

# hw6, BFS, debugging

CSE 331

Section 5 – 10/25/12

Slides by Kellen Donohue

# Agenda

- hw5 to graded in time for feedback to be used on hw6
- hw6 due next week
- Today
  - Asserts
  - hw6 data
  - 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



# Asserts

`assert true; // nothing happens`

`assert false; // program terminates with an  
// assertion failure`

# Asserts

You must manually turn on assert statements for them to be run in your code.

The command line flag is "-ea"

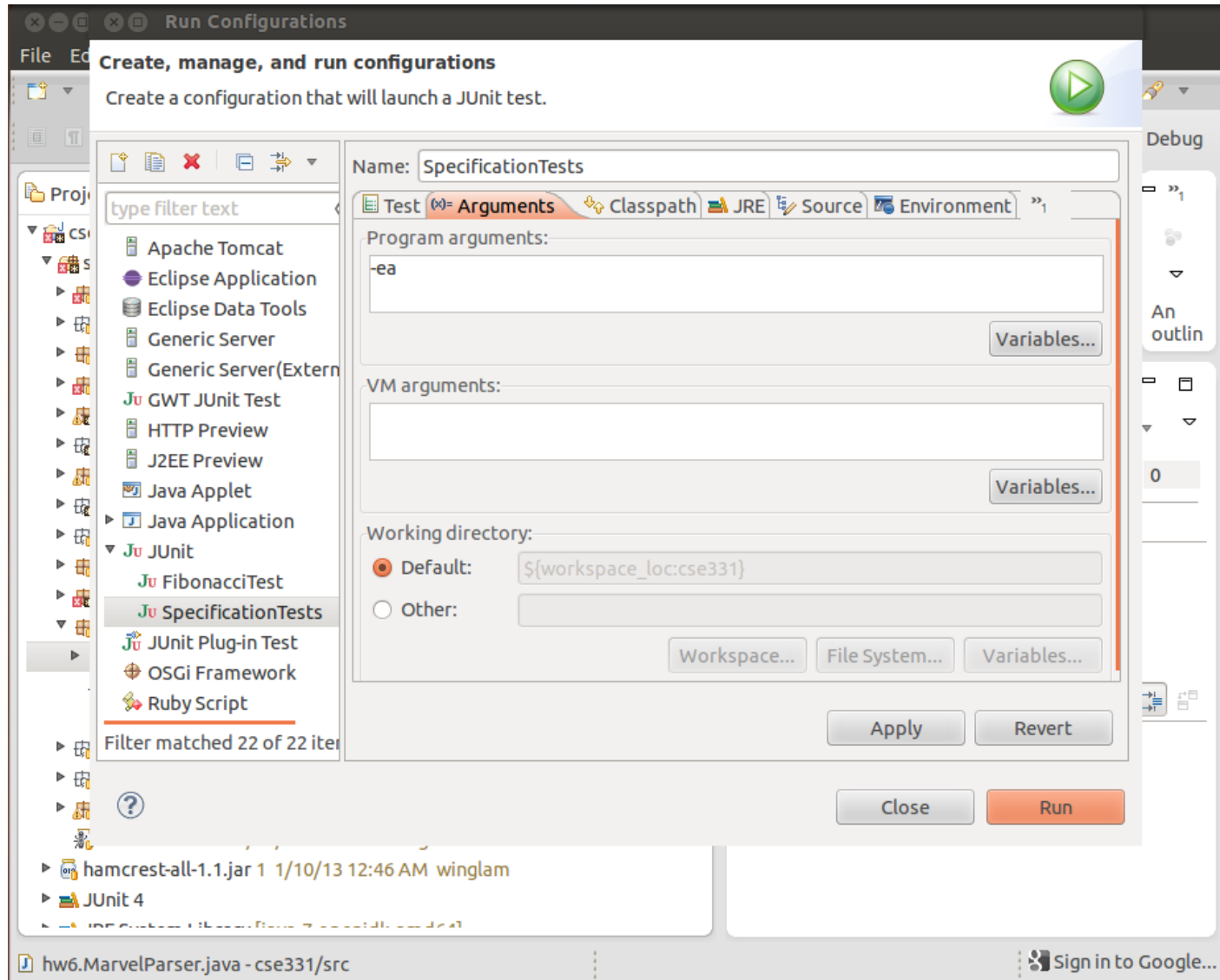
To set command line flags in eclipse:

The .java file you are running -> Run As -> Run Configurations

Arguments tab

Enter "-ea" under 'Program arguments'

# Asserts



# 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



# Homework 6

## The Data

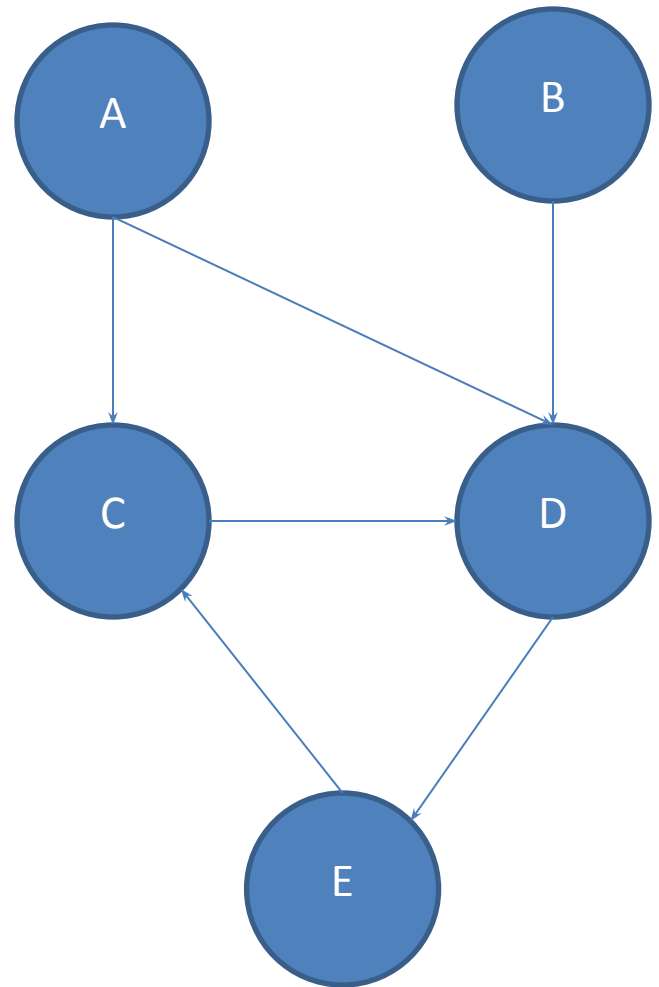
Download from HW6 assignment page

```
labeled_edges.tsv x
"FROST, CARMILLA" "AA2 35"
"KILLRAVEN/JONATHAN R" "AA2 35"
"M'SHULLA" "AA2 35"
"24-HOUR MAN/EMMANUEL" "AA2 35"
"OLD SKULL" "AA2 35"
"G'RATH" "AA2 35"
"3-D MAN/CHARLES CHAN" "M/PRM 35"
"3-D MAN/CHARLES CHAN" "M/PRM 36"
"3-D MAN/CHARLES CHAN" "M/PRM 37"
"HUMAN ROBOT" "WI? 9"
"MARVEL BOY III/ROBER" "WI? 9"
"GORILLA-MAN" "WI? 9"
"3-D MAN/CHARLES CHAN" "WI? 9"
"VENUS II" "WI? 9"
"HUMAN ROBOT" "AVF 4"
"GORILLA-MAN" "AVF 4"
"JONES, RICHARD MILHO" "AVF 4"
"3-D MAN/CHARLES CHAN" "AVF 4"
"WASP/JANET VAN DYNE " "AVF 4"
"LIBRA/GUSTAV BRANDT" "AVF 4"
"CAPTAIN AMERICA" "AVF 4"
"VENUS II" "AVF 4"
"VENUS II" "AVF 4"
"VENUS II" "AVF 4"
Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 INS
```



