

SECTION 2:

HW3 Setup

cse331-staff@cs.washington.edu

slides borrowed and adapted from Alex Mariakis, CSE 390a, Justin Bare, Deric Pang, Erin Peach, Vinod Rathnam

LINKS TO DETAILED SETUP AND USAGE INSTRUCTIONS

- Path to all references: **Course Homepage** → **Resources** → **CSE331 documents**
 - <http://courses.cs.washington.edu/courses/cse331/17su/docs/docs.html>
- Working from home (& setup info): **Java, Eclipse, SSH**
 - <http://courses.cs.washington.edu/courses/cse331/17su/docs/tools/WorkingAtHome.html>
- **Editing, Compiling, Running, and Testing Programs**
 - <http://courses.cs.washington.edu/courses/cse331/17su/docs/tools/UsingEclipse.html>
- **Eclipse Reference**
 - <http://courses.cs.washington.edu/courses/cse331/17su/docs/tools/EclipseTips.html>
- **Version Control - Git**
 - <http://courses.cs.washington.edu/courses/cse331/17su/docs/tools/VersionControl.html>
- **Assignment Submission**
 - <http://courses.cs.washington.edu/courses/cse331/17su/docs/tools/TurnInHomework.html>

**DOES EVERYONE HAVE A CSE
NETID?**

DEVELOPER TOOLS

- **Version Control**
- **Eclipse and Java versions**
- **Remote access**

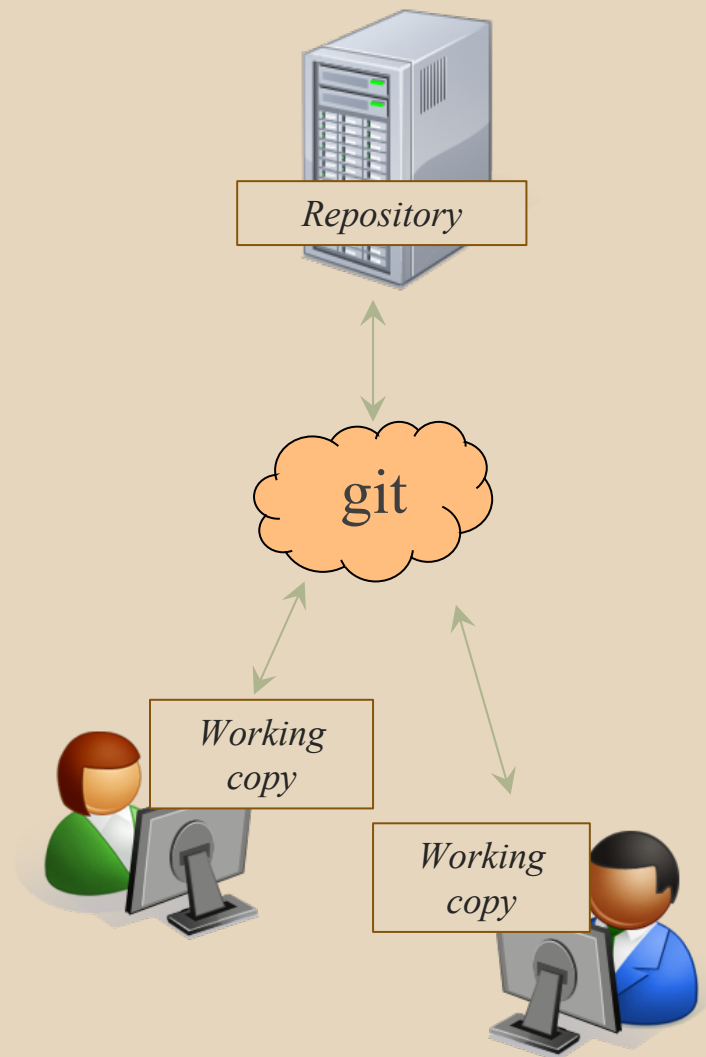
VERSION CONTROL

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

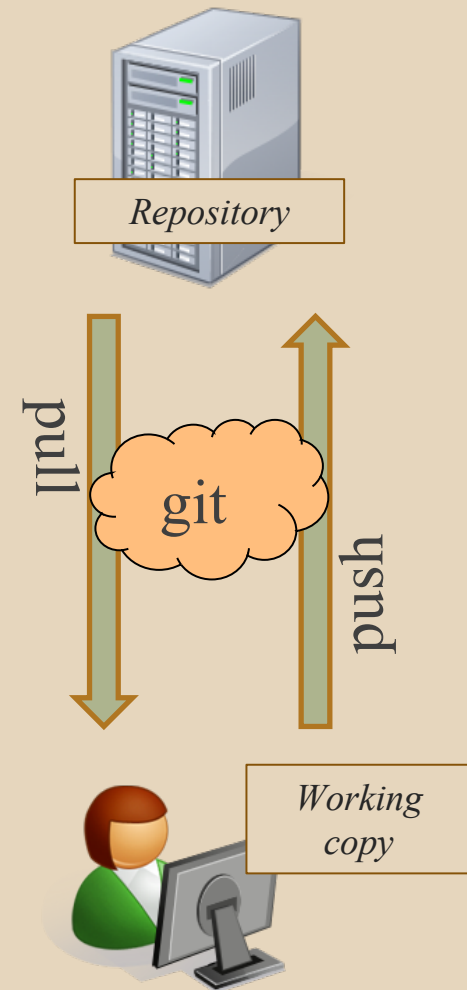
- Can create the repository anywhere
 - Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection
- But, usually you want the repository to be robust:
 - On a computer that's up and running 24/7
 - Everyone always has access 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:

