

# SECTION 2:

## HW3 Setup

[cse331-staff@cs.washington.edu](mailto: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

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

# DEVELOPER TOOLS

- Remote access
- Eclipse and Java versions
- Version Control

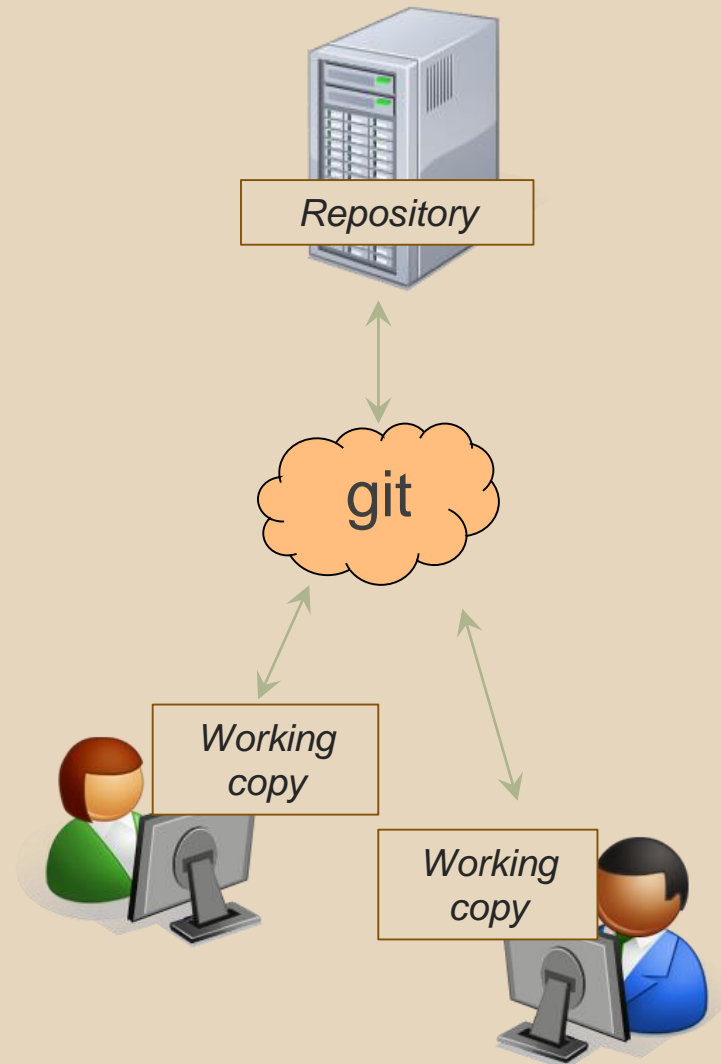
# **VERSION CONTROL**

# 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

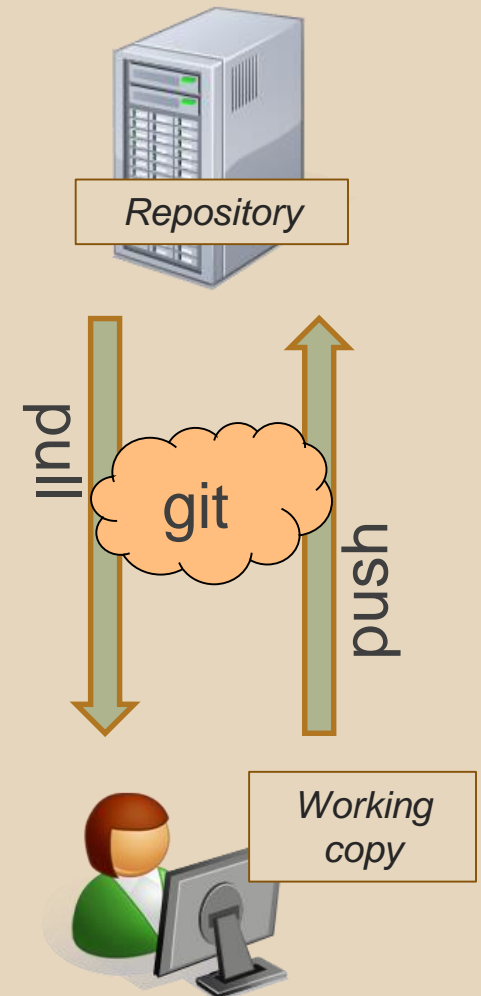
- Can create the repository anywhere
  - Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection
- But, usually you want the repository to be robust:
  - On a computer that's up and running 24/7
    - Everyone always has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



