

# hw6, BFS, debugging

CSE 331

Section 5 – 10/25/12

Slides by Kellen Donohue

# Agenda

- hw4 being graded
- hw5 may be graded first, for feedback to be used on hw6
- hw6 due next week
  
- Today
  - hw6
  - BFS
  - Debugging



# hashCode() and equals()



- Overriding these important for using classes you write in collections, e.g.
- Read Javadoc for requirements
  - Transitive, symmetric, etc. we'll discuss later in lecture
  - Usually must override hashCode() if you override equals()
- Eclipse can generate them for you
  - Right click in class source file
  - Source -> Generate hashCode() and equals()
  - Not always perfect – learn more later & in 332



# Special values vs. exceptions

```
Map<String, Integer> map =  
    new HashMap<String, Integer>();  
  
System.out.println(map.get("hi"));
```

# Special values vs. exceptions

```
Map<String, Integer> map =  
    new HashMap<String, Integer>();
```

```
System.out.println(map.get("hi"));
```

```
// prints null
```

# Special values vs. exceptions

```
Map<String, Integer> map =  
    new HashMap<String, Integer>();  
  
System.out.println(1 + map.get("hi"));
```

# Special values vs. exceptions

```
Map<String, Integer> map =  
    new HashMap<String, Integer>();  
  
System.out.println(1 + map.get("hi"));  
  
// throws NullPointerException
```

# Special values vs. exceptions

```
Map<String, String> map =  
    new HashMap<String, String>();
```

```
map.put("hi", null);
```

```
System.out.println(map.get("hi"));  
System.out.println(map.get("bye"));
```

```
// Prints null twice
```



# Special values vs. exceptions – C#

```
Dictionary<string, int> map =  
    new Dictionary<string, int>();  
  
Console.WriteLine(map["hi"]);
```

# Special values vs. exceptions – C#

```
Dictionary<string, int> map =  
    new Dictionary<string, int>();
```

```
Console.WriteLine(map["hi"]);
```

```
// C# -- Throws exception
```

```
// (Java -- prints null)
```

# Special values vs. exceptions – C#

```
Dictionary<string, int> map =  
    new Dictionary<string, int>();  
  
map["hi"] = 6;  
int hiVal, byeVal;  
  
bool hiSuccess =  
    map.TryGetValue("hi", out hiVal);  
bool byeSuccess =  
    map.TryGetValue("bye", out byeVal);
```

# Special values vs. exceptions – C#

```
Dictionary<string, int> map =  
    new Dictionary<string, int>();
```

```
map["hi"] = 6;
```

```
int hiVal = 4, byeVal, = 5;
```

```
map.TryGetValue("hi", out hiVal);
```

```
map.TryGetValue("bye", out byeVal);
```

```
// hiVal gets 6, byeVal gets 0 (!)
```

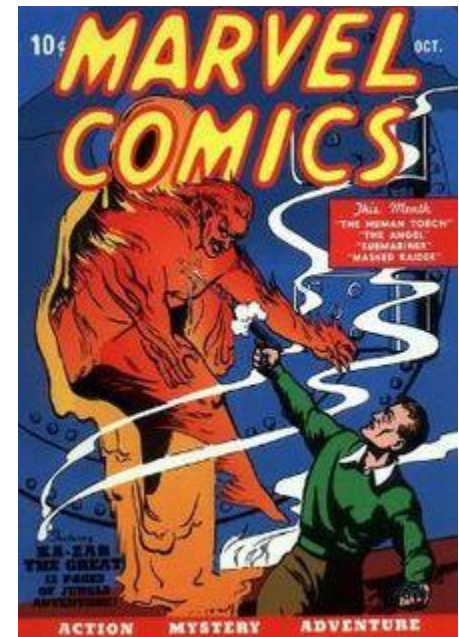
```
// (Java throws exception)
```

# Special values vs. exceptions

- Morals of the story
  - There's more than one right answer to special values vs. exceptions, and you may find more than one in practice
  - Be sure to read the specs to know what happens
  - e.g. What happens with null keys?

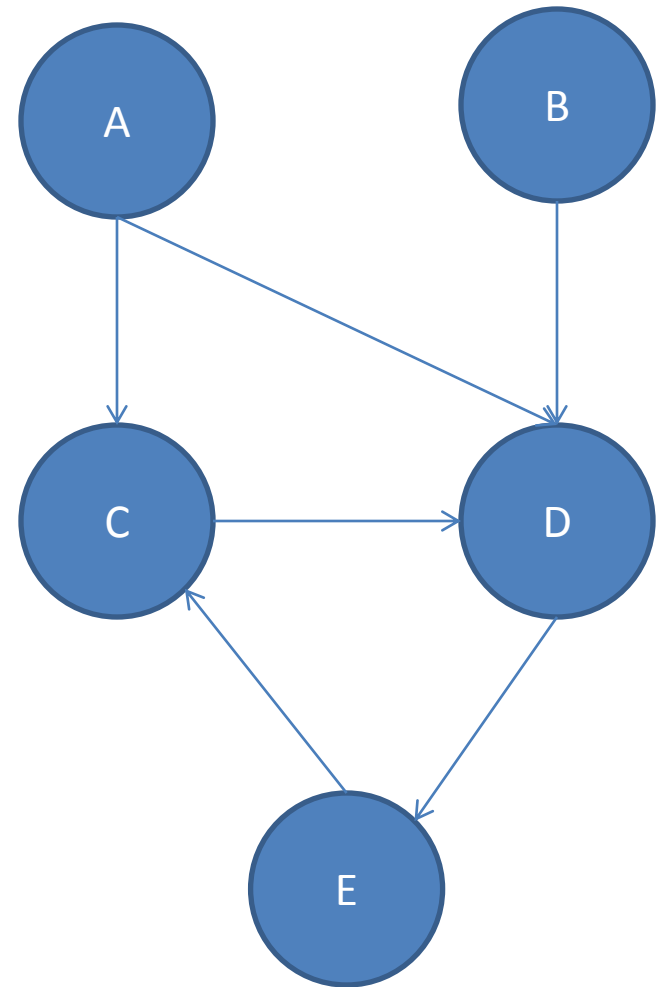
# Homework 6

- Use Graph ADT from hw5
- Fill with Marvel Data
  - Nodes = characters
  - Edges = books
    - Labeled with title
    - Connecting characters if both characters appeared in that book
- Turns out to model real life social graphs



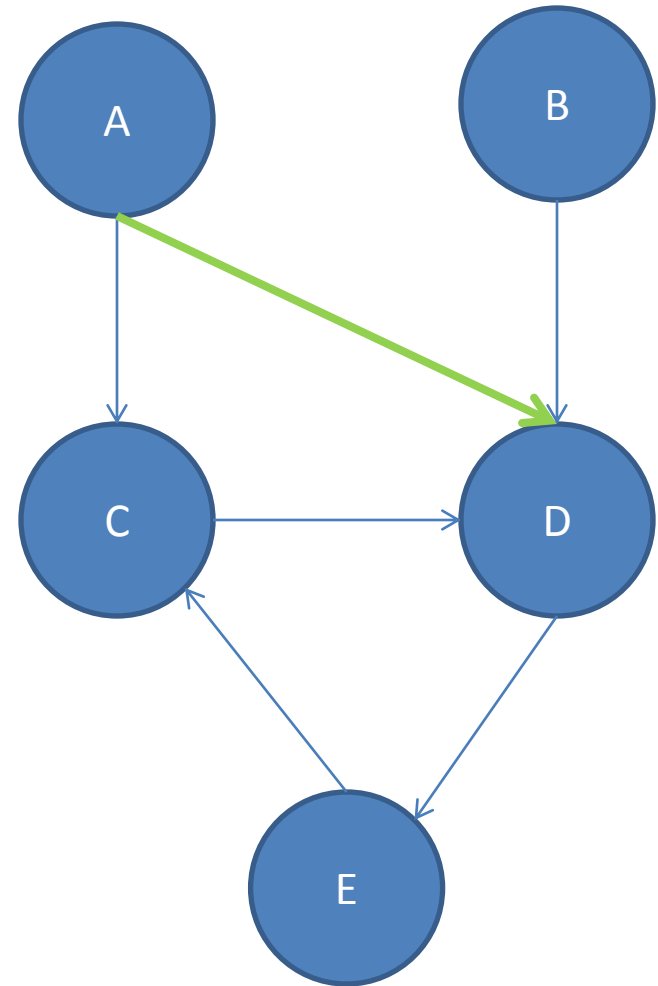
# Graph paths

- List of nodes travelled to get from one node to another, moving along edges, respecting direction



# Graph paths

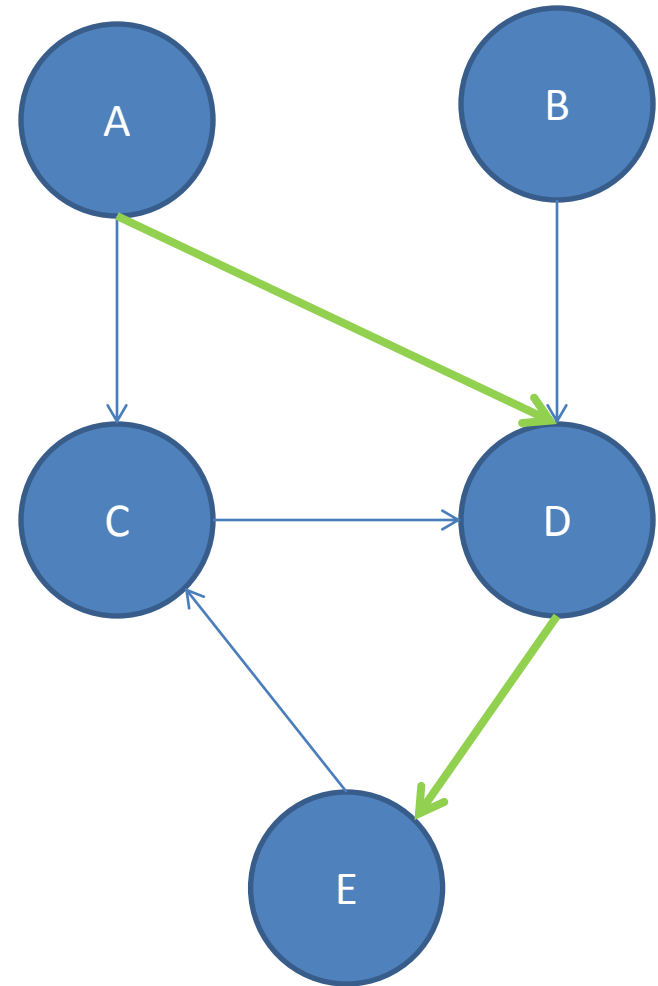
- List of nodes travelled to get from one node to another, moving along edges, respecting direction