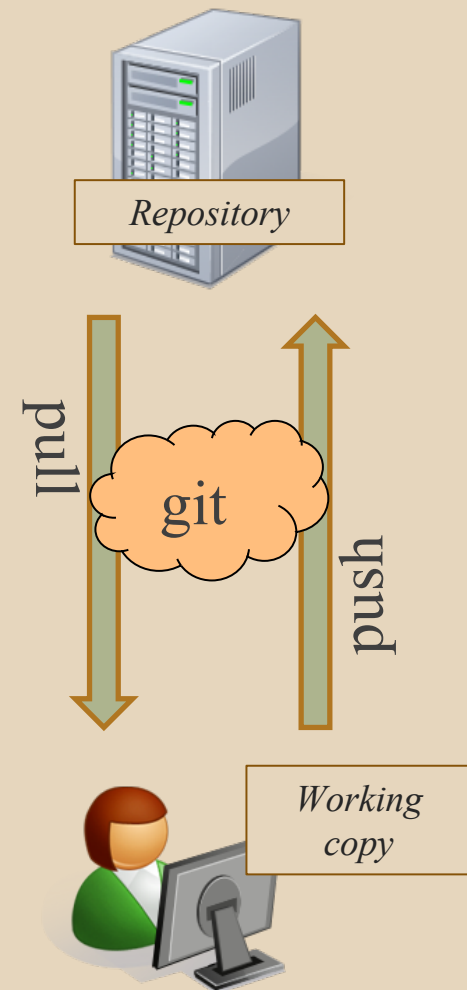
- **add / commit / push**
 - integrate changes *from* your working copy *into* the repository
- **pull**
 - integrate changes *into* your working copy *from* the repository



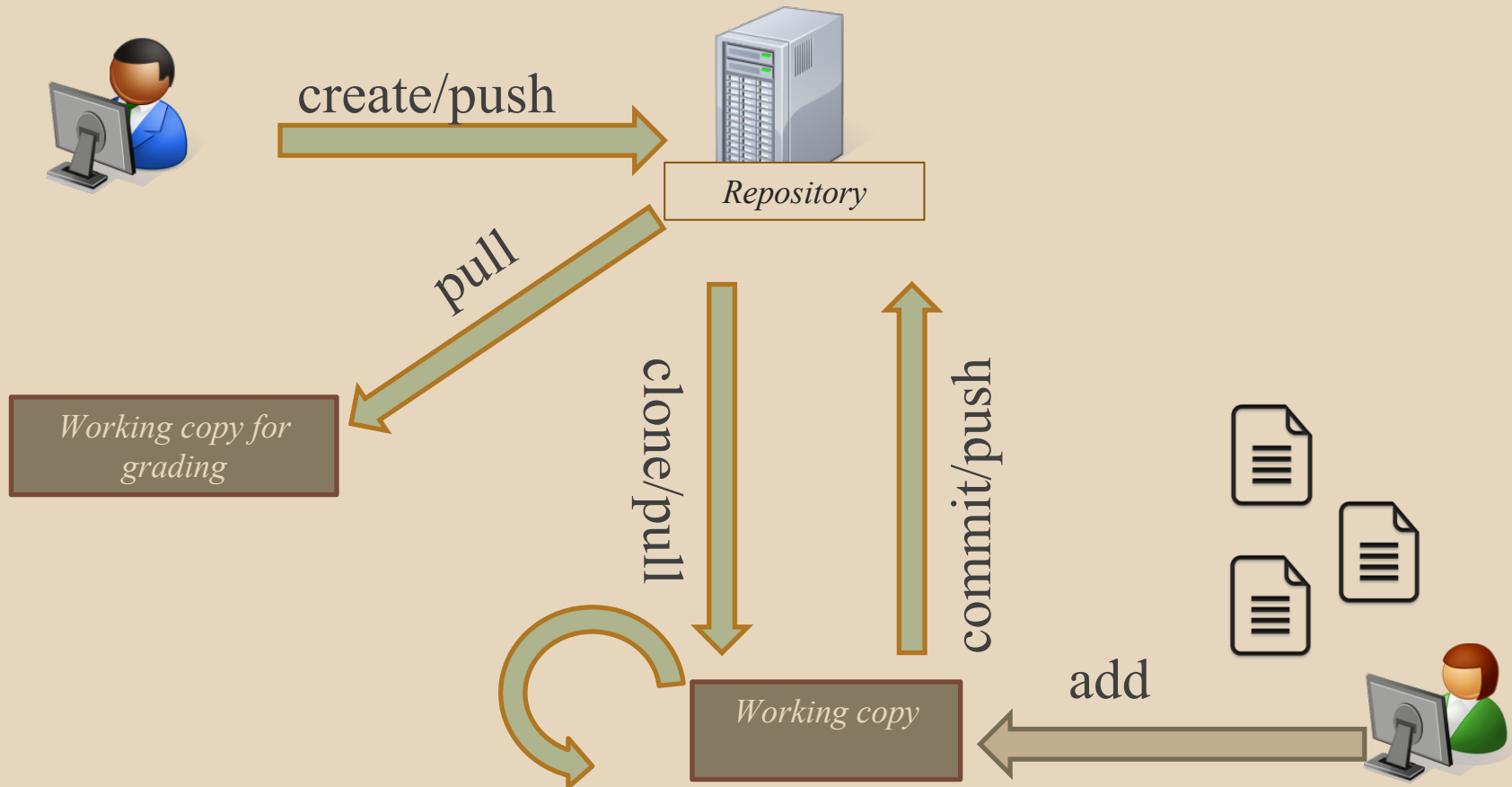
VERSION CONTROL UPDATING FILES

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



331 VERSION CONTROL



GIT BEST PRACTICES

- Add/commit/push your code **EARLY** and **OFTEN!!!**
 - You really, really, really don't want to deal with merge conflicts
 - Keep your repository up-to-date all the time
- Use the combined 'Commit and Push' tool in Eclipse
- Do not rename folders and files that we gave you – this will mess up our grading process and you could get a bad score
- Use the repo only for the homework
 - Adding other stuff (like notes from lecture) may mess up our grading process

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
 - Do this after committing and pushing your files
- You will **validate** your homework by **SSHing** onto attu, cloning your repo, and running an Ant build file

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/cse331/17su/docs/tools/WorkingAtHome.html> - **Step 1**

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/cse331/17su/hws/hw3/hw3.html>
- Cloning your repo: [Instructions](#)
- Committing changes: [Instructions](#)
 - How you turn in your assignments
- Updating changes: [Instructions](#)
 - How you retrieve new assignments

Turning in HW3

- [Instructions](#)
- Add/commit/push your final code
- Create a **hw3-final tag** on the last commit and push the tag to the repo (this can and should be done in Eclipse)
 - You can push a new hw3-final tag that overwrites the old one if you realize that you still need to make changes (Demo)
 - In Eclipse, just remember to check the correct checkboxes to overwrite existing tags
 - But keep track of how many late days you have left!
- 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
 - May need to create an SSH key on attu and add to GitLab: instructions
- 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 icons and a 'Quick Access' search bar. The main workspace is divided into several panels:

- Debug Console:** Located at the top left, it shows a stack of frames from the current thread. The top frame is `DelegatingMethodAccessorImpl.invoke(Object, Object[])`. Other frames include `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:** Located at the top right, it shows the current thread's variables. A table with two columns, 'Name' and 'Value', contains one entry: `this` with the value `RatPolyStackTest (id=33)`.
- Source Editor:** The main window shows the source code for `RatPolyStackTest.java`. The current line of execution is highlighted in green: `157 RatPolyStack stk1 = stack("3");`. The code includes comments for duplication and a `@Test` annotation. The method `testDupWithOneVal()` contains several assertions and stack operations.
- Outline View:** Located at the bottom right, it shows a list of methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal`, `testDupWithOneVal` (which is selected), `testDupWithTwoVal`, and `testIntegrate()`.

ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar includes standard development icons. The main workspace is divided into several panes:

- Debug Console:** Shows a stack trace of the current execution, including methods like `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 the following content:

Name	Value
this	RatPolyStackTest (id=33)
- Code Editor:** Displays the source code for `RatPolyStackTest.java`. Line 57 is highlighted with a green background, and a breakpoint icon (a small circle with a right-pointing arrow) is visible in the left margin. The code includes comments and assertions, such as `assertStackIs(stk1, "1123");`.
- Outline View:** Shows the class structure of the project.

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

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

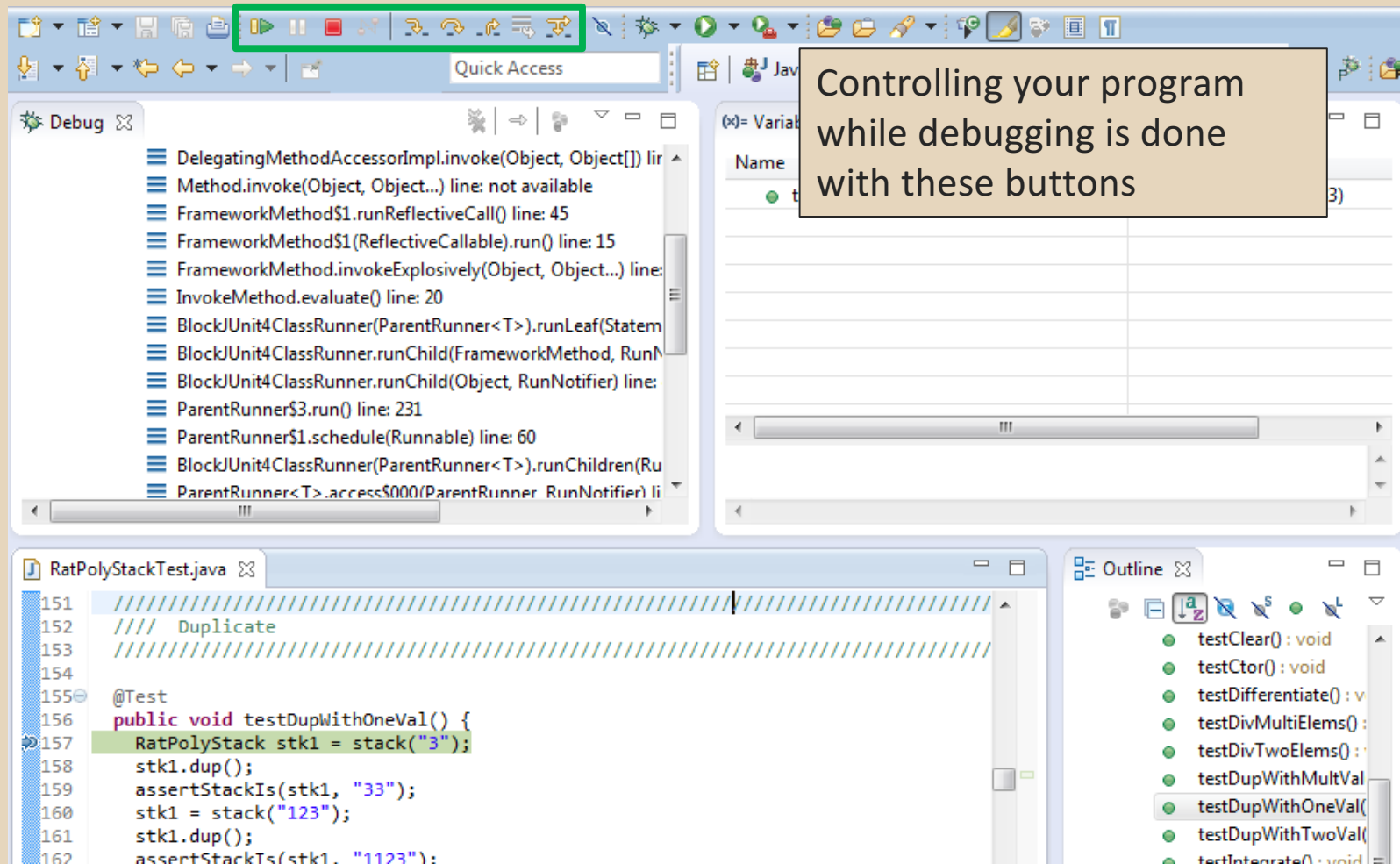
Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints.

```
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");
}
```

Value
RatPolyStackTest (id=33)

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

ECLIPSE DEBUGGING



ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar features a red square icon for stopping the debugger, which is highlighted with a green box. A text box with a white background and black border is overlaid on the toolbar, containing the text: "Play, pause, stop work just like you'd expect".

