

CSCI165 Computer Science II

Debugging, OOD and JUnit Lab

Please read this document thoroughly *before* asking questions. Figure out what ambiguities need to be addressed before leaving today. There are new concepts being introduced in this document. Be sure to read

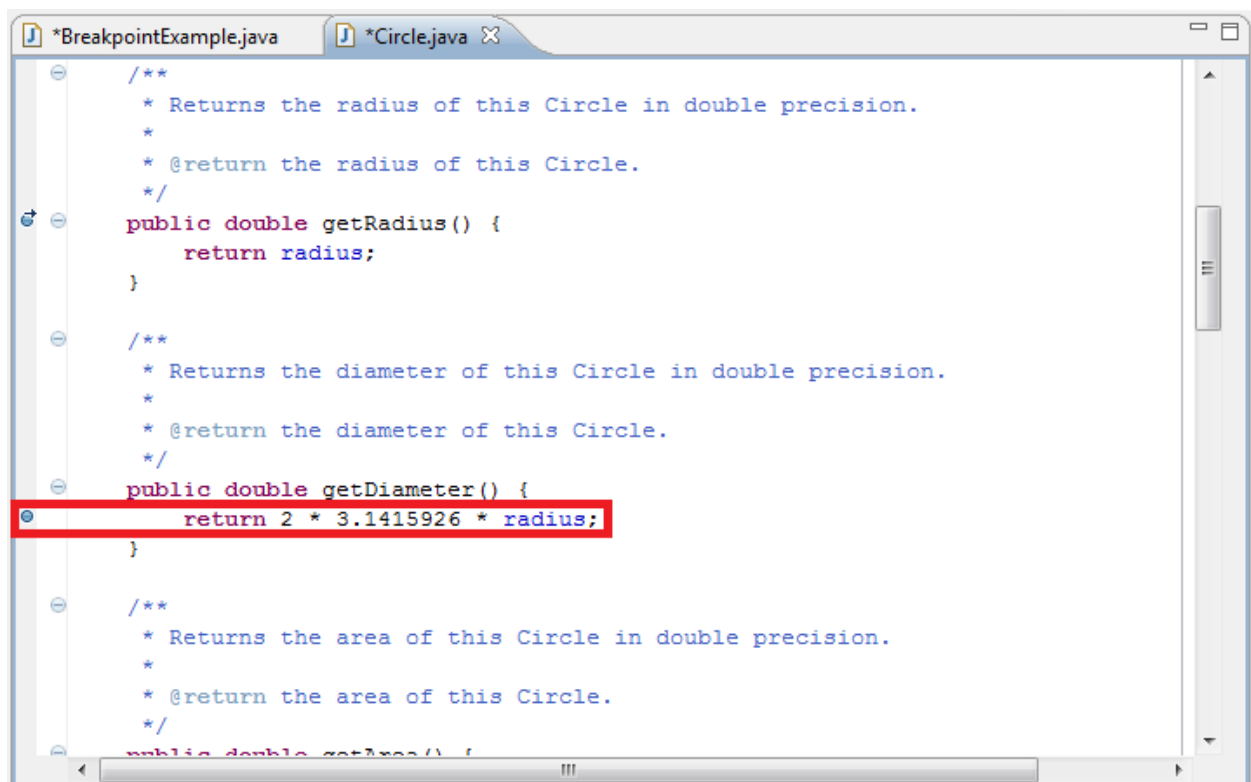
Debugging

A debugger is yet another invaluable tool for a software engineer. It provides a powerful mechanism for peering into the inner workings of complex programs and often comes paired with a user-friendly graphical user interface that makes it easy to use a debugger's important features.

