

# SECTION 2:

## HW3 Setup

**Justin Bare and Deric Pang**

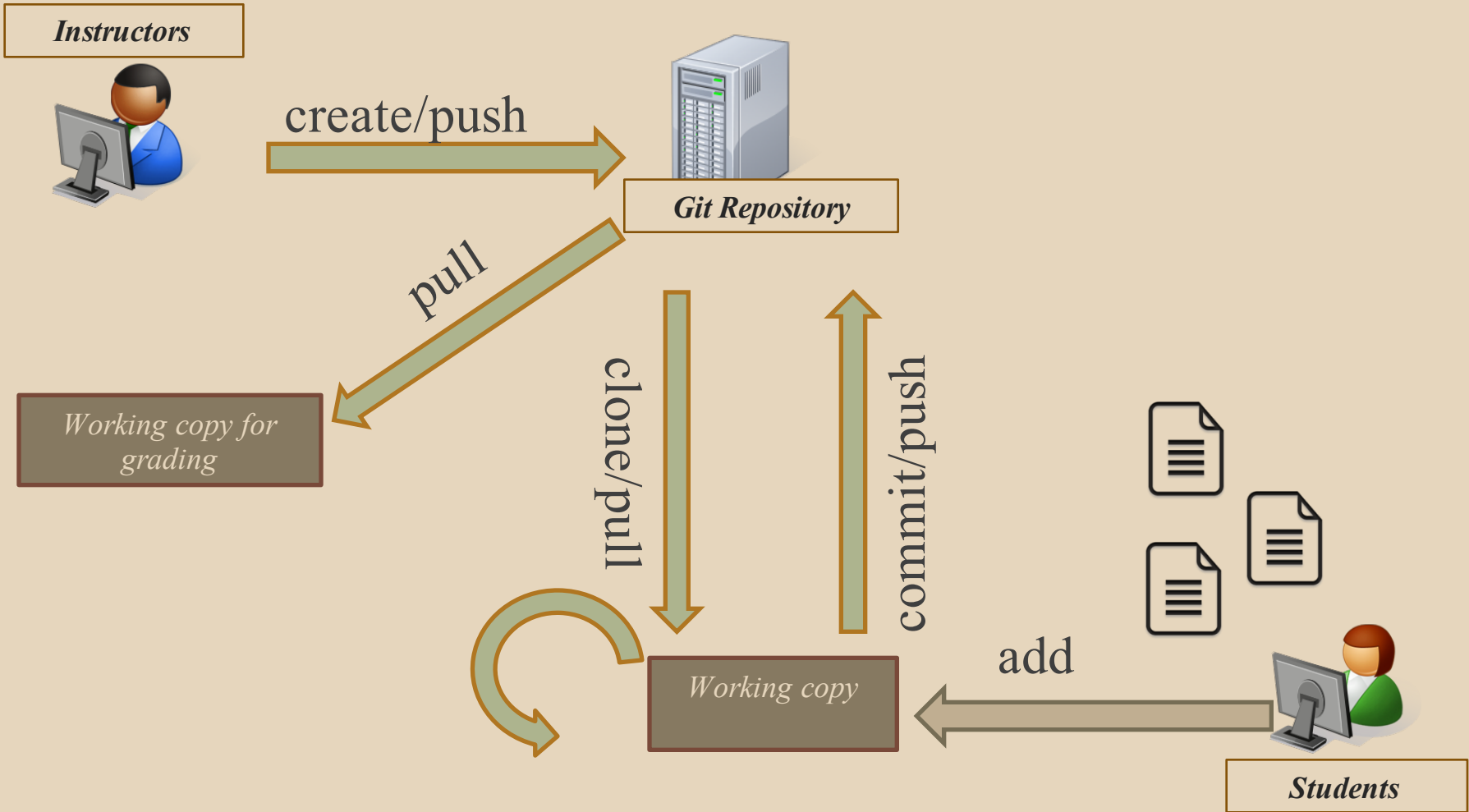
# OUTLINE

- **Version Control – Git**
- **Eclipse and Java versions**
- **SSH**
- **Ant**
- **Eclipse Debugging**

# LINKS TO DETAILED SETUP AND USAGE INSTRUCTIONS

- **All References**
  - <http://courses.cs.washington.edu/courses/cse331/16wi/#ref>
- **Basic Setup**
  - <http://courses.cs.washington.edu/courses/cse331/16wi/tools/basicSetup.html>
- **Working from home: Java, Eclipse, SSH**
  - <http://courses.cs.washington.edu/courses/cse331/16wi/tools/WorkingAtHome.html>
- **Editing, Compiling, Running, and Testing Programs**
  - <http://courses.cs.washington.edu/courses/cse331/16wi/tools/editing-compiling.html>
- **Eclipse Reference**
  - [http://courses.cs.washington.edu/courses/cse331/16wi/tools/eclipse\\_reference.html](http://courses.cs.washington.edu/courses/cse331/16wi/tools/eclipse_reference.html)
- **Version Control - Git**
  - <http://courses.cs.washington.edu/courses/cse331/16wi/tools/versioncontrol.html>
- **Assignment Submission**
  - <http://courses.cs.washington.edu/courses/cse331/16wi/tools/turnin.html>

