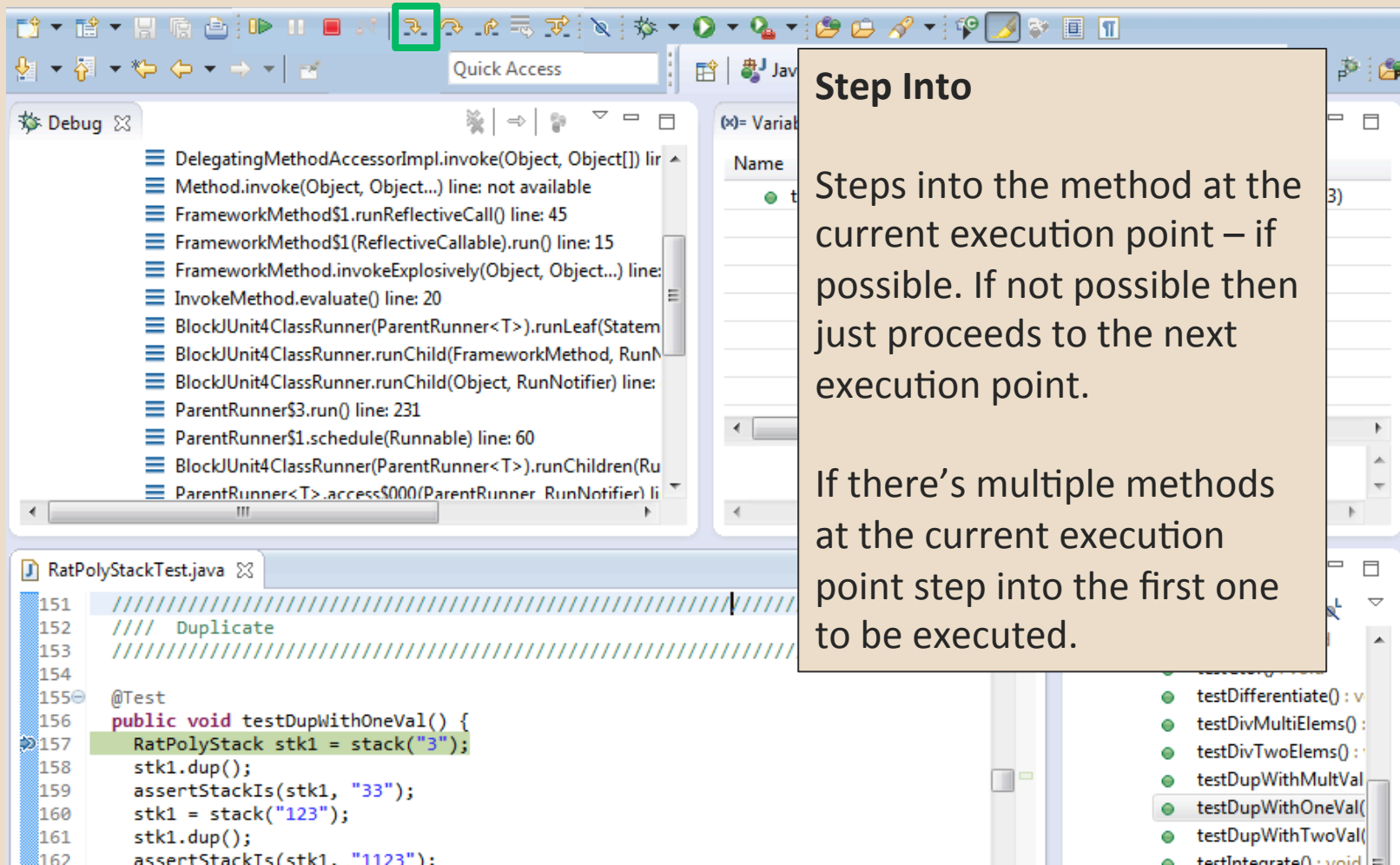
- DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: ...
- Method.invoke(Object, Object...) line: not available
- FrameworkMethod\$1.runReflectiveCall() line: 45
- FrameworkMethod\$1(ReflectiveCallable).run() line: 15
- FrameworkMethod.invokeExplosively(Object, Object...) line: ...
- InvokeMethod.evaluate()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

```
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:

- 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 during a debug session. The top toolbar has a green box around the 'Step Into' icon (a blue arrow pointing into a square). The 'Debug' console on the left shows a stack trace of method calls. The main editor window displays the source code for `RatPolyStackTest.java`, with line 157 highlighted in green: `RatPolyStack stk1 = stack("3");`. A callout box on the right explains the 'Step Into' action.

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");
}
```