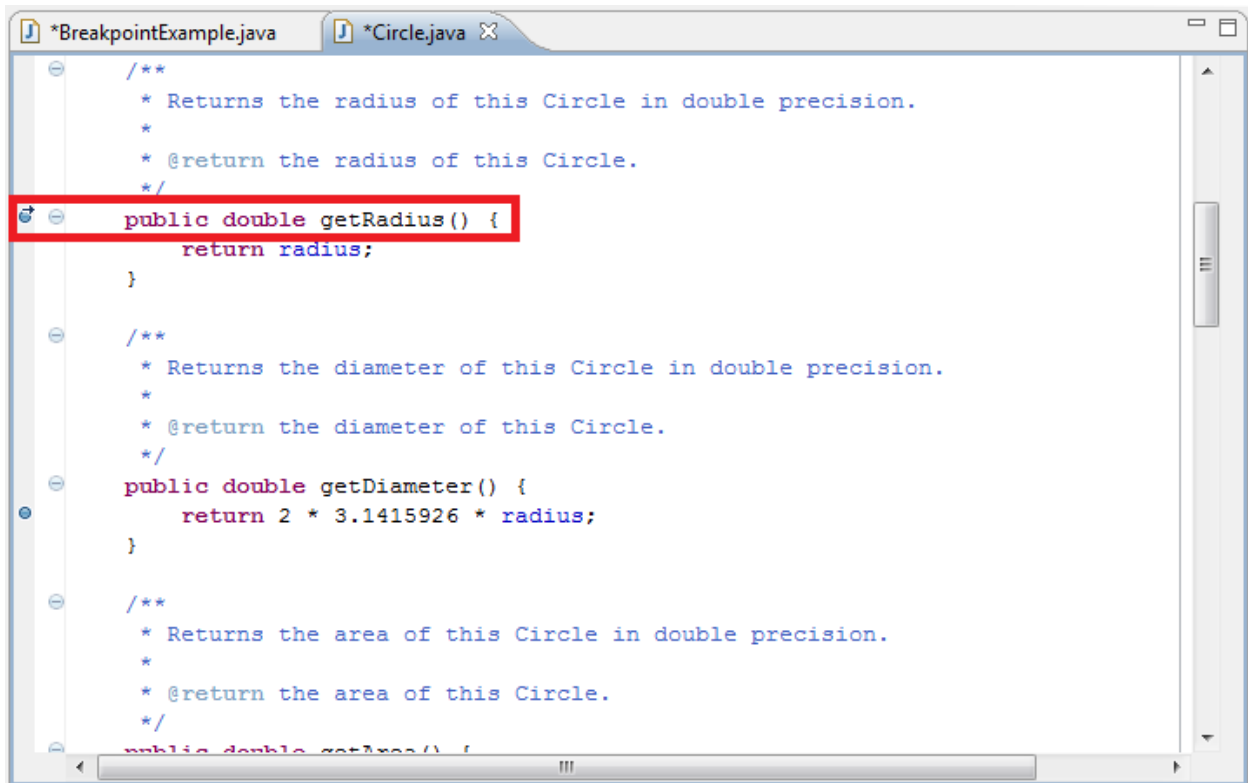
Read: <https://www.vogella.com/tutorials/EclipseDebugging/article.html>

