SECTION 2:
Loop Reasoning & HW3 Setup

cse331-staff@cs.washington.edu

slides borrowed and adapted from CSE 331 Spring 2018, CSE 391, and many more

Review: Reasoning about loops

- What is a loop invariant?
  - An assertion that always holds at the top of a loop
- Why do we need invariants?
  - Most code is not straight line
  - Most programs aren't guaranteed to terminate
  - Therefore: We need invariants to prove the correctness of most programs we can encounter
  - Additionally, invariants help us write correct programs!

Loop Invariants & Hoare Triples

- We can write a Hoare Triple involving a loop
  - {P} while(B) S {Q}
- The three key ingredients for a valid loop Hoare triple are:
  - The Invariant holds initially (precondition implies invariant)
    - P => I
  - Loop body must re-establish the invariant (Inv holds each time we execute)
    - $\{I \wedge B\}$ S {I}
  - Upon exiting the loop (test is false), the invariant must establish post-condition
    - $\{I \wedge !B\} \Rightarrow Q$

Loop Invariants ct.

- We want a goldilocks invariant
  - not too strong – false and cannot be proven
  - not too weak – cannot satisfy our postcondition
- No sure-fire way to find a loop invariant
  - Bad: Coding first and defining the invariant later
  - Good: think of invariant --> code the body --> code the loop condition --> code the initialization
- The common types of problems involving loop invariants include:
  - Given the code, fill in the assertions / invariant
  - Given a proof, find the error(s) in it if it is incorrect
  - Given the invariant, fill in the code

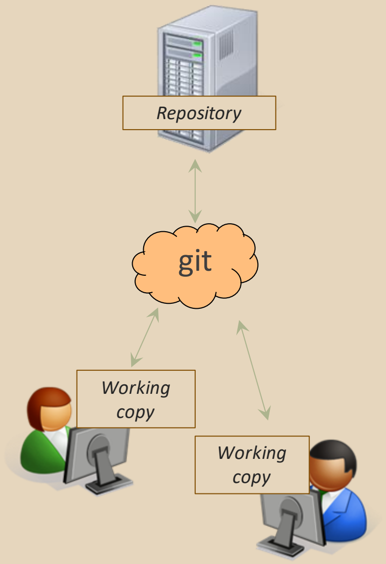
VERSION CONTROL

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)



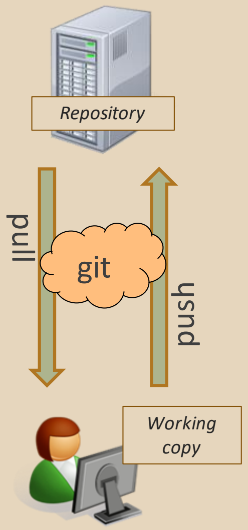
REPOSITORY

- 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:

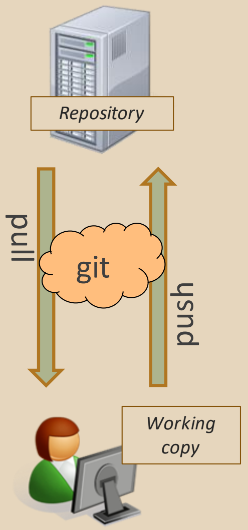
- add / commit / push
  - integrate changes *from* your working copy *into* the repository
- pull
  - integrate changes *into* your working copy *from* the repository



VERSION CONTROL UPDATING FILES

In a bit more detail:

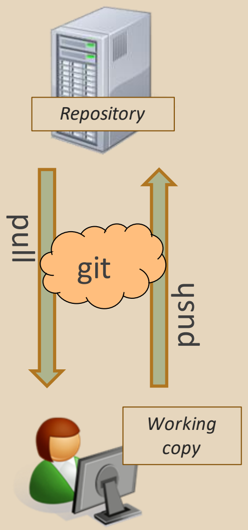
- You make some local changes, test them, etc., then…
- git add – tell git which changed files you want to save in repo
- git commit – save all files you've "add"ed in the local repo copy as an identifiable update
- git push – synchronize with the GitLab repo by pushing local committed changes



VERSION CONTROL COMMON ACTIONS (CONT.)

Other common commands:

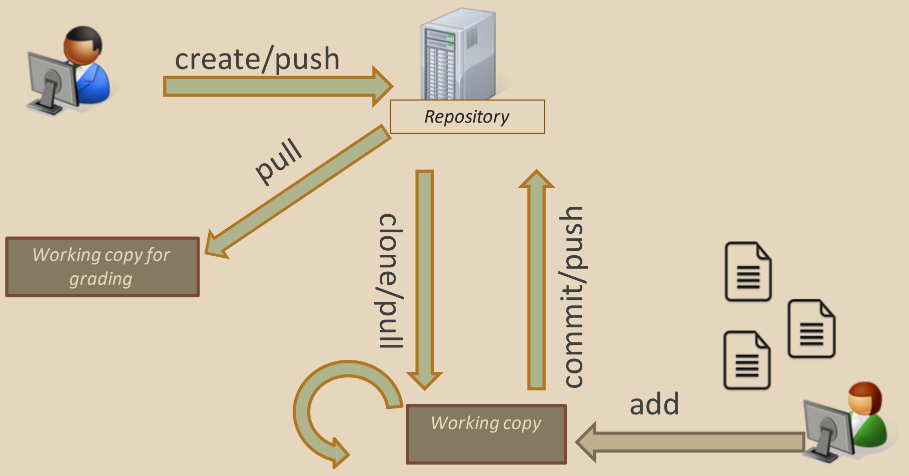
- add, rm
  - add or delete a file in the working copy
  - just putting a new file in your working copy does not add it to the repo!
  - still need to commit to make permanent



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



Your Local Repository

LINKS TO DETAILED SETUP AND USAGE INSTRUCTIONS

- All References
  - https://courses.cs.washington.edu/courses/cse331/18su/#resources
- Machine Setup: Java, Eclipse, SSH
  - https://courses.cs.washington.edu/courses/cse331/18su/machine-setup.html
- Editing, Compiling, Running, and Testing Programs
  - https://courses.cs.washington.edu/courses/cse331/18su/tools/editing-compiling.html
- Eclipse Reference
  - https://courses.cs.washington.edu/courses/cse331/18su/tools/eclipse_reference.html
- Version Control – Git (includes setting up gitlab locally)
  - https://courses.cs.washington.edu/courses/cse331/18su/tools/versioncontrol.html
  - https://gitlab.cs.washington.edu/help/ssh/README.md
- Assignment Submission
  - https://courses.cs.washington.edu/courses/cse331/18su/tools/turnin.html

DEVELOPER TOOLS

- Remote access
- Eclipse and Java versions
- Version Control

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!