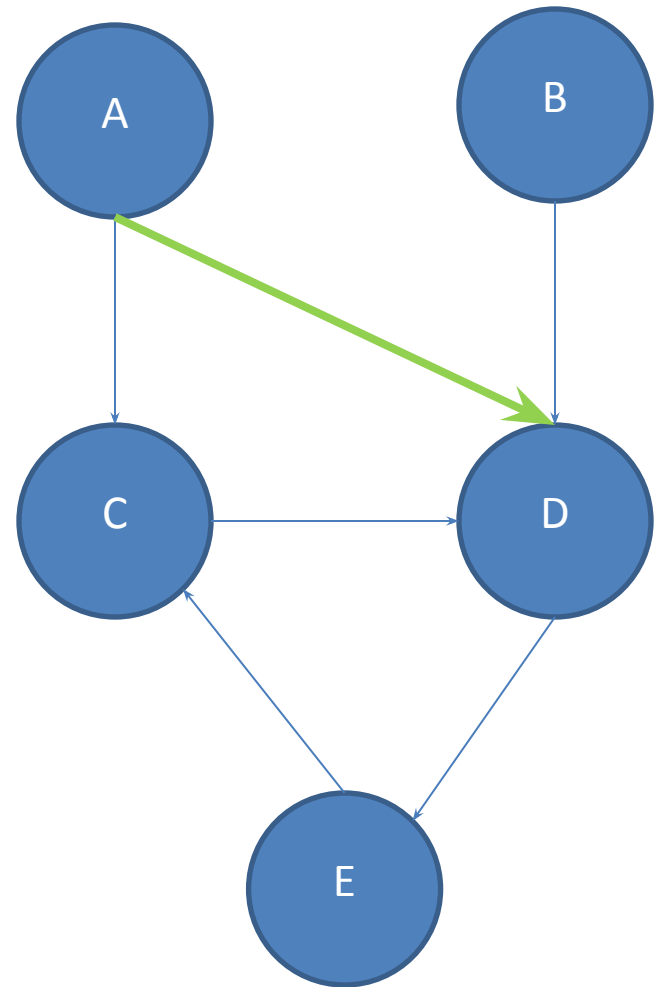
# Graph paths

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



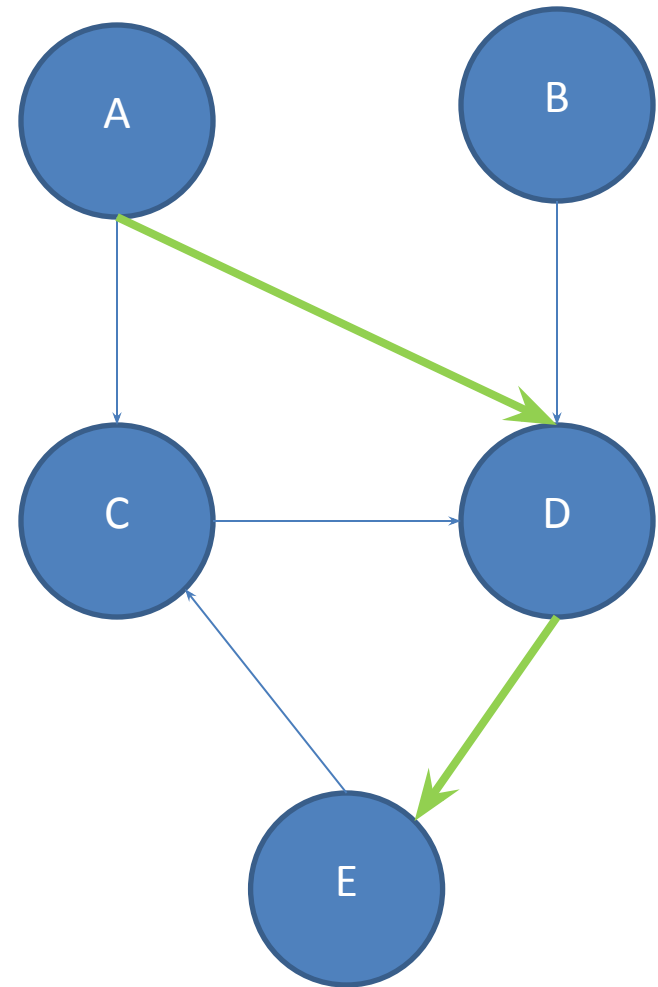
# Graph paths

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



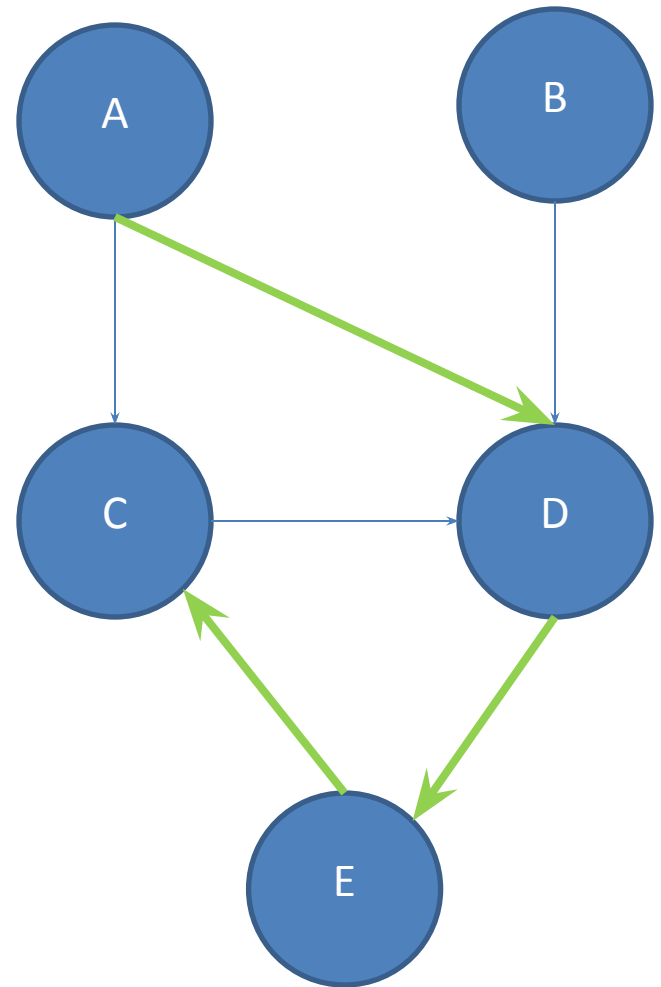
# Graph paths

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



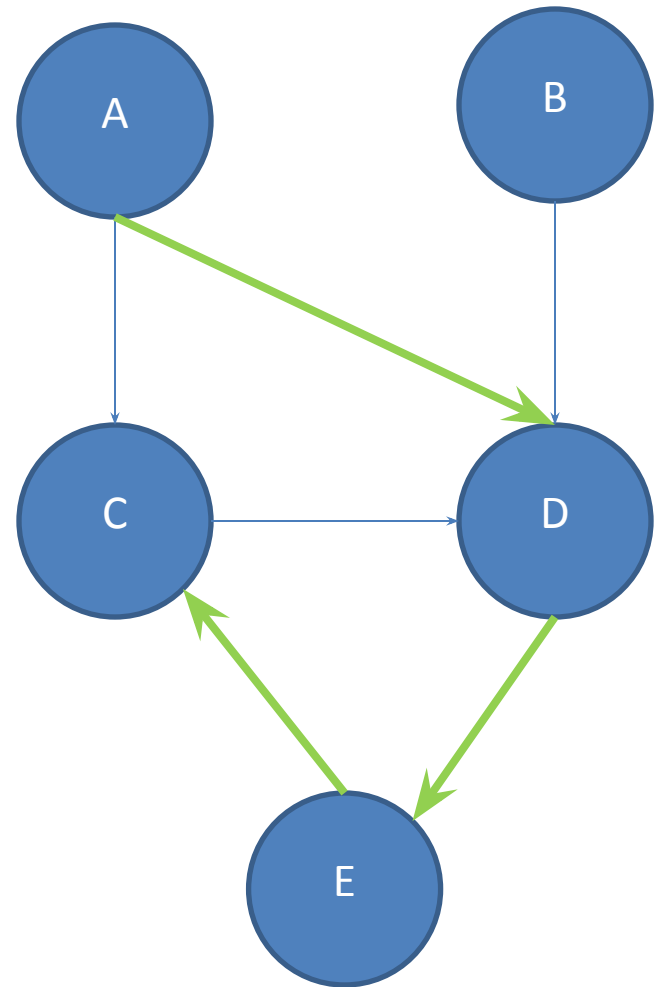
# Graph paths

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



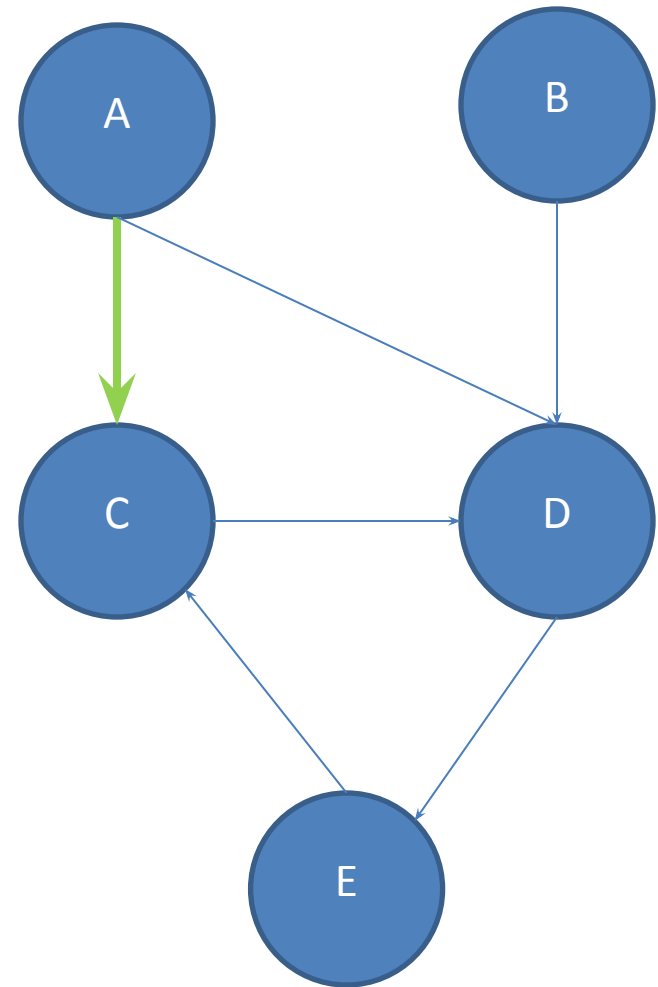