# 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

# 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)
- 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
- Please use **Eclipse 4.5 (Mars)**, “Eclipse for Java Developers”



# DEMO

- Eclipse
- Git cloning

# DEVELOPMENT PROCESS

- 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 update your files on the repo by **adding** them to the repo, **committing** your changes, and eventually **pushing** accumulated changes to GitLab
- You “turn in” an assignment when you’re finished by **tagging** your repo and **pushing** the tag to GitLab
- You will validate your homework submission by **SSHing** onto attu, **cloning** your repo, and running **ant validate**

# HW 3

- Many small exercises to get you used to version control and tools and a Java refresher
- More information on homework instructions

# DEMO

- Editing HW3
- Git pull in Eclipse
- Add/commit/push in Eclipse

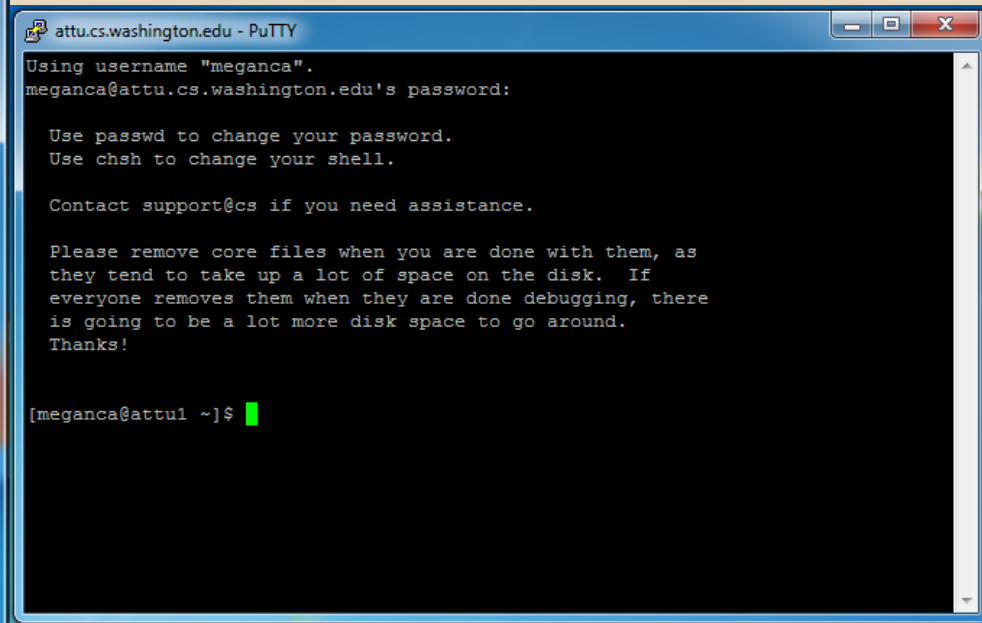
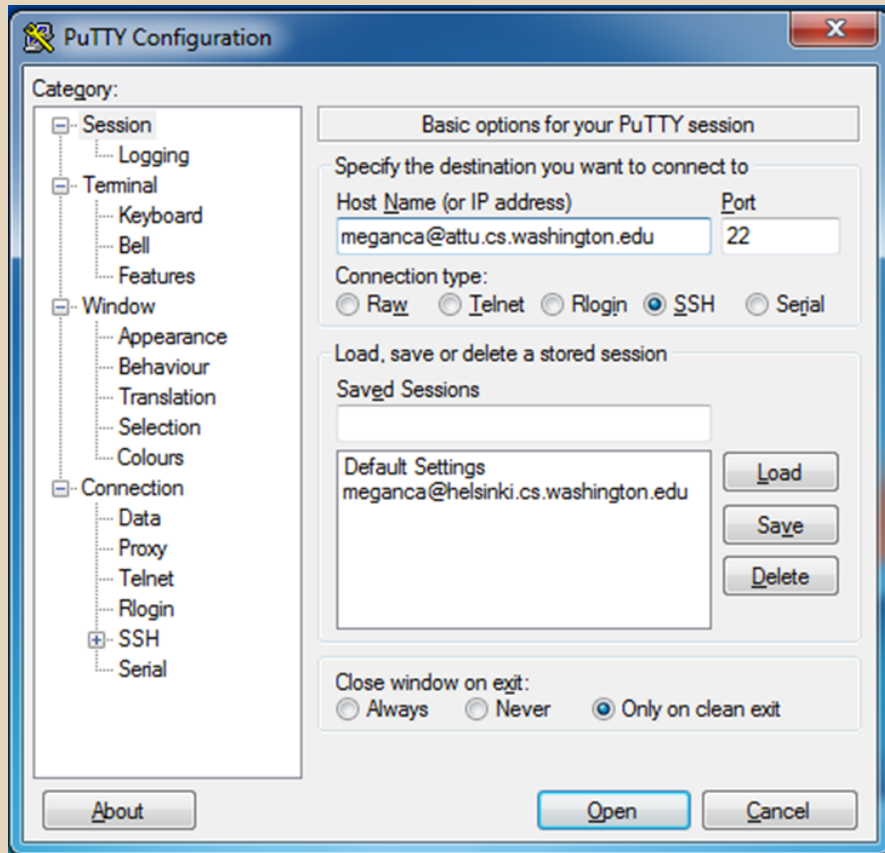
# Turning in HW3