Breakpoint Types

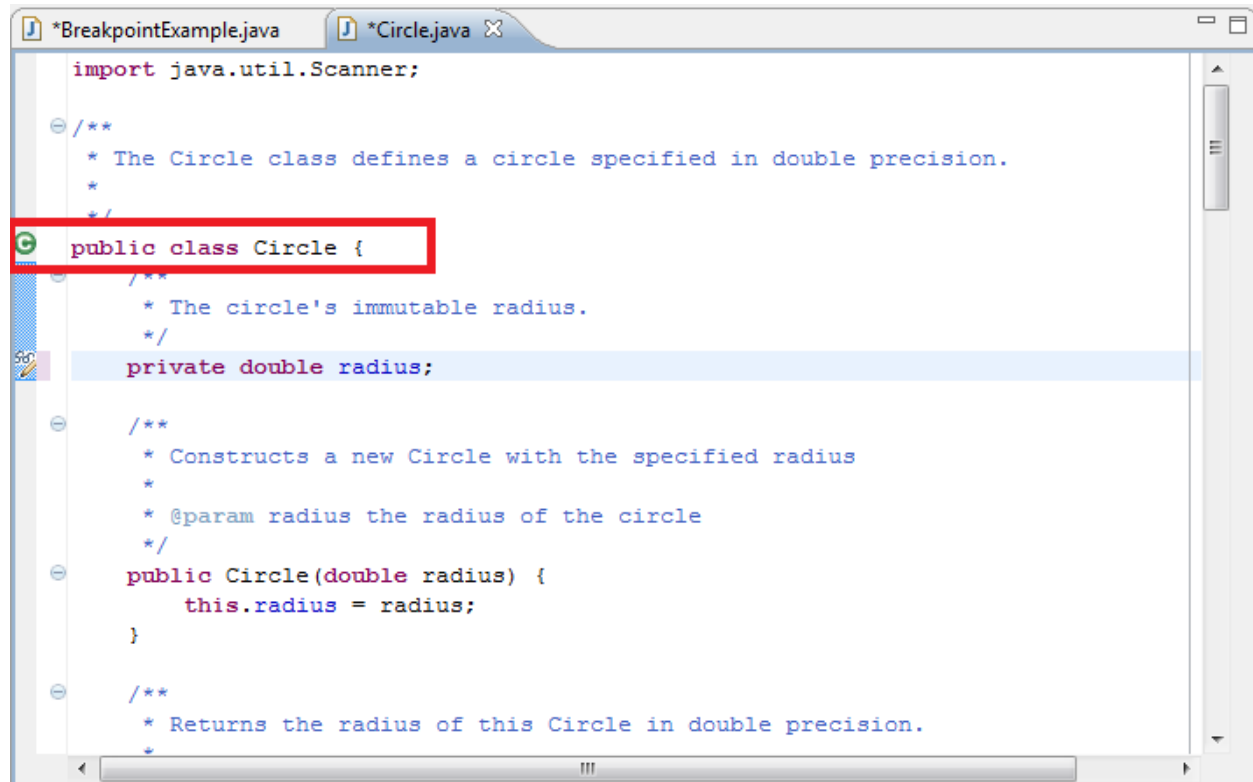
Line Breakpoint: This is the most familiar of the breakpoint types, which halts execution when a specified line in the source code is encountered during program execution. To set a line breakpoint in Eclipse, double-click in the margin to the left of the line where you want the breakpoint to be placed.



Method Breakpoint: This type of breakpoint is used to halt execution when a specified method is invoked. To set a method breakpoint in Eclipse, double-click in the margin to the left of the line defining the method name.