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

Live Demo of Setup!

HW 3

- Many small exercises to get you used to version control and tools and a Java refresher
- More information on homework instructions: https://courses.cs.washington.edu/courses/cse331/18su/hws/hw03/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
- 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

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, IntelliJ IDEA

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 ☺
Ctrl + Space	Autocomplete
Ctrl + S	Save (Eclipse does not autosave!)

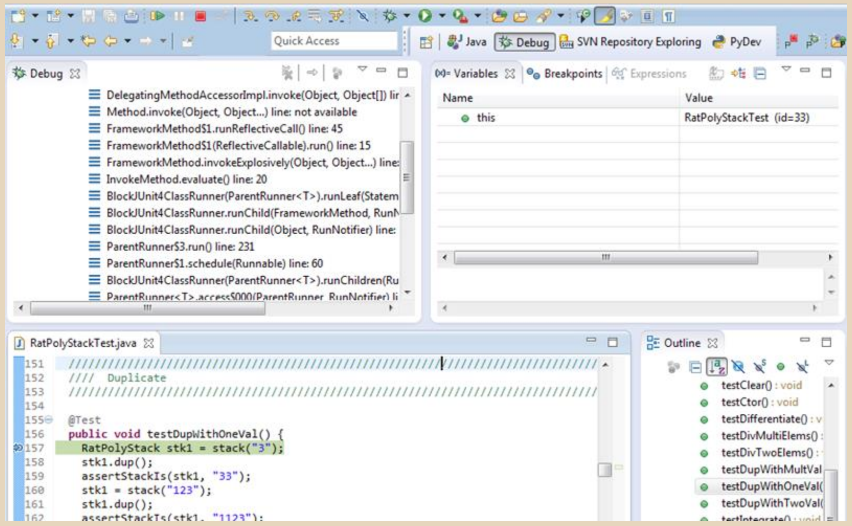
ECLIPSE and Java

- 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 Oxygen, "Eclipse for Java Developers"**
- **Instructions:** https://courses.cs.washington.edu/courses/cse331/18su/machine-setup.html#get-jdk

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



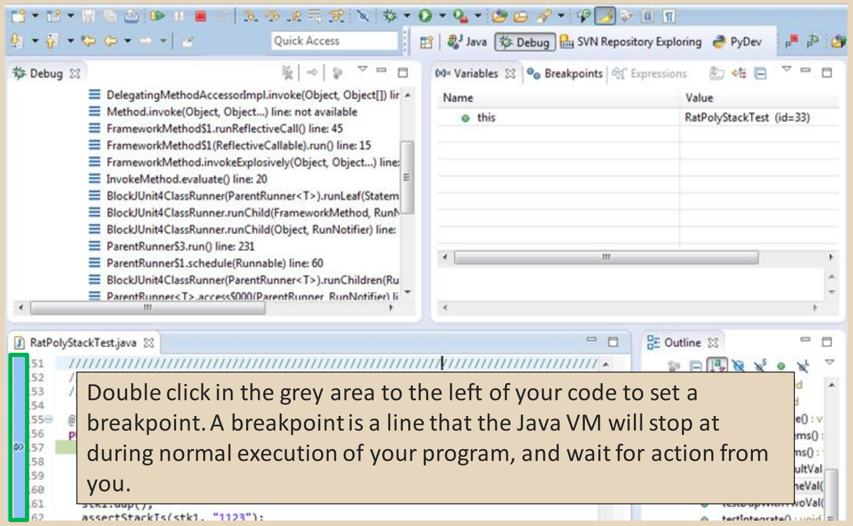
```
DelegatingMethodAccessorImpl.invoke(Object, Object[]) lir
Method.invoke(Object, Object...) line: not available
FrameworkMethod$1.runReflectiveCall() line: 45
FrameworkMethod$1(ReflectiveCallable).run() line: 15
FrameworkMethod.invokeExplosively(Object, Object...) line:
InvokeMethod.evaluate() line: 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
```

Name	Value
this	RatPolyStackTest (id=33)

```java
////////////////////////////////////////////////////////////////////////////////
////  Duplicate
////////////////////////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
  RatPolyStack stk1 = stack("3");
  stk1.dup();
  assertStackIs(stk1, "33");
  stk1 = stack("123");
  stk1.dup();
  assertStackIs(stk1, "1123");
```

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

ECLIPSE DEBUGGING

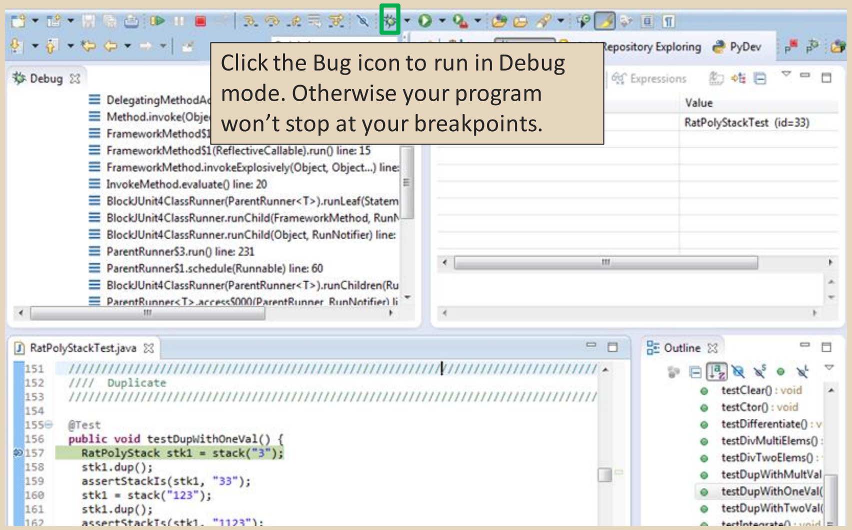


Double click in the grey area to the left of your code to set a breakpoint. A breakpoint is a line that the Java VM will stop at during normal execution of your program, and wait for action from you.

```
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
```