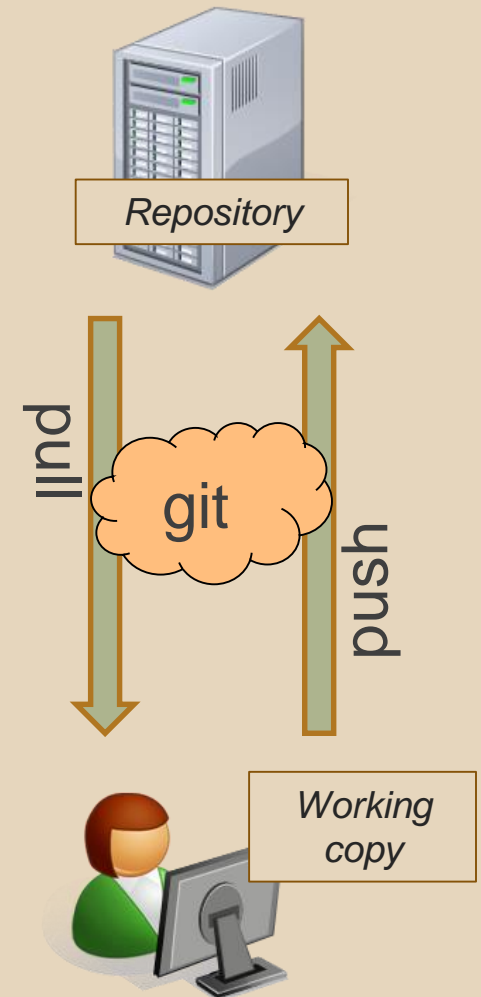


# VERSION CONTROL

## 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

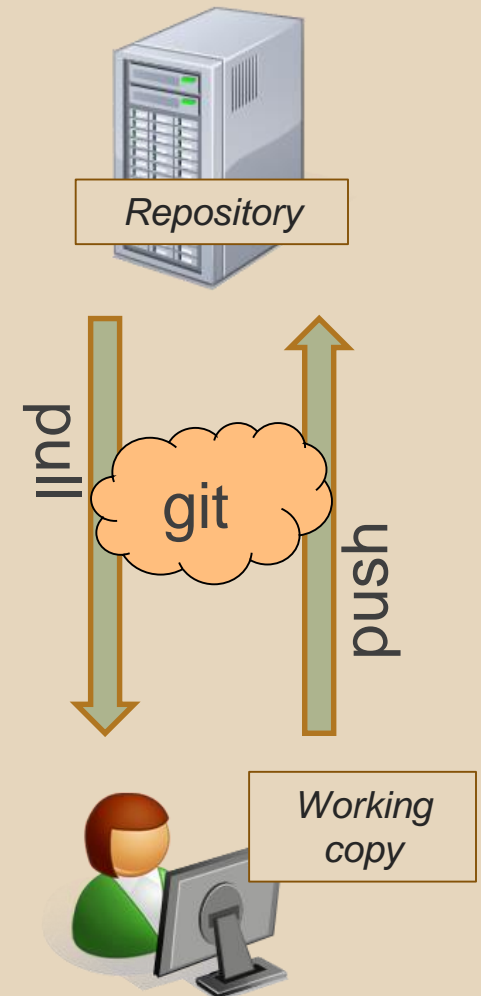


# VERSION CONTROL

## COMMON ACTIONS (CONT.)

Other common commands:

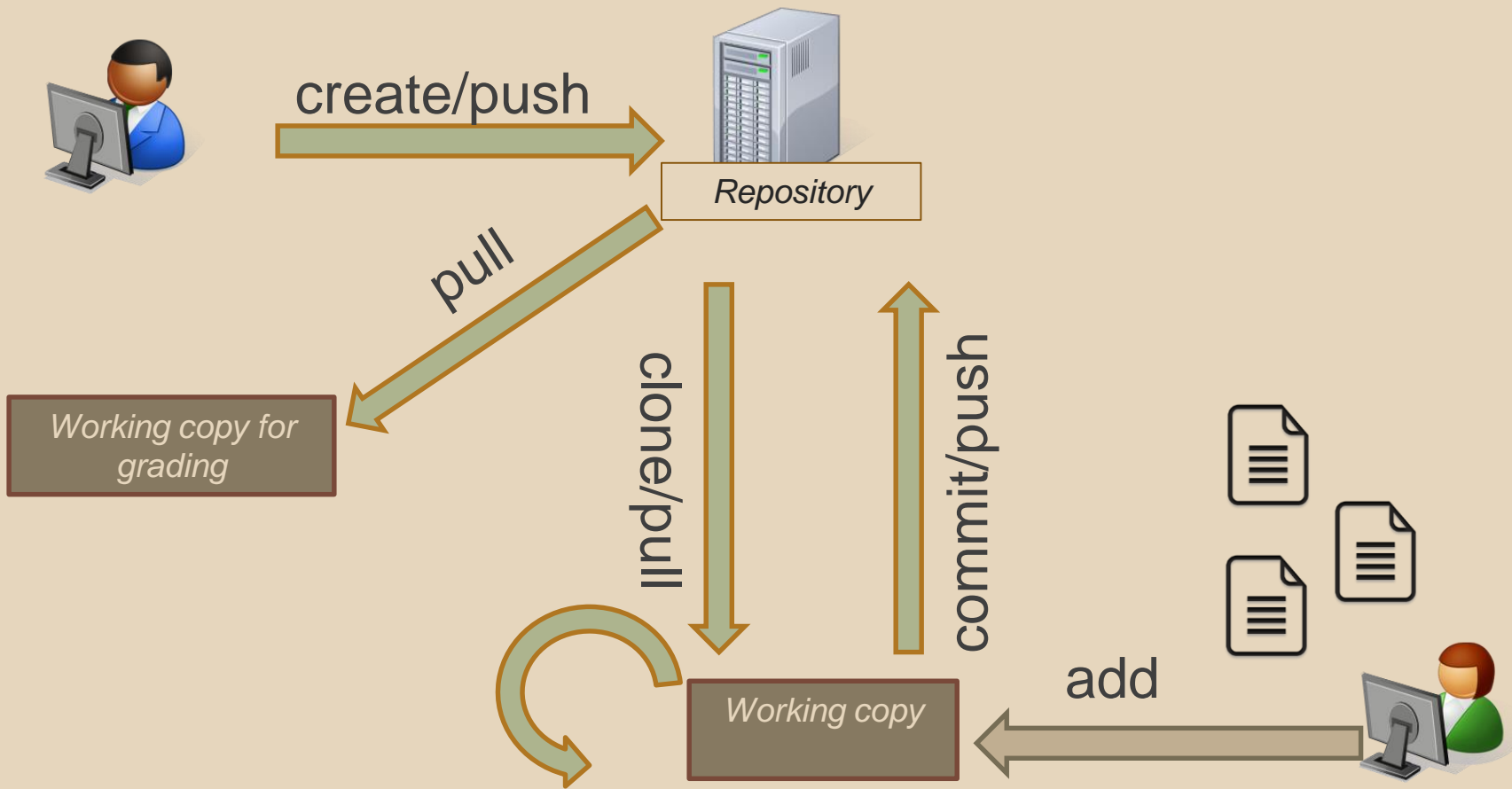
- **add, rm**
  - add or delete a file in the working copy
  - just putting a new file in your working copy does not add it to the repo!
  - still need to commit to make permanent



# 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
  - 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

# 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`  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/cse331/16au/tools/WorkingAtHome.html#Step\\_1](http://courses.cs.washington.edu/courses/cse331/16au/tools/WorkingAtHome.html#Step_1)



# 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!
  - Do this regularly for backup even before you're done!
- 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/cse331/16au/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

# 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

# Turning in HW3

- [Instructions](#)
- 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

# 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 (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:** 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 the current state of variables:

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 of method calls, with the current execution point at `ParentRunner$1.schedule(Runnable) line: 60`.
- Variables View:** Displays the current state of variables, showing `this` as `RatPolyStackTest (id=33)`.
- Code Editor:** Shows the source code for `RatPolyStackTest.java`. A breakpoint is set on line 57, indicated by a red dot in the left margin. A green vertical bar highlights the area around the breakpoint.