ECLIPSE DEBUGGING

Step Over

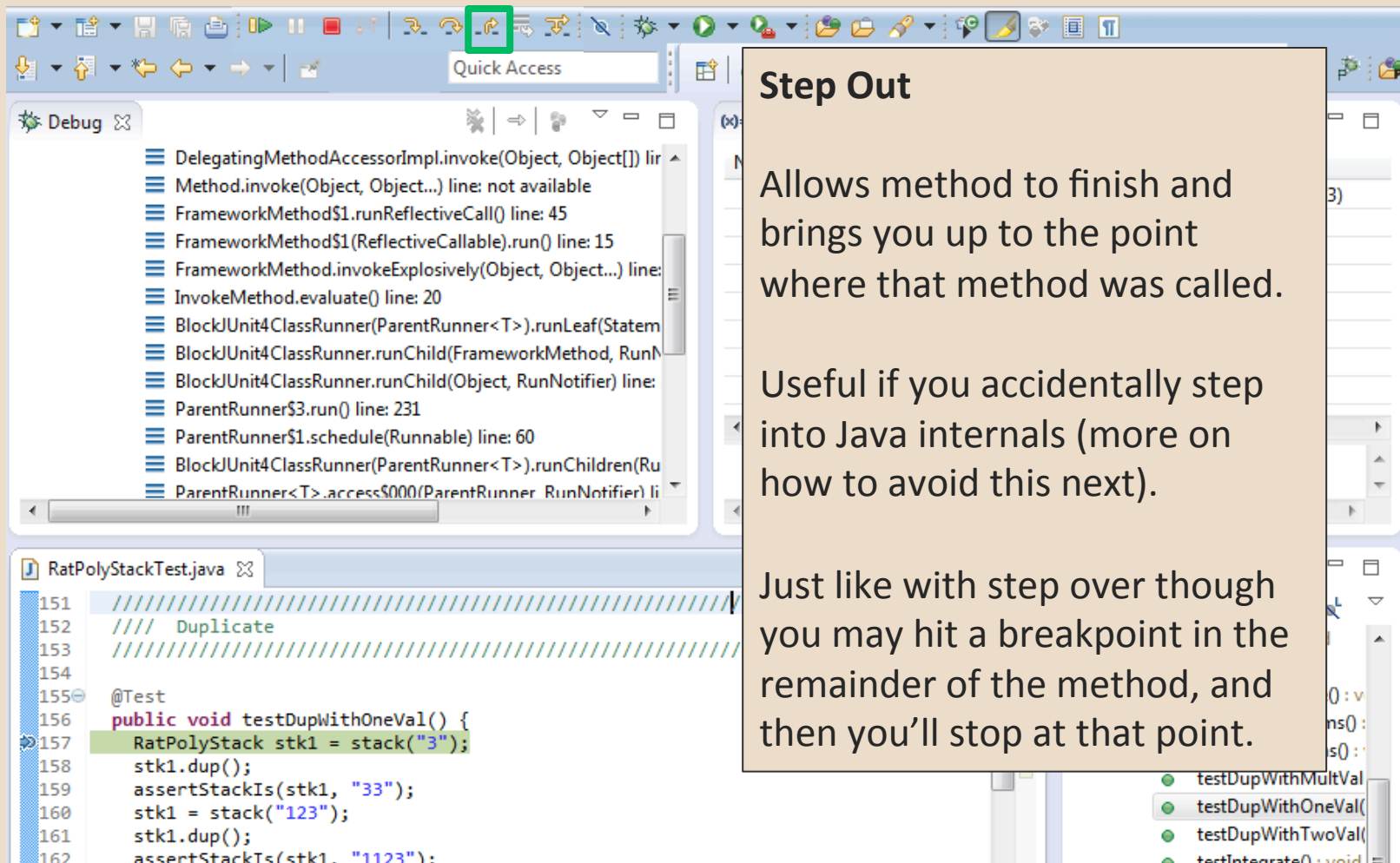
Steps over any method calls at the current execution point.