Name	Value
this	RatPolyStackTest (id=33)

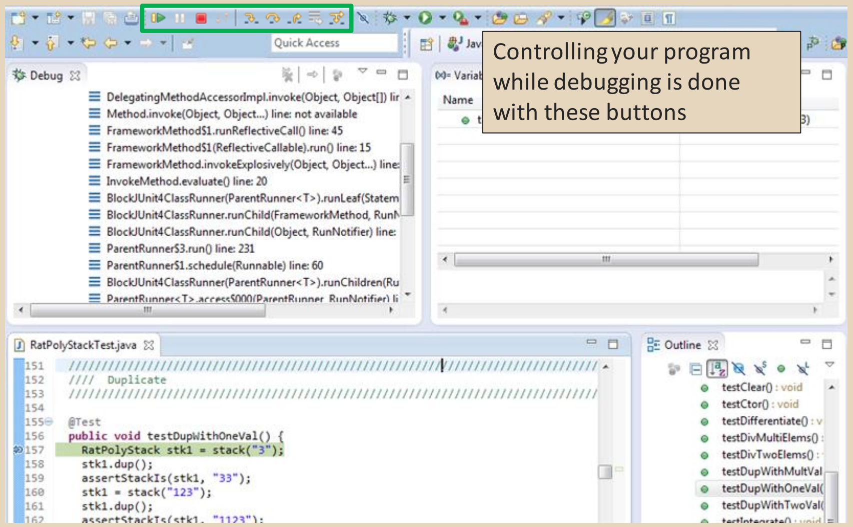
ECLIPSE DEBUGGING



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

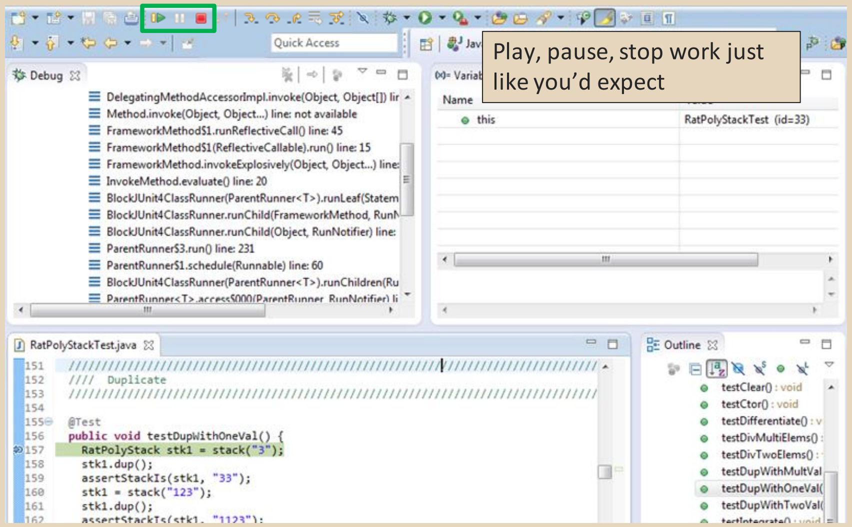
```
////////////////////////////////////////////////////////////////////////////
////  Duplicate
////////////////////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
  RatPolyStack stk1 = stack("3");
  stk1.dup();
  assertStackIs(stk1, "33");
  stk1 = stack("123");
  stk1.dup();
  assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



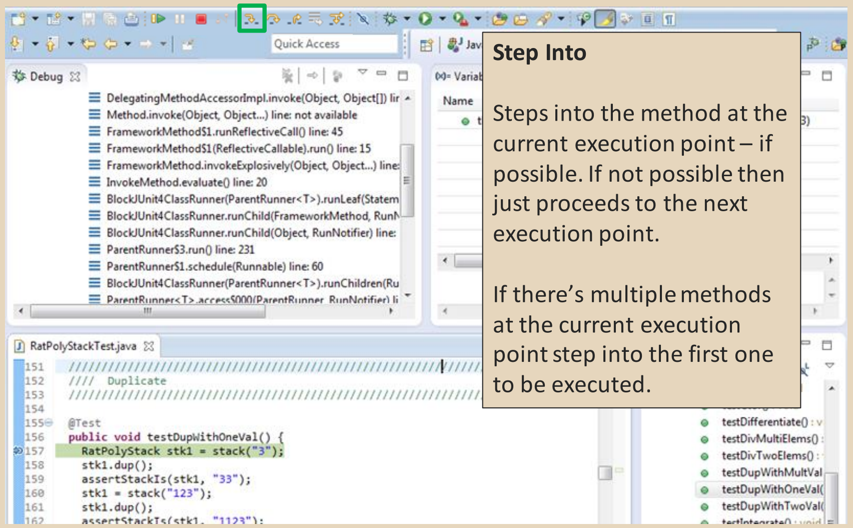
Controlling your program while debugging is done with these buttons

ECLIPSE DEBUGGING



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

ECLIPSE DEBUGGING



