

# Debugging

CMSC 202

# Overview

- Debugging
- Error Types
- Stack Traces
- Tracing
  - Via print statements
  - Java logging facilities
- Eclipse debugger

# Debugging

- Debugging is a ***methodical process*** of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected
- Debugging tends to be ***harder when*** various subsystems are ***tightly coupled***, as changes in one may cause bugs to emerge in another

—Wikipedia

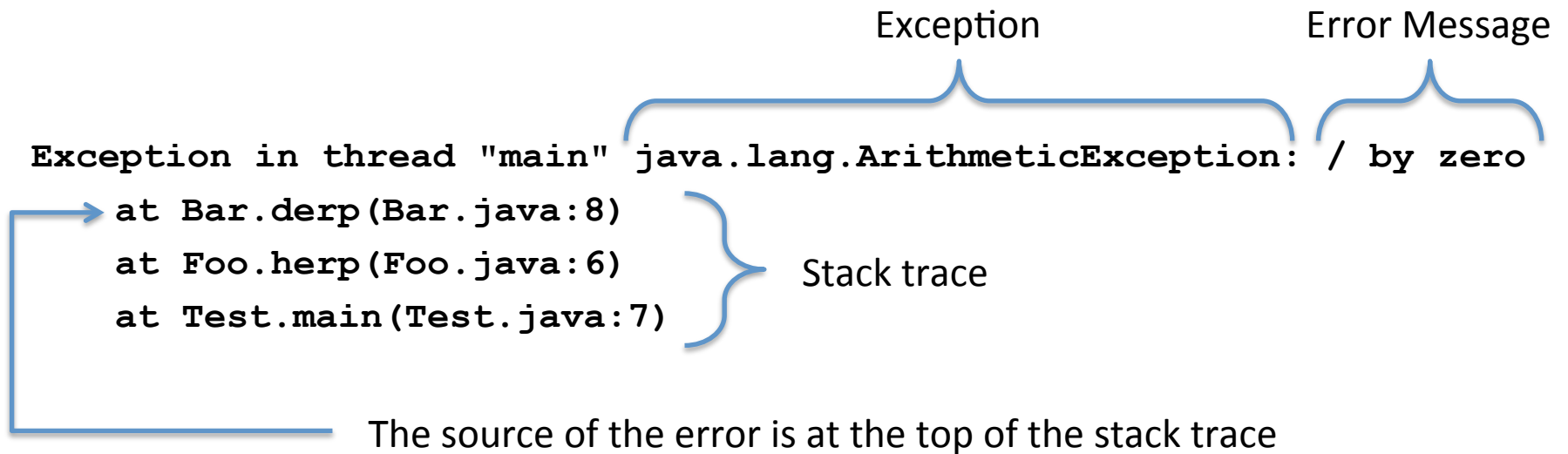
# Error/Bug Types

- Compile time errors
  - Bugs caught by compiler
    - Syntax errors
- Runtime error that terminates program
  - Bugs caught by the runtime
    - NullPointerException
    - ArrayIndexOutOfBoundsException
- Runtime error that does not terminate program
  - Bugs not caught by the runtime, hopefully caught by developer
    - Logic errors

# Stack Trace

- A stack trace is a dump of the active stack frames at a given point in time
- In Java, when the JVM detects an error condition (such as trying to invoke a method on a null reference) it raises an exception resulting in a stack trace
- This stack trace shows you where the error originated from and how it came to be executed

# Reading a Stack Trace



# Errors Inside a Java Provided Class

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 12, Size: 0
    at java.util.ArrayList.RangeCheck(ArrayList.java:547)
    at java.util.ArrayList.remove(ArrayList.java:387)
    at Bar.derp(Bar.java:12)
    at Foo.herp(Foo.java:6)
    at Test.main(Test.java:7)
```

Scan from the top down looking for the first reference to your code, that's usually a good place to start looking

Sometimes errors originate in a class that's provided by Java

This is where the error manifested itself, though the cause is almost always in your code up the stack

# Tracing with Print Statements

- Print (a.k.a. tracing, probing ) debugging is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process

—Wikipedia



# Tracing with Print Statements

- Once you've identified the location of the error (by reading the stack trace), start printing out variables
- This can be as basic as simply printing out all local variables, members, objects, parameters, etc. using `System.out.println()`
- Having a working `toString()` method for all of your objects really aids in this debugging process

# Java Logging API

- Another more sophisticated option to simple print statements would be to use a logging framework
- Java has a built-in logging API that can be used to quickly turn logging statements on and off

# Declaring & Using a Logger

- To add logging capabilities to a class, you typically add a logger as a static member of each class you'd like to add logging to...