Theoretically program proceeds just to the next line.

BUT, if you have any breakpoints set that would be hit in the method(s) you stepped over, execution will stop at those points instead.

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

ECLIPSE DEBUGGING



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

Step Out

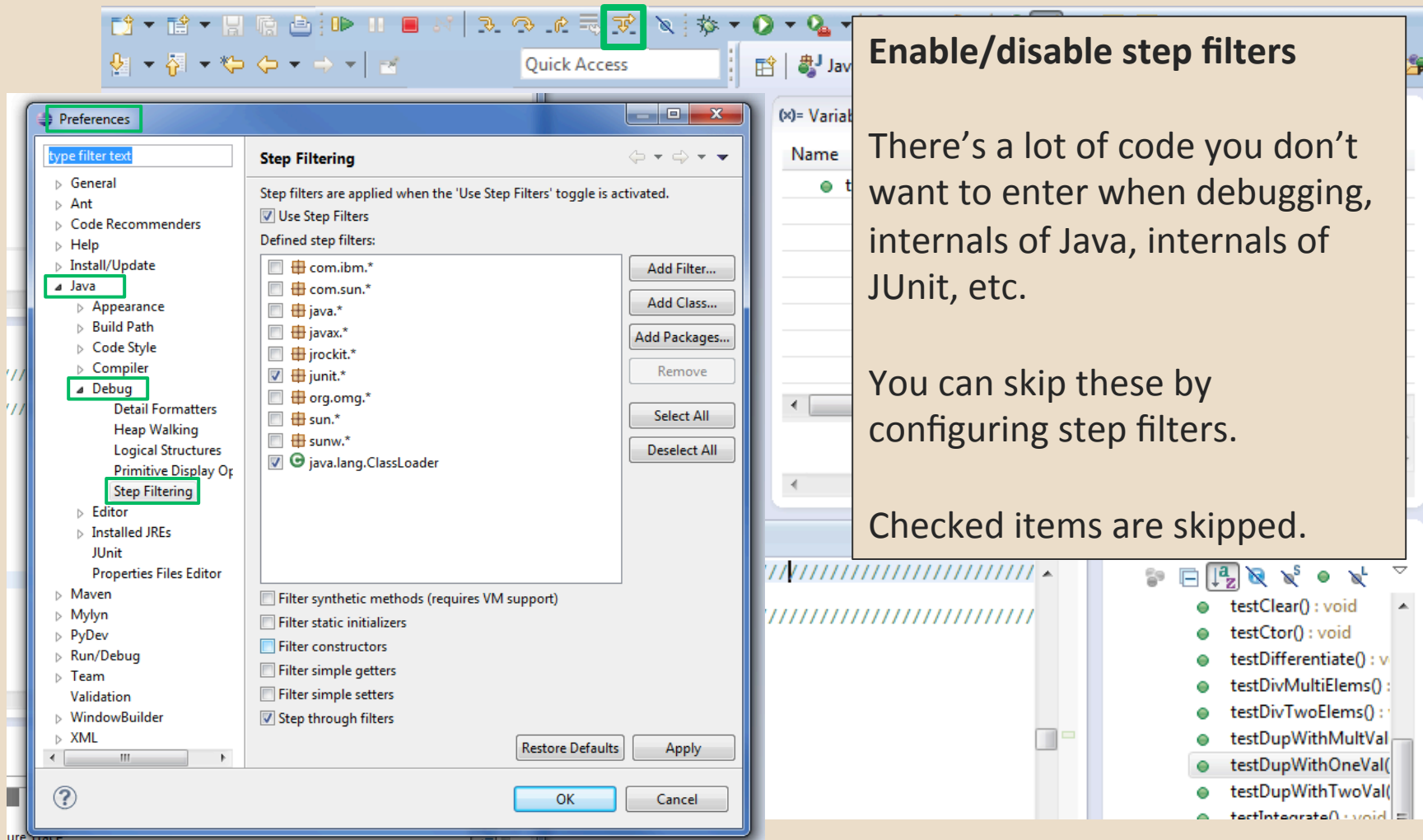
Allows method to finish and brings you up to the point where that method was called.

Useful if you accidentally step into Java internals (more on how to avoid this next).