```
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

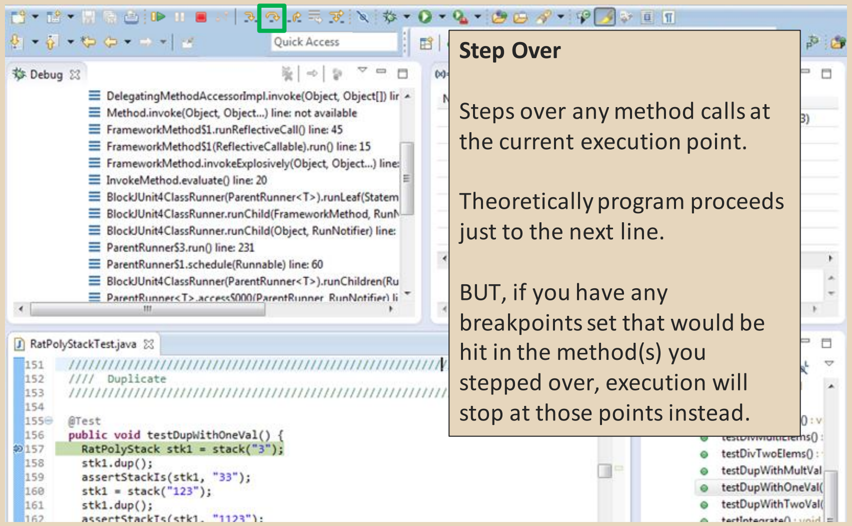
```java
/////////////////////////////////////////////////////////////////
////  Duplicate
/////////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
  RatPolyStack stk1 = stack("3");
  stk1.dup();
  assertStackIs(stk1, "33");
  stk1 = stack("123");
  stk1.dup();
  assertStackIs(stk1, "1123");
```

Step Into

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

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

ECLIPSE DEBUGGING



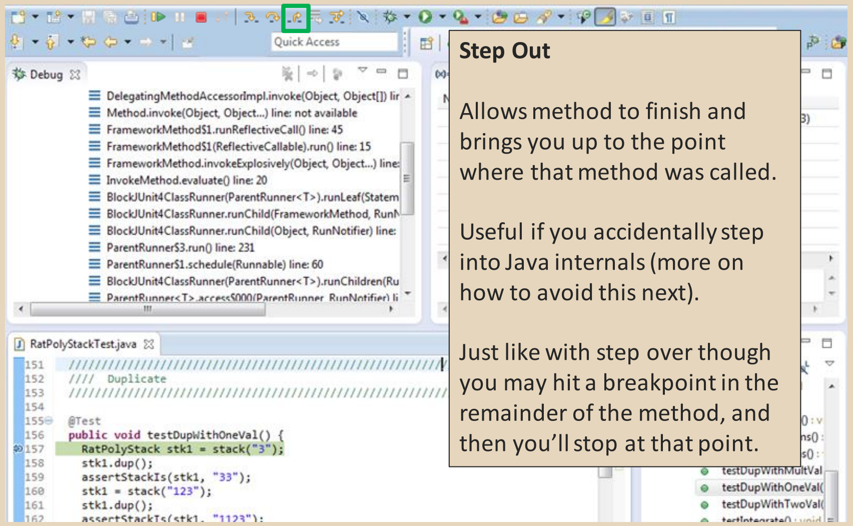
Step Over

Steps over any method calls at the current execution point.

Theoretically program proceeds just to the next line.

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

ECLIPSE DEBUGGING



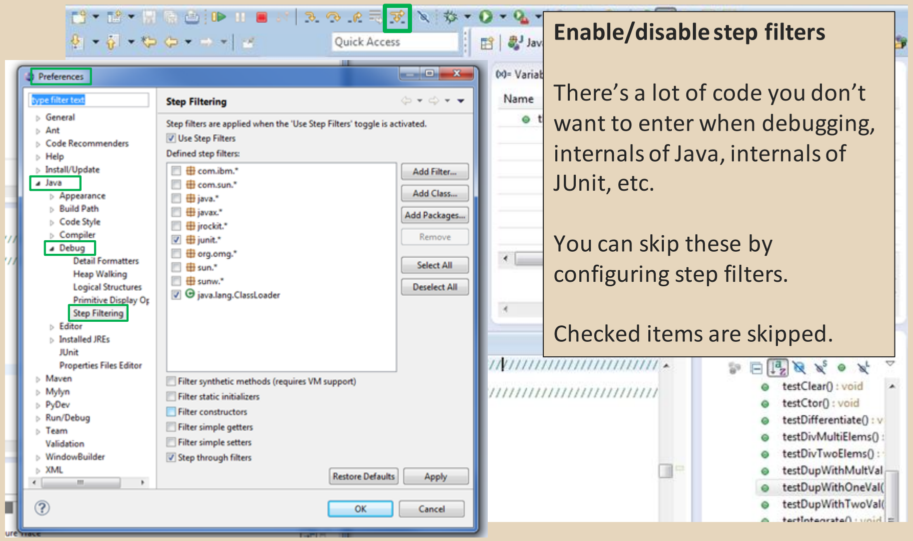
Step Out

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.

ECLIPSE DEBUGGING



Preferences

Step Filtering

Step filters are applied when the 'Use Step Filters' toggle is activated.

[✓] Use Step Filters

Defined step filters:

- [ ] com.ibm.*
- [ ] com.sun.*
- [ ] java.*
- [ ] javax.*
- [ ] jrockit.*
- [✓] junit.*
- [ ] org.omg.*
- [ ] sun.*
- [ ] sunw.*
- [✓] java.lang.ClassLoader

- [ ] Filter synthetic methods (requires VM support)
- [ ] Filter static initializers
- [ ] Filter constructors
- [ ] Filter simple getters
- [ ] Filter simple setters
- [✓] Step through filters