A text box overlaid on the code editor provides 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

The image shows the Eclipse IDE interface in Debug mode. The top toolbar contains various icons, with the Bug icon highlighted by a green box. A callout box with a black border and white background contains the text: "Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints." Below the toolbar, the Run and Debug console shows a stack of frames, including `BlockJUnit4ClassRunner.runLeaf`, `ParentRunner.run`, and `ParentRunner.schedule`. The central editor displays the source code for `RatPolyStackTest.java`, with line 157 highlighted: `RatPolyStack stk1 = stack("3");`. The right sidebar shows the Outline view with a list of test methods: `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal`, `testDupWithOneVal`, `testDupWithTwoVal`, and `testIntegrate()`.

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");
```

# ECLIPSE DEBUGGING

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar contains several icons, with a green box highlighting the Run, Break, and Step-Through buttons. Below the toolbar, the 'Debug' console shows a stack trace of method calls. The main editor window displays the source code of 'RatPolyStackTest.java', with line 157 highlighted in green. The 'Outline' view on the right lists the methods of the class, with 'testDupWithOneVal()' selected. A text box with a black border and white background is overlaid on the right side of the image, containing the text: 'Controlling your program while debugging is done with these buttons'.

Controlling your program while debugging is done with these buttons

```
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 displays the Eclipse IDE interface during a debugging session. The top toolbar contains several icons, with the Play (Run), Pause, and Stop (Terminate) icons highlighted by a green box. A text box with a black border is overlaid on the Variables view, containing the text: "Play, pause, stop work just like you'd expect".

The Debug console (top left) shows a stack trace of the current execution, listing various framework and class runner methods. The Variables view (top right) shows a table with one variable named "this" of type "RatPolyStackTest (id=33)".

The code editor (bottom left) shows the source code for "RatPolyStackTest.java". The following code is visible:

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

The Outline view (bottom right) shows a list of test methods, with "testDupWithOneVal()" selected and highlighted in green.