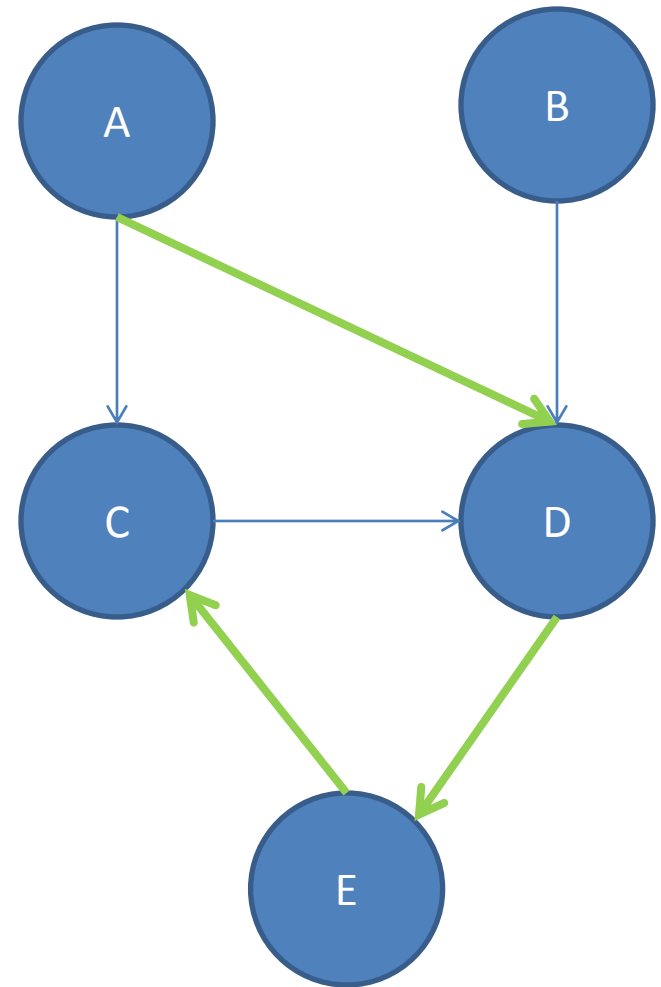
# Graph paths

- List of nodes travelled to get from one node to another, moving along edges, respecting direction



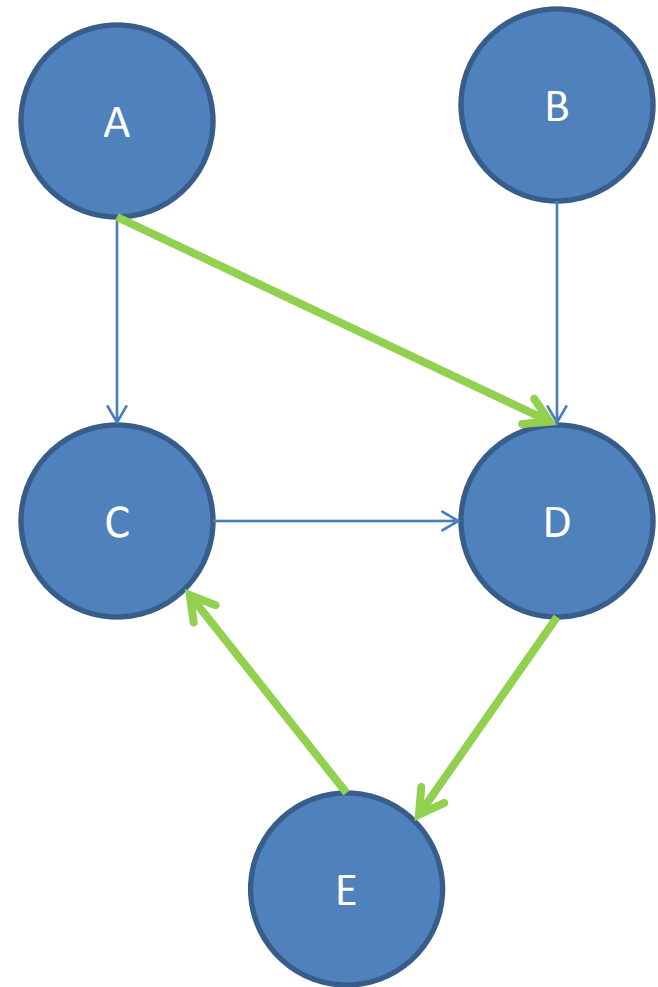
# Graph paths

- List of nodes travelled to get from one node to another, moving along edges, respecting direction



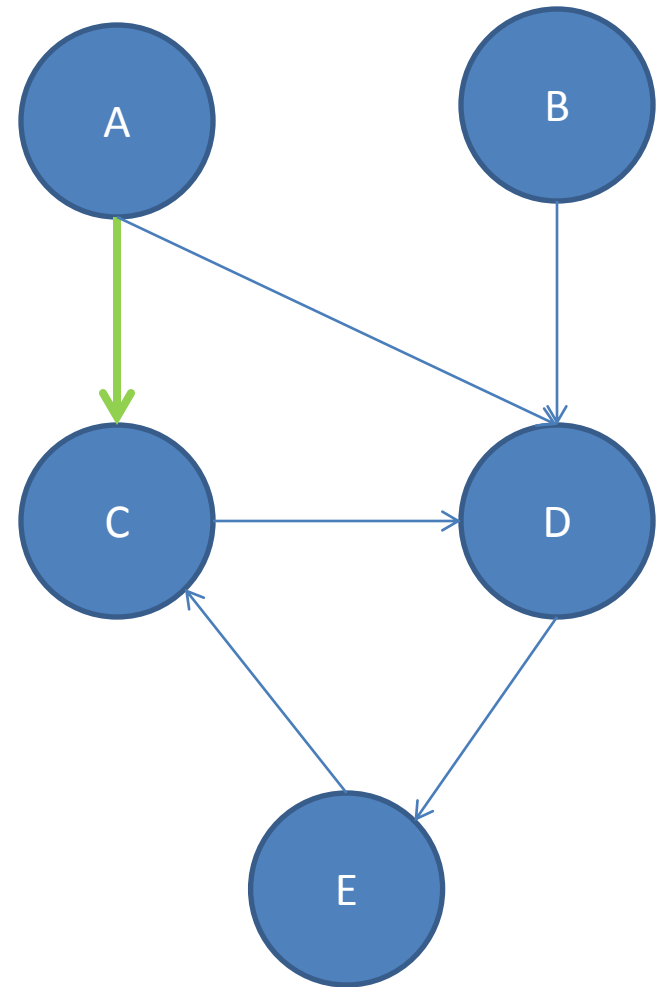
# Graph paths

- List of nodes travelled to get from one node to another, moving along edges, respecting direction
- ADEC is a path A to C



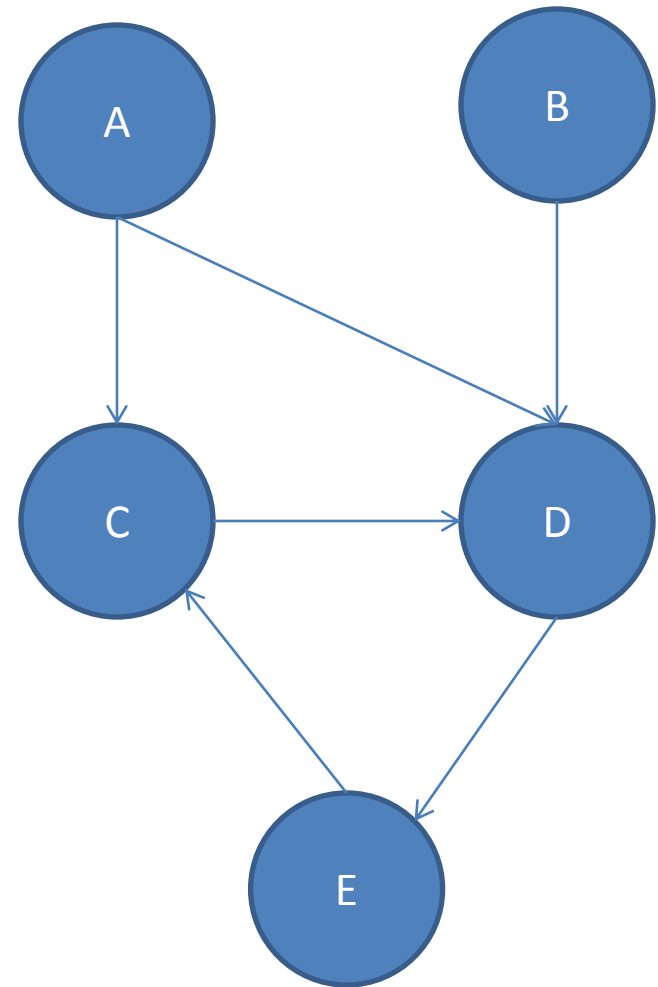
# Graph paths

- List of nodes travelled to get from one node to another, moving along edges, respecting direction
- ADEC is a path A to C
- AC is a path A to C