Just like with step over though you may hit a breakpoint in the remainder of the method, and then you'll stop at that point.

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


ECLIPSE DEBUGGING



Enable/disable step filters

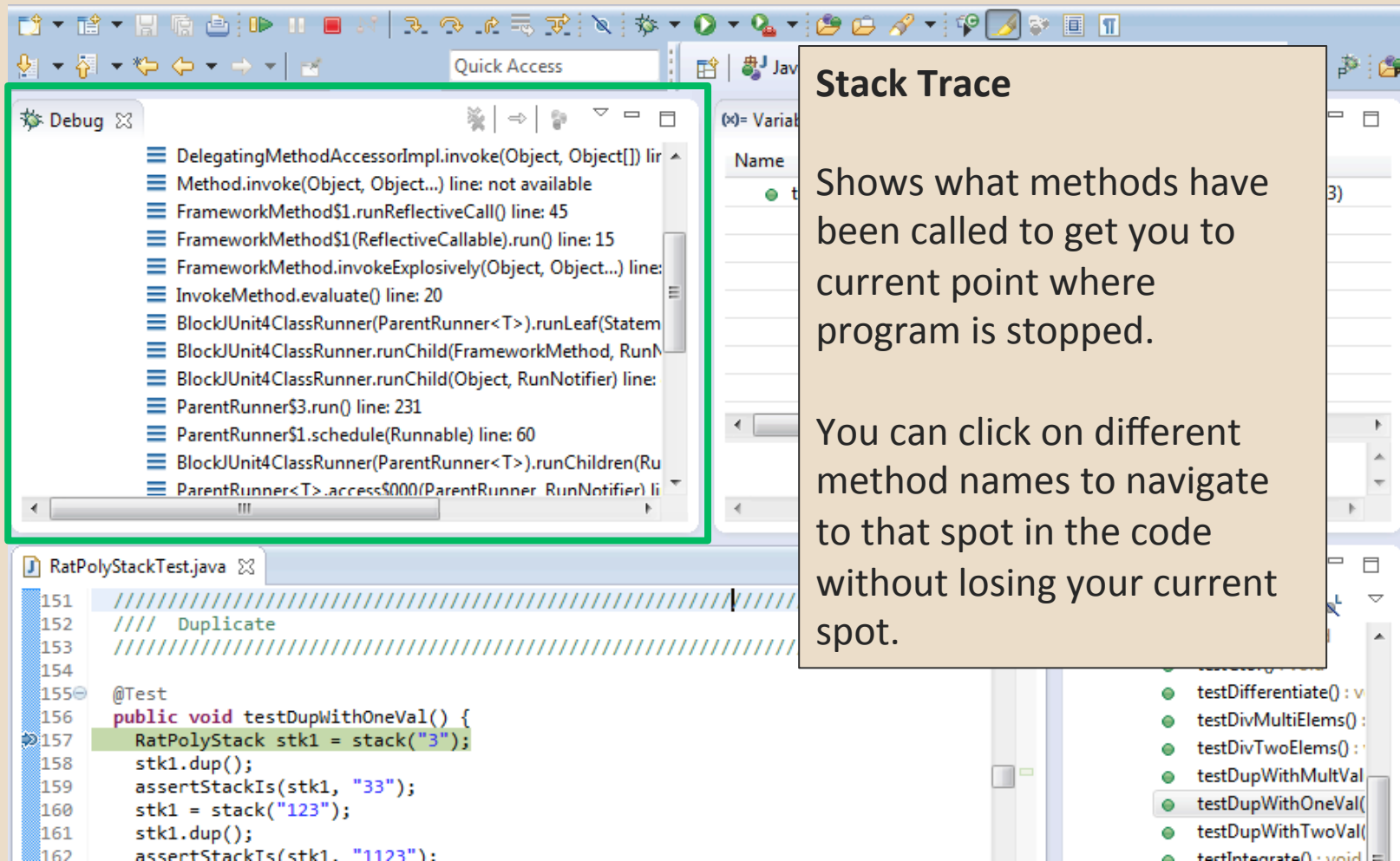
There's a lot of code you don't want to enter when debugging, internals of Java, internals of JUnit, etc.

You can skip these by configuring step filters.

Checked items are skipped.