# 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 square with a white arrow pointing down). The 'Debug' console on the left shows a stack trace of method calls. The main editor window displays the source code of `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");
}
```

Stack Trace (Debug Console):

- 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

Test Results (Bottom Right):

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



# ECLIPSE DEBUGGING

The screenshot shows the Eclipse IDE interface. The top toolbar has a green box around the 'Step Over' icon (a right-pointing arrow with a horizontal line above it). Below the toolbar, the 'Debug' console displays a stack trace of method calls. The bottom editor shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted in green, corresponding to the 'Step Over' action. The code on line 157 is `RatPolyStack stk1 = stack("3");`. The right-hand side of the IDE shows a list of test methods, with 'testDupWithOneVal()' selected.

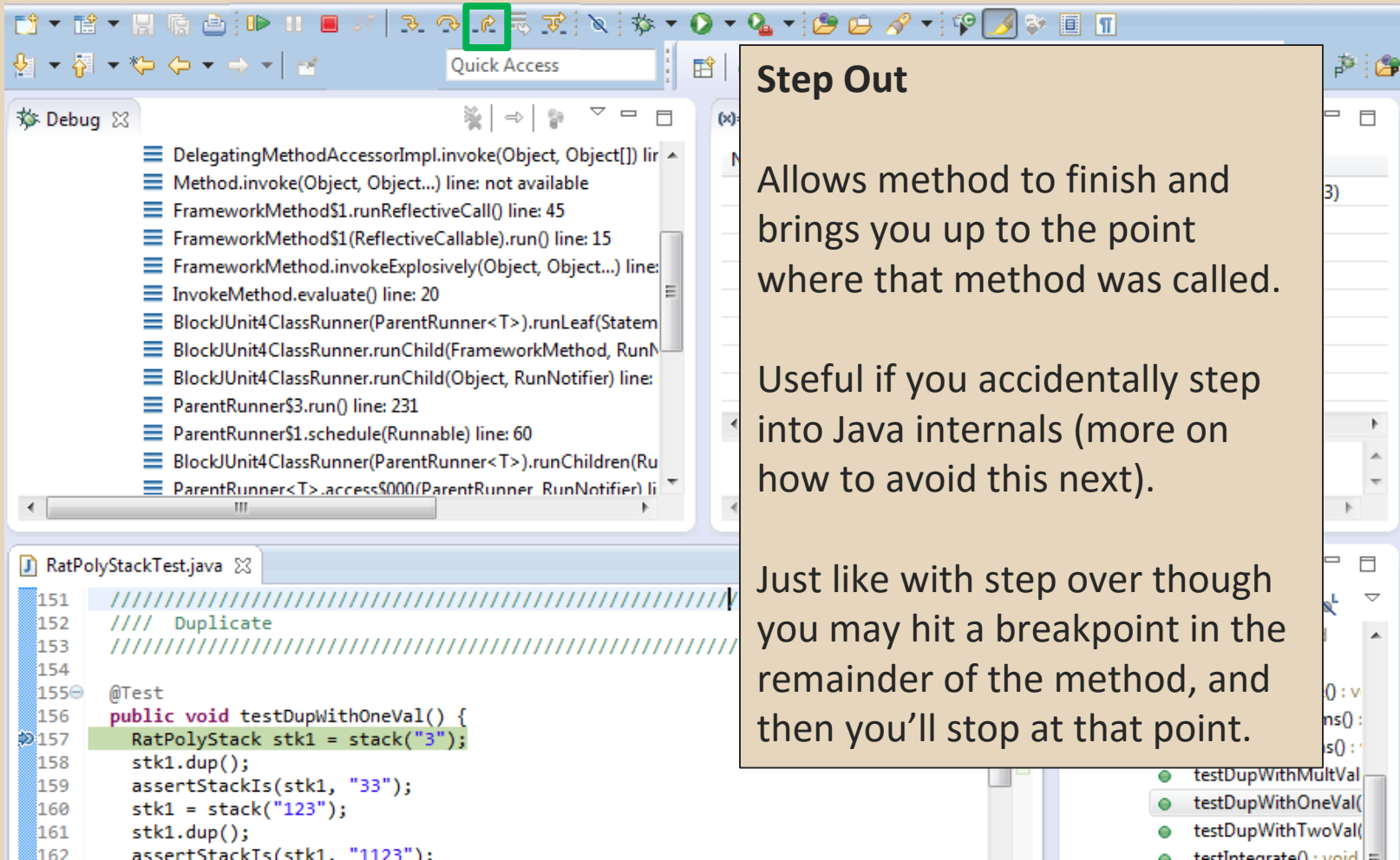
**Step Over**

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.

# ECLIPSE DEBUGGING



**Step Out**

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

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

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

```
Debug Console:  
DelegatingMethodAccessorImpl.invoke(Object, Object[]) lir  
Method.invoke(Object, Object...) line: not available  
FrameworkMethod$1.runReflectiveCall() line: 45  
FrameworkMethod$1(ReflectiveCallable).run() line: 15  
FrameworkMethod.invokeExplosively(Object, Object...) line:  
InvokeMethod.evaluate() line: 20  
BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf(Statem  
BlockJUnit4ClassRunner.runChild(FrameworkMethod, RunN  
BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line:  
ParentRunner$3.run() line: 231  
ParentRunner$1.schedule(Runnable) line: 60  
BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Ru  
ParentRunner<T>.access$000(ParentRunner, RunNotifier) li  
  