# Graph paths

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



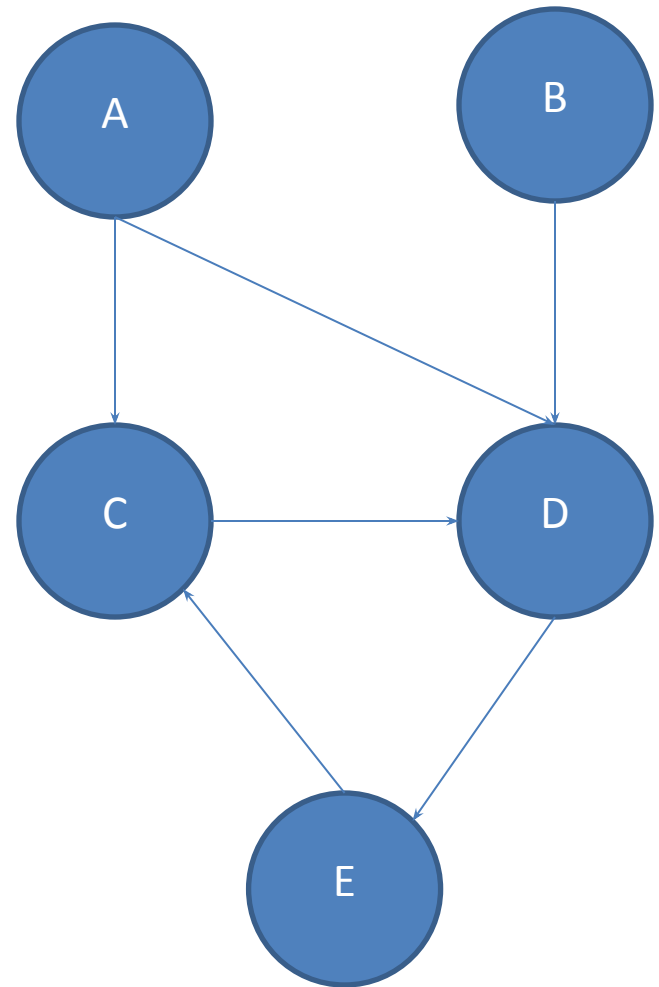
# Graph paths

- 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



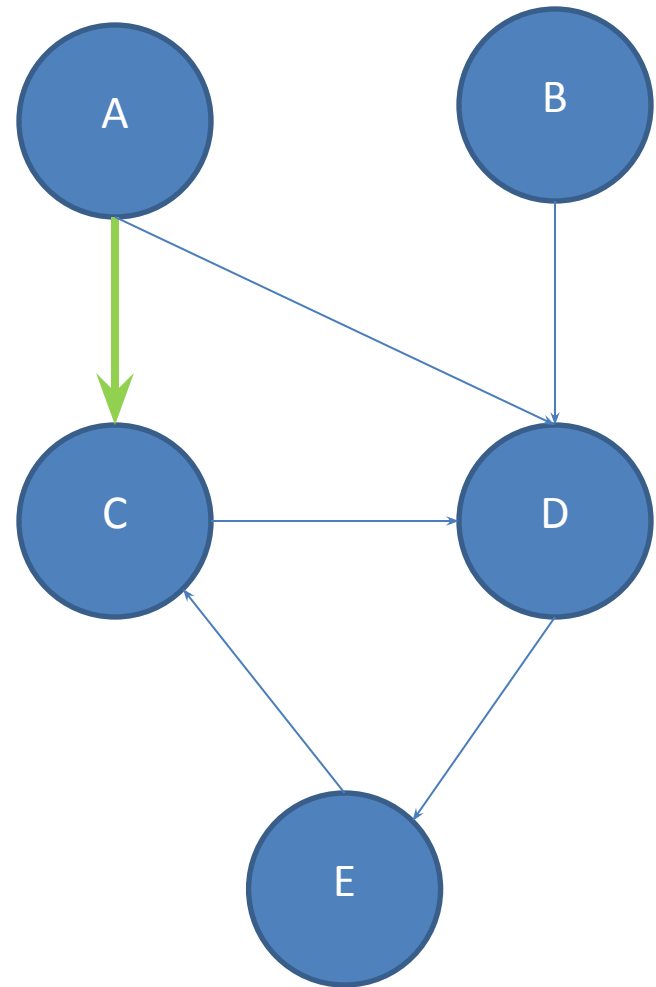
# Graph paths

- 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

## Pseudo code

Put 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 a path from start node to goal node

Return false

# Breadth-first search

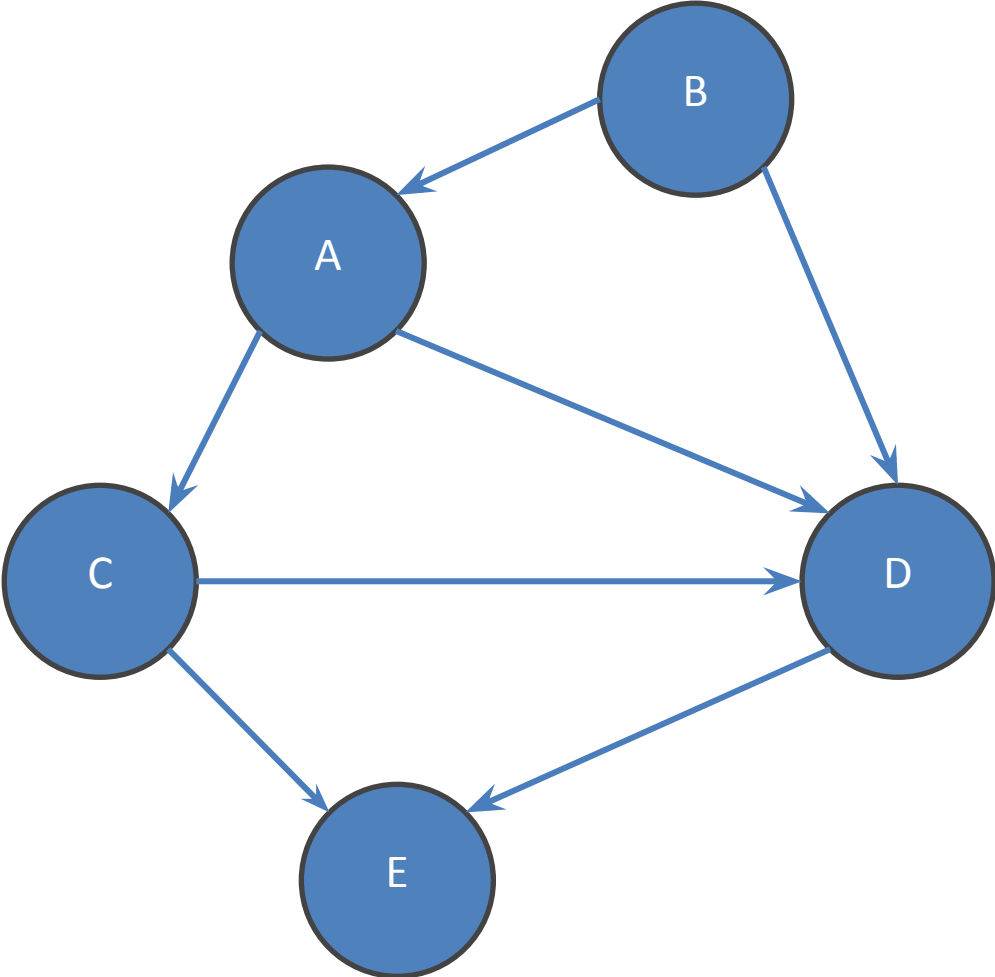
We often want to find the shortest path between two nodes