The screenshot shows the Eclipse IDE interface. The top toolbar has a green box around the 'Step Filtering' icon. The Preferences dialog is open, with the 'Debug' category selected in the left sidebar. The 'Step Filtering' sub-category is expanded, and the 'Use Step Filters' checkbox is checked. The 'Defined step filters' list includes 'java.lang.ClassLoader' which is checked. Other filters like 'com.ibm.*', 'com.sun.*', 'java.*', 'javax.*', 'jrockit.*', 'junit.*', 'org.omg.*', 'sun.*', and 'sunw.*' are unchecked. The 'Filter synthetic methods (requires VM support)', 'Filter static initializers', 'Filter constructors', 'Filter simple getters', and 'Filter simple setters' are also unchecked. The 'Step through filters' checkbox is checked. The 'OK' button is highlighted in blue.

ECLIPSE DEBUGGING



The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons for file operations and debugging. Below the toolbar is the 'Quick Access' search bar. The main workspace is divided into several panes. On the left, the 'Debug' console displays a stack trace with the following entries:

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

On the right, the 'Variables' pane shows a table with a 'Name' column and a 'Value' column. Below the stack trace, the 'RatPolyStackTest.java' code editor is visible, showing 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");
163 }
```

The 'testDupWithOneVal()' method is highlighted in green. The 'Debug' console is also highlighted with a green border. A text box on the right explains the stack trace:

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");
```

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.

Name	Value
this	RatPolyStackTest (id=33)

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 View (top right):** A table showing the current state of variables. The 't' variable is expanded to show its fields: 'coeff' and 'expt'. The 'expt' field is highlighted in yellow, indicating it has changed since the last breakpoint. The value of 'expt' is 5.
- Code Editor (bottom left):** Shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted, corresponding to the current execution point: `RatPolyStack stk1 = stack("3");`
- Outline View (bottom right):** Shows a list of methods in the current class, including `testDupWithOneVal()`.

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

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

ECLIPSE DEBUGGING

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

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

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

- Variables Window (top right, green border):** A table showing the current state of variables. The 'Value' tab is active. The variable 't' is expanded to show its fields: 'coeff' (RatNum) and 'expt' (RatNum). The 'expt' field is highlighted in yellow, indicating it has changed since the last breakpoint. The value of 'expt' is 5. Below the table, the expression $-2*x^5$ is visible.
- 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 comments, a `@Test` annotation, and a `testDupWithOneVal()` method.
- Outline (bottom right):** Shows a list of methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

ECLIPSE DEBUGGING

There's a powerful right-click menu.

- See all references to a given variable
- See all instances of the variable's class
- Add watch statements for that variable's value (more later)

The screenshot shows the Eclipse IDE interface during a debug session. The Variables view is open, showing a tree structure with 'this' (RatTermTest (id=33)) and a local variable 't'. Under 't', there are two variables: 'coeff' and 'expt', with 'expt' selected. A right-click context menu is displayed 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 highlighted. In the background, the Java editor shows a test method 'testDupWithOneVal()' with a breakpoint at line 157, where the variable 'stk1' is assigned the value '33'.

```
151 ///////////////////////////////////////////////////////////////////
152 /// Duplicate
153 ///////////////////////////////////////////////////////////////////Runner.class
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("33");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
163 }
```

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 running, stepping through code, and other debugging actions. The main editor displays the source code for `RatPolyStackTest.java`, with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The Variables view on the right is expanded to show the logical structure of the `stk1` object, which is a `RatPolyStack` (id=44). It contains a `polys` field (id=49) of type `Stack<E>`, which contains a `polys` array (id=719) of `RatPoly` objects. The first element of the array (id=728) is an `ArrayList<E>` containing a `RatTerm` (id=731) object. The `RatTerm` object has a `coeff` field (id=733) of type `RatNum` and an `expt` field with the value 0. A green box highlights the expanded logical structure of the `stk1` object. The bottom right pane shows a list of methods available for the current object, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

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

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

- testClear(): void
- testCtor(): void
- testDifferentiate(): 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.

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 includes options for Hit count, Suspend thread, Suspend VM, and Conditional (Suspend when 'true' or Suspend when value changes). A text field shows the condition `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.