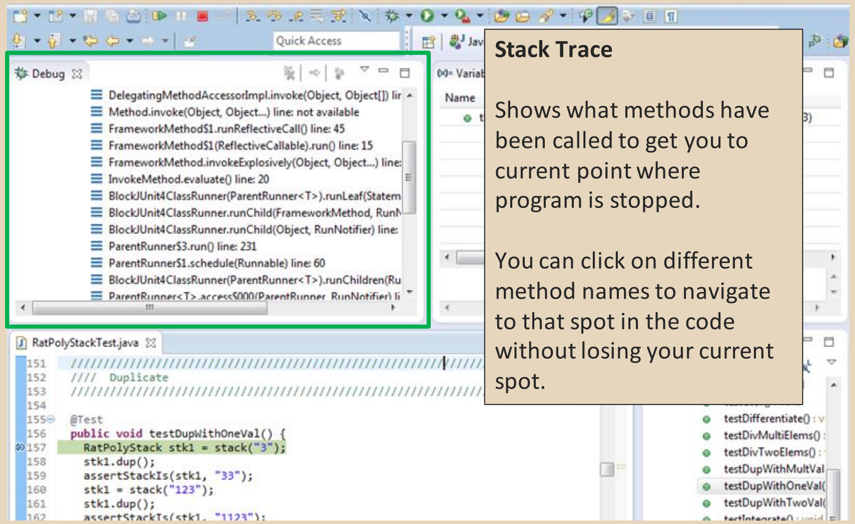
Enable/disable step filters

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



Stack Trace

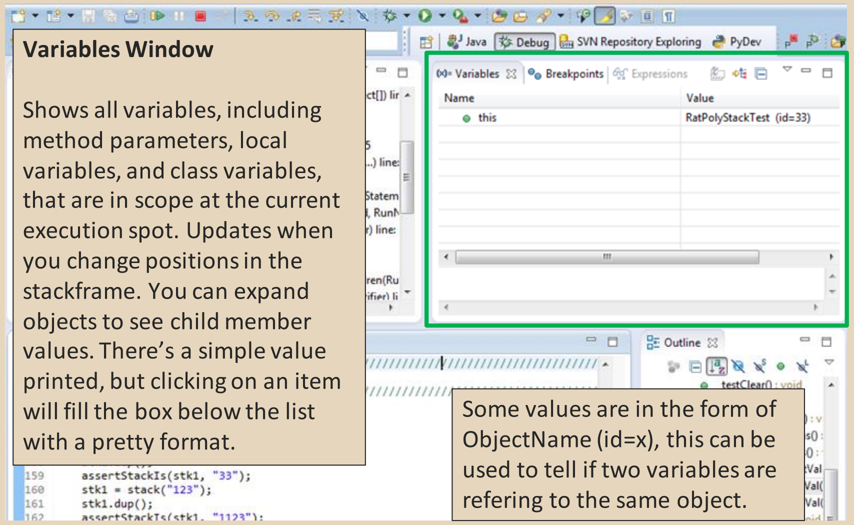
Shows what methods have been called to get you to current point where program is stopped.

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

```
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
```

```java
//////////////////////////////////////////////////////////////
////  Duplicate
//////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



Variables Window

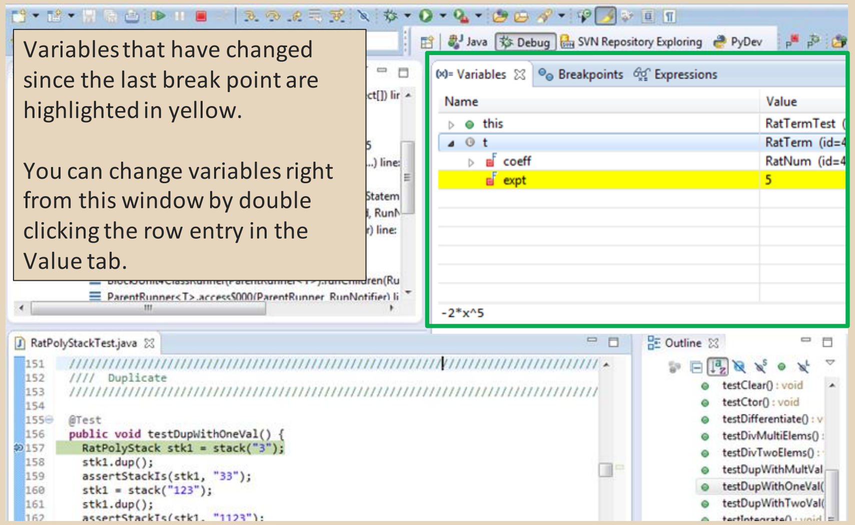
Shows all variables, including method parameters, local variables, and class variables, that are in scope at the current execution spot. Updates when you change positions in the stackframe. You can expand objects to see child member values. There's a simple value printed, but clicking on an item will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

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

Name	Value
this	RatPolyStackTest (id=33)

```java
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



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

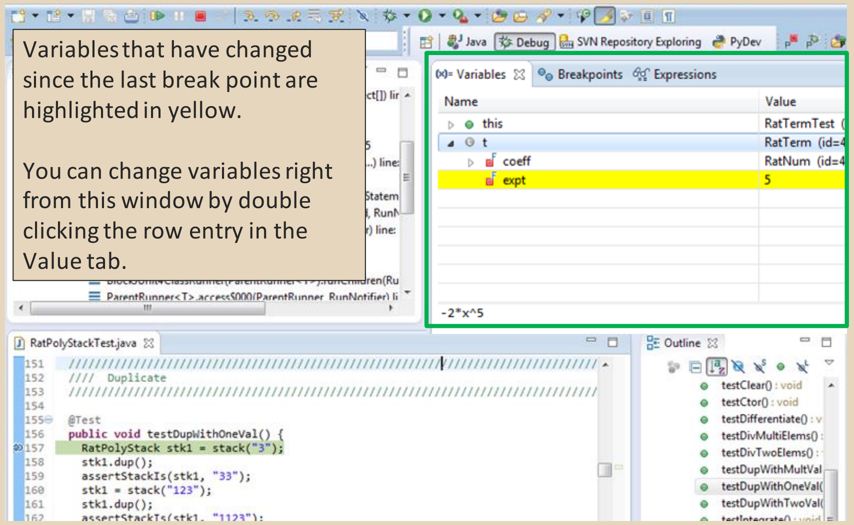
You can change variables right from this window by double clicking the row entry in the Value tab.