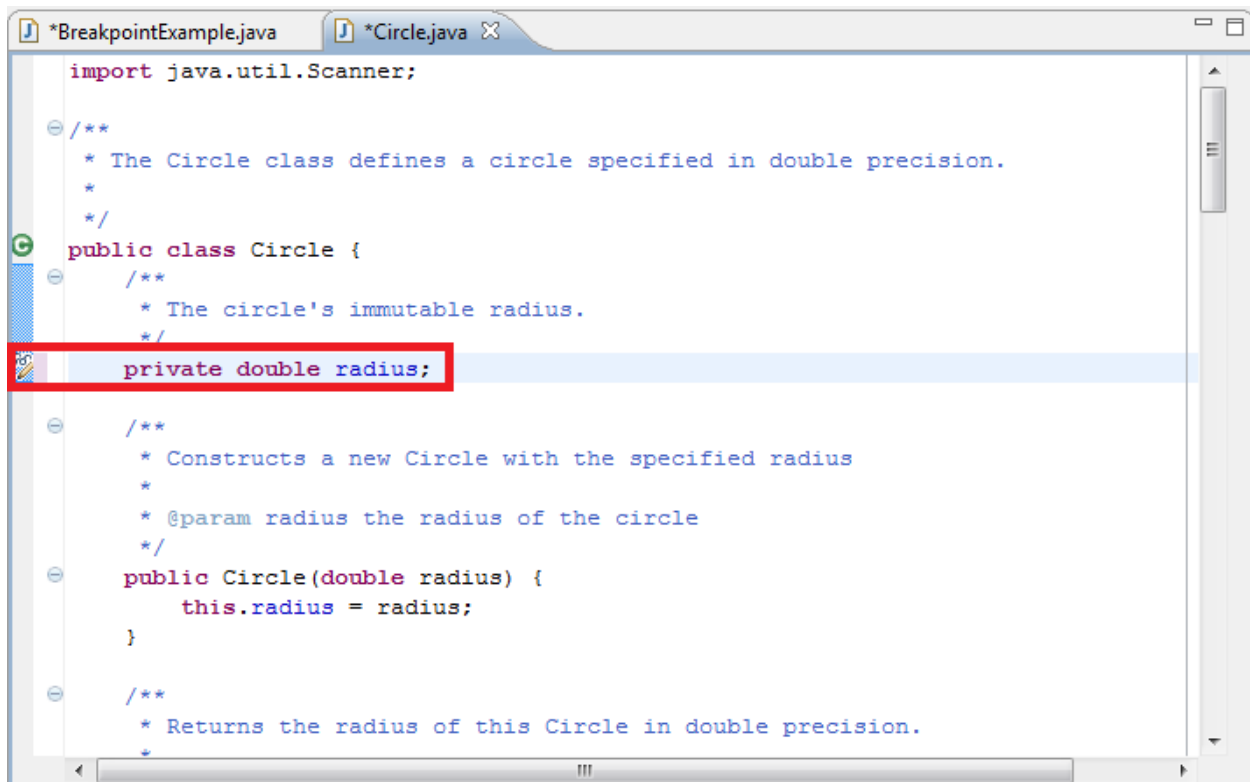


Class Load Breakpoint: This type of breakpoint is used to halt execution when a specified class is loaded. This is especially useful when classes are loaded dynamically at runtime. To set a class load breakpoint in Eclipse, double-click in the margin to the left of the line defining the class name.



Watchpoint: This type of breakpoint is used to halt execution when a specified field is read or modified. This is especially useful because it allows you to watch a field without needing to add a breakpoint everywhere that it is read or modified throughout the code. To set a watchpoint in Eclipse, double-click in the margin to the left of the line defining the field's name.

A list of all currently defined breakpoints is available in Eclipse's "Breakpoints" view. The "Breakpoints" view is loaded automatically in the "Debug" perspective (usually in the same tab grouping as "Variables"), or can be opened manually in any perspective by selecting "Window -> Show View -> Other... -> Debug -> Breakpoints". The "Breakpoints" view also provides a convenient way to enable/disable breakpoints and to edit their properties (see section on properties below).