```
private static final Logger LOGGER =  
    Logger.getLogger(<class name>.class.getCanonicalName());
```

- To log a message to the console, simply use the logger like so...

```
LOGGER.info("message to log");
```

# Logging Levels

## Levels

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)

## Methods

```
LOGGER.severe("message");
```

```
LOGGER.warning("message");
```

```
LOGGER.info("message");
```

```
LOGGER.config("message");
```

```
LOGGER.fine("message");
```

```
LOGGER.finer("message");
```

```
LOGGER.finest("message");
```

# Running the App

- When you run the app, you should see your logging messages on stderr
- In Eclipse they show up in red in the Console window like so...

```
This is Baz's toString method
Apr 5, 2011 9:29:37 PM foo.bar.Baz main
SEVERE: message
Apr 5, 2011 9:29:37 PM foo.bar.Baz main
WARNING: message
Apr 5, 2011 9:29:37 PM foo.bar.Baz main
INFO: message
Apr 5, 2011 9:29:37 PM foo.bar.Baz toString
INFO: blah
```

# Logging Levels

- You can easily change the logging level for all classes within your application like so...

```
Logger.getLogger("").setLevel(Level.<LEVEL>);
```

- When the logger is set at a given level, all logging statements at that level or higher are printed out
- The following special values turn it completely on or off...

Level.*ALL*

Level.*OFF*

# Turning Logging On/Off

- You can add the following code to your main() to support turning logging on/off at the command line...

```
String levelStr = System.getProperty("debug");
Level level = (levelStr == null) ? Level.OFF : Level.parse(levelStr);
Logger.getLogger("").setLevel(level);
for(Handler h : Logger.getLogger("").getHandlers()) {
    h.setLevel(level);
}
```

- Then to turn it on at a given level...