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

```
-2*x^5
```

```java
//////////////////////////////////////////////////////////////
////  Duplicate
//////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



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

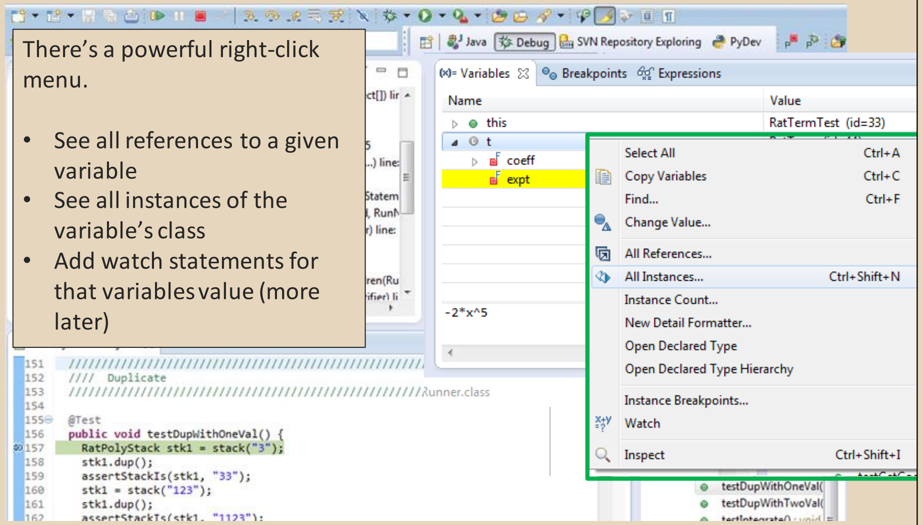
You can change variables right from this window by double clicking the row entry in the Value tab.

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

```
-2*x^5
```

```java
//////////////////////////////////////////////////////////////
////  Duplicate
//////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



There's a powerful right-click menu.

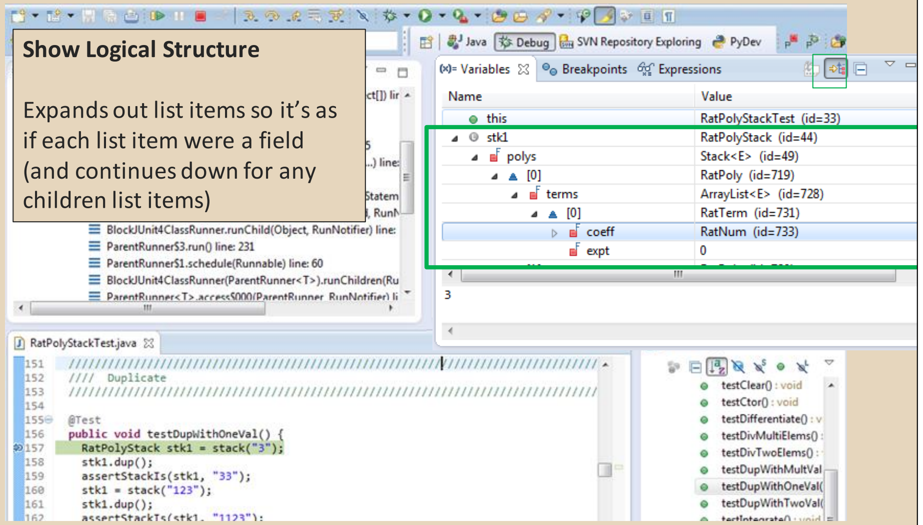
- See all references to a given variable
- See all instances of the variable's class
- Add watch statements for that variables value (more later)

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

-2*x^5

```
//////////////////////////////////////////////////////////////
////  Duplicate
//////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



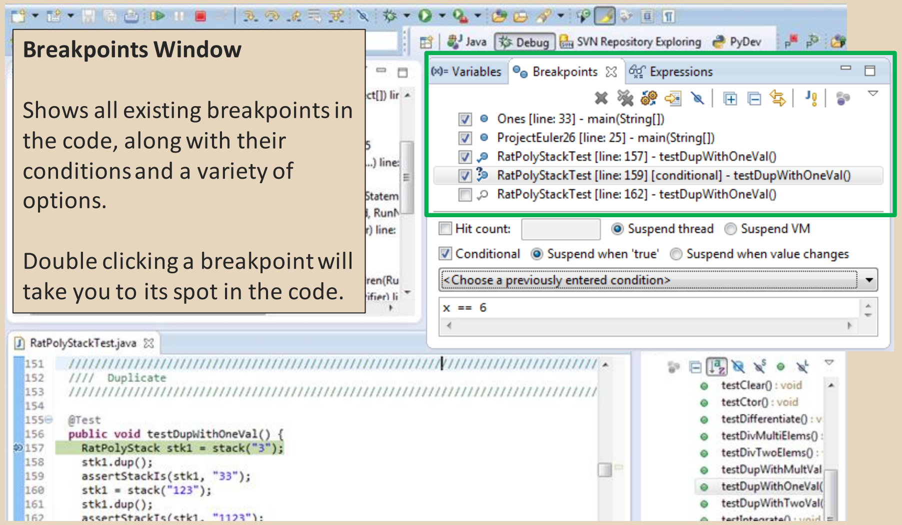
Show Logical Structure

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

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

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

ECLIPSE DEBUGGING



Breakpoints Window

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

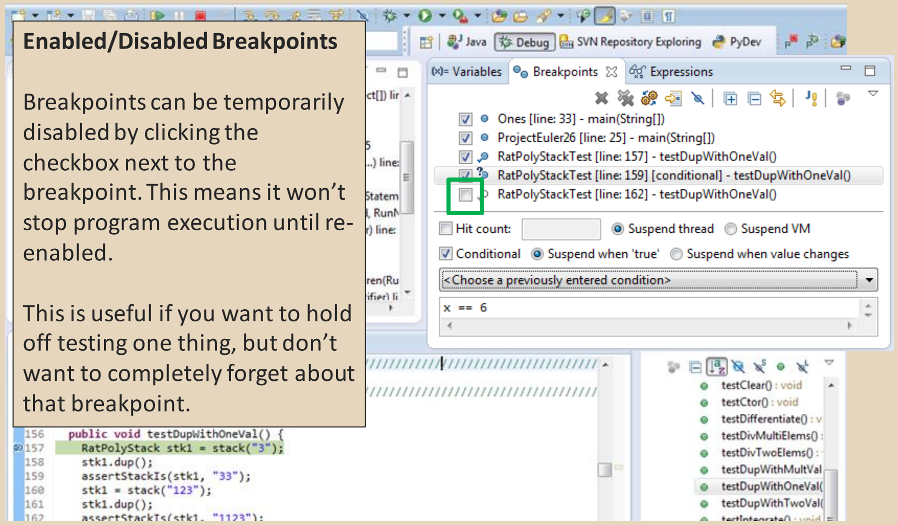
Double clicking a breakpoint will take you to its spot in the code.