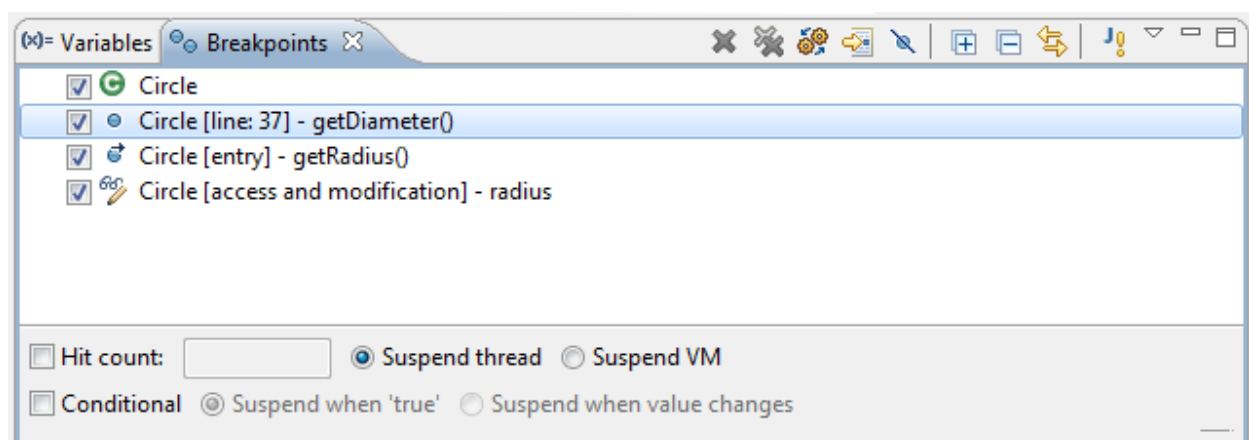


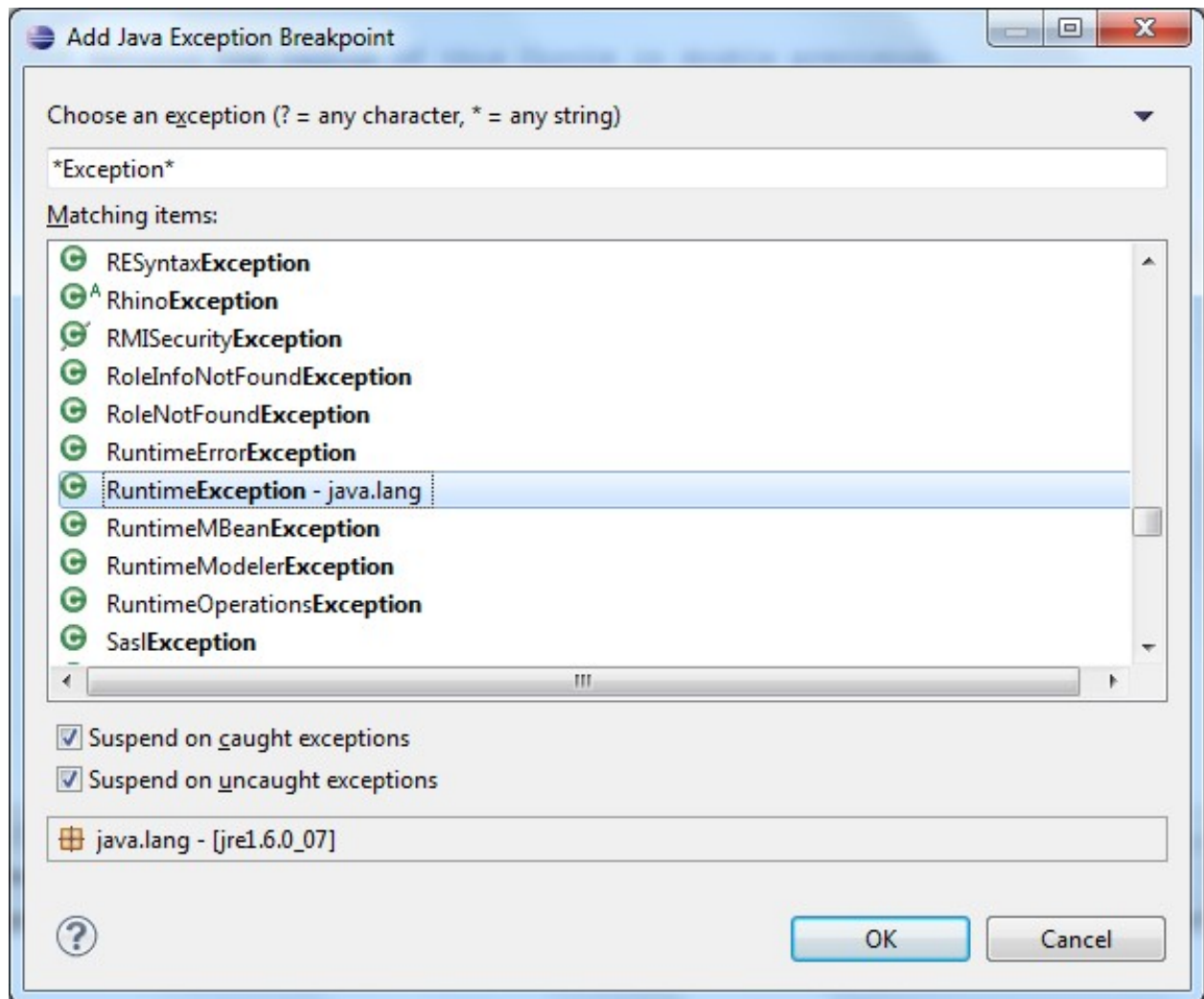
```
import java.util.Scanner;

/**
 * The Circle class defines a circle specified in double precision.
 */
public class Circle {
    /**
     * The circle's immutable radius.
     */
    private double radius;

    /**
     * Constructs a new Circle with the specified radius
     *
     * @param radius the radius of the circle
     */
    public Circle(double radius) {
        this.radius = radius;
    }

    /**
     * Returns the radius of this Circle in double precision.
     */
}
```