```
% java -Ddebug=SEVERE foo.bar.Baz
% java -Ddebug=INFO foo.bar.Baz
% java -Ddebug=ALL foo.bar.Baz
```

# Debugger

- A special program used to find errors (bugs) in other programs
- A debugger allows a programmer to stop a program at any point and examine and change the values of variables

—Webopedia



# Eclipse Debugger

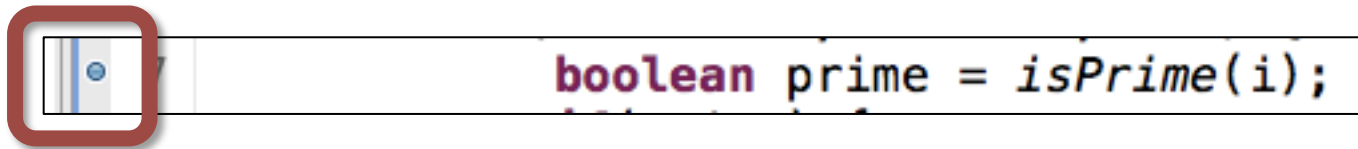
- Eclipse has a built-in perspective that's dedicated to debugging Java code
- To run a program in the debugger, simply right click on the class to run and select...
  - Debug As → Java Application
- Allow Eclipse to open the Debug perspective if it asks
- If you do nothing else, Eclipse will simply run your program just like a normal “Run As”

# Breakpoints

- Breakpoints can be used to pause your program at a certain point
- Once paused, you can examine (and even change) the state of variables
- There are many different ways to break...
  - Line
  - Method
  - Member change
  - Etc.

# Line Breakpoints

- To set a breakpoint on a line, simply double click in the gutter left of the line to stop on
- Once you do so, you'll see a small blue bubble in the gutter like so...



- Notes
  - Simply double click again to toggle the breakpoint off