```
☑ 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
```

ECLIPSE DEBUGGING

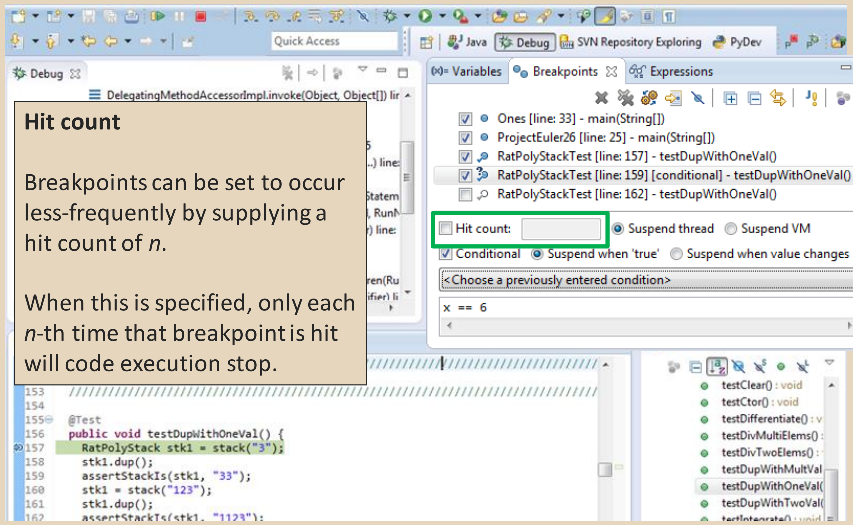


Enabled/Disabled Breakpoints

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.

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.

- [✓] 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

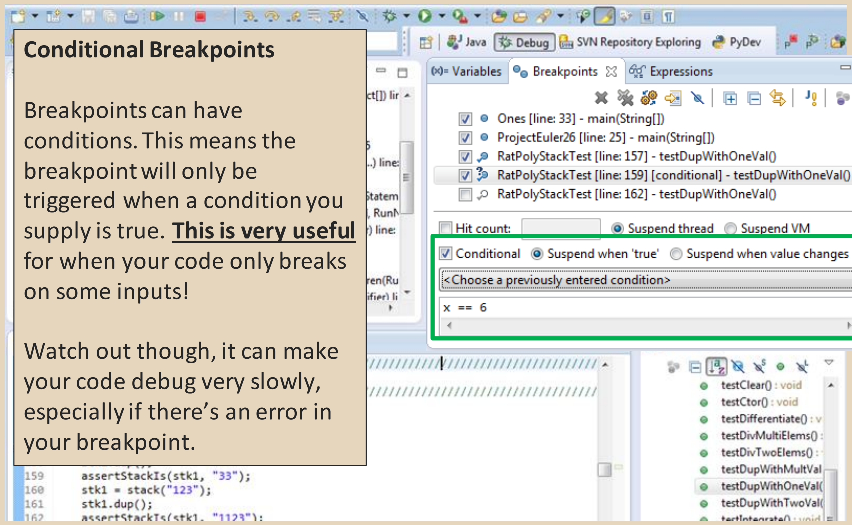
Conditional [✓] [✓ Suspend when 'true'] Suspend when value changes

<Choose a previously entered condition>

```
x == 6
```

```
@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING



Conditional Breakpoints

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

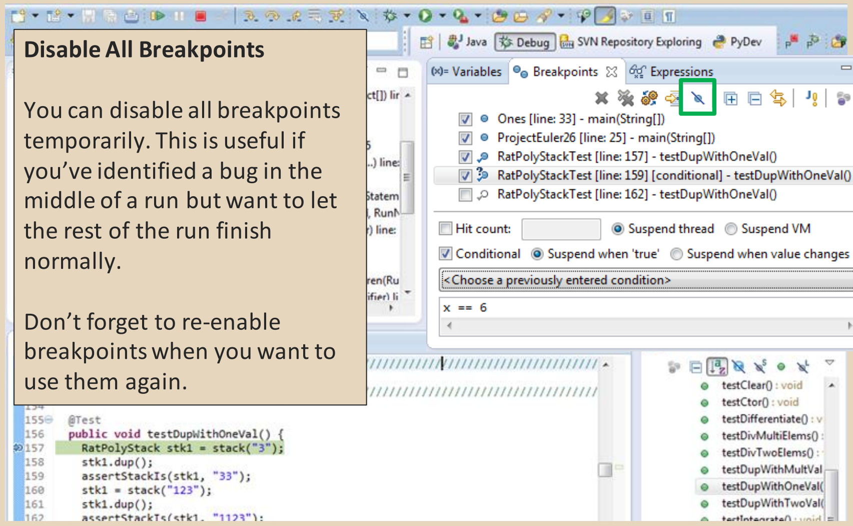
Watch out though, it can make your code debug very slowly, especially if there's an error in your breakpoint.