- Google Maps
- Optimal route through a maze

# Breadth-first search

Queue

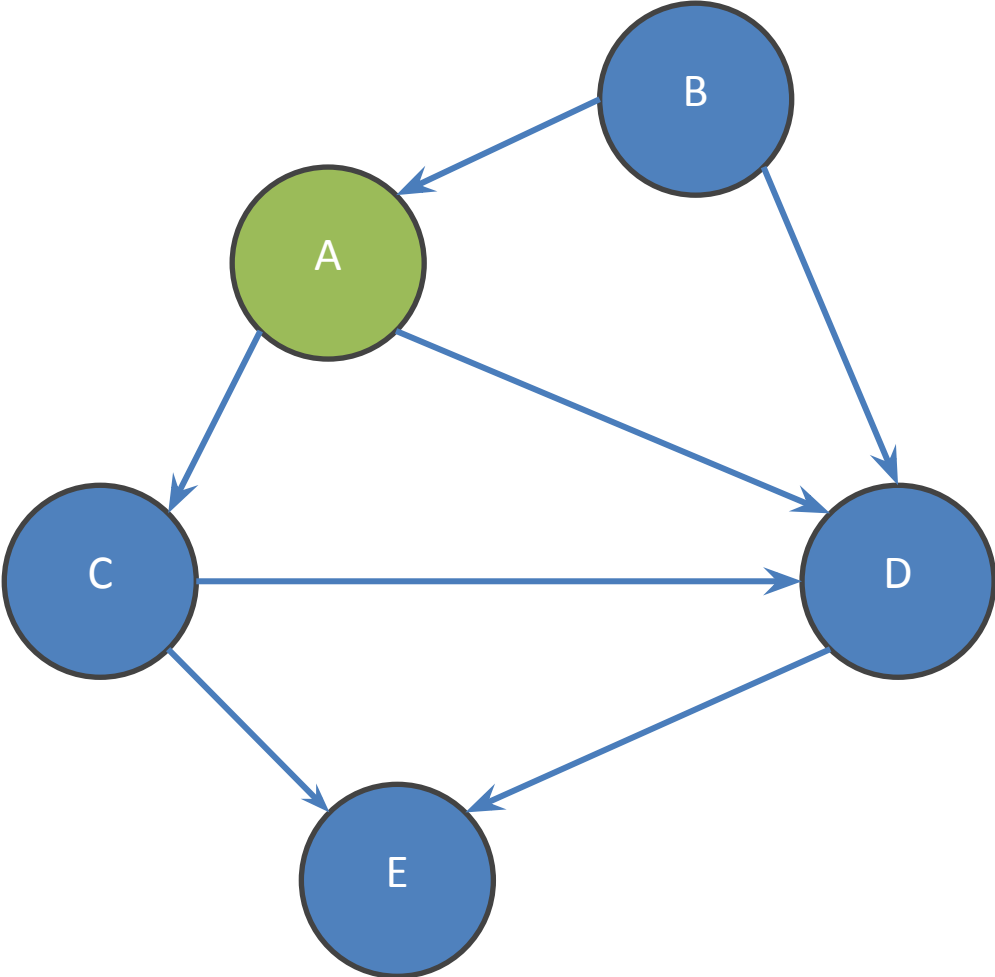
< <



# Breadth-first search

Queue

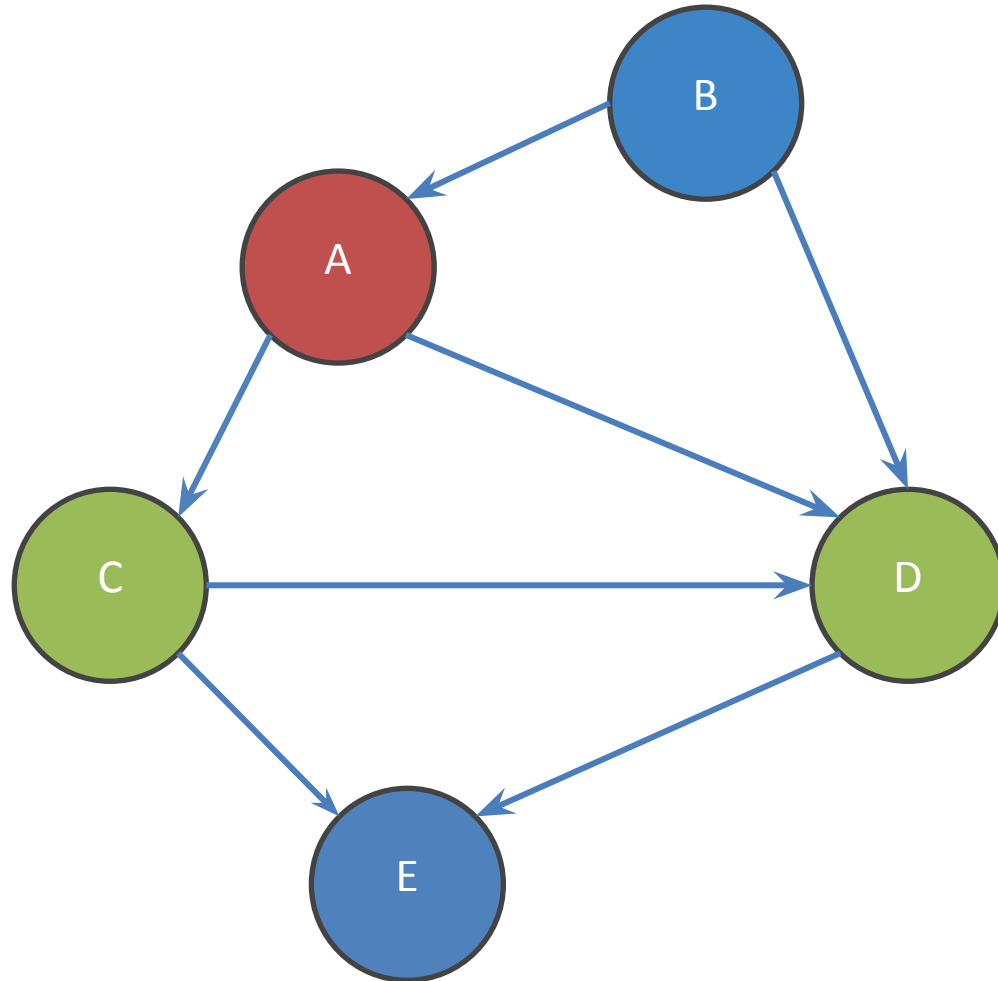
< A <



# Breadth-first search

Queue

< C D <



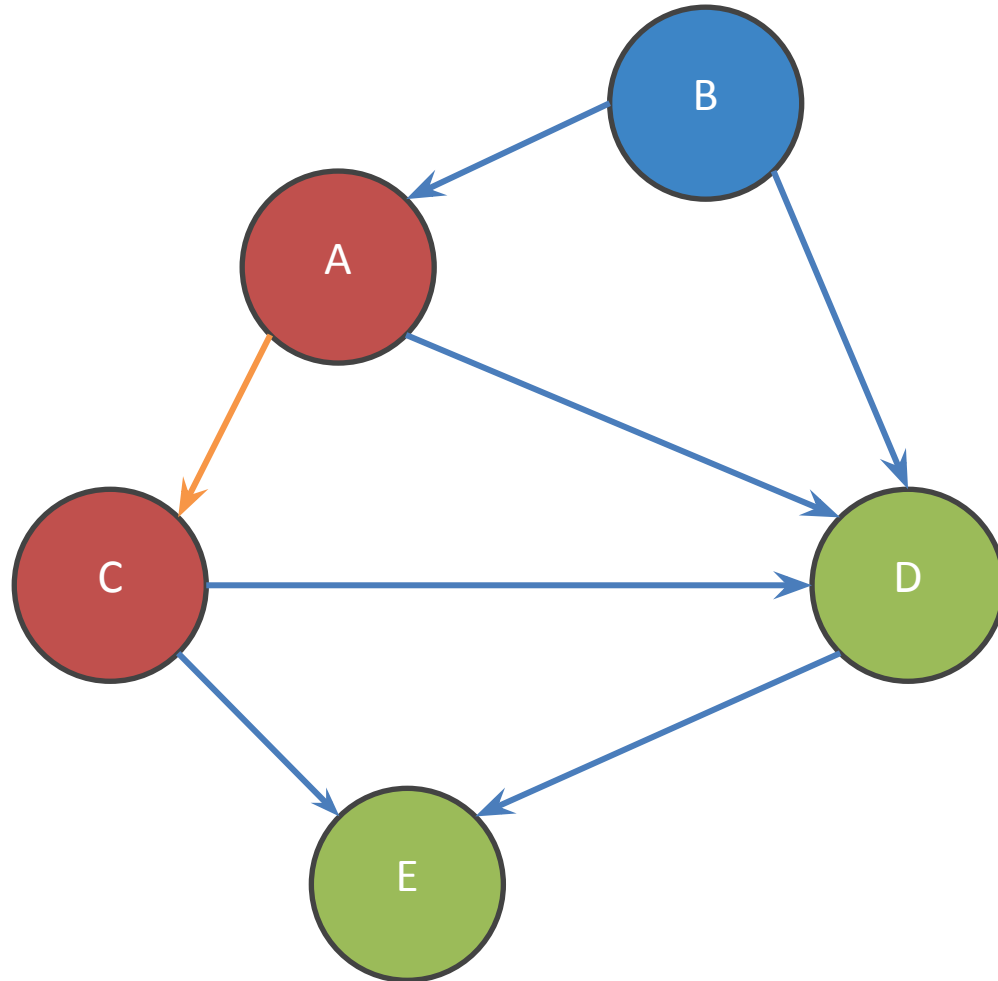
# Breadth-first search

Queue

< D E <

Reach C by path

A -> C



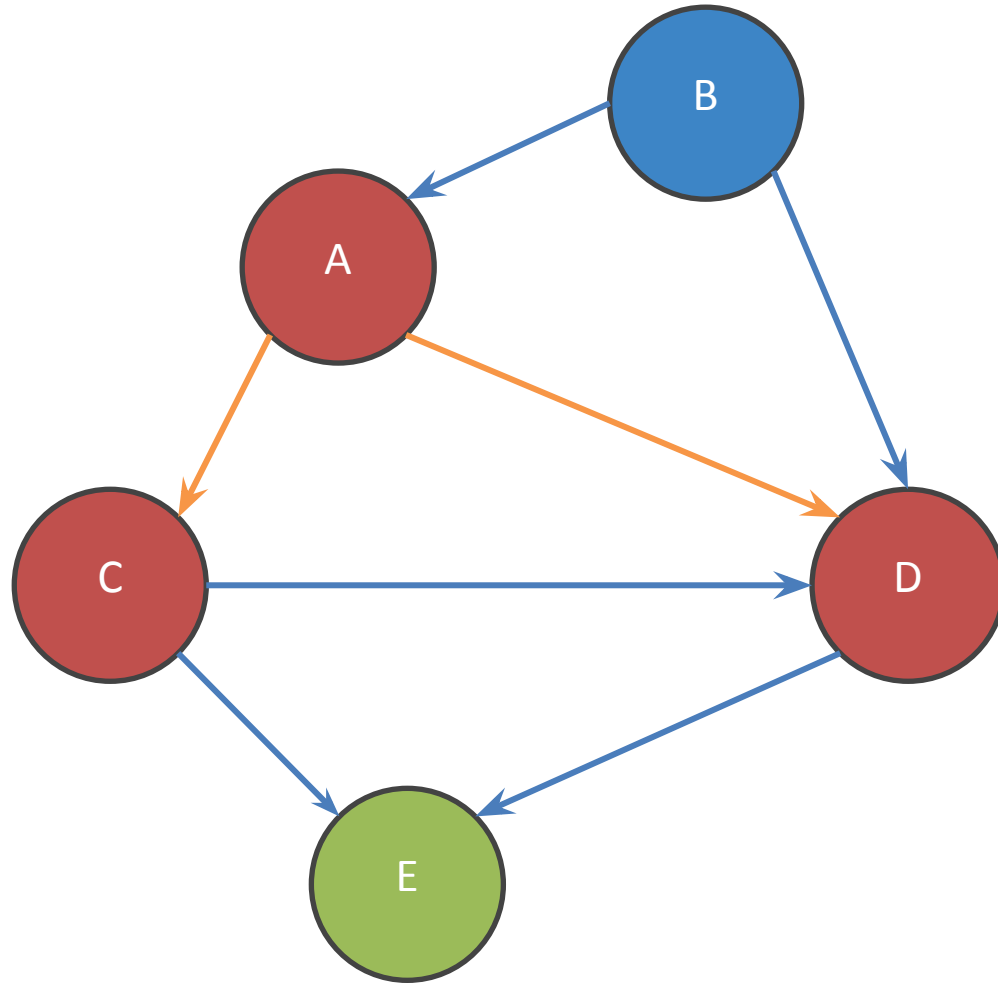
# Breadth-first search

Queue

< E <

Reach D by path

A -> D



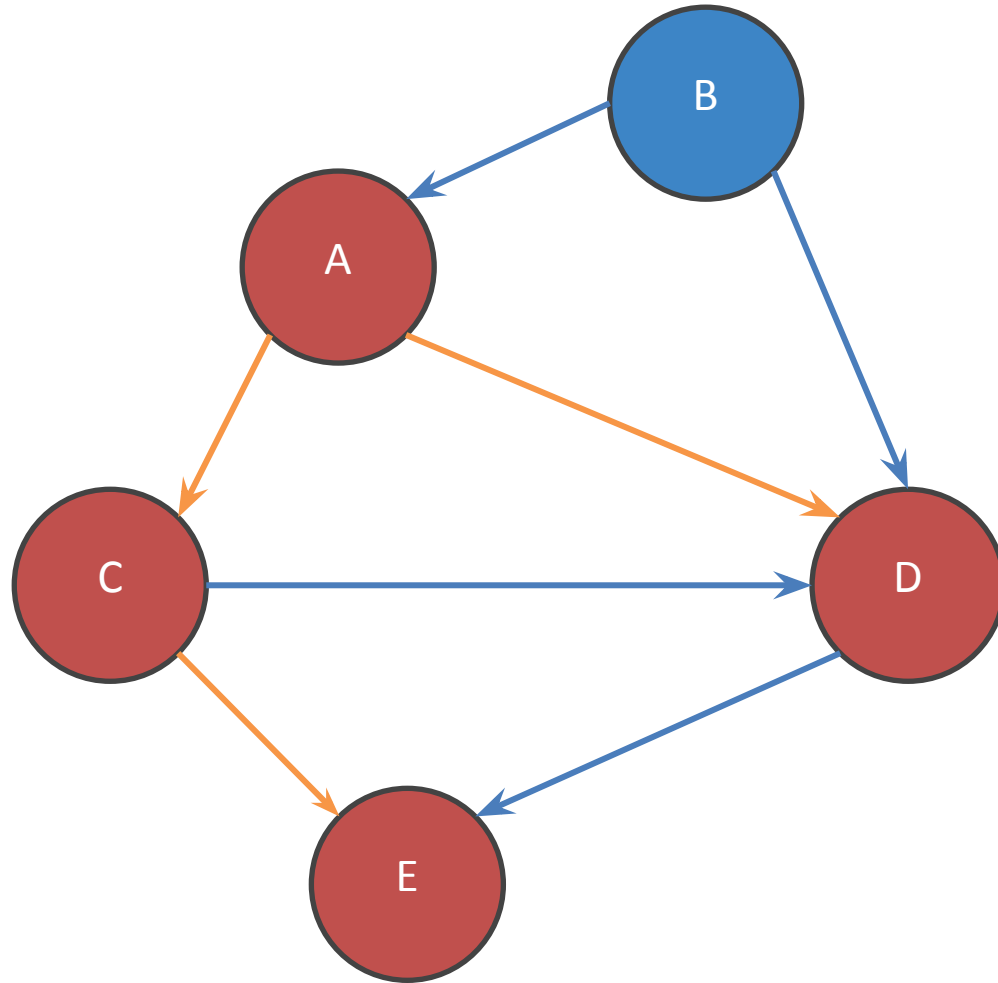