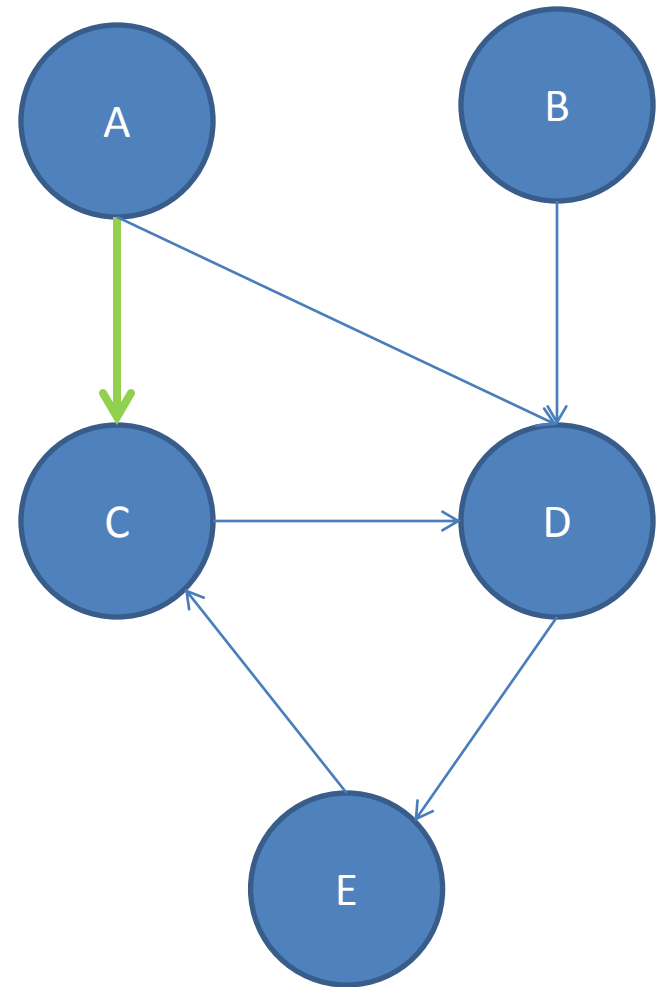
# Graph paths

- List of nodes travelled to get from one node to another, moving along edges, respecting direction
- ADEC is a path A to C
- AC is a path A to C
- There's no path A to B



# Graph paths

- We often want to find the shortest path between two nodes
  - Google Maps
  - Optimal route through a maze
- AC is the shortest path A to C

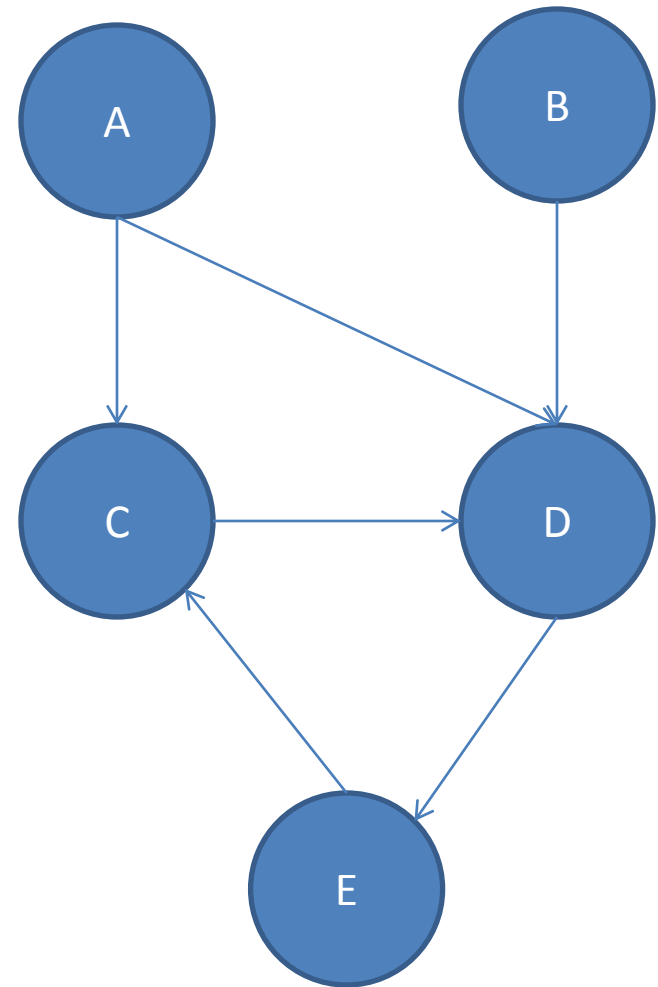


# Breadth-first search

- Informally, put the start node in a queue
  - While the queue isn't empty
    - Pick a node N off the queue
    - If N is the goal then return
    - Else, for each node O you can reach from N
      - If O isn't marked
        - » Add O to the queue
        - » Mark O
  - Couldn't find an path after exploring graph

# Breadth-first search

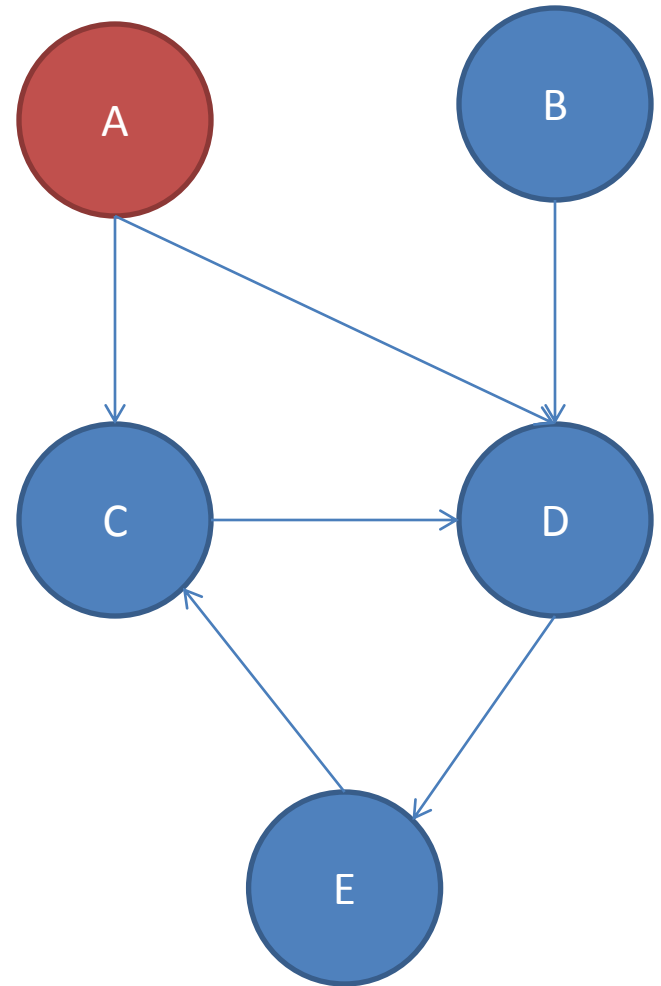
- We often want to find the shortest path between two nodes
  - Google Maps
  - Optimal route through a maze
- AC is the shortest path A to C