- 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
    - 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

# WHAT IS AN SSH CLIENT?

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

# PuTTY



# TERMINAL (LINUX, MAC)

```
meganca@charmander: ~  
meganca@charmander:~$ ssh meganca@attu.cs.washington.edu  
meganca@attu.cs.washington.edu's password:  
Last login: Wed Sep 24 17:13:13 2014 from c-24-19-57-209.hsd1.wa.comcast.net  
  
Use passwd to change your password.  
Use chsh to change your shell.  
  
Contact support@cs if you need assistance.  
  
Please remove core files when you are done with them, as  
they tend to take up a lot of space on the disk. If  
everyone removes them when they are done debugging, there  
is going to be a lot more disk space to go around.  
Thanks!  
  
[meganca@attu3 ~]$ █
```



# 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

# DEMO

- Tagging your final submission
- SSH into attu
- Running ant validate on attu

# 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:** 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:** A table showing the current state of variables:

Name	Value
this	RatPolyStackTest (id=33)
- Source Editor:** Displays the code for `RatPolyStackTest.java`. The current line is 157, which is highlighted in green:

```
151 ///////////////////////////////////////////////////  
152 /// Duplicate  
153 ///////////////////////////////////////////////////  
154  
155 @Test  
156 public void testDupWithOneVal() {  
157     RatPolyStack stk1 = stack("3");  
158     stk1.dup();  
159     assertStackIs(stk1, "33");  
160     stk1 = stack("123");  
161     stk1.dup();  
162     assertStackIs(stk1, "1123");
```
- Outline View:** Shows a list of methods in the class, with `testDupWithOneVal()` selected:
  - testClear(): void
  - testCtor(): void
  - testDifferentiate(): void
  - testDivMultiElems(): void
  - testDivTwoElems(): void
  - testDupWithMultVal(): void
  - testDupWithOneVal(): void**
  - testDupWithTwoVal(): void
  - testIntegrate(): void

# ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debug session. The top toolbar includes standard development icons. The 'Debug' toolbar is active, showing a 'Quick Access' search bar and tabs for 'Java', 'Debug', 'SVN Repository Exploring', and 'PyDev'. The 'Debug' console on the left lists the current stack of frames, with the current frame being 'ParentRunner\$1.schedule(Runnable) line: 60'. The 'Variables' view on the right shows a table with one entry: 'this' with the value 'RatPolyStackTest (id=33)'. The code editor at the bottom shows the source file 'RatPolyStackTest.java' with line numbers 51 to 62. A green arrow points to line 57, where a breakpoint has been set, indicated by a small green circle in the left margin. A text box is overlaid on the code editor, providing instructions on how to set a breakpoint.

Name	Value
this	RatPolyStackTest (id=33)

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

The screenshot shows the Eclipse IDE interface in Debug mode. At the top, the toolbar contains various icons, with the 'Bug' icon (a bug) highlighted by a green box. A callout box with a tan background and black border points to this icon, containing the text: "Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints." Below the toolbar, the 'Debug Console' panel is visible, showing a list of stack frames for the current thread. The frame for 'RatPolyStackTest (id=33)' is selected. Below this, the 'Expressions' panel shows the value of the selected frame. The main editor window displays the source code for 'RatPolyStackTest.java', with a breakpoint set at line 157: 'RatPolyStack stk1 = stack("3");'. The 'Outline' panel on the right shows a list of methods in the class, including 'testDupWithOneVal()'. The 'Console' panel at the bottom is empty.

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

# ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debugging session. At the top, the toolbar contains several icons for controlling the program's execution, which are highlighted with a green box. These icons include a play button (run), a pause button (suspend), a red square (stop), a magnifying glass (search), a refresh icon, a double arrow (step over), a single arrow (step into), a double arrow (step out), and a double arrow (step return).

Below the toolbar, the **Debug** console is visible, showing a stack trace of the current execution. The stack trace includes 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, 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

To the right of the Debug console is the **Variables** view, which shows the current state of variables in the program. A text box overlaid on this view contains the text: "Controlling your program while debugging is done with these buttons".

At the bottom of the IDE, the **Code Editor** shows the source code for `RatPolyStackTest.java`. A breakpoint is set at line 157, which is highlighted in green. The code snippet is as follows:

```
151 ////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
163 }
```

On the right side of the IDE, the **Outline** view shows a list of methods in the current class, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() : void`, `testDivTwoElems() : void`, `testDupWithMultVal() : void`, `testDupWithOneVal() : void`, `testDupWithTwoVal() : void`, and `testIntegrate() : void`.

# ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar features a green box around the play, pause, and stop icons. The Debug console on the left shows a stack trace of method calls. The Variables view on the right shows a table with the variable 'this' of type 'RatPolyStackTest (id=33)'. The main editor shows the source code of 'RatPolyStackTest.java' with line 157 highlighted. The Outline view on the bottom right lists the methods of the class.