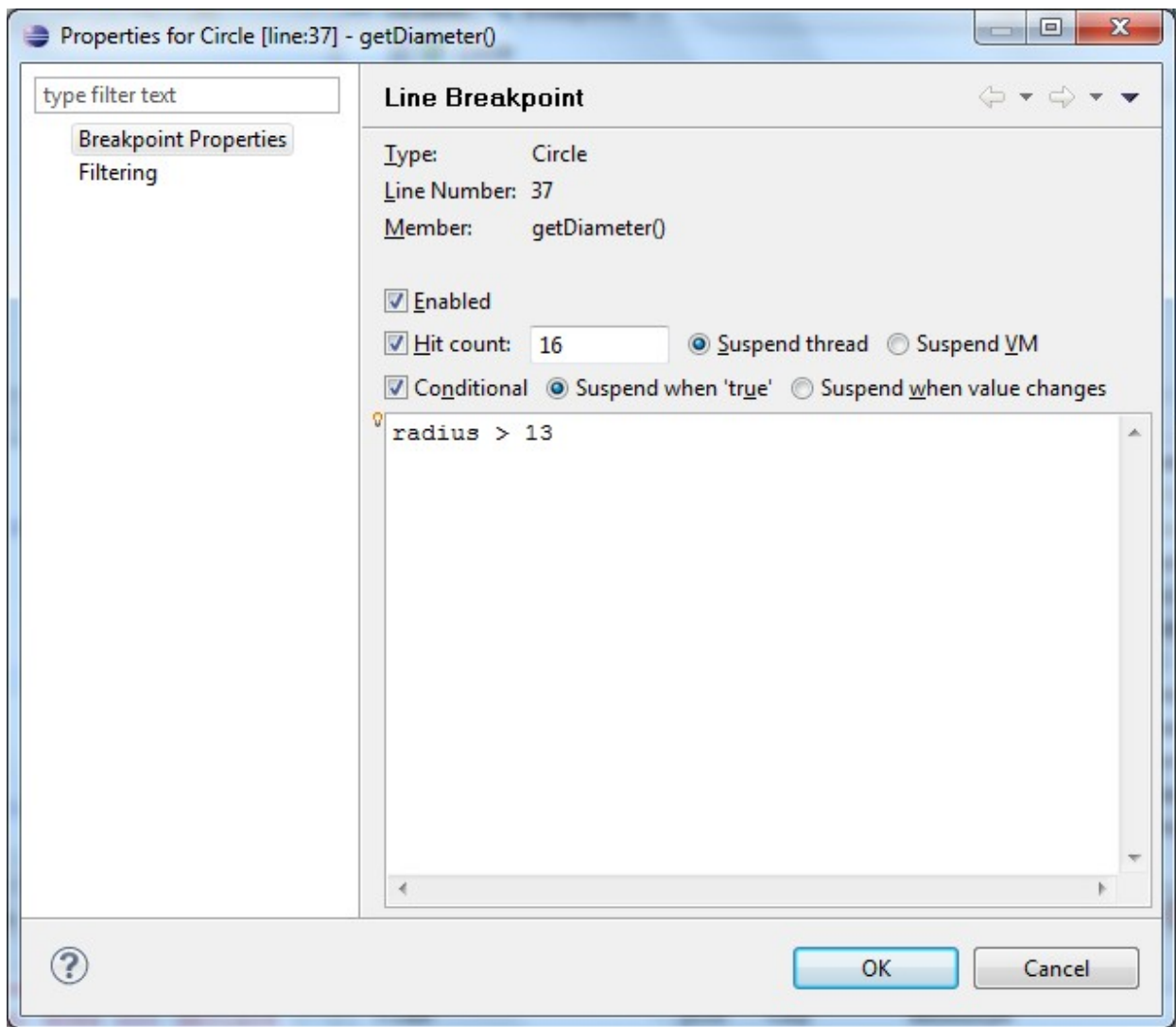


Exception Breakpoint: This type of breakpoint is used to halt execution when a specified exception type is thrown at any time during execution. To set an exception breakpoint in Eclipse, use the "Run -> Add Java Exception Breakpoint..." menu item.