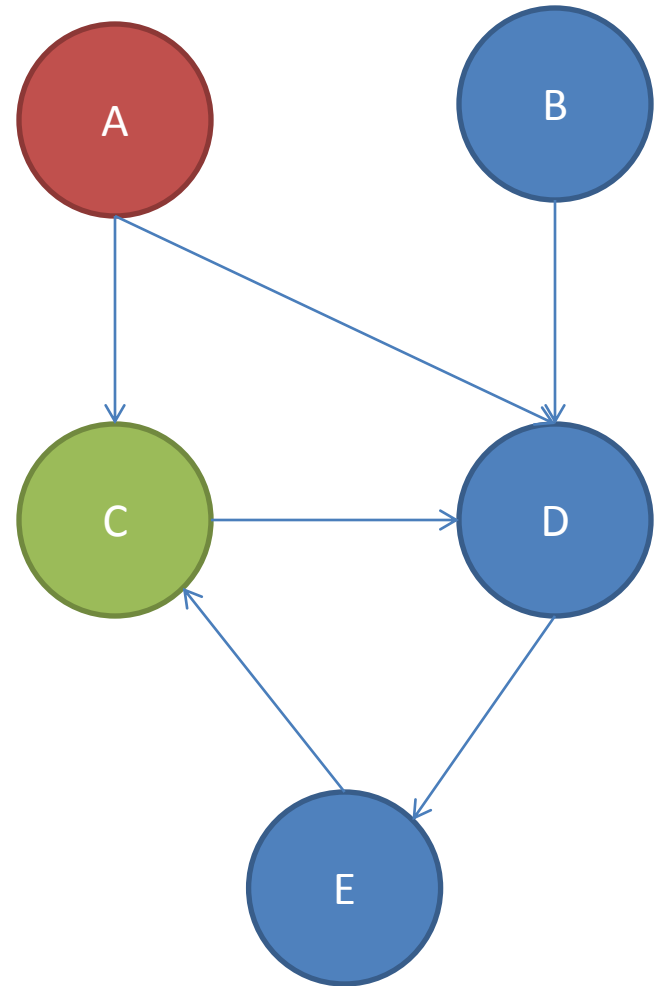
# Breadth-first search

- Queue



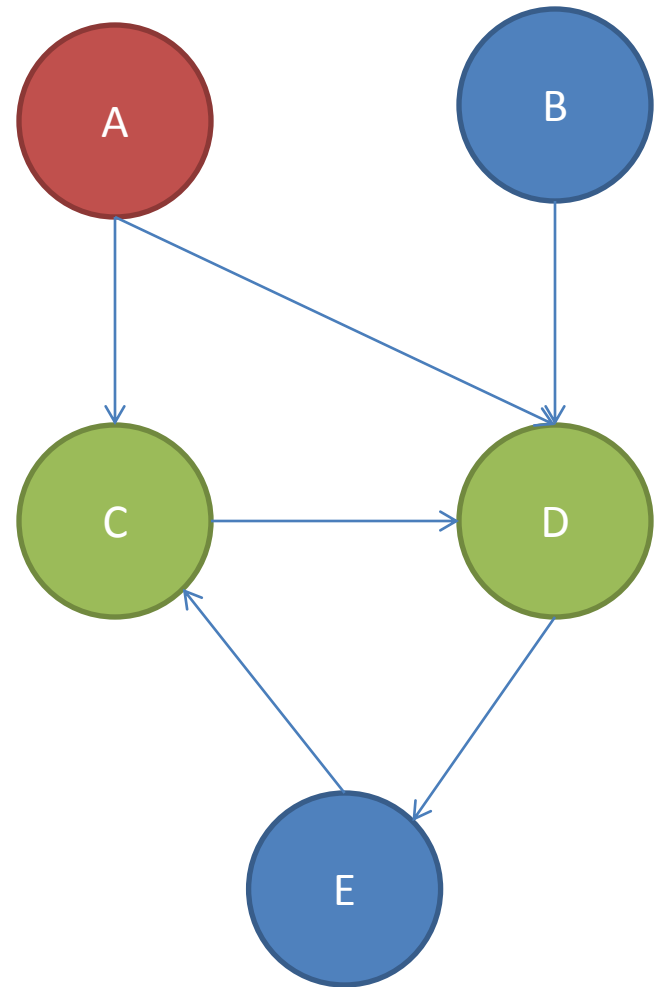
# Breadth-first search

- Queue
  - C



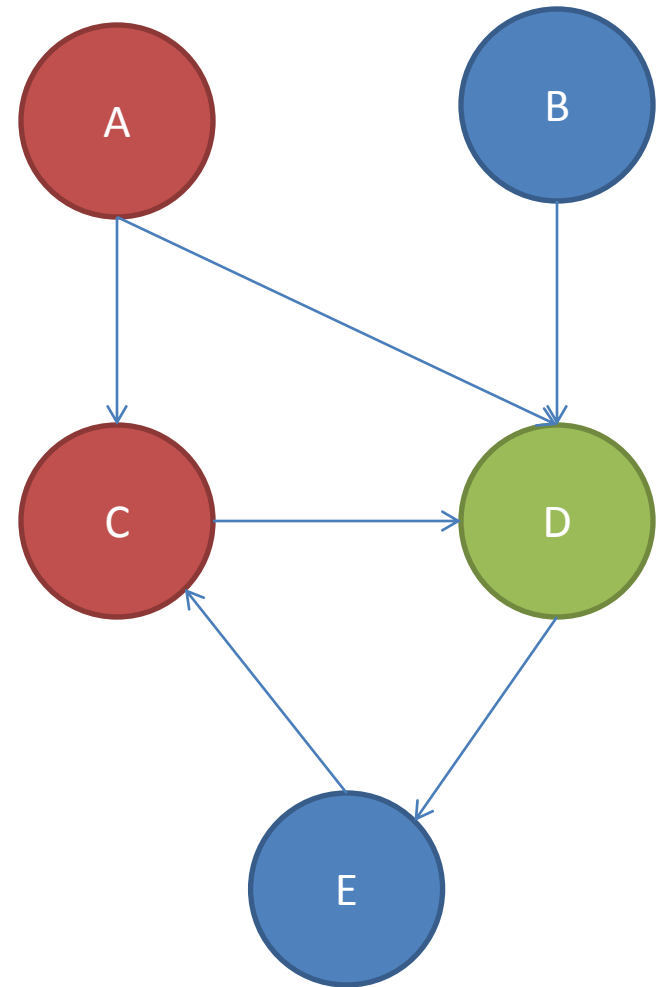
# Breadth-first search

- Queue
  - C
  - D



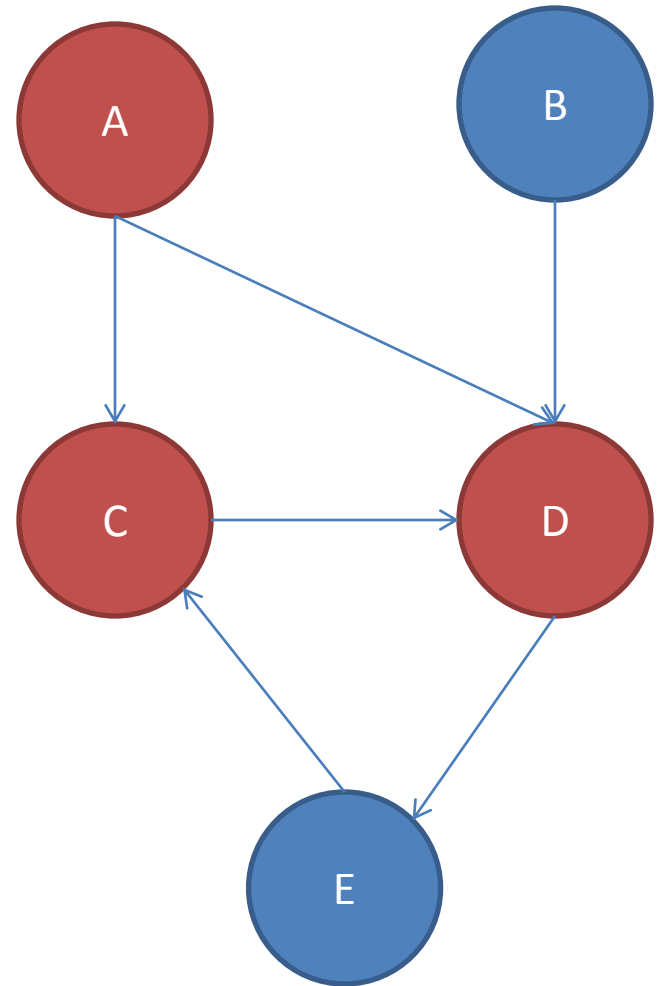
# Breadth-first search

- Queue
  - D



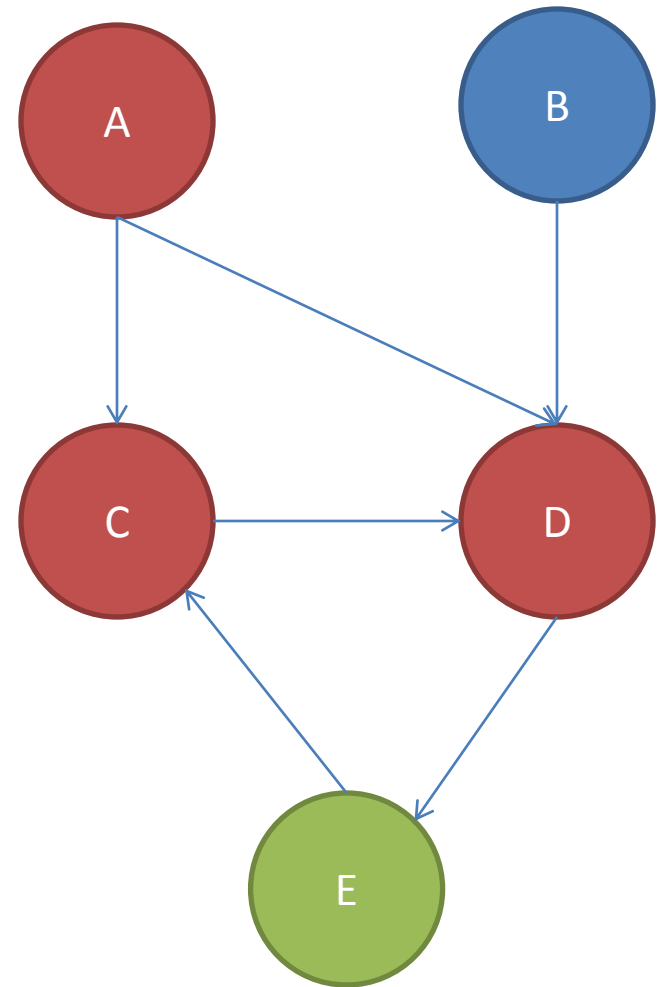
# Breadth-first search

- Queue



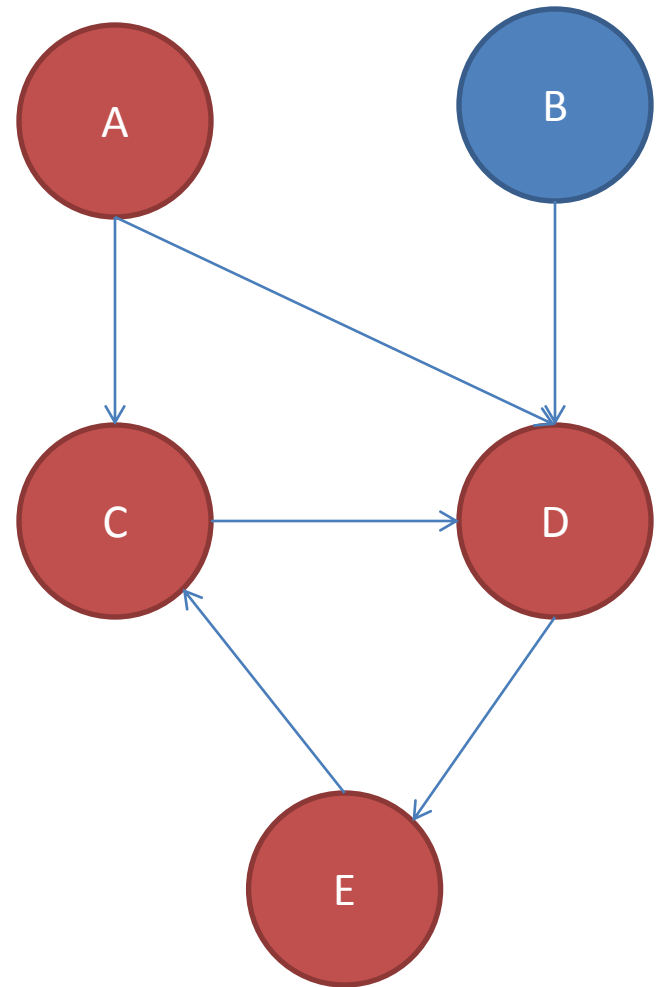
# Breadth-first search

- Queue
  - E



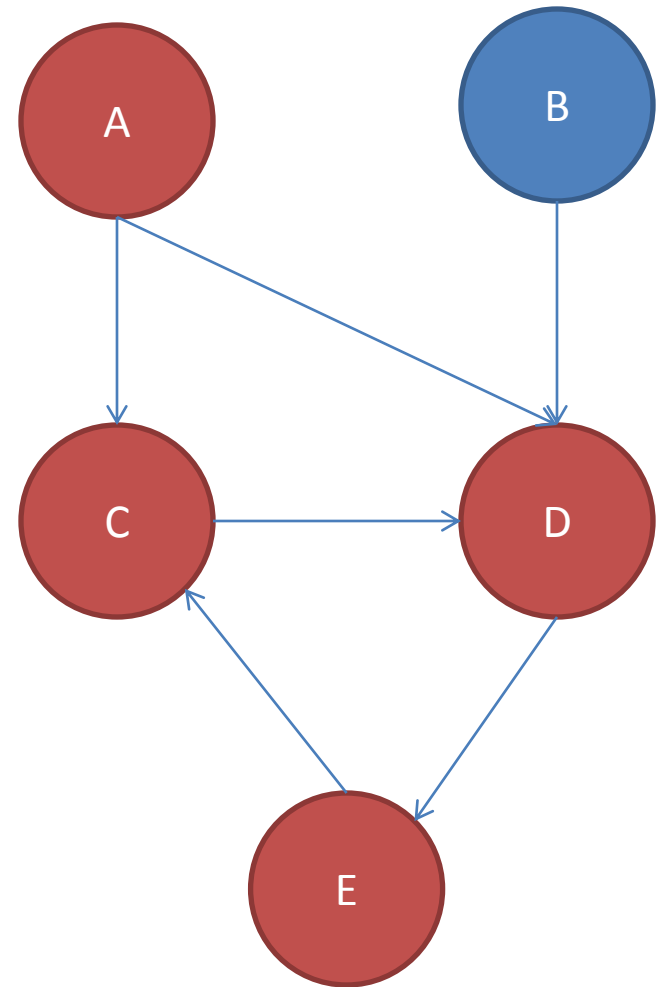
# Breadth-first search

- Queue



# Breadth-first search

- Queue empty, so no path A->B





# Breadth-first search

- Maze demo

# Breadth-first search

- Guaranteed to find shortest-path
  - In number of nodes
  - Not lowest cost path if edges have cost
- If a stack was used instead of queue = depth first search
- Very memory intensive for large graphs --  $O(b^d)$
- Will use in HW6 to find shortest paths between two characters

# Eclipse Debugging

- Hal will talk more in lecture tomorrow about debugging
  - In some sense debugging is last resort
  - Still want good tools for it
- Eclipse has a great debugger!
  - Complicated, hidden features
  - I'll demo a lot, but don't feel try to remember how to do everything – slides will be posted



# Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar includes standard IDE actions like Run, Stop, and Step Over. Below the toolbar, the 'Quick Access' field is visible. The main workspace is divided into several panels:

- Debug Console:** Shows the execution stack with the following entries:
  - DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: not available
  - Method.invoke(Object, Object...) line: not available
  - FrameworkMethod\$1.runReflectiveCall() line: 45
  - FrameworkMethod\$1(ReflectiveCallable).run() line: 15
  - FrameworkMethod.invokeExplosively(Object, Object...) line: not available
  - InvokeMethod.evaluate() line: 20
  - BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf(Statement) line: not available
  - BlockJUnit4ClassRunner.runChild(FrameworkMethod, Runnable) line: not available
  - BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line: not available
  - ParentRunner\$3.run() line: 231
  - ParentRunner\$1.schedule(Runnable) line: 60
  - BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(RunNotifier) line: not available
  - ParentRunner<T>.access\$000(ParentRunner, RunNotifier) line: not available