Conditional [✓] [✓ Suspend when 'true'] Suspend when value changes

<Choose a previously entered condition>

```
x == 6
```

ECLIPSE DEBUGGING

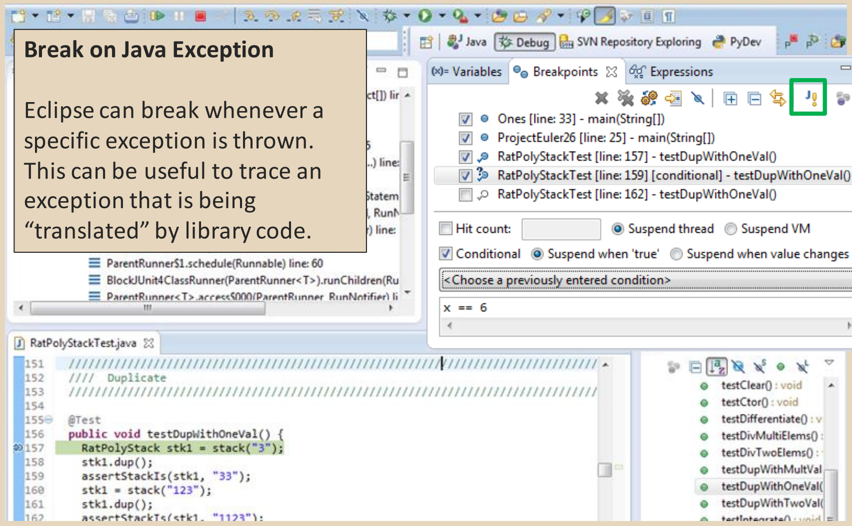


Disable All Breakpoints

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.

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
```

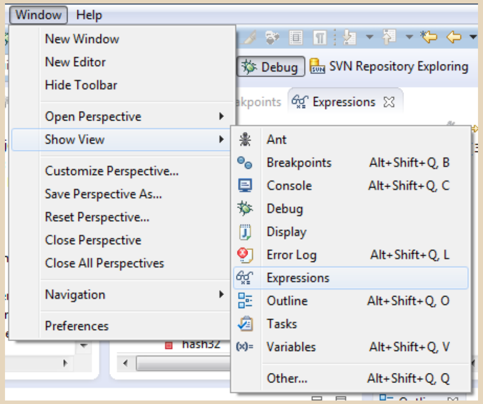
```
//// Duplicate
@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

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.



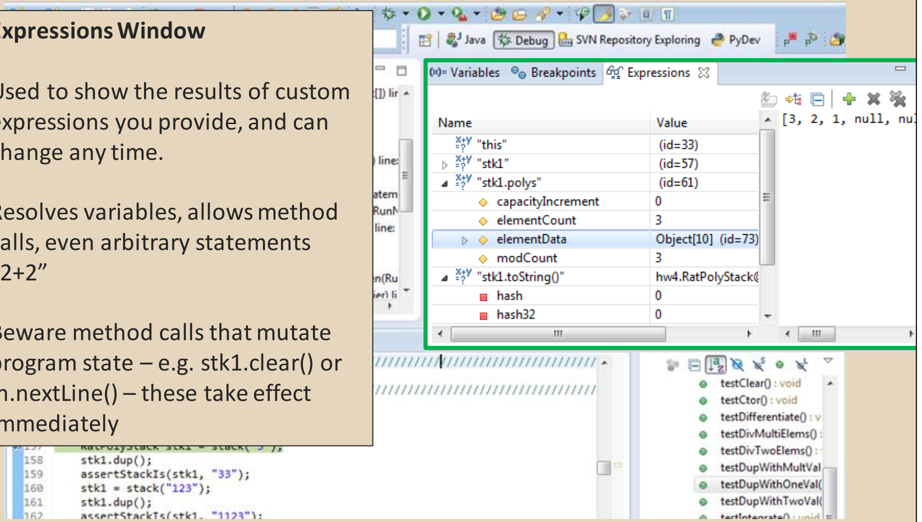
ECLIPSE DEBUGGING

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



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

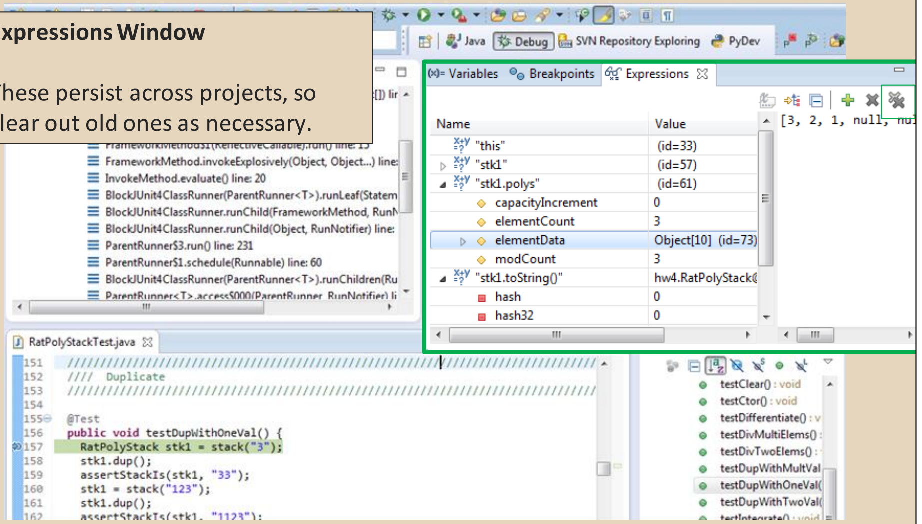
```
[3, 2, 1, null, nu
```

```
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```

ECLIPSE DEBUGGING

Expressions Window

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



```
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
```

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

```
[3, 2, 1, null, nu
```

```
////////////////////////////////////////////////////////////////////////
////  Duplicate
////////////////////////////////////////////////////////////////////////

@Test
public void testDupWithOneVal() {
    RatPolyStack stk1 = stack("3");
    stk1.dup();
    assertStackIs(stk1, "33");
    stk1 = stack("123");
    stk1.dup();
    assertStackIs(stk1, "1123");
```