  - In order to break on a line, there must be an executable statement (e.g. you cannot break on a curly brace)

Stack Frames

Variable/  
Breakpoint  
Tables

Current Line

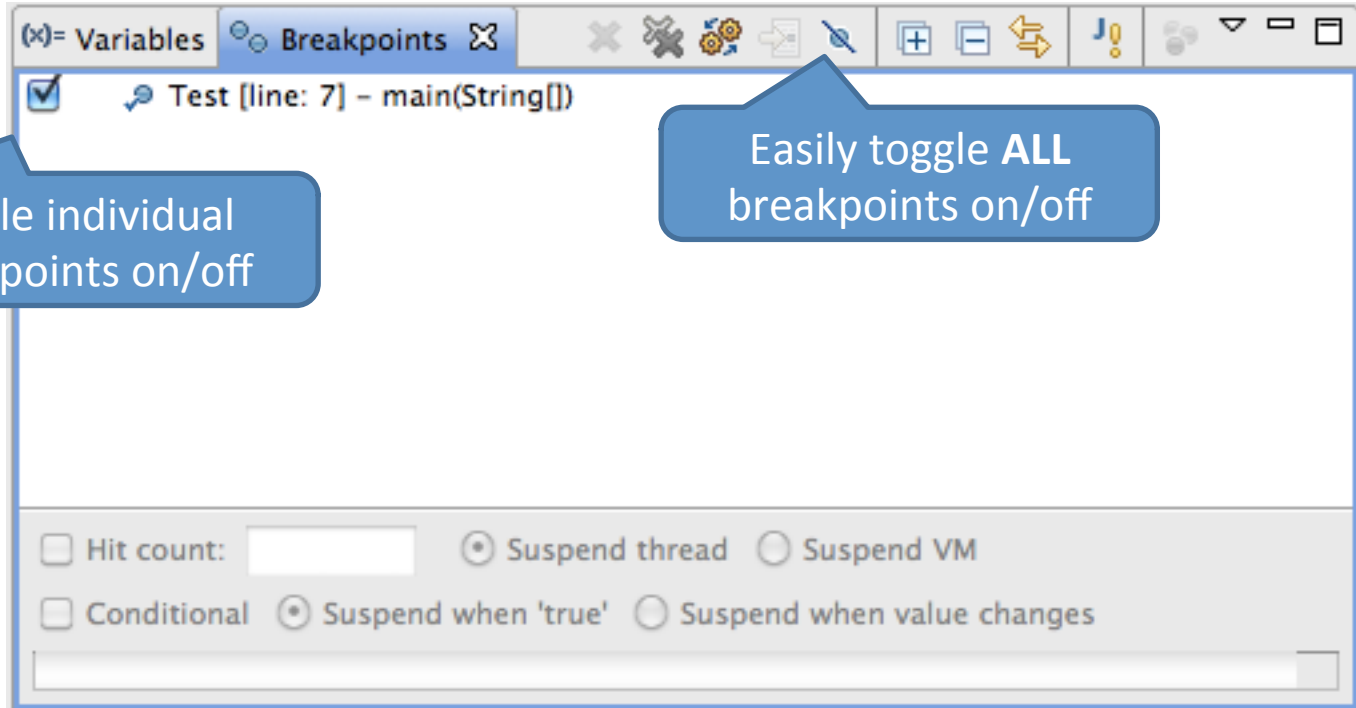
The screenshot shows the Eclipse IDE in a debug session. The top toolbar includes icons for file operations, running, and debugging. The main window is divided into several panes:

- Debug Console:** Shows the execution stack. The top frame is "Test (2) [Java Application]" with a sub-frame "Test at localhost:54107". The current thread is "Thread [main] (Suspended (breakpoint at line 7 in Test))" at "Test.main(String[]) line: 7".
- Variables and Breakpoints:** A table showing the current state of variables and breakpoints. The table has two columns: "Name" and "Value".
- Code Editor:** Displays the source code of "Test.java". Line 7 is highlighted in green, indicating the current execution point. The code is:

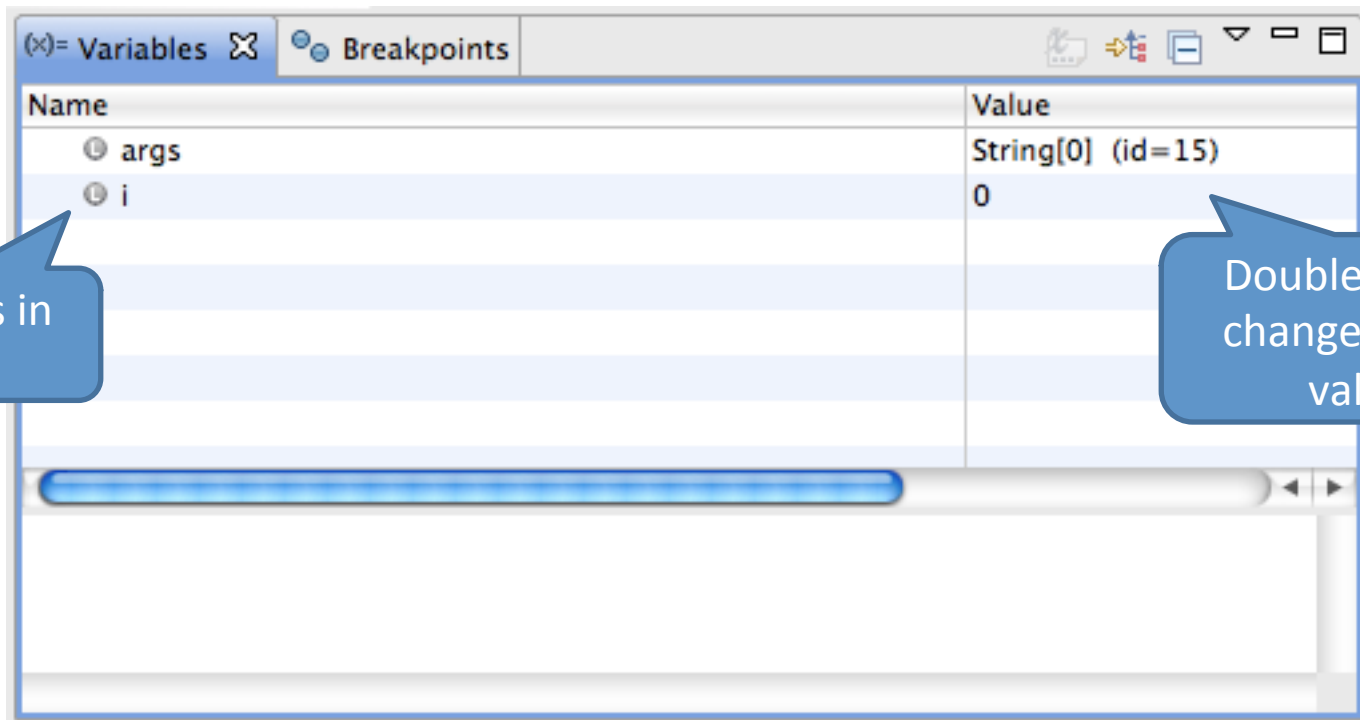
```
1  
2  
3 public class Test {  
4  
5     public static void main(String[] args) {  
6         for(int i = 0; i < 100; i++) {  
7             boolean prime = isPrime(i);
```
- Outline:** Shows the class structure of "Test" with methods "main(String[]): void" and "isPrime(int): boolean".