- Variables View:** Shows a table with the following content:

Name	Value
this	RatPolyStackTest (id=33)
- Code Editor:** Displays the source code for `RatPolyStackTest.java`. A breakpoint is set on line 57, indicated by a small circle in the left margin. The code includes comments and a call to `assertStackTs(stk1, "1123")`.

A text box with a black border is overlaid on the code editor, containing the following text:

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

# Eclipse Debugging

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

The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons, with the 'Debug' icon (a bug) highlighted by a green box. Below the toolbar, a text box contains the instruction: 'Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints.' The main workspace is divided into several panes. The left pane shows a stack trace with the following entries:

- DelegatingMethodA...
- Method.invoke(Object...
- FrameworkMethodS...
- FrameworkMethodS1(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

The right pane shows a 'Value' table with one entry: 'RatPolyStackTest (id=33)'. The bottom pane shows the source code for '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");
```

The 'Outline' pane on the right shows a list of test methods:

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

# Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. A red box highlights the top toolbar, which contains several icons for controlling the program's execution: a green play button (Run), a red square (Stop), a blue square (Debug), a blue play button (Resume), a blue square (Step Over), a blue square (Step Into), and a blue square (Step Out). A text box with a black border is overlaid on the right side of the IDE, containing the text: "Controlling your program while debugging is done with these buttons".

The Debug console on the left shows a list of stack frames, including:

- 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

The Java variable view on the right shows a table with columns for Name and Value. The Name column contains the letter 't'.

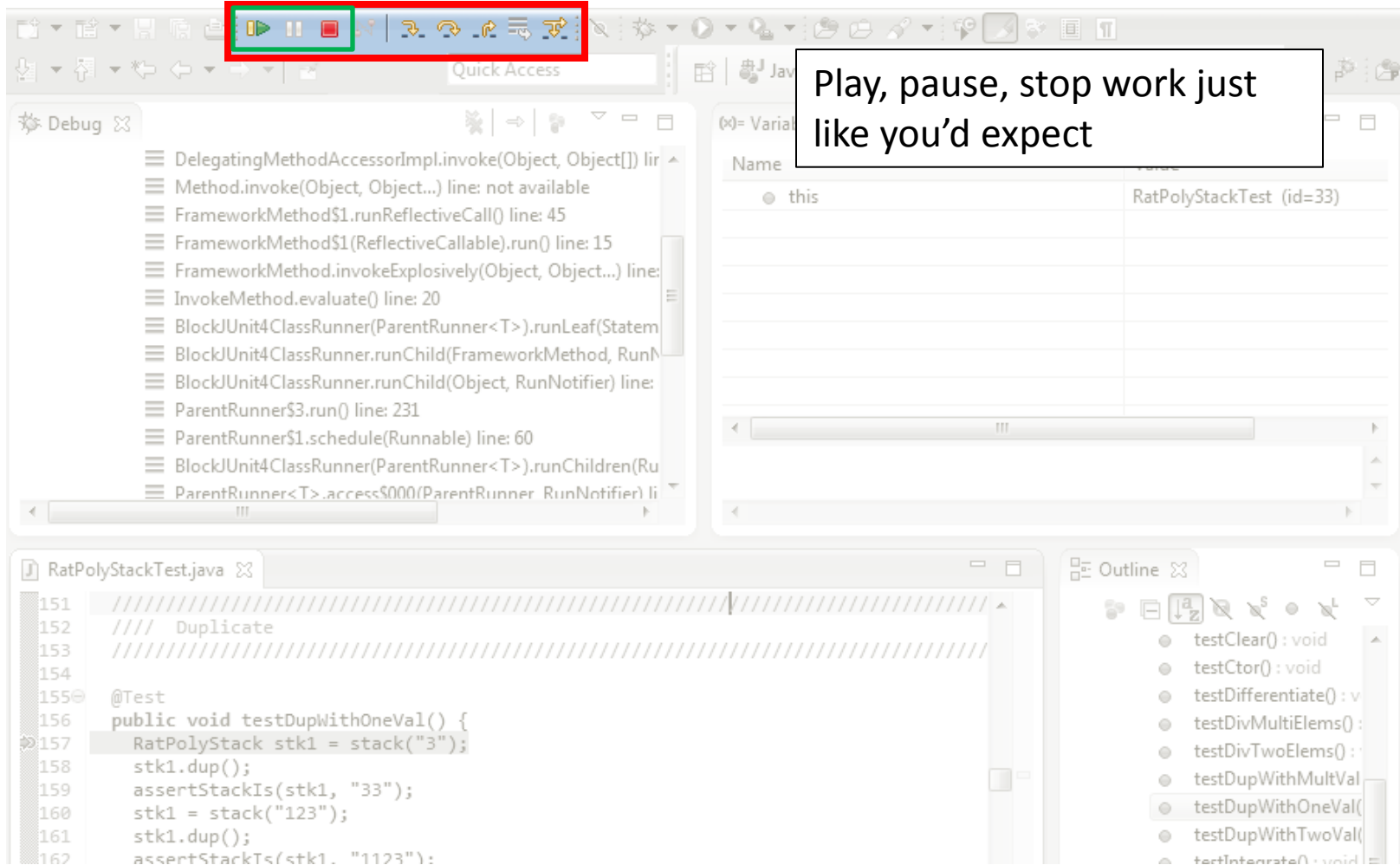
The code editor at the bottom shows the source code for `RatPolyStackTest.java`. The code is as follows:

```
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 Outline view on the right shows a list of methods in the `RatPolyStackTest` class:

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

# Eclipse Debugging





# Eclipse Debugging

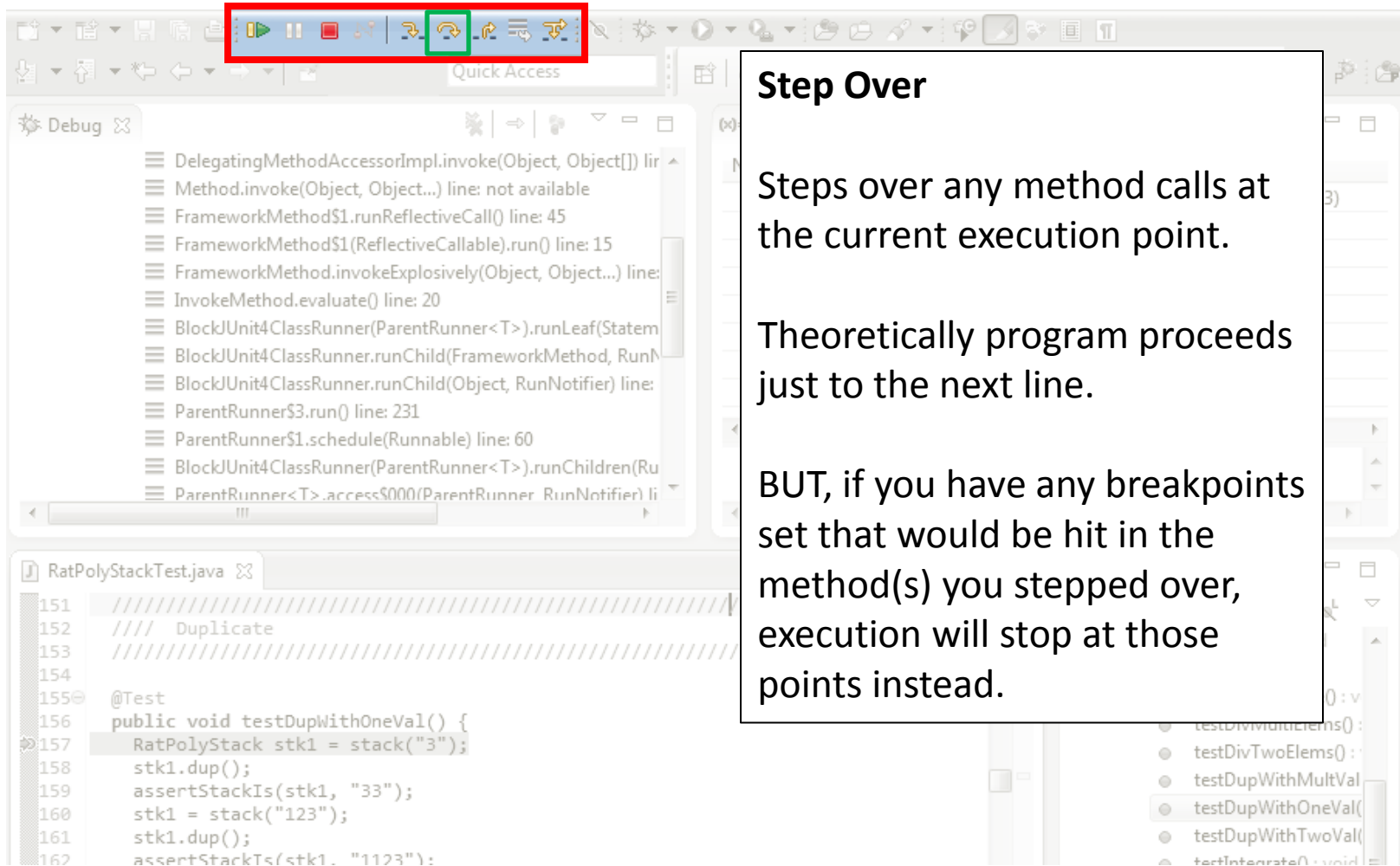
The screenshot shows the Eclipse IDE interface. At the top, the toolbar contains several icons, with a red box highlighting the 'Step Into' icon (a blue square with a white arrow pointing right) and a green box highlighting the 'Step Over' icon (a blue square with a white arrow pointing right). Below the toolbar, the 'Debug' console displays a stack trace of the current execution. The code editor shows the file 'RatPolyStackTest.java' with a breakpoint set at line 157. The code at line 157 is `RatPolyStack stk1 = stack("3");`. To the right of the code editor, a list of test methods is visible, including `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal()`, `testDupWithOneVal()`, `testDupWithTwoVal()`, and `testIntegrate()`.

**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



The screenshot shows the Eclipse IDE interface. At the top, a toolbar contains several icons, with a red box highlighting the 'Step Over' icon (a blue square with a white right-pointing arrow) and a green box highlighting the 'Step Into' icon (a blue square with a white right-pointing arrow). Below the toolbar, the 'Debug' console displays a stack trace of method calls. The bottom pane shows the source code for 'RatPolyStackTest.java', with line 157 highlighted. The code includes a test method 'testDupWithOneVal()' that creates a stack, duplicates it, and asserts its contents.

```
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");  
}
```

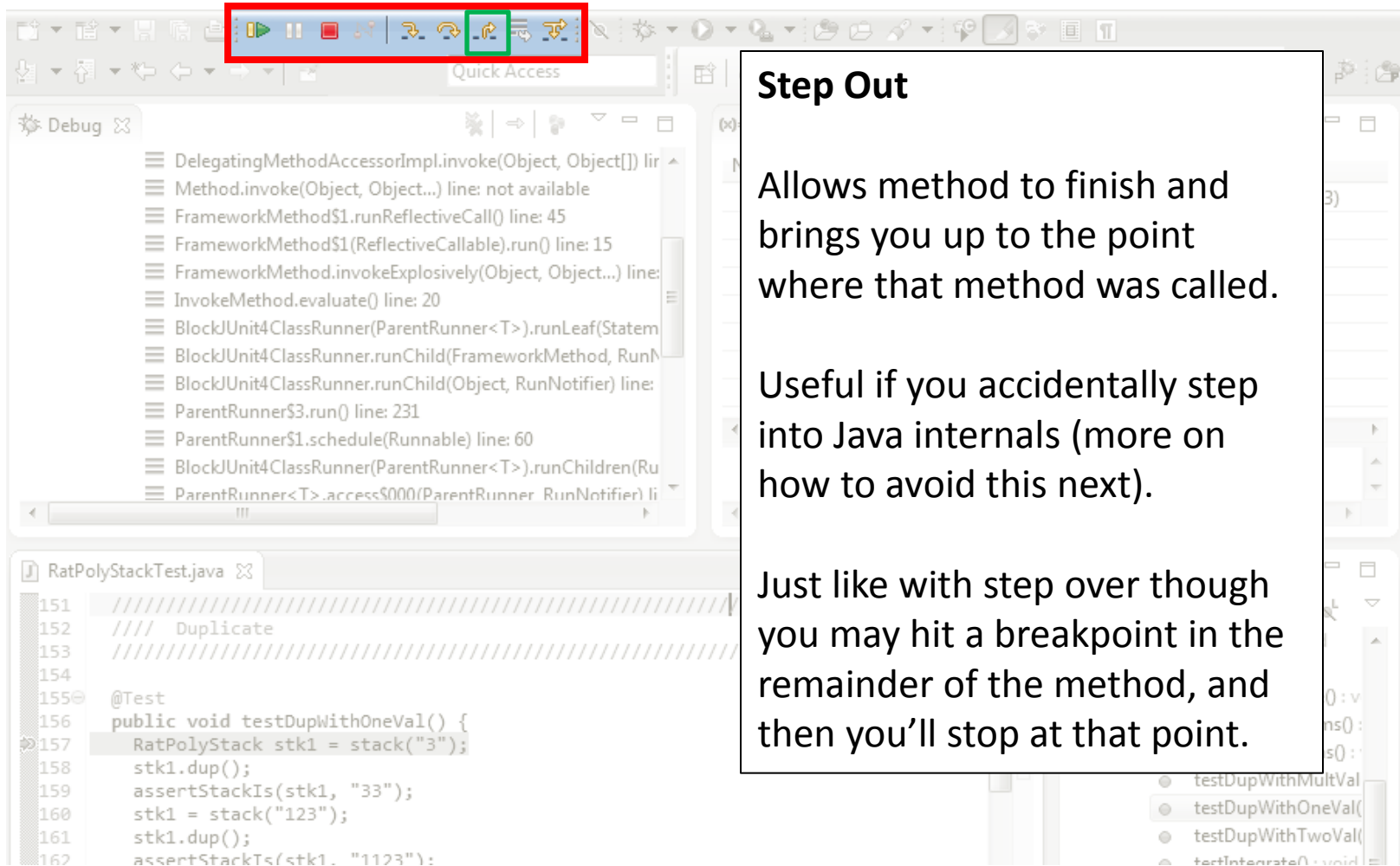
## Step Over

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



The screenshot shows the Eclipse IDE interface. At the top, a toolbar contains several icons, with a red box highlighting the 'Step Out' icon (a blue square with a white arrow pointing right) and a green box highlighting the 'Step Over' icon (a blue square with a white arrow pointing right). Below the toolbar, the 'Debug' console shows a stack trace of method calls. The bottom part of the image shows a code editor with the file 'RatPolyStackTest.java' open. 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 'testDupWithOneVal()' method is highlighted in the code editor. The 'Debug' console shows the following stack trace:

- 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

The 'Debug' console also shows a list of variables in the current frame:

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

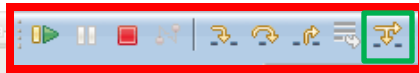
## Step Out

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

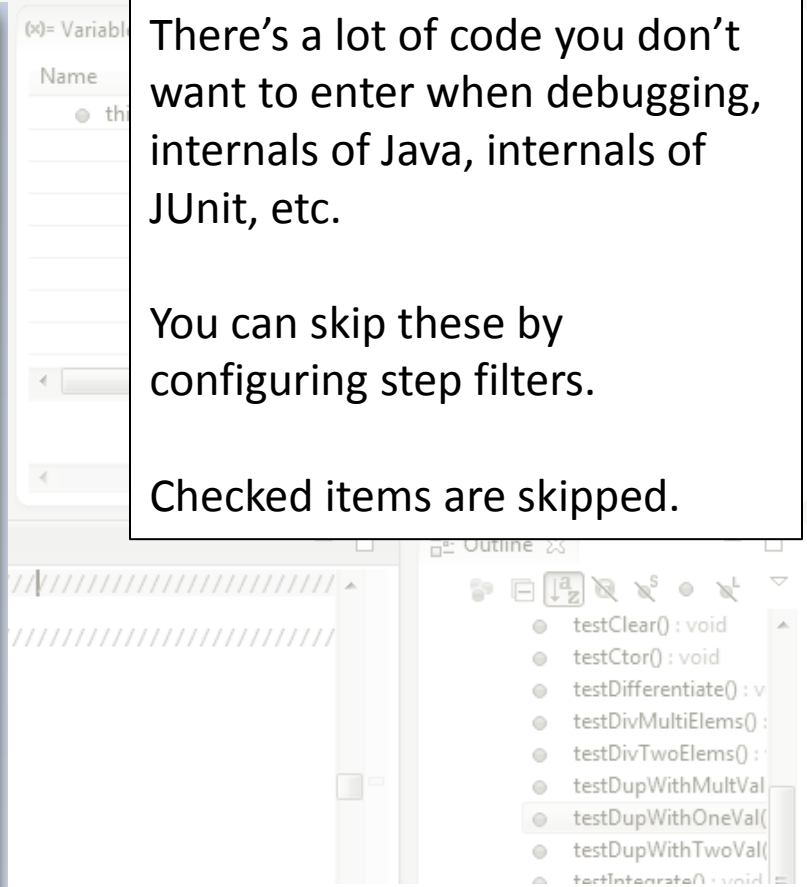
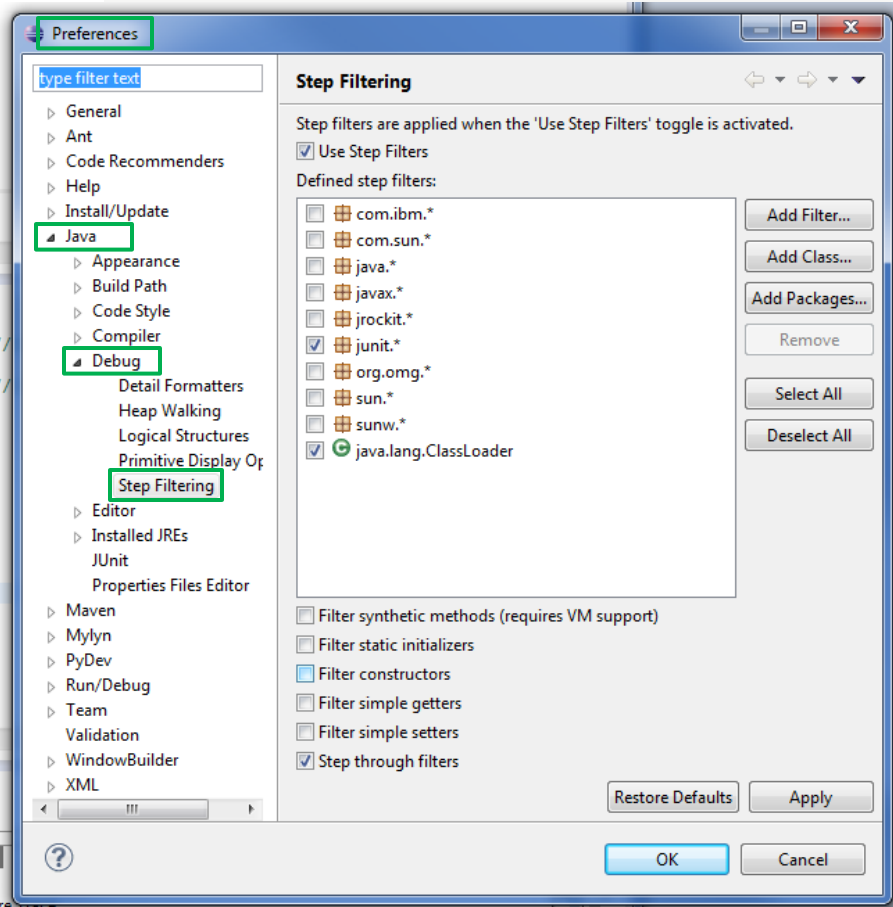


## Enable/disable step filters

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 of method calls. A red rectangle highlights this console. The stack trace lists methods such as `DelegatingMethodAccessorImpl.invoke`, `Method.invoke`, `FrameworkMethod$1.runReflectiveCall`, `FrameworkMethod$1(ReflectiveCallable).run`, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate`, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `BlockJUnit4ClassRunner.runChild`, `BlockJUnit4ClassRunner.runChild`, `ParentRunner$3.run`, `ParentRunner$1.schedule`, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren`, and `ParentRunner<T>.$access$000`. In the center, the 'Variables' pane shows a table with columns for 'Name' and 'Value'. On the right, the 'Stack Trace' pane lists several test methods, with `testDupWithOneVal` selected. At the bottom, the 'RatPolyStackTest.java' code editor is open, showing a test method `testDupWithOneVal` with a breakpoint set at line 157, which is highlighted in grey. The code includes comments for duplicating a stack and assertions for its contents.

**Stack Trace**

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

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

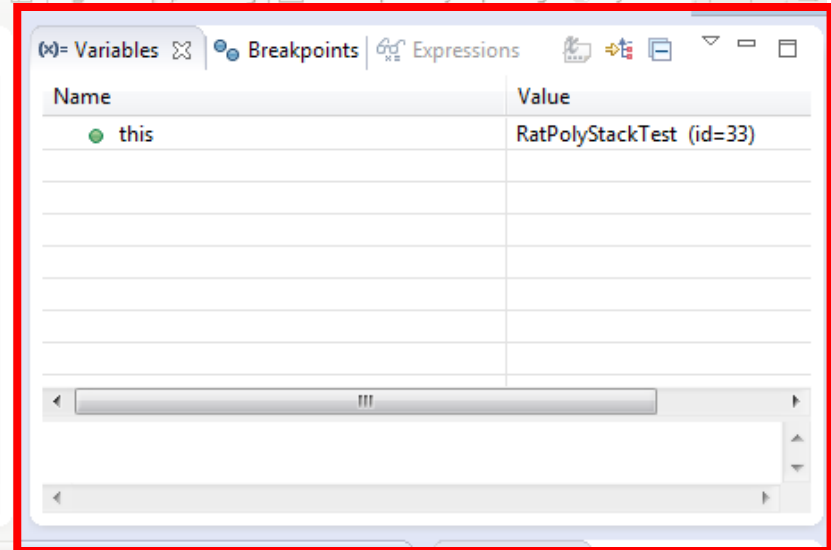
```
151 ////////////////////////////////////////////////////////////////////
152 /// Duplicate
153 ////////////////////////////////////////////////////////////////////
154
155 @Test
156 public void testDupWithOneVal() {
157     RatPolyStack stk1 = stack("3");
158     stk1.dup();
159     assertStackIs(stk1,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

# Eclipse Debugging

## Variables Window

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

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



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 'Variables' window is open, displaying a table of variables:

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

The 'expt' variable is highlighted in yellow, indicating it has changed since the last breakpoint. Below the table, the expression  $-2*x^5$  is visible.

The code editor shows the following Java 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

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 in a debug state. The 'Variables' window is open, displaying a table of variables. The 'expt' variable is highlighted in yellow, indicating it has changed since the last breakpoint. The code editor shows a Java method 'testDupWithOneVal()' with a breakpoint at line 157. The Outline view on the right shows the class structure.

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 'Variables' view is open, displaying a tree structure of variables. The variable 'expt' is selected and highlighted in yellow. A right-click context menu is open over 'expt', listing various actions such as 'Select All', 'Copy Variables', 'Find...', 'Change Value...', 'All References...', 'All Instances...', 'Instance Count...', 'New Detail Formatter...', 'Open Declared Type', 'Open Declared Type Hierarchy', 'Instance Breakpoints...', 'Watch', and 'Inspect'. The 'All Instances...' option is highlighted in blue. In the background, the source code editor shows a Java class named 'Runner.class' with a method 'testDupWithOneVal()' containing several lines of code involving 'RatPolyStack' objects.

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

```
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

## 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 with a Java class named `RatPolyStackTest.java` open. The code is at line 157, where `stk1 = stack("3");` is executed. The Variables view on the right shows the state of the program:

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

The red box highlights the expansion icon (a plus sign with a right-pointing arrow) in the top right corner of the Variables view, which is used to show the logical structure of the objects.

```

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

## 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, 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

## Enabled/Disabled Breakpoints

Breakpoints can be temporarily disabled by clicking the checkbox next to the breakpoint. This means it won't stop program execution until re-enabled.

This is useful if you want to hold off testing one thing, but don't want to completely forget about that breakpoint.

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

The screenshot shows the Eclipse IDE's Breakpoints view. The view is titled "Breakpoints" and contains a list of breakpoints for the method `testDupWithOneVal()`. 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 breakpoint at line 162 is disabled, indicated by a greyed-out checkbox. The breakpoint at line 159 is conditional and is currently enabled. The conditional expression `x == 6` is entered in the text field below the list. The background shows the Java editor with the code from the previous block.

# Eclipse Debugging

## Hit count

Breakpoints can be set to occur less-frequently by supplying a hit count of  $n$ .

When this is specified, only each  $n$ -th time that breakpoint is hit will code execution stop.

The screenshot shows the Eclipse IDE interface. The main editor displays a Java file with the following code:

```
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 'Breakpoints' dialog is open, showing a list of breakpoints. The selected breakpoint is 'RatPolyStackTest [line: 159] [conditional] - testDupWithOneVal()'. The 'Hit count' field is highlighted with a green box and contains the value '6'. The 'Suspend when' options are 'Suspend thread' (selected) and 'Suspend VM'. The 'Conditional' checkbox is checked, and the 'Suspend when true' radio button is selected. The condition field contains 'x == 6'. The background shows the Eclipse IDE with a Java file open and a list of test methods on the right.

# Eclipse Debugging

## Conditional Breakpoints

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

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

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

The screenshot shows the Eclipse IDE interface during a debug session. The Breakpoints view is open, displaying a list of breakpoints. A conditional breakpoint is highlighted for the method `RatPolyStackTest.testDupWithOneVal()` at line 159. The condition for this breakpoint is `x == 6`. The view also shows options for hit count, suspension (suspend thread or suspend VM), and suspension type (conditional, suspend when 'true', or suspend when value changes).