RatPolyStackTest.java:  
151 ///////////////////////////////////////////////////////////////////  
152 /// Duplicate  
153 ///////////////////////////////////////////////////////////////////  
154  
155 @Test  
156 public void testDupWithOneVal() {  
157 RatPolyStack stk1 = stack("3");  
158 stk1.dup();  
159 assertStackIs(stk1, "33");  
160 stk1 = stack("123");  
161 stk1.dup();  
162 assertStackIs(stk1, "1123");
```

# ECLIPSE DEBUGGING

The screenshot shows the Eclipse IDE interface. The top toolbar has a green box around the 'Step Through' icon. The 'Preferences' dialog is open, with 'Java' selected in the left sidebar, and 'Debug' > 'Step Filtering' selected. The 'Step Filtering' panel shows a list of defined step filters with checkboxes. The 'java.lang.ClassLoader' filter is checked and highlighted with a green circle. 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 list of test methods is visible, each with a green circle icon to its left, indicating they are skipped.

**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 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, List<Runnable>, RunNotifier) 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

On the right, the 'Variables' pane shows a table with a 'Name' column and a single entry 't'.

At the bottom, the 'RatPolyStackTest.java' 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");
}
```

The line `stk1 = stack("123");` is highlighted in green. The right sidebar shows a list of test methods, with `testDupWithOneVal()` 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.

# ECLIPSE DEBUGGING

## Variables Window

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

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

Name	Value
this	RatPolyStackTest (id=33)

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

# ECLIPSE DEBUGGING

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

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

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

- Variables Window (top right):** A table showing the current state of variables. The 'Value' tab is active. The variable 't' is expanded to show its fields: 'coeff' and 'expt'. The 'expt' field has a yellow background, indicating it has changed since the last breakpoint. The value of 'expt' is 5. 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 several `testDupWithOneVal()` method calls.
- Outline (bottom right):** Lists the methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal`, `testDupWithOneVal`, `testDupWithTwoVal`, and `testIntegrate()`.

# ECLIPSE DEBUGGING

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

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

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

- Variables Window (top right):** A table showing the current state of variables. The 'Value' tab is active, and the variable 'expt' is highlighted in yellow, indicating it has changed since the last breakpoint. The table is as follows:

Name	Value
▶ this	RatTermTest (
▶ t	RatTerm (id=4
▶ coeff	RatNum (id=4
▶ expt	5
- Code Editor (bottom left):** Shows the source code for 'RatPolyStackTest.java'. Line 157 is highlighted in green, corresponding to the breakpoint. The code snippet is:

```
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 (bottom right):** Lists the methods of the class, with 'testDupWithOneVal()' selected.

# 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 top toolbar includes icons for file operations, running, and debugging. The main editor displays Java code for a class named `Runner.class`. The code includes a `@Test` annotation and a `testDupWithOneVal()` method. Line 157 is highlighted, showing the assignment `RatPolyStack stk1 = stack("3");`. The `Variables` view on the right shows the current stack frame, with the variable `stk1` expanded to show its value `coeff` and `expt`. A right-click context menu is open over the `expt` variable, 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, indicating the user's intent to view all instances of the variable's class.

```
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");

```

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

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



# ECLIPSE DEBUGGING

## Show Logical Structure

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

```
BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line:  
ParentRunner$3.run() line: 231  
ParentRunner$1.schedule(Runnable) line: 60  
BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Ru  
ParentRunner<T>.access$000(ParentRunner RunNotifier) li
```

RatPolyStackTest.java

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

Variables Breakpoints Expressions

Name	Value
this	RatPolyStackTest (id=33)
stk1	RatPolyStack (id=44)
polys	Stack<E> (id=49)
[0]	RatPoly (id=719)
terms	ArrayList<E> (id=728)
[0]	RatTerm (id=731)
coeff	RatNum (id=733)
expt	0

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

# ECLIPSE DEBUGGING

## Breakpoints Window

Shows all existing breakpoints in the code, along with their conditions and a variety of options.

Double clicking a breakpoint will take you to its spot in the code.

The screenshot displays the Eclipse IDE interface. The 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. The Conditional option is checked, and the condition is set to `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

## 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 interface during a debugging session. The main window shows a code editor with the following Java code snippet:

```
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 'Hit count' dialog box is open, showing the following configuration:

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

The background shows the Eclipse IDE with the 'Debug' toolbar, the 'Variables' tab, and a list of test methods in the right-hand pane.

# 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. The main editor displays Java code with line numbers 159 to 162. The 'Breakpoints' view is open, showing a list of breakpoints. The selected breakpoint is for 'RatPolyStackTest [line: 159] - testDupWithOneVal()' with a conditional expression. The configuration for this breakpoint is highlighted with a green box, showing the 'Conditional' checkbox checked, the 'Suspend when 'true'' radio button selected, and the condition 'x == 6' entered in the text field. The 'Hit count' is set to 1. The 'Suspend thread' radio button is also selected. The 'Expressions' view is also visible, showing a list of expressions.