Breakpoint Properties

In addition to the breakpoint types listed above, each breakpoint has a set of properties that can be used to refine the conditions under which a breakpoint will halt program execution. These properties can be edited by right clicking on the breakpoint (either within the "Breakpoints" view or on the indicator that appears to the left within the editor) and selecting the "Breakpoint Properties..." context menu item. Here are some of the most commonly used properties:

Hit Count: If a hit count is specified, the debugger will keep track of the number of times a breakpoint is "hit" and will only halt execution when the specified number is reached. This is often useful when a line of code is in a loop, and you want to examine the program state at a specific iteration.



Enable Condition: An enable condition allows you to specify any boolean expression that will be evaluated each time a breakpoint is encountered. Program execution will only be halted if the boolean expression evaluates to true. This is useful in cases where a breakpoint may be hit many times in a program, but you only want to examine the program state under certain conditions.

Suspend Policy: This specifies whether the breakpoint should halt the execution of only the thread in which the breakpoint was encountered, or all threads within the virtual machine.

Debug Exercises:

For the remainder of this class and all subsequent CS courses you are REQUIRED to use the debugger. If you come to me and ask for help the very first thing I will do is to have you show me how you have attempted to debug the problem. You need to develop a systematic approach to locating and correcting errors. The time for trial and error is over.

1. Create an Eclipse Project and name it Debug:
2. Add the file **DebuggingExercise.java**
3. This program attempts to store values in an array. Compile and run the program.
4. Use the Eclipse debugger to find and fix the issue. **I am not interested in you finding this issue solely with your eyes.** That is possible but not the point of this exercise. Mastering the debugger comes with time and practice.
 1. Read Eclipse documentation on executing the debugger. Use the link above to access the documentation
 2. Set breakpoints on lines or methods where you feel there may be problems
 3. Execute the debugger and step through the logic, analyzing variables as you go.
5. Add the file **DebugHash.java** The program generates arbitrary hashes and prints them to the screen in an infinite loop. It has been written in such a way that any changes to the program are very likely to change the output - a scenario in which debuggers become a very important tool. Your task is to use a breakpoint to figure out the 49,791st hash value that will be printed. (**Hint:** Don't worry too much about how the values are generated, just find where the value to be printed is computed and use the breakpoint expertise introduced above to find the answer)
6. Remove **DebugHash.java** and add the **FibDebug.java** file to the project. Using the Eclipse debugger try to find the bug
7. Remove **FibDebug.java** and add **Marker.java** to the project. Read the comments in the file and use the Eclipse debugger to find the bug
8. Remove **Marker.java** and add **Account.java** and **AccountDebug.java**. Treat Account.java as if it was an object from the Java API. Execute **AccountDebug.java** and analyze the runtime messages. Use the Eclipse debugger to trace the code from class to class. Find the logic error and fix it.
9. Remove **Account.java** and **AccountDebug.java** and add **Person.java** and **PersonDebug.java**. Treat Person as if it was an object from the Java API. Compile and run

the code. Use the Eclipse debugger to find and fix the error

Submission:

Create a write up for this lab that provides answers to the following questions

- Specifically WHAT were the issues in each debugging situation?
- Specifically HOW did you use the Eclipse Debugger to find the issues?
- Specifically WHAT were the fixes that you applied to correct the actions?

For the debugging portion of this lab only submit the write up

Object Oriented Design Exercise

Write a Temperature class that has two instance variables:

- a temperature value (a floating-point number)
- an enumeration for the scale, either C for Celsius or F for Fahrenheit.
- Use proper encapsulation and information hiding techniques.