The Debug console on the left shows a stack trace of the current execution, with the following methods listed from top to bottom:

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

The Variables view on the right shows a table with the following content:

Name	Value
this	RatPolyStackTest (id=33)

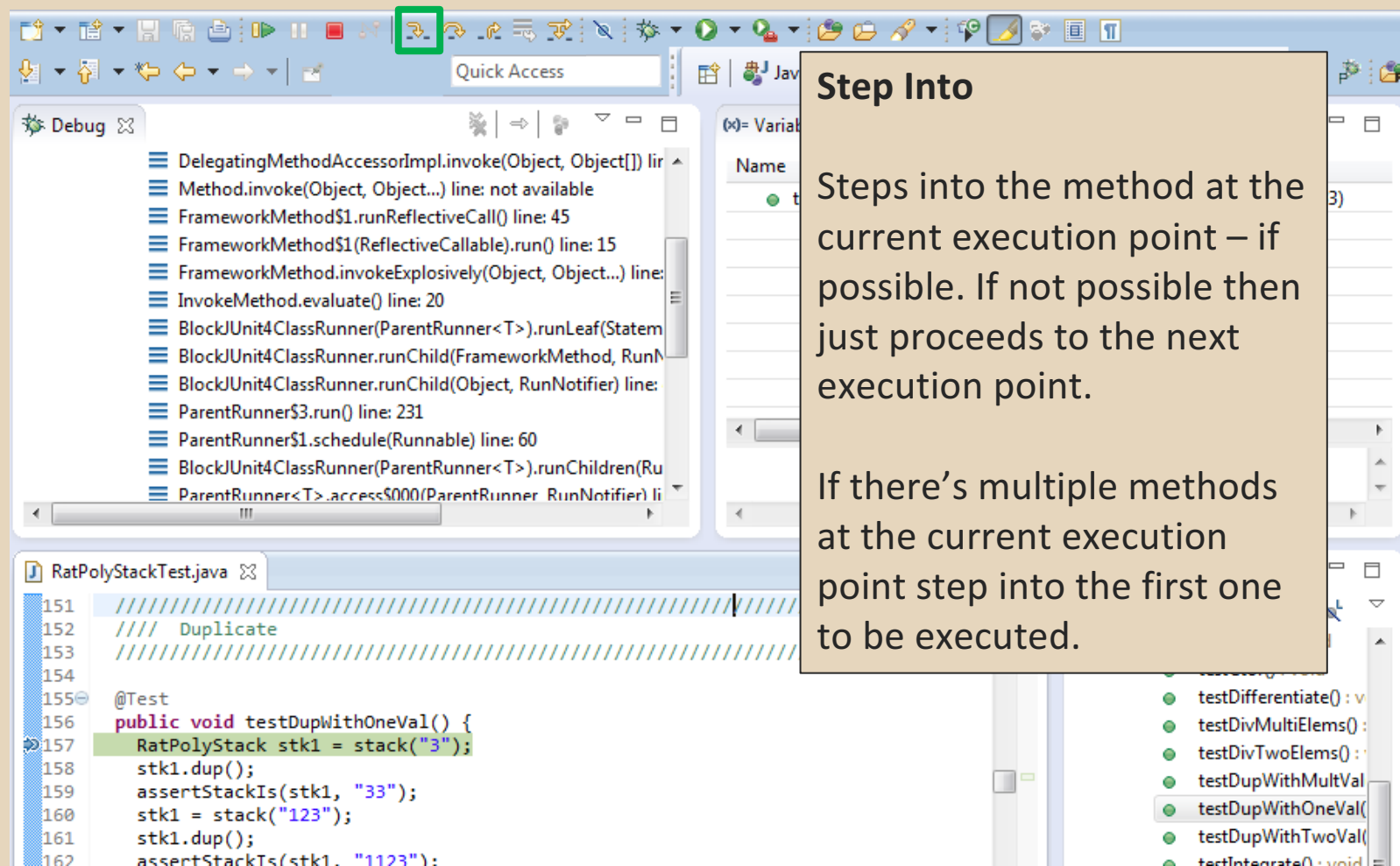
The Java code editor at the bottom shows the source code for `RatPolyStackTest.java`. The code is as follows:

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

The Outline view on the right shows a list of test methods, with `testDupWithOneVal()` selected:

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

ECLIPSE DEBUGGING



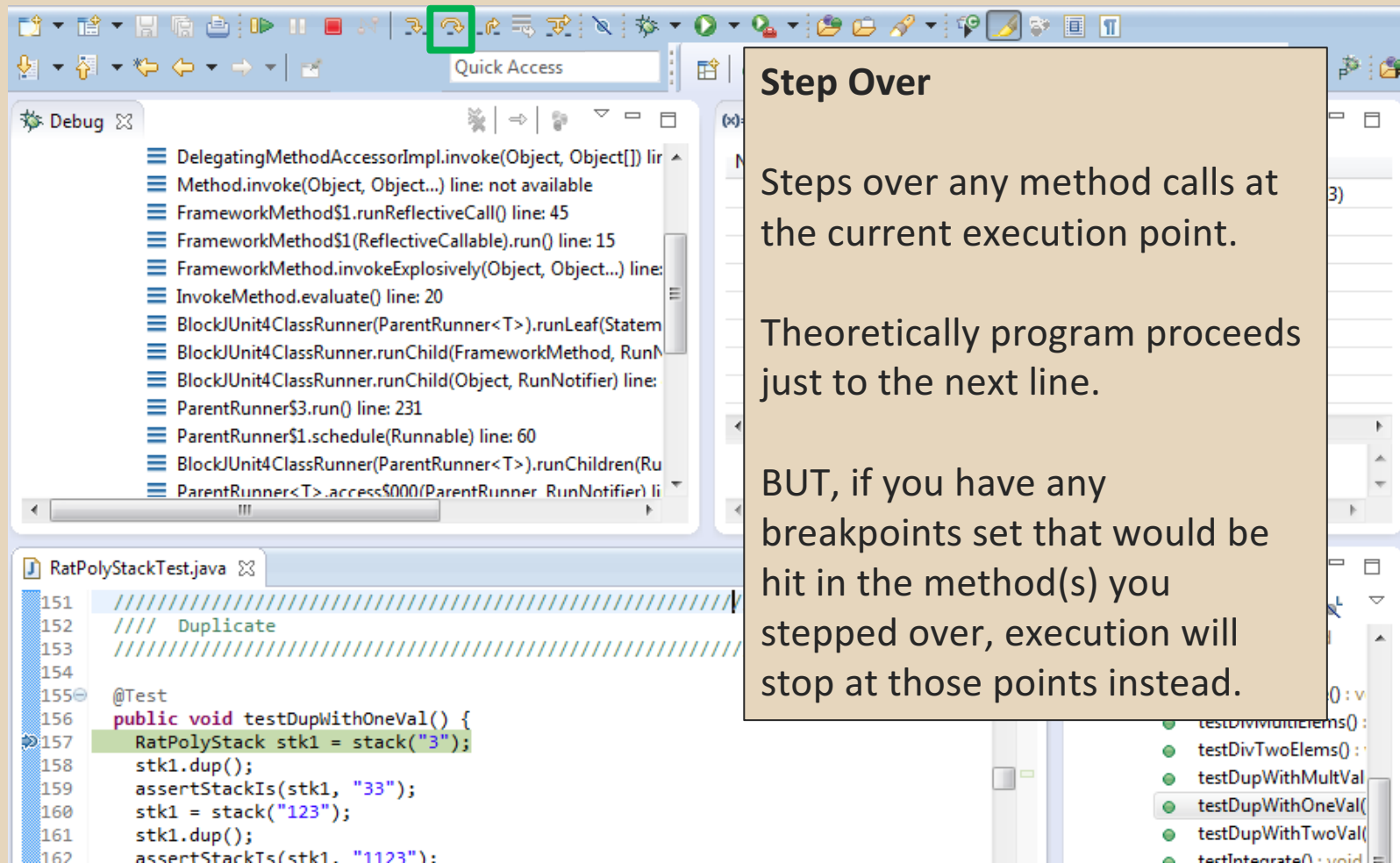
Step Into

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.

```
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

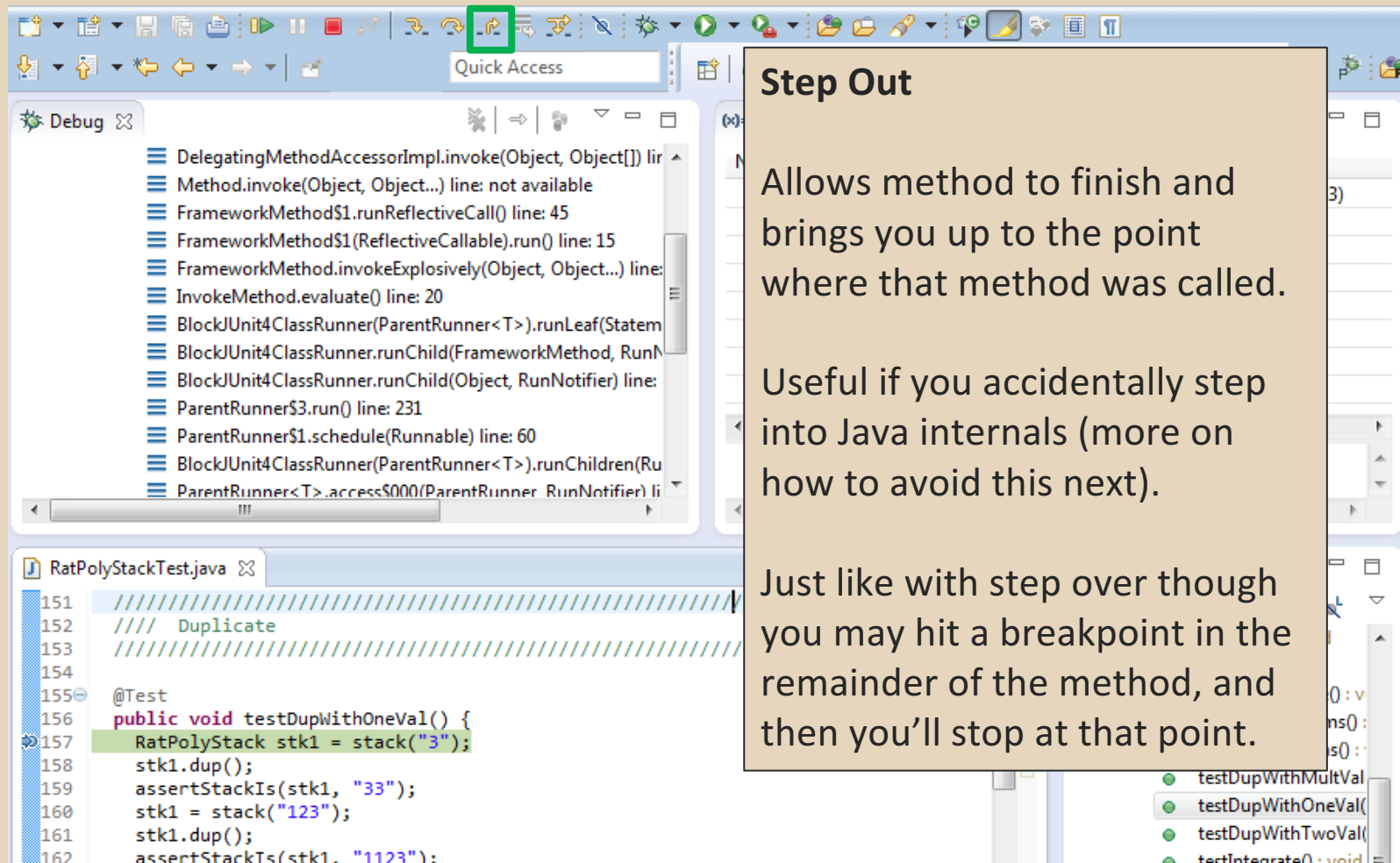
Steps over any method calls at the current execution point.

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



Step Out

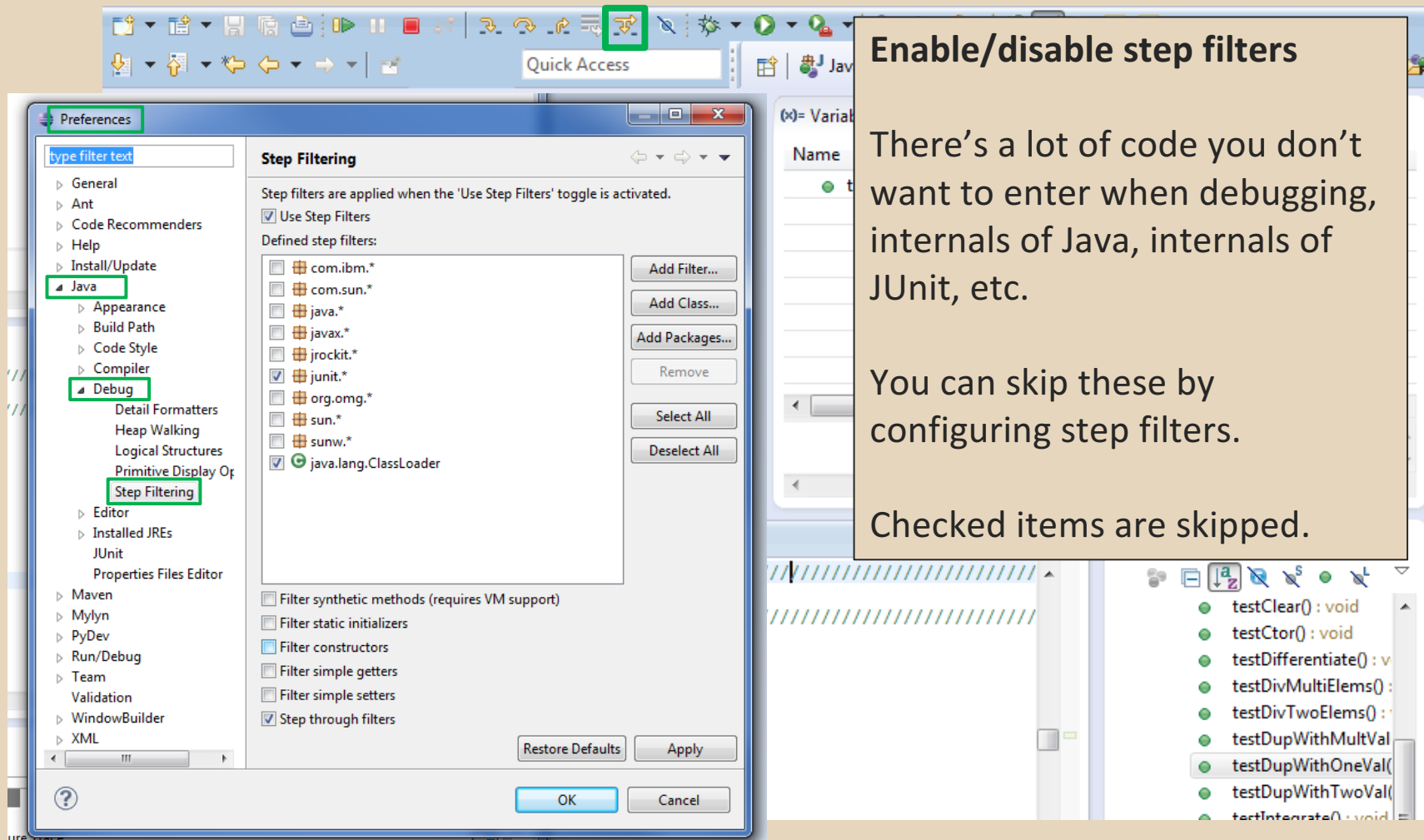
Allows method to finish and brings you up to the point where that method was called.

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 'Step Through' icon highlighted with a green box. The 'Preferences' dialog box is open, with the 'Debug' category selected in the left sidebar and 'Step Filtering' highlighted. The 'Step Filtering' section is active, showing a list of defined step filters. The 'Use Step Filters' checkbox is checked. The list includes 'com.ibm.*', 'com.sun.*', 'java.*', 'javax.*', 'jrocket.*', 'junit.*', 'org.omg.*', 'sun.*', 'sunw.*', and 'java.lang.ClassLoader'. The 'java.lang.ClassLoader' item is checked. Below the list are buttons for 'Add Filter...', 'Add Class...', 'Add Packages...', 'Remove', 'Select All', and 'Deselect All'. At the bottom of the dialog are 'Restore Defaults', 'Apply', 'OK', and 'Cancel' buttons. In the background, a code editor shows a list of test methods, with 'testDupWithOneVal()' selected.