# Breadth-first search

Queue

< <

Reach E by path

A -> C -> E



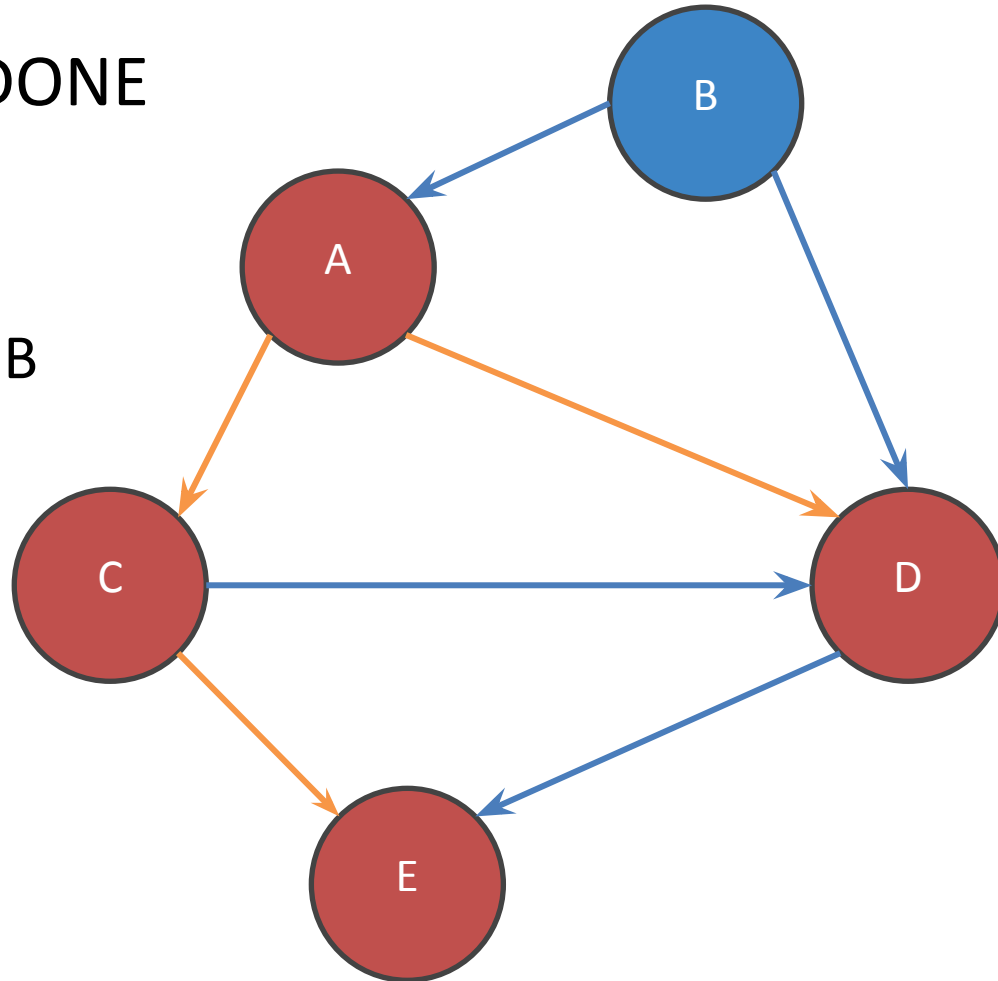


# Breadth-first search

Queue empty: DONE

< <

No path from A to B



# Breadth-first search

## **Guaranteed to find shortest-path**

- In number of nodes
- Not lowest cost path if edges have cost

Breadth First Search uses a Queue. Change to a Stack to change it to a 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

Eclipse has a great debugger!