The screenshot displays the Eclipse IDE interface. The Breakpoints view is open, showing a list of breakpoints. The breakpoint for 'RatPolyStackTest [line: 162] - testDupWithOneVal()' is disabled, indicated by a green box around its unchecked checkbox. The code editor shows the corresponding Java code with line 162 highlighted.

```
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.

```
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(): 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.

The screenshot shows the Eclipse IDE interface with the Breakpoints view open. A conditional breakpoint is configured for the method `testDupWithOneVal()` at line 159. The breakpoint is checked, and the configuration options are set to "Conditional" and "Suspend when 'true'". The condition field contains the expression `x == 6`. The background shows the source code editor with lines 159-162 and a list of test methods.

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

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

ECLIPSE DEBUGGING

Disable All Breakpoints

You can disable all breakpoints temporarily. This is useful if you've identified a bug in the middle of a run but want to let the rest of the run finish normally.

Don't forget to re-enable breakpoints when you want to use them again.

The screenshot shows the Eclipse IDE interface during a debug session. The 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 1, and the 'Suspend' mode is set to 'Suspend thread'. The 'Conditional' checkbox is checked, and the 'Suspend when' mode is set to 'true'. The condition field contains the expression `x == 6`.

The main editor shows the source code for `@Test public void testDupWithOneVal()`. Line 157, `RatPolyStack stk1 = stack("3");`, is highlighted in green, indicating the current execution point. The bottom right pane shows a list of methods, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate() : void`.

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 with a conditional breakpoint configured. The Breakpoints view on the right lists several breakpoints, with the one at line 159 of RatPolyStackTest.java selected and highlighted. The configuration for this breakpoint is as follows:

- Hit count: (empty field)
- Suspend thread (selected), Suspend VM (unselected)
- Conditional (checked), Suspend when 'true' (selected), Suspend when value changes (unselected)
- Condition: `x == 6`

The main editor window 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");
```

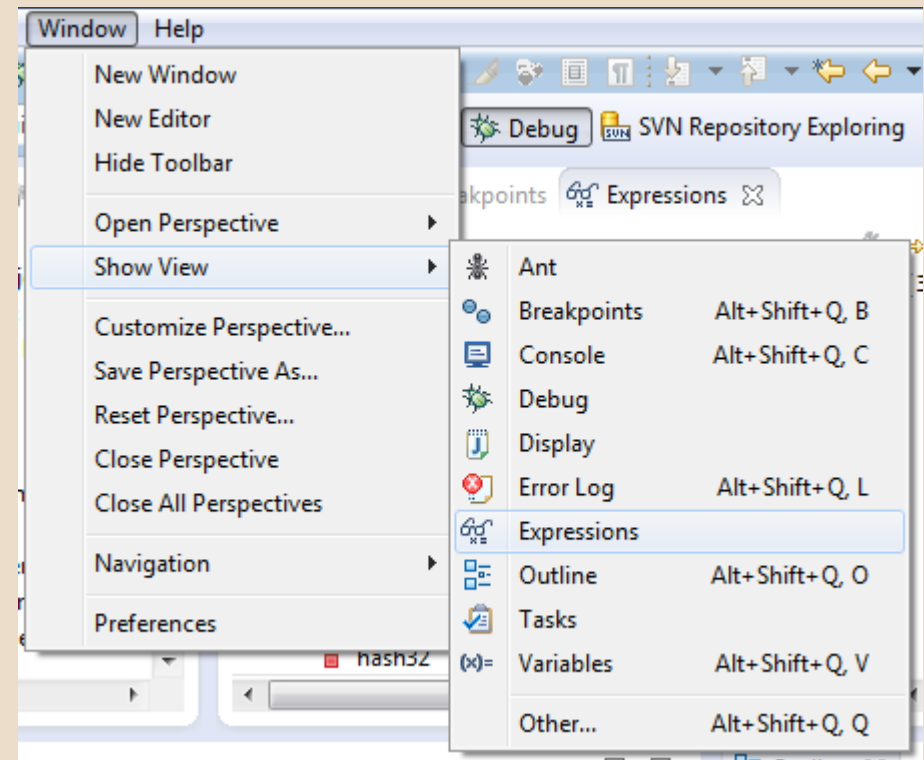
The bottom right pane shows a list of methods for the current class, including `testDupWithOneVal()`.

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

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

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

The Expressions window is highlighted with a green border. The background shows the Java editor with the following code snippet:

```
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 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