ECLIPSE DEBUGGING

The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons for file operations and debugging. Below the toolbar, the 'Quick Access' search bar is visible. 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 top entry being 'DelegatingMethodAccessorImpl.invoke(Object, Object[])'. The stack trace is highlighted with a green border.
- Code Editor:** A window titled 'RatPolyStackTest.java' showing a Java test method. The method is annotated with '@Test' and is named 'testDupWithOneVal()'. The code includes comments and several lines of Java code, with line 157 highlighted in green.
- Variables View:** A window titled '(*)= Variables View' showing a list of variables. The variable 't' is visible.
- Test Runner:** A window showing a list of test methods, including 'testDifferentiate()', 'testDivMultiElems()', 'testDivTwoElems()', 'testDupWithMultVal()', 'testDupWithOneVal()', 'testDupWithTwoVal()', and 'testIntegrate()'. The 'testDupWithOneVal()' method is selected.

Stack Trace

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

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

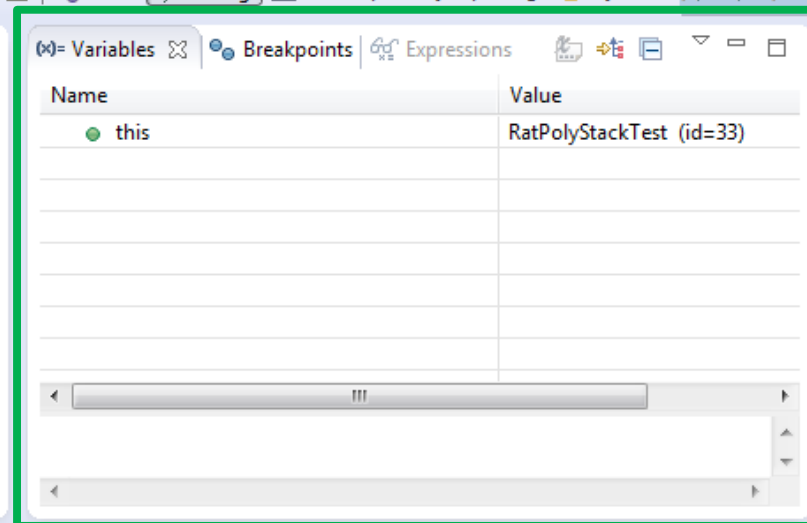
```
151 ////////////////////////////////////////////////////
152 // Duplicate
153 ////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");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