- Console:** Shows the output of the program, including the date and time: "Test (2) [Java Virtual Machines/1.6.0.jdk/Contents/Home/bin/java (Apr 6, 2011 1:38:43 PM)".

Name	Value
args	String[0] (id=15)
i	0

# Breakpoint View



# Variable View



# Hover to View Variable Values

The screenshot shows an IDE window with two tabs: 'Fraction.java' and 'Test.java'. The 'Test.java' tab is active, displaying the following code:

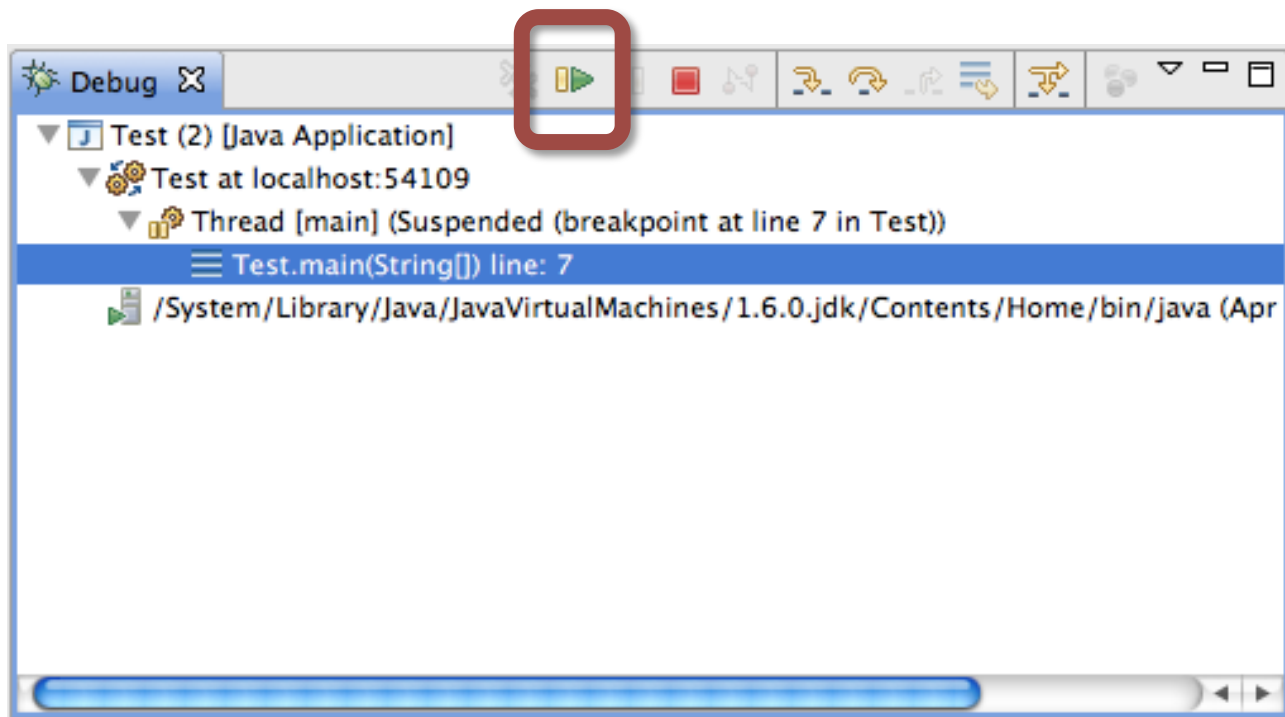
```
3 public class Test {  
4  
5     public static void main(String[] args) {  
6         for(int i = 0; i < 100; i++) {  
7             boolean prime = isPrime(i);  
8             if(prime) {  
9                 System.out.println(i
```

The line `boolean prime = isPrime(i);` is highlighted in green. A mouse cursor is hovering over the variable `prime`, which has triggered a tooltip window. The tooltip window has a title bar that reads `i = 0` and contains the value `0`.

The IDE interface includes a 'Console' tab at the bottom left, a 'Tasks' tab, and an 'Outline' view on the right side. The 'Outline' view shows a tree structure for the 'Test' class with two methods: `main(String[] args)` and `isPrime(int)`.