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

Breakpoints:

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

Configuration for selected breakpoint:

- Conditional  Suspend when 'true'  Suspend when value changes
- <Choose a previously entered condition>
- x == 6
- Hit count: 1
- Suspend thread  Suspend VM

Expressions:

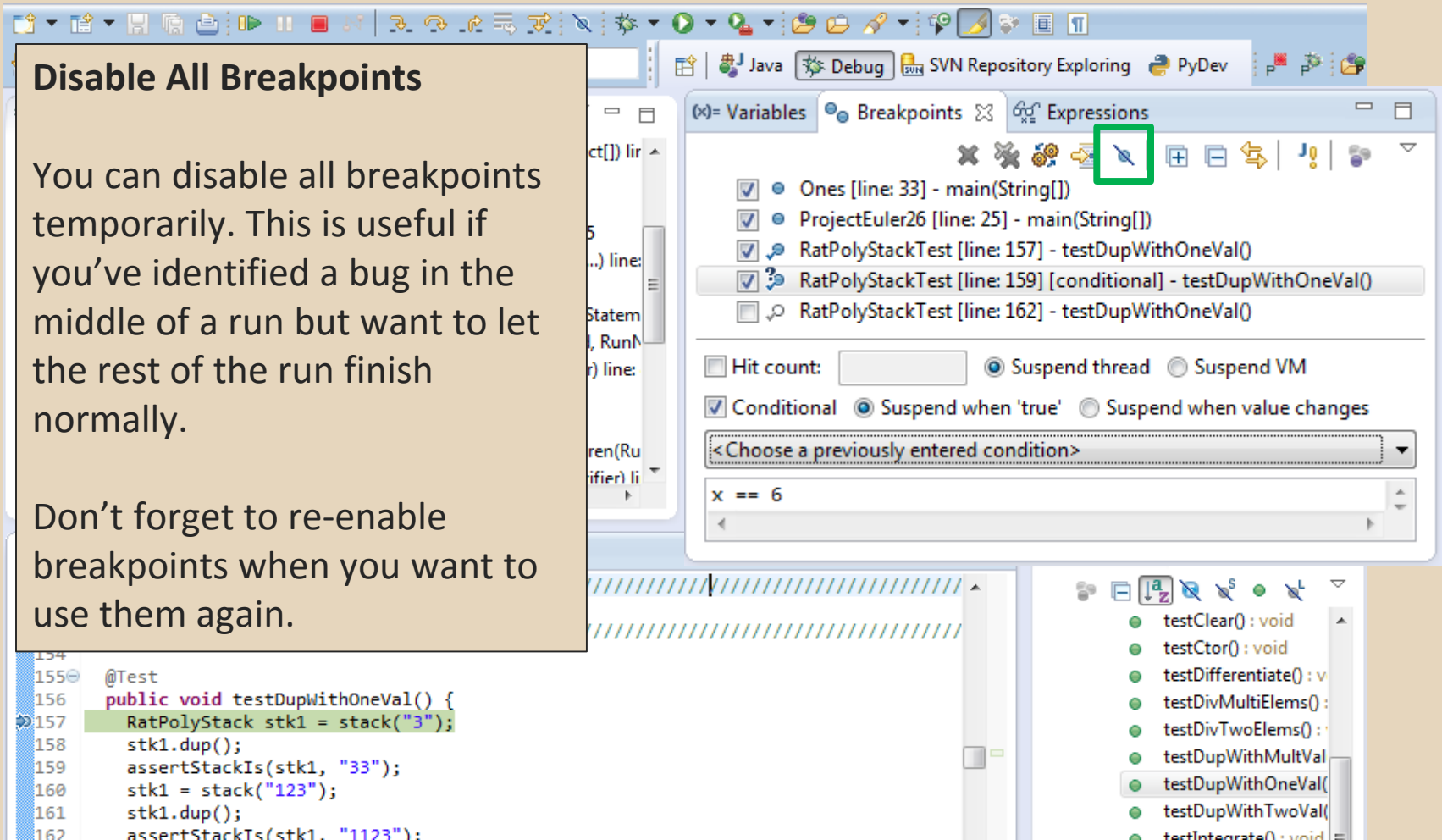
- testClear(): void
- testCtor(): void
- testDifferentiate(): v
- testDivMultiElems(): :
- testDivTwoElems(): :
- testDupWithMultVal
- testDupWithOneVal()**
- testDupWithTwoVal()
- testIntegrate(): void