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

Name	Value
this	RatPolyStackTest (id=33)

```
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

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 box). The Debug console on the left shows a stack trace of the current execution. The code editor at the bottom shows the file `RatPolyStackTest.java` with a breakpoint set at line 157, which is highlighted in green. The code at line 157 is `RatPolyStack stk1 = stack("3");`. The variable viewer on the right shows a list of variables, with `testDupWithOneVal()` selected.

**Step Into**

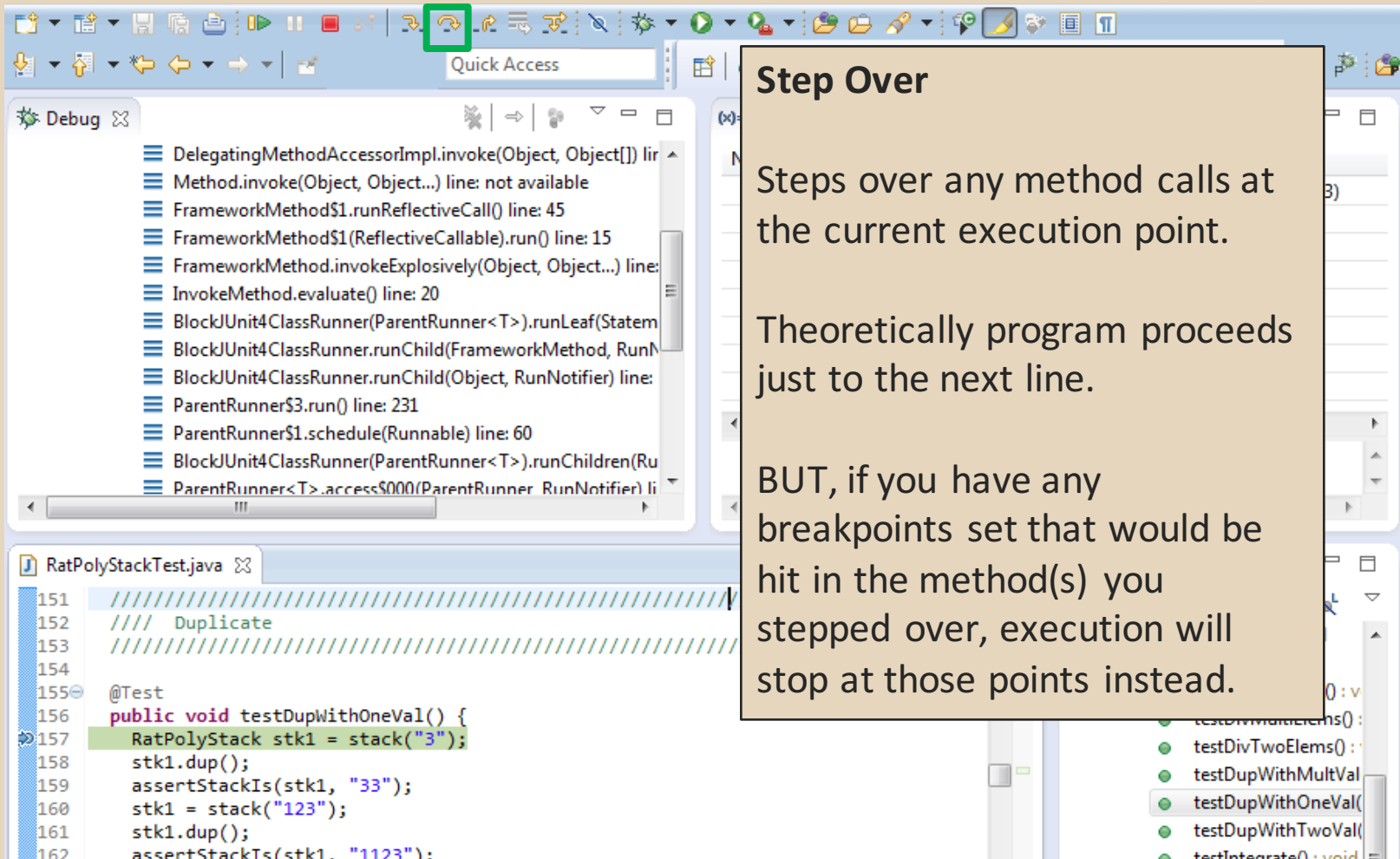
Steps into the method at the current execution point – if possible. If not possible then just proceeds to the next execution point.

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

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

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

# ECLIPSE DEBUGGING



# ECLIPSE DEBUGGING

**Step Out**

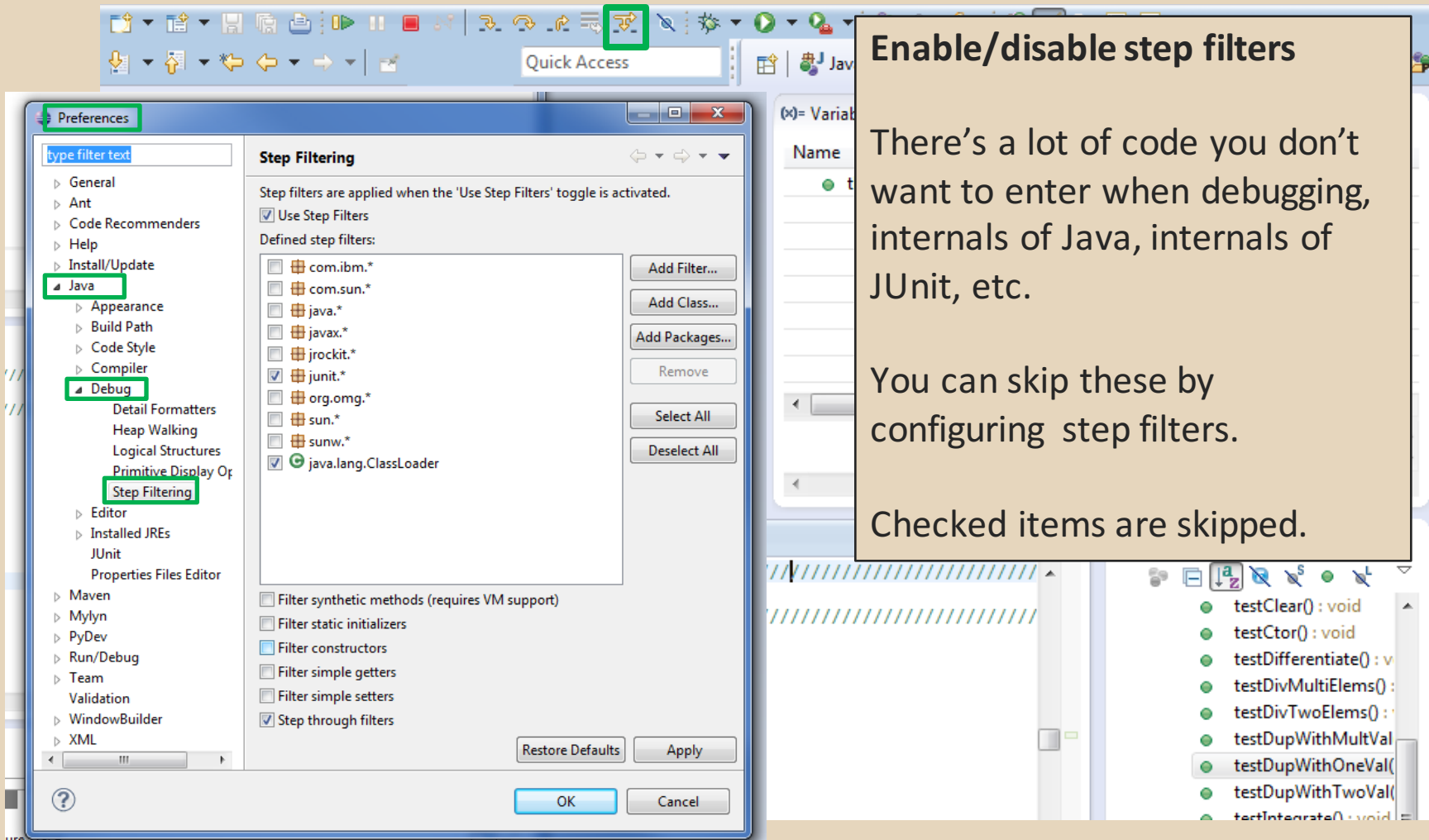
Allows method to finish and brings you up to the point where that method was called.

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. On the left, the 'Debug' console is open, displaying a stack trace. The stack trace is highlighted with a green border. The stack trace lists several methods, including `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate`, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `BlockJUnit4ClassRunner.runChild(FrameworkMethod, Runnable)`, `BlockJUnit4ClassRunner.runChild(Object, Runnable)`, `ParentRunner$3.run`, `ParentRunner$1.schedule(Runnable)`, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Runnable)`, and `ParentRunner<T>.access$000(ParentRunner, Runnable)`. In the center, the 'Variables' pane is partially visible, showing a table with columns for 'Name' and 'Value'. On the right, the 'Stack Trace' pane is open, displaying a list of methods with green circular icons next to them. The 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 line `157 RatPolyStack stk1 = stack("3");` is highlighted in green. The 'Stack Trace' pane on the right contains the following list of methods:

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

## 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 interface during a debug session. The top toolbar includes icons for running, stepping through code, and other debugging actions. The main window is divided into several panes:

- Variables View (top right, highlighted with a green border):** A table showing the current state of variables. The 'Value' tab is active. The variable 't' is expanded to show its fields: 'coeff' and 'expt'. The 'expt' field is highlighted in yellow, indicating it has changed since the last breakpoint. Its value is '5'.

Name	Value
this	RatTermTest (...)
t	RatTerm (id=4...)
coeff	RatNum (id=4...)
expt	5
- Code Editor (bottom left):** Shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted in green, corresponding to the current execution point. The code includes a test method 'testDupWithOneVal()' that creates a 'RatPolyStack' and performs operations on it.