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>

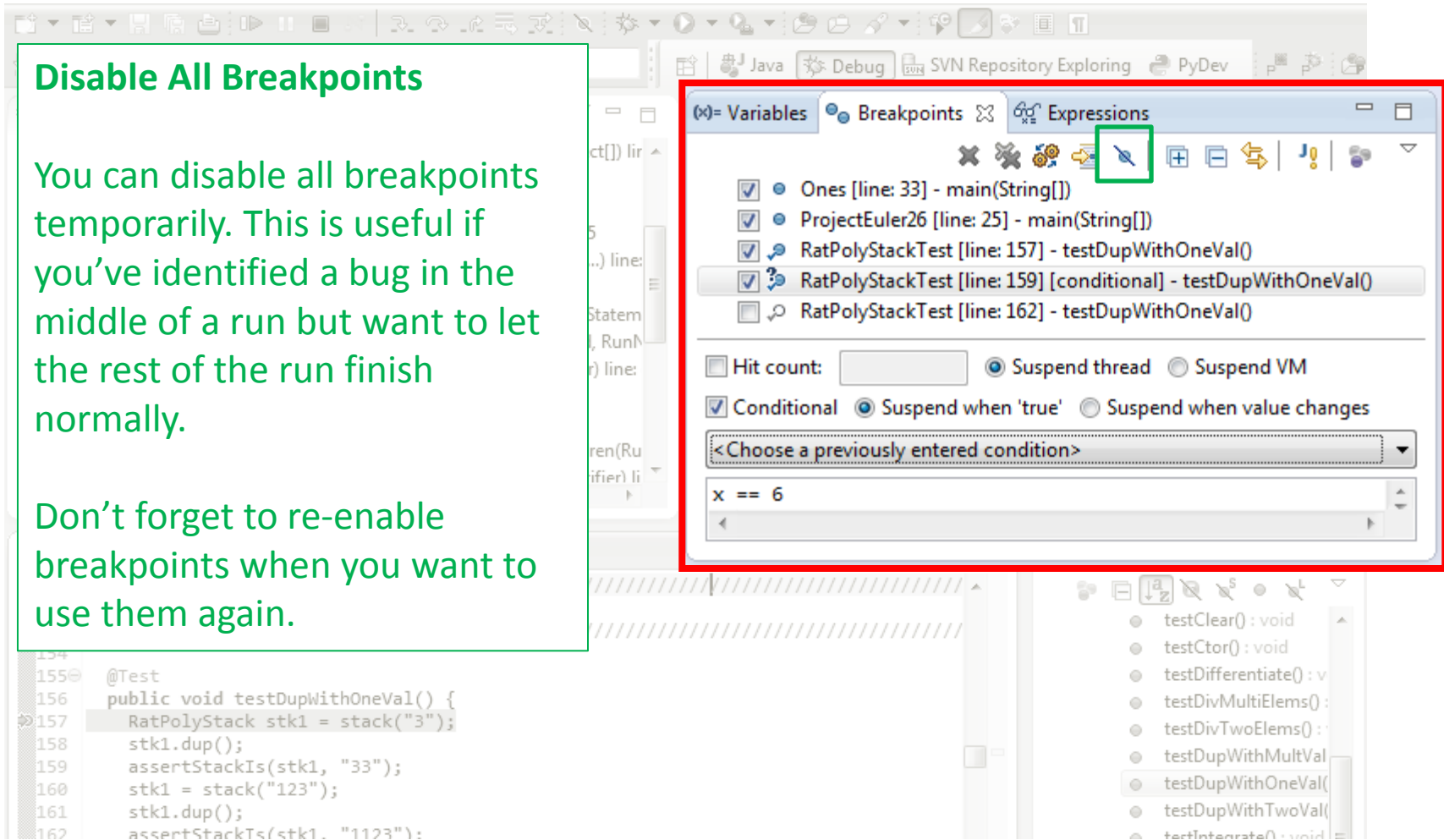
`x == 6`

# 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 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 view is currently set to "Conditional" and "Suspend when 'true'". The "Hit count" field is empty. A red box highlights the "Hit count" and "Conditional" options, and a green box highlights the "Hit count" field. The code editor shows the following code:

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

# Eclipse Debugging

## Break on Java Exception

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

The screenshot displays the Eclipse IDE interface during a debugging session. The main editor shows the source code for `RatPolyStackTest.java`, with line 159 highlighted: `assertStackIs(stk1, "33");`. The `Breakpoints` view is open, showing a list of breakpoints. A conditional breakpoint is set on `RatPolyStackTest [line: 159] - testDupWithOneVal()`. The breakpoint configuration is as follows:

- Hit count: [ ]
- Suspend thread
- Suspend VM
- Conditional
- Suspend when 'true'
- Suspend when value changes
- Condition: `x == 6`

The `Expressions` view is also visible, showing a list of expressions, including `testClear() : void`, `testCtor() : void`, `testDifferentiate() : void`, `testDivMultiElems() :`, `testDivTwoElems() :`, `testDupWithMultVal`, `testDupWithOneVal(`, `testDupWithTwoVal(`, and `testIntegrate() : void`.

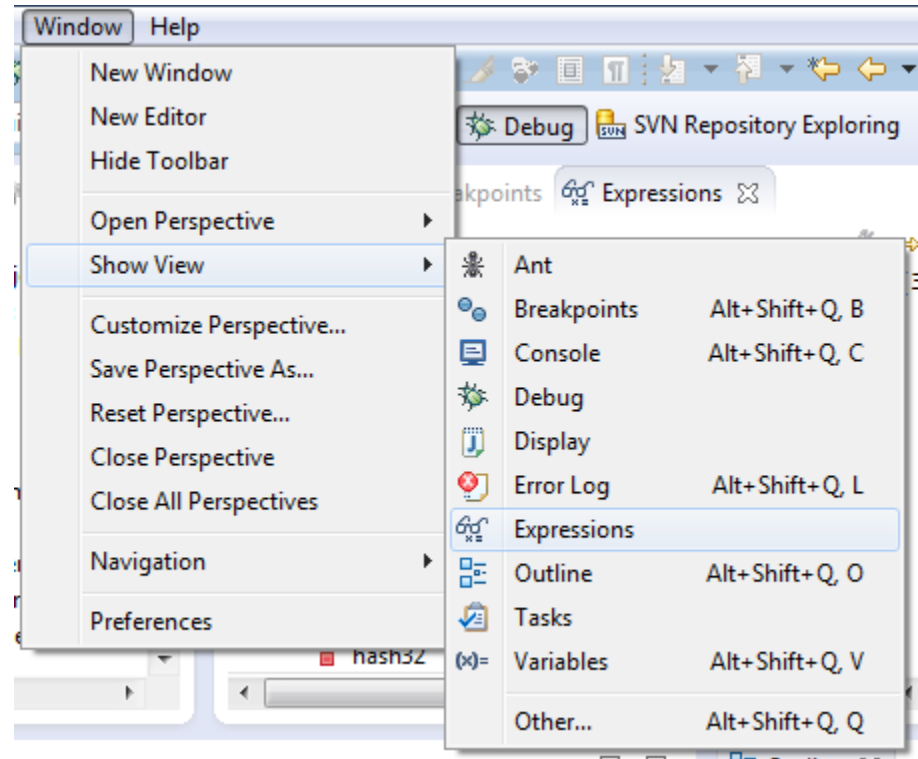


# Eclipse Debugging

## Expressions Window

Used to show the results of custom expressions you provide, and can change any time.

Not shown by default but highly recommended.



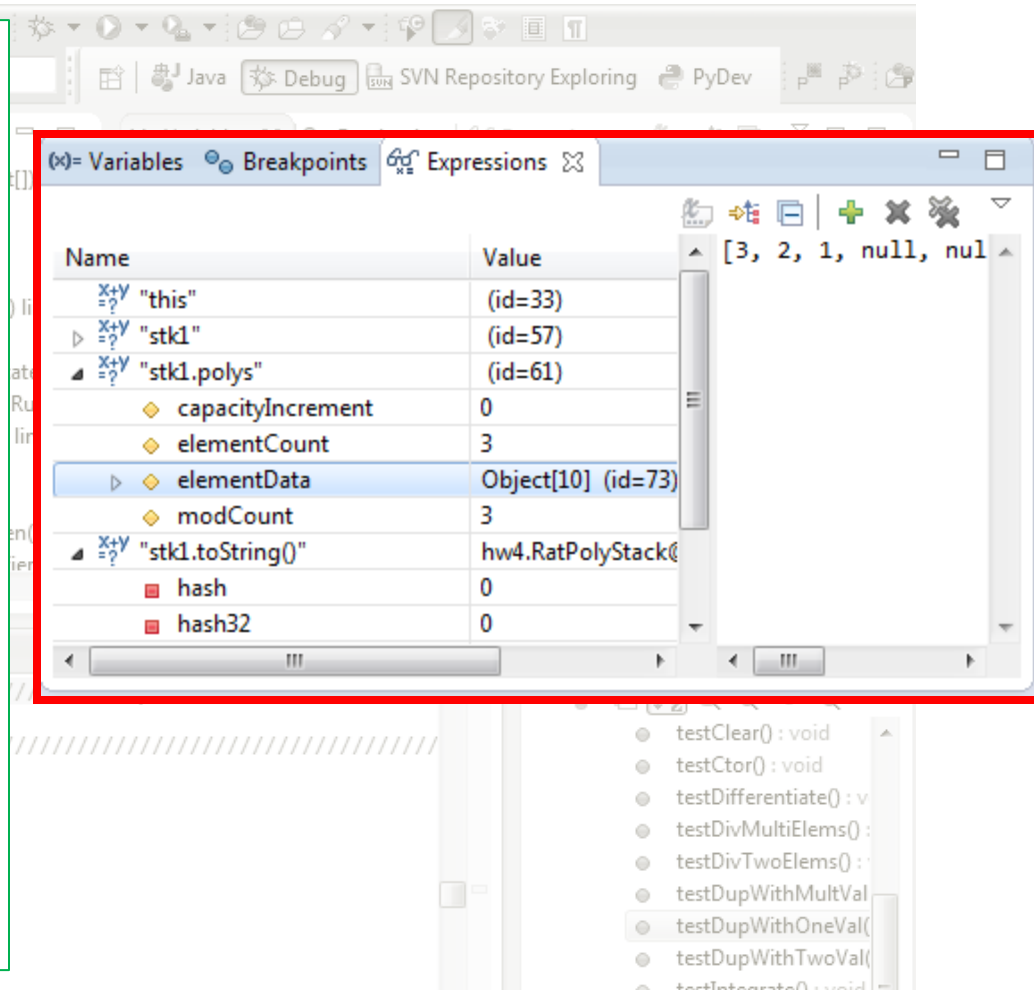
# Eclipse Debugging

## 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



162     assertStackTs(stk1, "1123");

# 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 and highlighted with a red border. It displays a table of variables and their values. The 'this' variable has a value of (id=33). The 'stk1' variable has a value of (id=57). The 'stk1.polys' variable has a value of (id=61) and is expanded to show its contents: capacityIncrement (0), elementCount (3), elementData (Object[10] (id=73)), and modCount (3). The 'stk1.toString()' variable has a value of hw4.RatPolyStack(). The 'hash' and 'hash32' variables have values of 0. The Expressions window also has a toolbar with icons for adding, deleting, and refreshing expressions. A green box highlights the delete icon in the toolbar.

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

```
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");
}
```

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

# Eclipse Debugging

- Demo