# 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 green box highlights the 'Disable All Breakpoints' icon (a crossed-out pencil) in the toolbar of the Breakpoints view. The list of breakpoints includes:

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

Below the list, the 'Hit count' is set to 0, and the 'Suspend thread' option is selected. The 'Conditional' checkbox is checked, and the 'Suspend when 'true'' option is selected. The condition field contains the expression `x == 6`.

The 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.

```
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 main editor displays the source code for `RatPolyStackTest.java`. Line 157 is highlighted, showing the code: `RatPolyStack stk1 = stack("3");`. The Breakpoints view on the right is open, showing a list of breakpoints. The breakpoint for `RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()` is selected and highlighted. The configuration for this breakpoint is shown below the list:  Hit count:   Suspend thread  Suspend VM  Conditional  Suspend when 'true'  Suspend when value changes. A dropdown menu is open, showing the option `<Choose a previously entered condition>`. The expression field contains `x == 6`. The bottom right pane shows a list of methods, including `testDupWithOneVal()`.

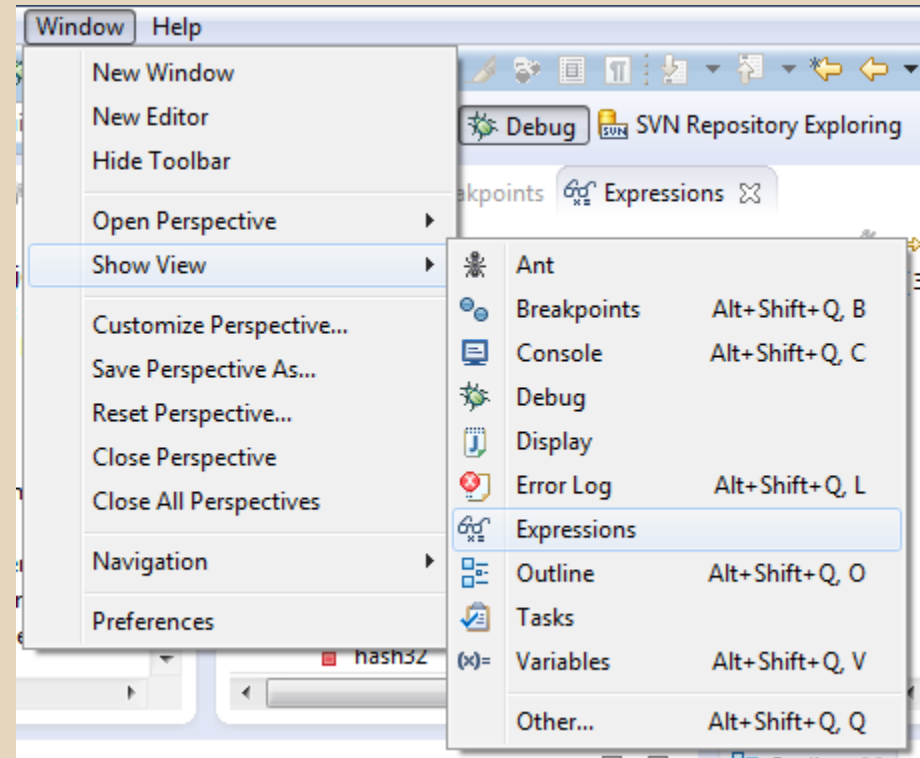


# ECLIPSE DEBUGGING

## 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

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

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

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

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");
```

# ECLIPSE DEBUGGING

## Expressions Window

These persist across projects, so clear out old ones as necessary.

The screenshot displays the Eclipse IDE interface during a debug session. The Expressions window is highlighted with a green border and contains the following data:

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

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

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