```
151 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1, "33");
160     stk1 = stack("123");
161     stk1.dup();
162     assertStackIs(stk1, "1123");
163 }
```
- Outline View (bottom right):** Lists the methods in the current class, including 'testClear()', 'testCtor()', 'testDifferentiate()', 'testDivMultiElems()', 'testDivTwoElems()', 'testDupWithMultVal()', 'testDupWithOneVal()', 'testDupWithTwoVal()', and 'testIntegrate()'. The 'testDupWithOneVal()' method is highlighted.

# 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 with the following components:

- Variables View (top right):** A table listing variables and their values. The variable `expt` is highlighted in yellow, indicating it has changed since the last breakpoint.
- Main Editor (bottom left):** Shows the source code for `RatPolyStackTest.java`. Line 157 is highlighted, corresponding to the current breakpoint.
- Outline View (bottom right):** Lists the methods in the 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

There's a powerful right-click menu.

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

The screenshot shows the Eclipse IDE interface during a debug session. The main editor displays Java code for a test method. The Variables view on the right shows the current stack frame with variables 'this' and 't'. The 't' variable is expanded to show 'coeff' and 'expt'. A right-click context menu is open over the 'expt' variable, listing various actions such as 'All References...', 'All Instances...', and 'Watch'.

```
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");

```

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

- Select All (Ctrl+A)
- Copy Variables (Ctrl+C)
- Find... (Ctrl+F)
- Change Value...
- All References...
- All Instances... (Ctrl+Shift+N)
- Instance Count...
- New Detail Formatter...
- Open Declared Type
- Open Declared Type Hierarchy
- Instance Breakpoints...
- Watch
- Inspect (Ctrl+Shift+I)

# ECLIPSE DEBUGGING

## Show Logical Structure

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

The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar contains various icons for file operations, running, and debugging. The main window is divided into several panes:

- Left Pane:** A list of stack frames for the current thread. The top frame is `BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line: 5`. Below it are `ParentRunner$3.run() line: 231`, `ParentRunner$1.schedule(Runnable) line: 60`, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Ru`, and `ParentRunner<T>.access$000(ParentRunner RunNotifier) li`.
- Bottom-Left Pane:** The source code editor for `RatPolyStackTest.java`. Line 157 is highlighted: `RatPolyStack stk1 = stack("3");`. Other lines include `stk1.dup();`, `assertStackIs(stk1, "33");`, `stk1 = stack("123");`, `stk1.dup();`, and `assertStackIs(stk1, "1123");`.
- Right Pane:** The **Variables** view. It shows a tree structure of variables. The `stk1` variable is expanded to show its `polys` list. The `polys` list is further expanded to show its `terms` list. The `terms` list is expanded to show its `coeff` field, which is a `RatNum` object with a value of `0`. The `expt` field is also visible with a value of `0`. A green box highlights the `polys` and `terms` sections.
- Bottom-Right Pane:** The **Method List** view, showing a list of methods for the current class, including `testClear(): void`, `testCtor(): void`, `testDifferentiate(): v`, `testDivMultiElems():`, `testDivTwoElems(): :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `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 shows the Eclipse IDE interface. The Breakpoints window is open, displaying a list of breakpoints for the file `RatPolyStackTest.java`. The breakpoints are:

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

The Breakpoints window also shows options for Hit count, Suspend thread, Suspend VM, Conditional, Suspend when 'true', and Suspend when value changes. A dropdown menu is open showing a list of previously entered conditions, with `x == 6` selected.

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");
```