- Complicated, hidden features
- I'll demo, 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 contains various icons for file operations, running, and debugging. The 'Debug' console shows a stack trace of method calls, 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 'Variables' view shows a table with the following data:

Name	Value
this	RatPolyStackTest (id=33)

The 'RatPolyStackTest.java' editor shows the source code with line 157 highlighted:

```
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 shows a list of test methods:

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

# Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar includes standard development icons. Below it, the 'Quick Access' search bar is visible. The main workspace is divided into several panels:

- Debug Console:** Shows a stack trace of the current execution. The top frame is `DelegatingMethodAccessorImpl.invoke(Object, Object[])` at line 1. Other frames include `Method.invoke`, `FrameworkMethod$1.runReflectiveCall()` at line 45, `FrameworkMethod$1(ReflectiveCallable).run()` at line 15, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate()` at line 20, `BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf`, `BlockJUnit4ClassRunner.runChild`, `BlockJUnit4ClassRunner.runChild`, `ParentRunner$3.run()` at line 231, `ParentRunner$1.schedule` at line 60, `BlockJUnit4ClassRunner(ParentRunner<T>).runChildren`, and `ParentRunner<T>.access$000`.
- Variables Panel:** Shows a table with two columns: 'Name' and 'Value'. The only entry is `this` with the value `RatPolyStackTest (id=33)`.
- Code Editor:** Displays the source code for `RatPolyStackTest.java`. Line 57 is highlighted in green, and a breakpoint (a small circle) is set on this line. A tooltip is overlaid on the editor, explaining how to set a breakpoint.
- Outline Panel:** Shows the class structure of the project.

**Breakpoint Tooltip:**

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

The screenshot displays the Eclipse IDE interface. At the top, the toolbar contains a green bug icon, which is highlighted with a green box. A text box with a black border and white background is overlaid on the toolbar, containing the text: "Click the Bug icon to run in Debug mode. Otherwise your program won't stop at your breakpoints." Below the toolbar, the "Debug" console is visible, showing a list of stack frames from the Java runtime. The main editor window shows the source code for "RatPolyStackTest.java". The code includes a test method "testDupWithOneVal()" with several assertions and stack operations. Line 157, "RatPolyStack stk1 = stack("3");", is highlighted in green. To the right of the editor, the "Outline" view shows a list of methods, with "testDupWithOneVal()" selected. The "Expressions" view at the top right shows the value "RatPolyStackTest (id=33)".

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");
}
```

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

# Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. At the top, the toolbar contains several icons for controlling the program's execution, which are highlighted with a green box. A text box with a black border is overlaid on the toolbar, containing the text: "Controlling your program while debugging is done with these buttons".

The main workspace is divided into several panes:

- Debug Console:** Located on the left, it shows a stack of frames from the current thread. The top frame is `DelegatingMethodAccessorImpl.invoke(Object, Object[])` at line 1. Other frames include `Method.invoke`, `FrameworkMethod$1.runReflectiveCall()` at line 45, `FrameworkMethod$1(ReflectiveCallable).run()` at line 15, `FrameworkMethod.invokeExplosively`, `InvokeMethod.evaluate()` at line 20, `BlockJUnit4ClassRunner.runLeaf`, `BlockJUnit4ClassRunner.runChild`, `ParentRunner$3.run()` at line 231, `ParentRunner$1.schedule` at line 60, and `ParentRunner.access$000`.
- Variables View:** Located on the right, it shows a table with a header "Name" and a single entry "t" with a green dot next to it.
- Code Editor:** The bottom-left pane shows the source code of `RatPolyStackTest.java`. Line 157 is highlighted in green, indicating the current execution point. The code includes a `@Test` annotation and a `testDupWithOneVal()` method that creates a `RatPolyStack` object, duplicates it, and asserts its state.
- Outline View:** Located on the bottom-right, it shows a list of methods in the current class, including `testClear()`, `testCtor()`, `testDifferentiate()`, `testDivMultiElems()`, `testDivTwoElems()`, `testDupWithMultVal`, `testDupWithOneVal()` (which is selected), `testDupWithTwoVal()`, and `testIntegrate()`.

# Eclipse Debugging

The screenshot displays the Eclipse IDE interface during a debugging session. The top toolbar features a green box around the play, pause, and stop icons. The Debug console on the left shows a stack trace of the current execution. The Variables view on the right shows the current object being debugged. The main editor shows the source code of `RatPolyStackTest.java` with a breakpoint at line 157.