Include the following:

1. two accessor (getter) methods to return the temperature
 - a. one to return the degrees Celsius, the other to return the degrees Fahrenheit
 - b. use the following formulas to write the two methods, round to the nearest tenth of a degree:

$$\text{DegreesC} = 5(\text{degreesF} - 32)/9$$
$$\text{DegreesF} = (9(\text{degreesC})/5) + 32$$

2. Three mutator (setter) methods:
 - a. one to set the value
 - b. one to set the scale (F or C)
 - c. one to set both
3. Two comparison methods:
 - a. an equals method to test whether two temperatures are equal. The method header should match this exactly **public boolean equals(Temperature t)**
 - i. this method will compare the argument "t" to the calling instance "this"
 - ii. Note that a Celsius temperature can be equal to a Fahrenheit temperature as indicated by the above formulas
 - b. An **int compareTo(Temperature t)** method that determines sorting order of temperature values
 - i. This method should accept a Temperature instance and compare it to "this" instance
 - ii. If "this" (calling instance) temp is greater than the temp argument return 1
 - iii. If "this" (calling instance) temp is less than the temp argument return -1
 - iv. If they are identical return 0

- v. Note that a Celsius temperature can be equal to a Fahrenheit temperature as indicated by the above formulas
 - vi. This is a commonly defined method in Java programming and is the basis of the Comparable design pattern. More on that later.
4. a suitable toString() method.
 5. Write a Driver class that creates a few instances of the Temperature class and shows that you can call methods, pass arguments and display results. Each method must be demonstrated.
 6. Write a series of JUnit tests that test each of the methods described above. Here is a great overview of unit testing, the JUnit framework and JUnit integration with Eclipse

<https://www.vogella.com/tutorials/JUnit/article.html>

0.0 degrees C = 32.0 degrees F,
-40.0 degrees C = -40.0 degrees F,
100.0 degrees C = 212.0 degrees F.

Submission: Push Driver.java and Temperature.java and Unit Test files. Do not submit screen shots or class files

Object Oriented Programming Exercise: Parsing 2016 Election Results

The file **2016_US_County_Level_Presidential_Results.csv** contains a county by county listing of the 2016 presidential election results. I make NO CLAIM as to the accuracy of these data.

Here is your task.

Define the class **CountyResults2016.java**

- Fields:
 - double demVotes
 - double gopVotes
 - double totalVotes
 - double percentDem
 - double percentGOP
 - double difference
 - double percentDifference
 - String stateAbbreviation
 - String county
 - int fips
- Methods:
 - getTotalVotes
 - getDemVotes

- getGOPVotes
- getDifference
- getPercentDifference
- getState
- getCounty
- toString
- Constructor to accept all fields for initialization

Define the class ***ElectionDriver.java***

- Fields
 - an ArrayList of type CountyResults2016

ArrayList<CountyResults2016> results = new ArrayList<CountyResults2016>();

- Methods
 - **void fillList():**
 - opens the file *2016_US_County_Level_Presidential_Results.csv*
 - For each line in the file, create an instance of **CountyResults2016** by calling the overloaded constructor.
 - Add the instance to the list
 - **CountyResults2016 findLargestMargin():** finds and returns the county record showing the largest margin of victory for the entire country
 - **CountyResults2016 findLargestMargin(String state):** overloaded method that returns the county record showing largest margin of victory by state
 - **String[] getStateTotals():** returns an array showing the total results per state. Each element will contain:
 - State, Total Dem Votes, Total GOP votes, Margin of Victory, Winning Party
- Unit Tests
 - Write unit tests for the following methods
 - findLargestMargin
 - findLargestMargin(State)
 - getStateTotals
 - How will you determine what the correct output will be? Using a spreadsheet application to parse, filter, sort and otherwise process the file would be a good idea. **I do not know the answers to these questions. It is up to you to reliably figure this out.**
- Prove beyond a shadow of a doubt that your code functions correctly. I want to see intelligently design output that I easy to read and undertand.

Submission: Push CountyResults2016.java, ElectionDriver.java and your unit tests

Extra Challenge (NOT REQUIRED): Use a charting API like JFreeChart to create chart graphics illustrating interesting views of the election results: <http://www.jfree.org/jfreechart/>