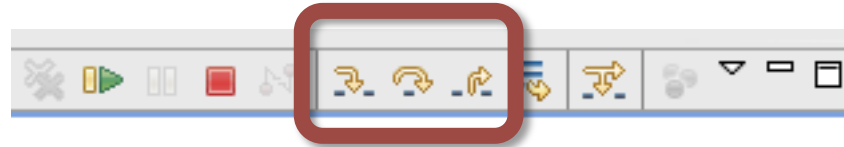
# Continuing Execution

- If you click the resume button in the Debug view, execution will continue until the next breakpoint is encountered





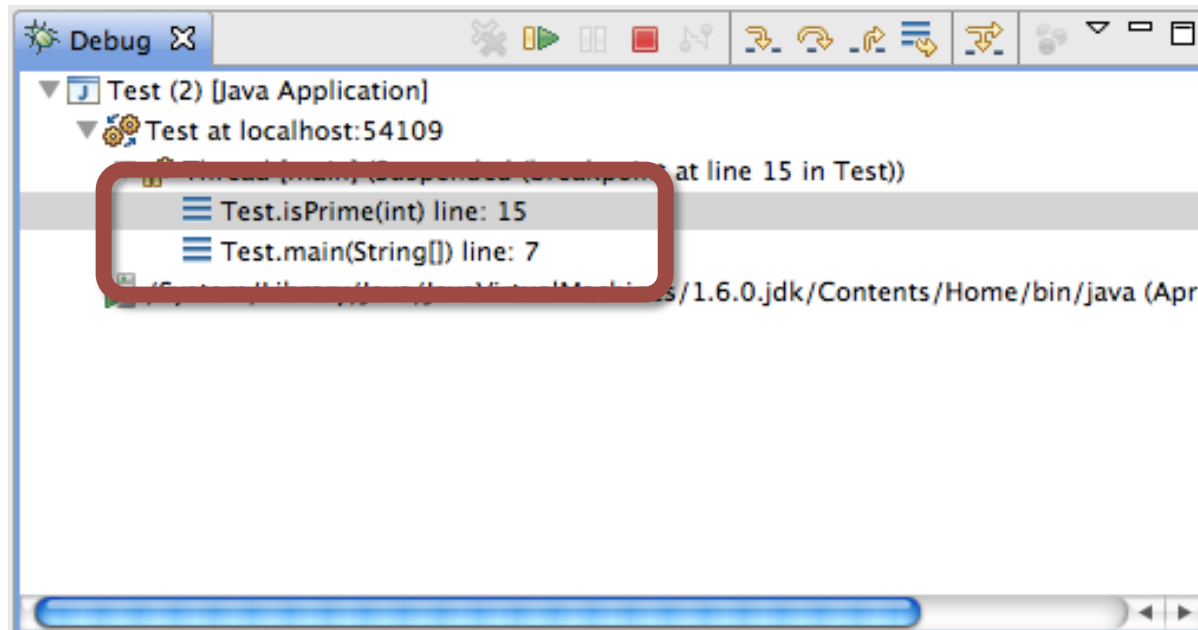
# Stepping



- There several step buttons that allow you to walk through the execution of your code
  - Step Into
    - If the line contains a method call, step into that method and pause execution
  - Step Over
    - Completely execute this line (including any method calls) and pause execution at the next line (in the same method)
  - Step Return
    - Complete the current method and pause execution where the method was called from

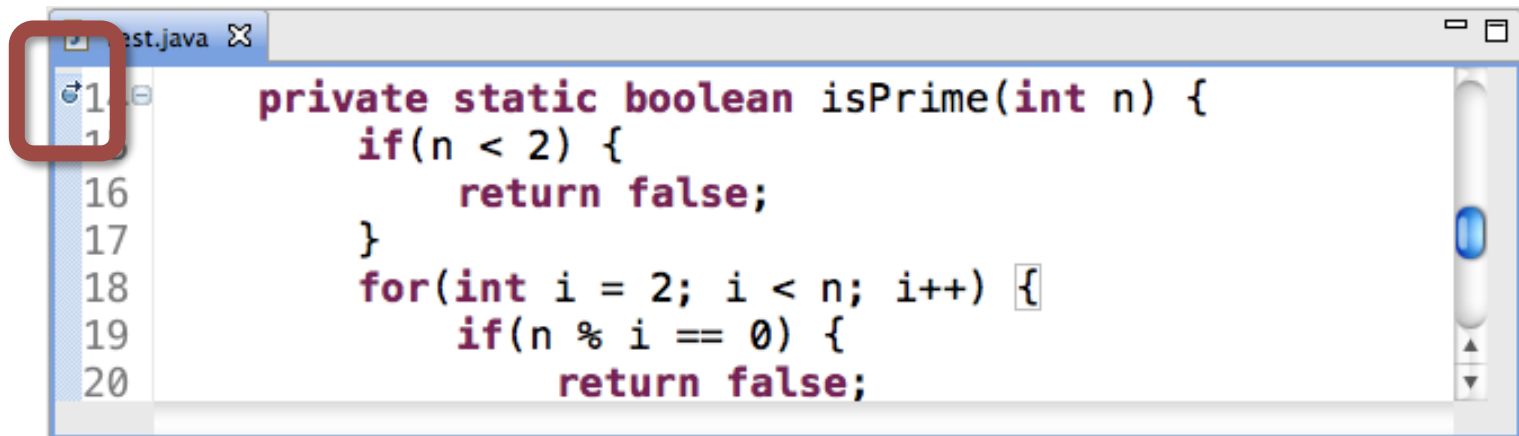
# Stack Frames

- Eclipse's Debug view also shows you stack frames so you can see how you got somewhere
  - Current stack frame is at the top, main should be at the bottom



# Method Breakpoints

- Double clicking on margin next to method will create a method entry breakpoint
  - Right click → Breakpoint Properties... allows you to also set exit breakpoint



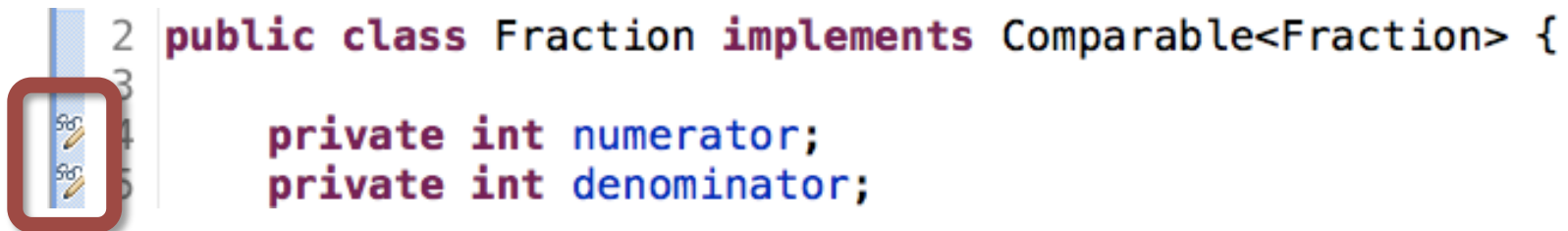
The screenshot shows an IDE window titled "test.java". The code is as follows:

```
14 private static boolean isPrime(int n) {  
15     if(n < 2) {  
16         return false;  
17     }  
18     for(int i = 2; i < n; i++) {  
19         if(n % i == 0) {  
20             return false;  
21     }  
22 }
```

A red square highlights the margin area next to line 14, where a breakpoint icon (a small circle with a plus sign) is visible, indicating that a breakpoint has been set at the start of the `isPrime` method.

# Watching Members

- You can also set breakpoints (also called watch points) to see when a member is being accessed or changed
- Simply double click next to the member and it will set both breakpoints
  - Double click again to toggle off
  - Right click → Breakpoint Properties... to change



```
2 public class Fraction implements Comparable<Fraction> {  
3  
4     private int numerator;  
5     private int denominator;  
}
```

# Additional References

- Java Logging Overview
  - <http://download.oracle.com/javase/1.5.0/docs/guide/logging/overview.html>
- Lars Vogel's Java Logging API Tutorial
  - <http://www.vogella.de/articles/Logging/article.html>
- Lars Vogel's Java Debugging with Eclipse Tutorial
  - <http://www.vogella.de/articles/EclipseDebugging/article.html>