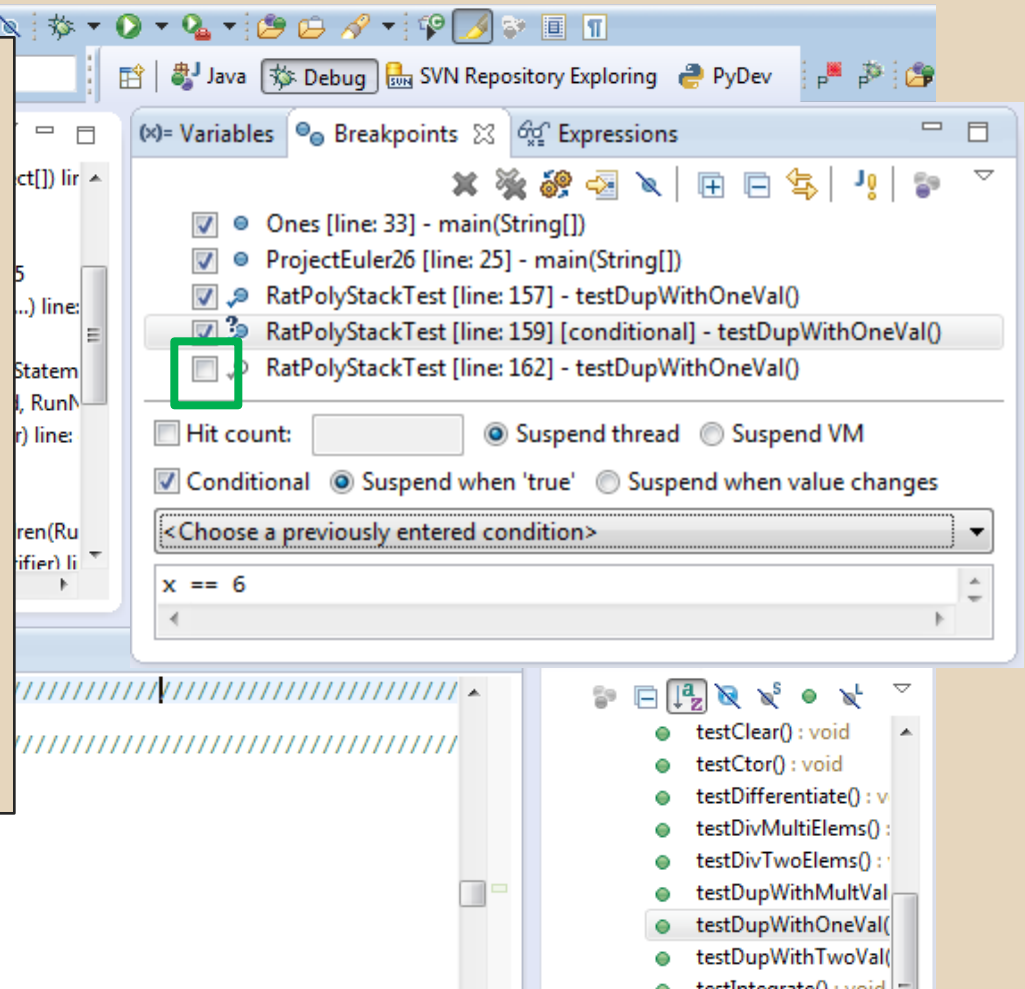
# ECLIPSE DEBUGGING

## 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");
```



# 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 and contains the value "6". The "Suspend when 'true'" option is selected. The background shows the Eclipse IDE interface with a code editor and a variables window.

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

Variables window:

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

Conditional expression: `x == 6`



# 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 with the Breakpoints view open. The Breakpoints view lists several breakpoints, with the one at line 159 of RatPolyStackTest being highlighted. This breakpoint is conditional and is set to suspend when the condition is true. The condition is `x == 6`. The Breakpoints view also shows options for Hit count, Suspend thread, and Suspend VM. The Expressions view is also visible, showing a list of expressions.

Breakpoints:

- 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`

Expressions:

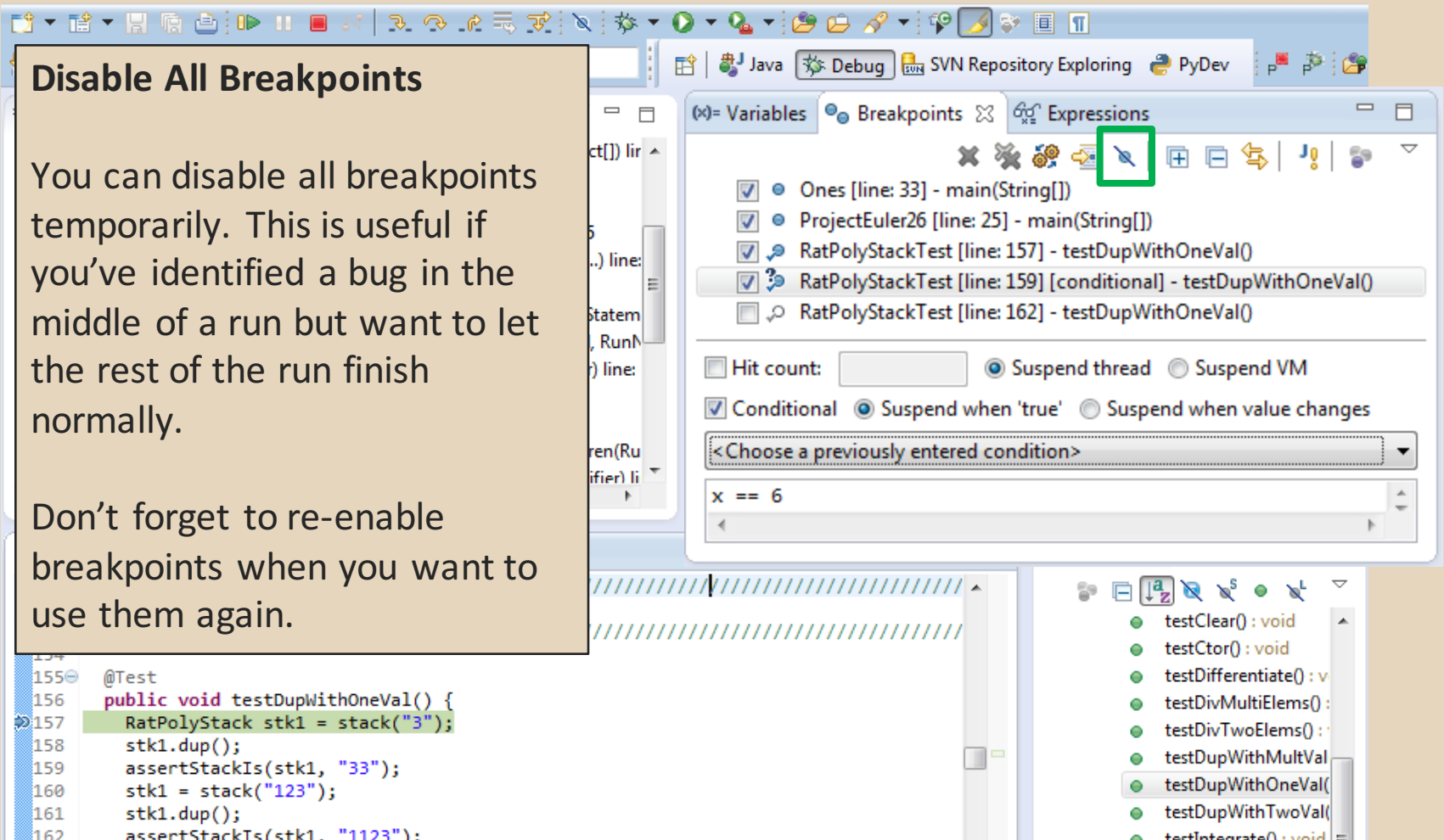
- testClear(): void
- testCtor(): void
- testDifferentiate(): void
- testDivMultiElems(): void
- testDivTwoElems(): void
- testDupWithMultVal(): void
- testDupWithOneVal(): void**
- testDupWithTwoVal(): void
- testIntegrate(): void

# 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 `RatPolyStackTest` with line 157 highlighted:

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

# ECLIPSE DEBUGGING

## Break on Java Exception

Eclipse can break whenever a specific exception is thrown. This can be useful to trace an exception that is being “translated” by library code.

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

The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for file operations, running, and debugging. The main window is divided into several panes:

- Breakpoints:** A list of breakpoints is shown, including:
  - Ones [line: 33] - main(String[])
  - ProjectEuler26 [line: 25] - main(String[])
  - RatPolyStackTest [line: 157] - testDupWithOneVal()
  - RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()** (highlighted)
  - RatPolyStackTest [line: 162] - testDupWithOneVal()
- Conditional:** The 'Conditional' checkbox is checked. The 'Suspend when 'true'' radio button is selected. A dropdown menu shows '<Choose a previously entered condition>'. Below it, the condition `x == 6` is entered.
- Hit count:** A text box for setting the hit count.
- Suspend:** Radio buttons for 'Suspend thread' (selected) and 'Suspend VM'.
- Expressions:** A pane for defining expressions to evaluate at the breakpoint.

The bottom pane shows the source code for `RatPolyStackTest.java`. Line 159 is highlighted, corresponding to the selected breakpoint:

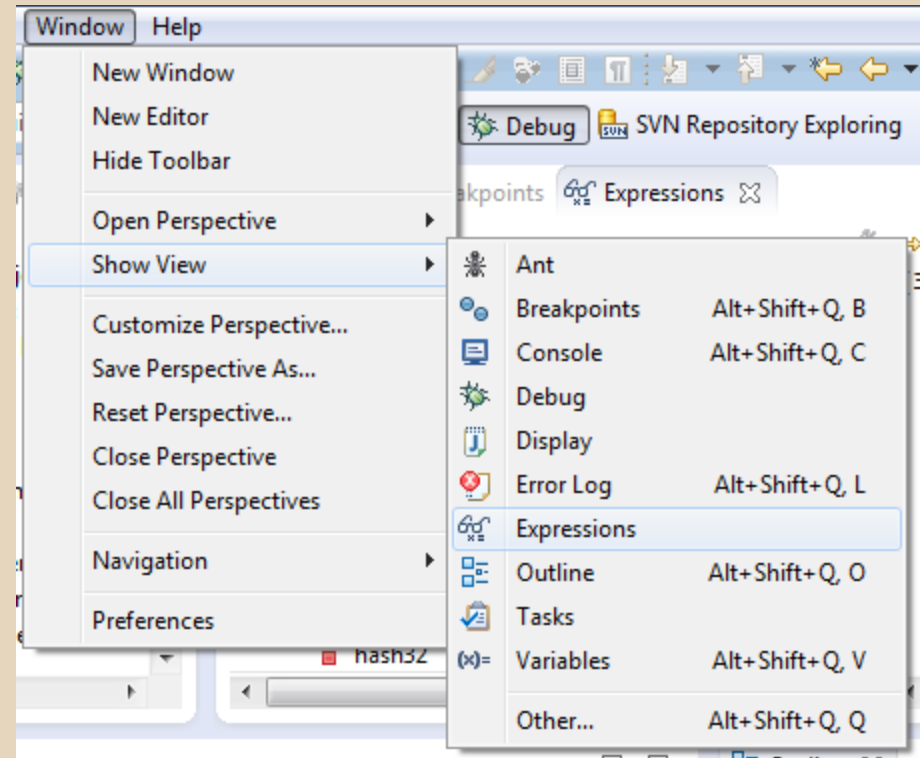
```
155 @Test
156 public void testDupWithOneVal() {
157 RatPolyStack stk1 = stack("3");
158 stk1.dup();
159 assertStackIs(stk1, "33");
160 stk1 = stack("123");
161 stk1.dup();
162 assertStackIs(stk1, "1123");
```

# 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 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 listed are:

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

The background shows the Eclipse IDE with the Java editor displaying code lines 158-162:

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

The bottom right corner shows a list of test methods:

- testClear(): void
- testCtor(): void
- testDifferentiate(): void
- testDivMultiElems(): void
- testDivTwoElems(): void
- testDupWithMultVal(): void
- testDupWithOneVal(): void
- testDupWithTwoVal(): void
- 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 listed 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 on the right side, including testClear(), testCtor(), testDifferentiate(), testDivMultiElems(), testDivTwoElems(), testDupWithMultVal(), testDupWithOneVal(), testDupWithTwoVal(), and testIntegrate().

The background shows the source code of RatPolyStackTest.java, with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The stack trace on the left indicates the current execution point is at `ParentRunner$3.run() line: 231`.

# 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