ECLIPSE DEBUGGING

Variables Window

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

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



Name	Value
this	RatPolyStackTest (id=33)

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

ECLIPSE DEBUGGING

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

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

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes icons for Java, Debug, SVN Repository Exploring, and PyDev. The main window is divided into several panes:

- Variables View:** A table showing the current state of variables. The 'expt' variable is highlighted in yellow, indicating it has changed since the last breakpoint. The 'Value' column shows '5'.
- Code Editor:** Shows the source code for `RatPolyStackTest.java`. A breakpoint is set at line 157, which is highlighted in green. The code includes a test method `testDupWithOneVal()` that creates a `RatPolyStack` object and performs operations on it.
- Outline View:** Shows the class structure, including methods like `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

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 displays the Eclipse IDE interface during a debug session. The top toolbar includes icons for Java, Debug, SVN Repository Exploring, and PyDev. The main window is divided into several panes:

- Variables View:** A table showing the current state of variables. The 'expt' variable is highlighted in yellow, indicating it has changed since the last breakpoint. The 'Value' column shows '5'.
- Code Editor:** Shows the source code for 'RatPolyStackTest.java'. A breakpoint is set at line 157, which is highlighted in green. The code includes a test method 'testDupWithOneVal()' that creates a 'RatPolyStack' object and performs operations on it.
- Outline View:** Shows the class structure, including methods like 'testClear()', 'testCtor()', 'testDifferentiate()', 'testDivMultiElems()', 'testDivTwoElems()', 'testDupWithMultVal()', 'testDupWithOneVal()', 'testDupWithTwoVal()', and 'testIntegrate()'. The 'testDupWithOneVal()' method is selected.

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

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

ECLIPSE DEBUGGING

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 variables value (more later)

The screenshot shows the Eclipse IDE in a debug state. 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'. The 'expt' variable is selected, and a right-click context menu is displayed over it. The menu items include: 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, and Inspect (Ctrl+Shift+I). The background shows a code editor with a Java class 'Runner.class' containing a test method 'testDupWithOneVal()' with several lines of code, including 'RatPolyStack stk1 = stack("3");' and 'stk1.dup();'.

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

```
151 ////////////////////////////////////////////////////
152 // Duplicate
153 ////////////////////////////////////////////////////
154 ////////////////////////////////////////////////////Runner.class
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

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 in a debug state. The top toolbar includes icons for Java, Debug, SVN Repository Exploring, and PyDev. The 'Variables' view is open, displaying a tree structure of variables. A green box highlights the 'stk1' variable and its nested structure, including 'polys', 'terms', and 'coeff'. The 'coeff' variable is expanded to show its value as 'RatNum (id=733)'. The 'Expressions' view is also visible, showing a list of test methods. The code editor at the bottom shows the source code for 'RatPolyStackTest.java', with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The stack trace on the left shows the current execution point in the `testDupWithOneVal()` method.

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

```
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. The breakpoint at line 162 is disabled, indicated by an unchecked checkbox. The Breakpoints view also shows options for hit count, suspend thread, suspend VM, and conditional execution. The conditional execution is set to 'Suspend when true' with the condition 'x == 6'. The code editor shows the source code for the test 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.

The screenshot shows the Eclipse IDE interface during a debugging session. The main window displays the source code of a Java test method `testDupWithOneVal()` in `RatPolyStackTest`. A breakpoint is set at line 159, which is highlighted in the code editor. The breakpoint configuration dialog is open, showing the following settings:

- Breakpoint: `RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()`
- Hit count: (highlighted with a green box)
- Suspend thread (selected)
- Conditional (checked)
- Condition: `x == 6`

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

ECLIPSE DEBUGGING

Conditional Breakpoints

Breakpoints can have conditions. This means the breakpoint will only be triggered when a condition you supply is true. This is very useful for when your code only breaks on some inputs!

Watch out though, it can make your code debug very slowly, especially if there's an error in your breakpoint.

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

The screenshot shows the Eclipse IDE interface. The Breakpoints view is open, displaying a list of breakpoints. A conditional breakpoint is highlighted with a green box. The condition is set to "x == 6". The breakpoint is for the method "testDupWithOneVal()" in the class "RatPolyStackTest" at line 159. The breakpoint is currently disabled (unchecked).

Breakpoints list:

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

Breakpoint configuration for the selected breakpoint:

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

Code editor view (bottom):