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

```
Debug
  DelegatingMethodAccessorImpl.invoke(Object, Object[]) line:
  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

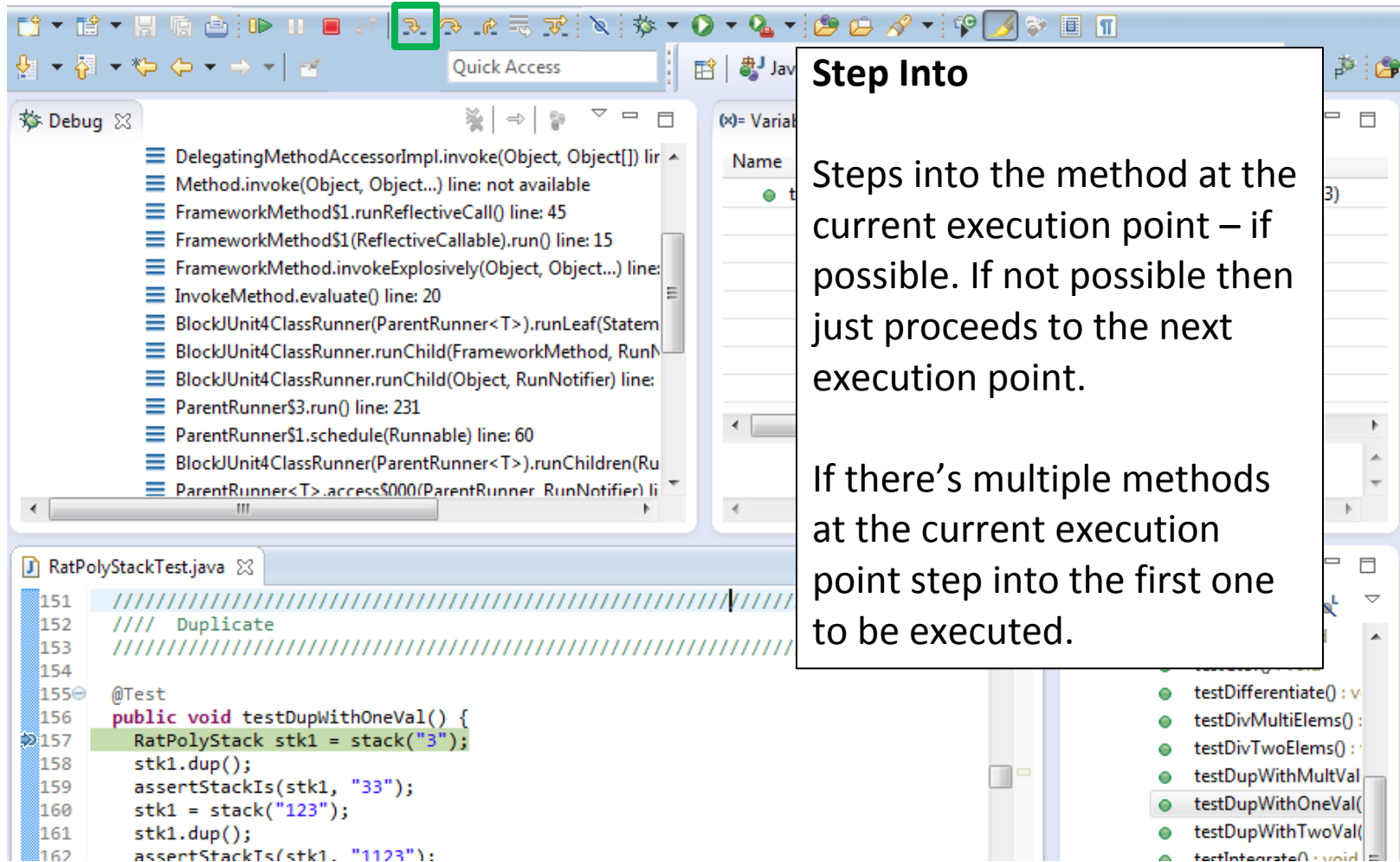
Variables
  Name
  this RatPolyStackTest (id=33)

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");

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



# Eclipse Debugging



**Step Into**

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

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

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

# Eclipse Debugging

The screenshot shows the Eclipse IDE interface. At the top, the toolbar contains various icons, with the 'Step Over' icon (a circular arrow) highlighted with a green box. Below the toolbar is the 'Quick Access' search bar. The main area is divided into several panes. On the left, the 'Debug' console shows a stack trace of the current execution point, with the top line being 'DelegatingMethodAccessorImpl.invoke(Object, Object[])'. In the center, the 'RatPolyStackTest.java' source file is open, showing a Java method 'testDupWithOneVal()' with a breakpoint set at line 157. The code in the source file is as follows:

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

On the right side of the IDE, a list of test methods is visible, including 'testDivWithTwoElems()', 'testDivTwoElems()', 'testDupWithMultVal()', 'testDupWithOneVal()', 'testDupWithTwoVal()', and 'testIntegrate(): void'.

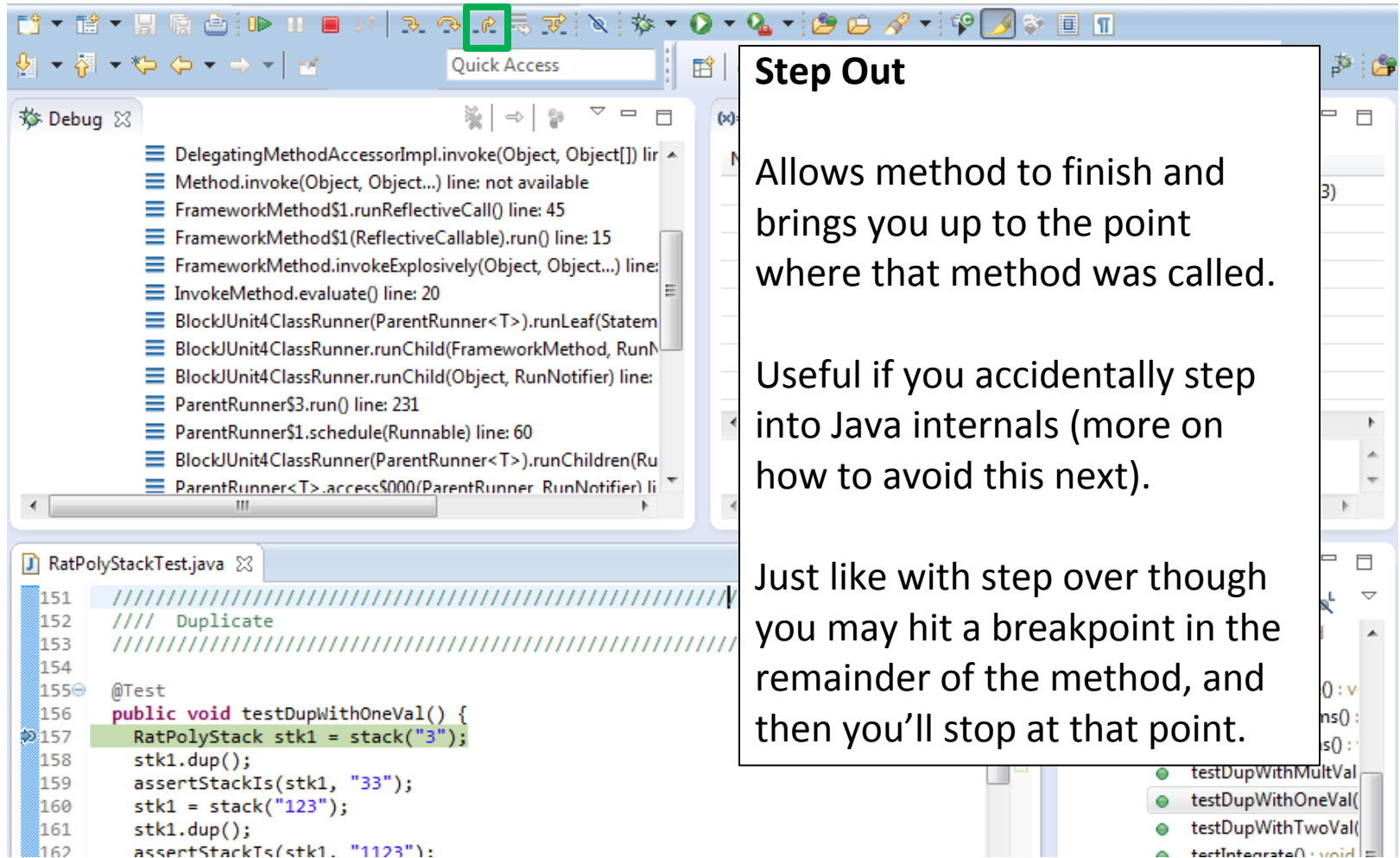
## Step Over

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**

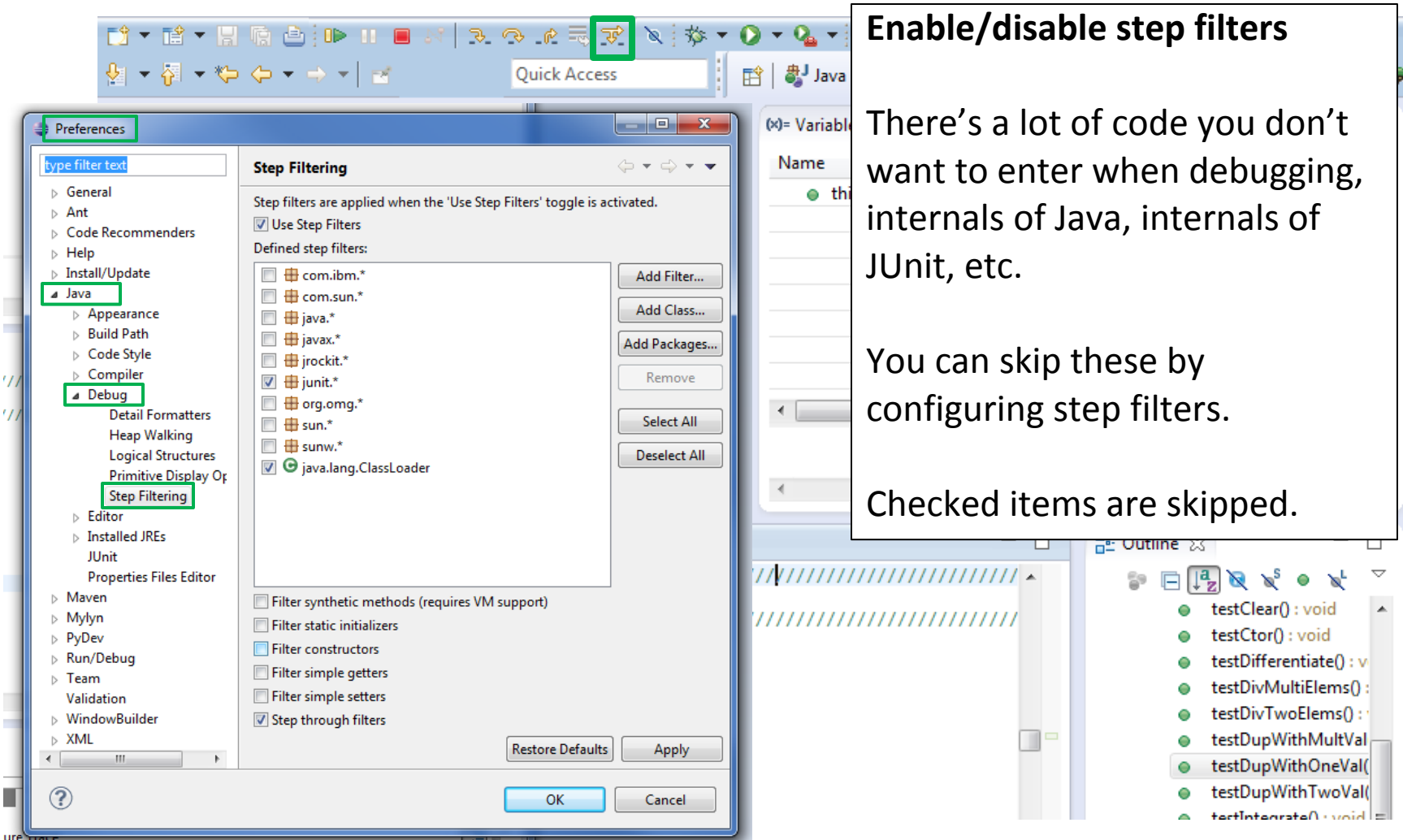
Allows method to finish and brings you up to the point where that method was called.

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.

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

# Eclipse Debugging



**Enable/disable step filters**

There's a lot of code you don't want to enter when debugging, internals of Java, internals of JUnit, etc.

You can skip these by configuring step filters.

Checked items are skipped.

# Eclipse Debugging

The screenshot shows the Eclipse IDE interface. The top toolbar contains various icons for file operations and debugging. Below the toolbar is the 'Quick Access' search bar. The main workspace is divided into several panes. On the left, the 'Debug' console displays a stack trace with a green border around it. The stack trace lists several methods, including 'DelegatingMethodAccessorImpl.invoke', 'Method.invoke', 'FrameworkMethod\$1.runReflectiveCall', 'FrameworkMethod\$1(ReflectiveCallable).run', 'FrameworkMethod.invokeExplosively', 'InvokeMethod.evaluate', 'BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf', 'BlockJUnit4ClassRunner.runChild(FrameworkMethod, RunN', 'BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line:', 'ParentRunner\$3.run() line: 231', 'ParentRunner\$1.schedule(Runnable) line: 60', 'BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(Ru', and 'ParentRunner<T>.access\$000(ParentRunner RunNotifier) li'. In the center, the 'Variables' pane shows a table with 'Name' and 'Value' columns. On the right, the 'Stack Trace' pane lists several test methods: 'testDifferentiate(): v', 'testDivMultiElems():', 'testDivTwoElems():', 'testDupWithMultVal', 'testDupWithOneVal(', 'testDupWithTwoVal(', and 'testIntegrate(): void'. At the bottom, the 'RatPolyStackTest.java' 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");
```

## 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, Debug, and Breakpoints. 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`. A breakpoint is set at line 157, which is highlighted in green. The code is:

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

# 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, Debug, and Breakpoints. The main window is divided into several panes:

- Variables View (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`. A breakpoint is set at line 157, which is highlighted in green. The code snippet is:

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



# Eclipse Debugging

There's a powerful right-click menu.

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

The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar includes icons for file operations, running, and debugging. The main window is divided into several panes: a code editor at the bottom left showing a Java method `testDupWithOneVal()` with line numbers 154-162; a 'Variables' view in the center showing a tree structure with 'this' (RatTermTest) and 't' (containing 'coeff' and 'expt'); and a 'Breakpoints' and 'Expressions' view at the top right. A right-click context menu is open over the 'expt' variable, listing 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, and its keyboard shortcut 'Ctrl+Shift+N' is visible. 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

## 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");
```

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

# Eclipse Debugging

## Breakpoints Window

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

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

The screenshot shows the Eclipse IDE interface. The Breakpoints window is open, displaying a list of breakpoints for the project. The breakpoints are:

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

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

The code editor shows the following code:

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

# Eclipse Debugging

## Enabled/Disabled Breakpoints

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

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

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

The screenshot shows the Eclipse IDE interface during a debug session. The Breakpoints view is open, displaying a list of breakpoints for the current project. The breakpoint for `RatPolyStackTest [line: 162] - testDupWithOneVal()` is highlighted with a green box, and its checkbox is unchecked, indicating it is disabled. The other breakpoints are checked. The Breakpoints view also shows options for hit count, suspension type (Suspend thread or Suspend VM), and conditional execution (Suspend when 'true' or Suspend when value changes). The conditional expression `x == 6` is entered in the text field below the dropdown menu. The background shows the source code editor with the `testDupWithOneVal()` method highlighted, and the Package Explorer on the right showing the project structure.

# 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.