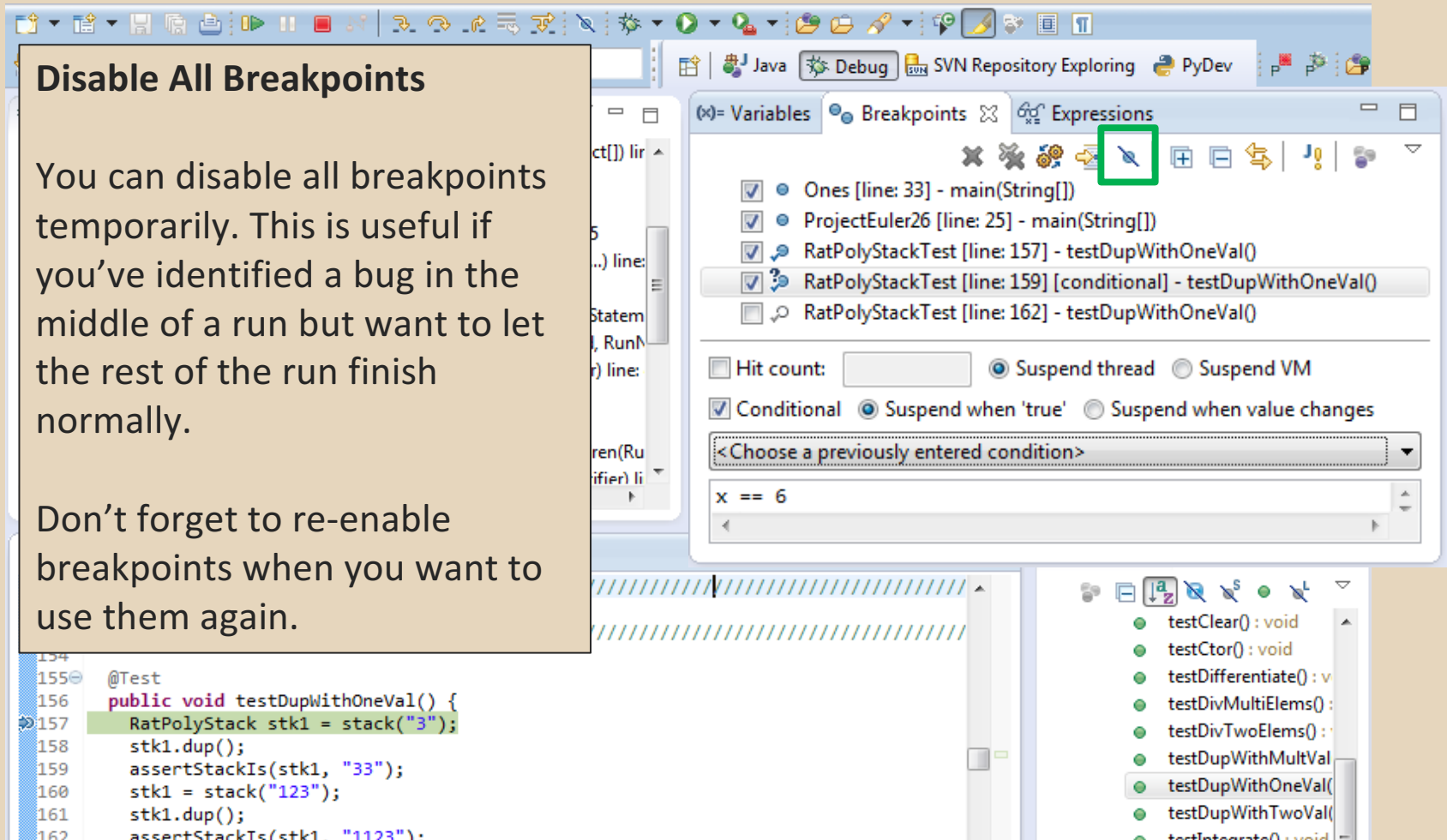
- testClear() : void
- testCtor() : void
- testDifferentiate() : void
- testDivMultiElems() :
- testDivTwoElems() :
- testDupWithMultVal
- testDupWithOneVal(
- testDupWithTwoVal(
- testIntegrate() : void

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 red box highlights the 'Disable All Breakpoints' icon (a blue circle with a diagonal slash) in the toolbar of the Breakpoints view. The list of breakpoints includes:

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

Below the list, the 'Hit count' is set to 1, and the 'Suspend' mode is set to 'Suspend thread'. The 'Conditional' checkbox is checked, and the 'Suspend when 'true'' radio button is selected. The condition field contains the expression `x == 6`.

In the background, the source code editor shows the following code snippet:

```
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. The top toolbar includes icons for Java, Debug, SVN Repository Exploring, and PyDev. The main window displays the 'Breakpoints' view, which lists several breakpoints for the 'RatPolyStackTest' class. A conditional breakpoint is highlighted at line 159, with the condition 'x == 6' entered in the text field below. The 'Suspend thread' option is selected. The 'Hit count' field is empty. The 'Conditional' section is checked, and 'Suspend when 'true'' is selected. A dropdown menu shows '<Choose a previously entered condition>'. The 'Expressions' view is also visible, showing a list of expressions. The 'Variables' view is also visible, showing a list of variables. The 'RatPolyStackTest.java' editor is open, showing the following code:

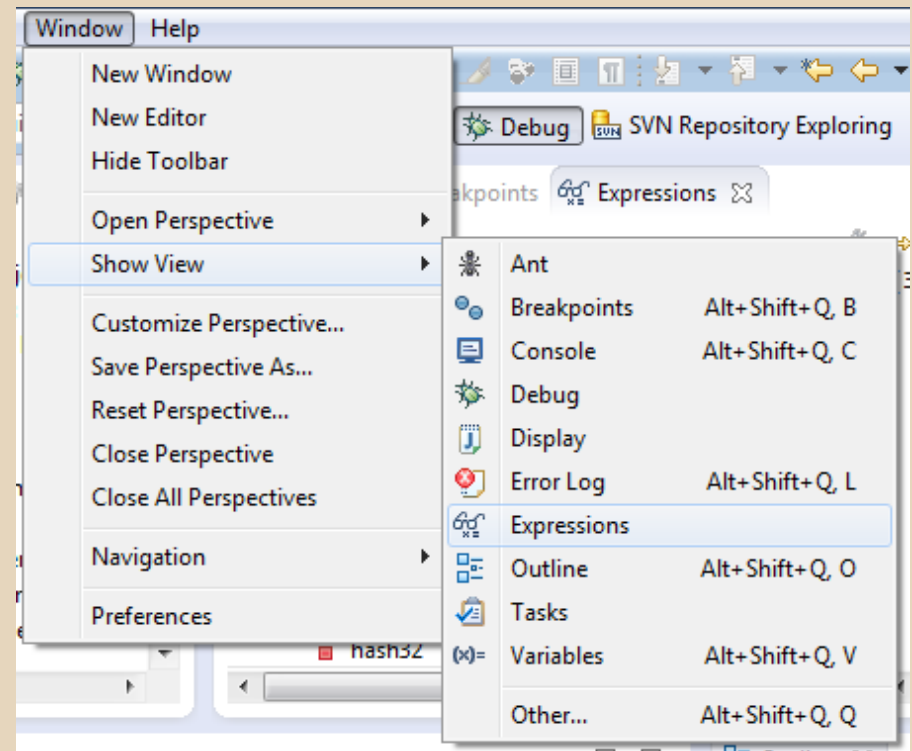
```
151 ////////////////////////////////////////////////////
152 // Duplicate
153 ////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");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

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 with the Expressions Window open. The Expressions Window displays a table of variables and their values:

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

The background shows a code editor with the following code snippet:

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

The Expressions Window also shows a list of test methods in the background:

- `testClear() : void`
- `testCtor() : void`
- `testDifferentiate() : void`
- `testDivMultiElems() :`
- `testDivTwoElems() :`
- `testDupWithMultVal`
- `testDupWithOneVal(`
- `testDupWithTwoVal(`
- `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 source code of `RatPolyStackTest.java` with a breakpoint at line 157. The code snippet is:

```
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 }
```