```
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");

```

- 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

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

# 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. The breakpoint at line 159 is selected and highlighted. Below the list, the configuration for this breakpoint is shown, including the condition 'x == 6' and the option 'Suspend when 'true''.

Breakpoints List:

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

Configuration for the selected breakpoint:

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

Method List:

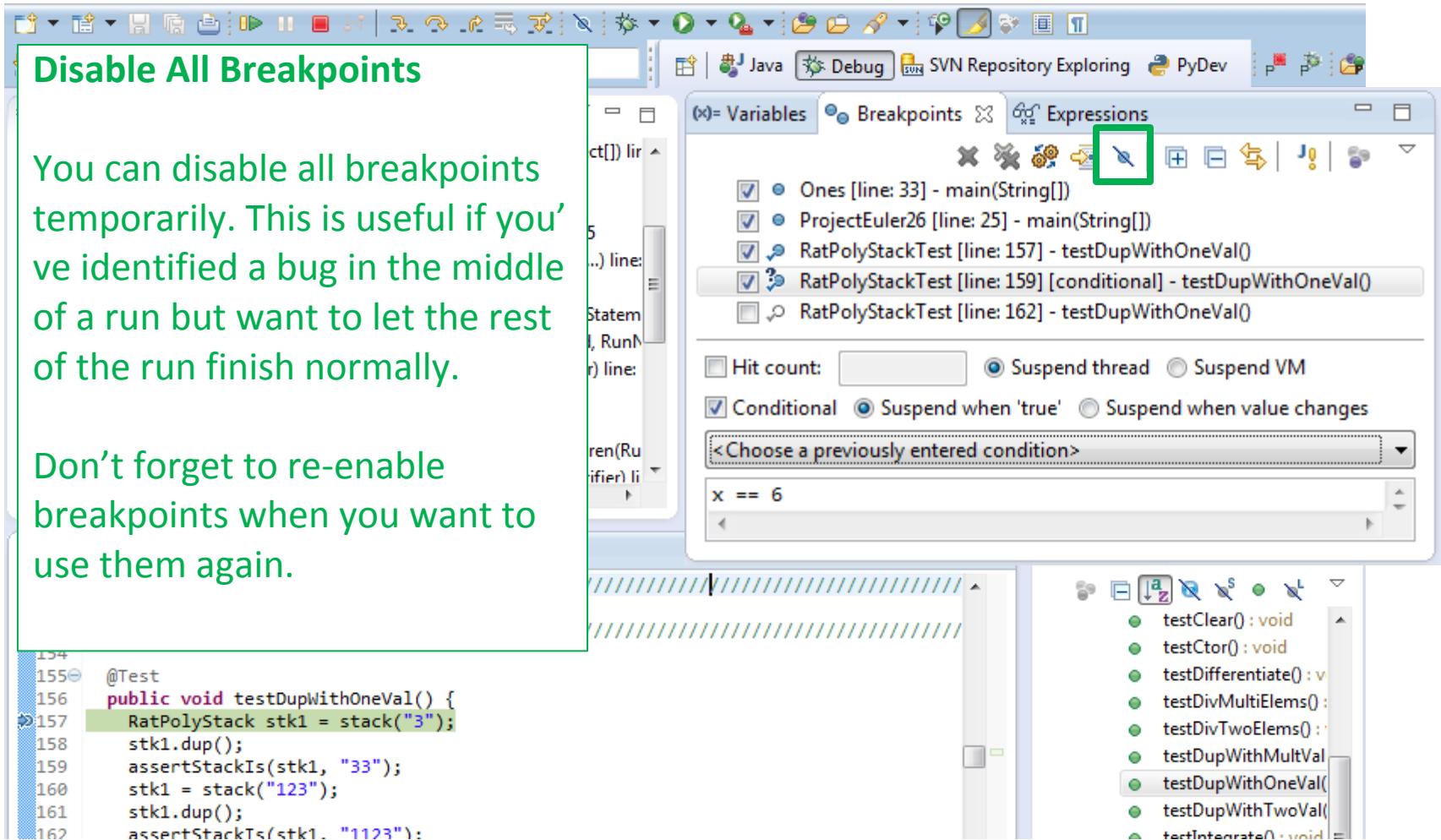
- testClear(): void
- testCtor(): void
- testDifferentiate(): void
- testDivMultiElems(): void
- testDivTwoElems(): void
- testDupWithMultiVal(): void
- testDupWithOneVal(): void**
- testDupWithTwoVal(): void
- testIntegrate(): void

# Eclipse Debugging

## Disable All Breakpoints

You can disable all breakpoints temporarily. This is useful if you've identified a bug in the middle of a run but want to let the rest of the run finish normally.

Don't forget to re-enable breakpoints when you want to use them again.



The screenshot shows the Eclipse IDE interface during a debug session. The Breakpoints view is open, displaying a list of breakpoints for the current project. A red 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`.

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

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

# Eclipse Debugging

## Break on Java Exception

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

```
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");
```

Breakpoints window showing a list of 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 the selected breakpoint:

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

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

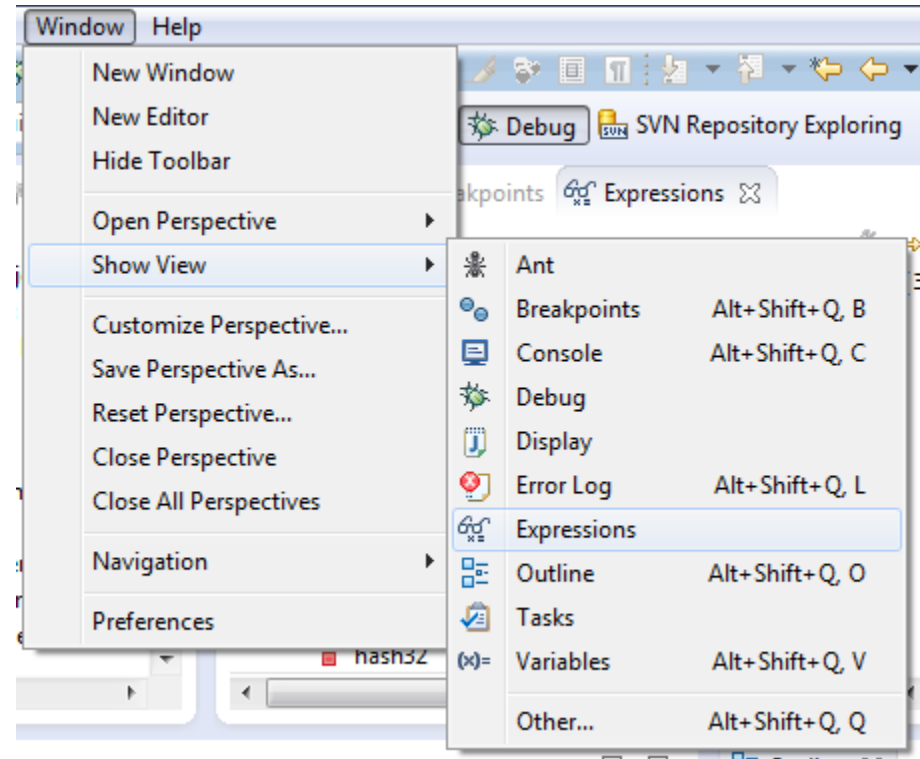


# Eclipse Debugging

## 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

The screenshot shows the Eclipse IDE interface with the Expressions Window open. The window displays a table of variables and their values. The 'Name' column lists variables like 'this', 'stk1', 'stk1.polys', and 'stk1.toString()', along with their fields like 'capacityIncrement', 'elementCount', 'elementData', 'modCount', 'hash', and 'hash32'. The 'Value' column shows the corresponding values, such as '(id=33)', '(id=57)', '(id=61)', and 'Object[10] (id=73)'. The 'elementData' field is expanded, showing an array of 10 elements: [3, 2, 1, null, null, null, null, null, null, null]. The background shows the Eclipse IDE with the 'Debug' mode selected and a list of methods in the bottom right corner.

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

Background methods list:

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

# Eclipse Debugging

## Expressions Window

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

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

```
FrameworkMethod.invokeExplosively(Object, Object...) line: 13
InvokeMethod.evaluate() line: 20
BlockJUnit4ClassRunner(ParentRunner<T>).runLeaf(State
BlockJUnit4ClassRunner.runChild(FrameworkMethod, Ru
BlockJUnit4ClassRunner.runChild(Object, RunNotifier) line:
ParentRunner$3.run() line: 231
ParentRunner$1.schedule(Runnable) line: 60
BlockJUnit4ClassRunner(ParentRunner<T>).runChildren(
ParentRunner<T>.access$000(ParentRunner RunNotifie
```

```
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(): v
- testDivMultiElems():
- testDivTwoElems():
- testDupWithMultVal
- testDupWithOneVal(
- testDupWithTwoVal(
- testIntegrate(): void

# Eclipse Debugging

- Demo