CSCI165 Computer Science II
Debugging Lab

Please read this document thoroughly *before* asking questions

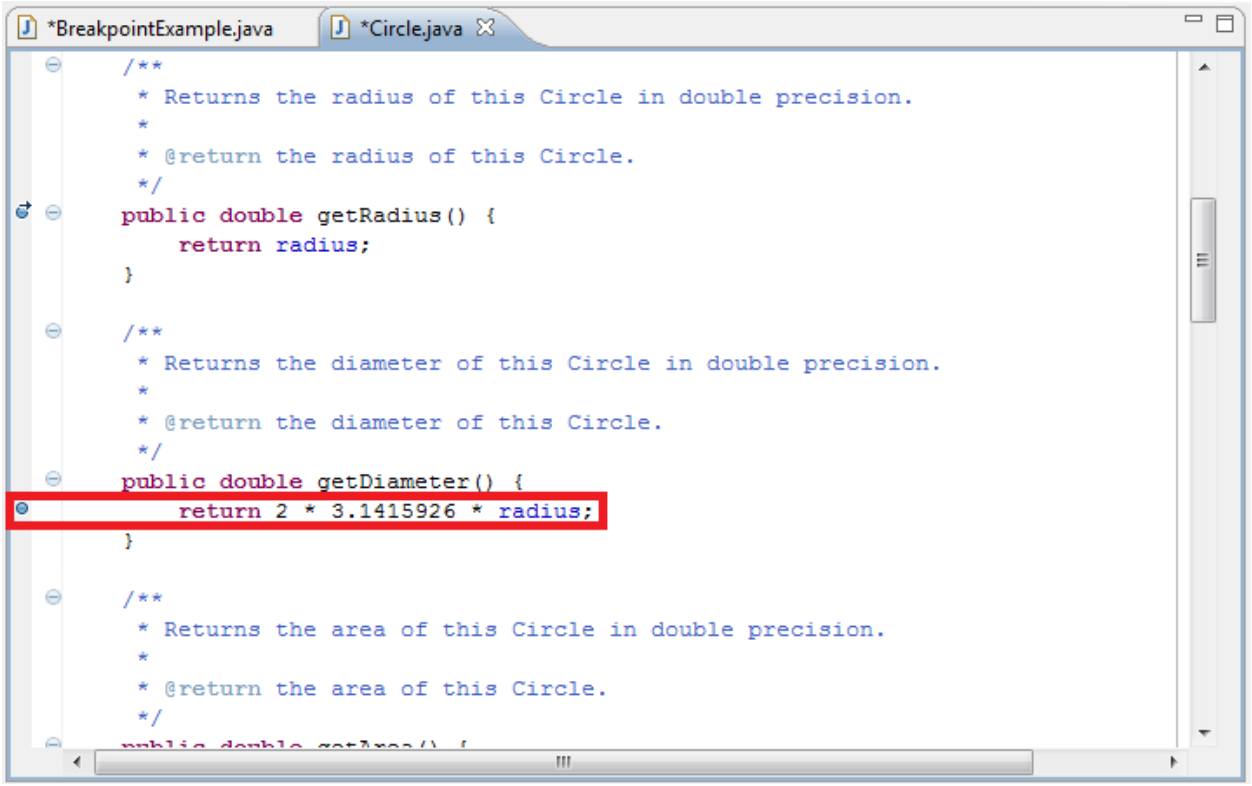
Debugging

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.

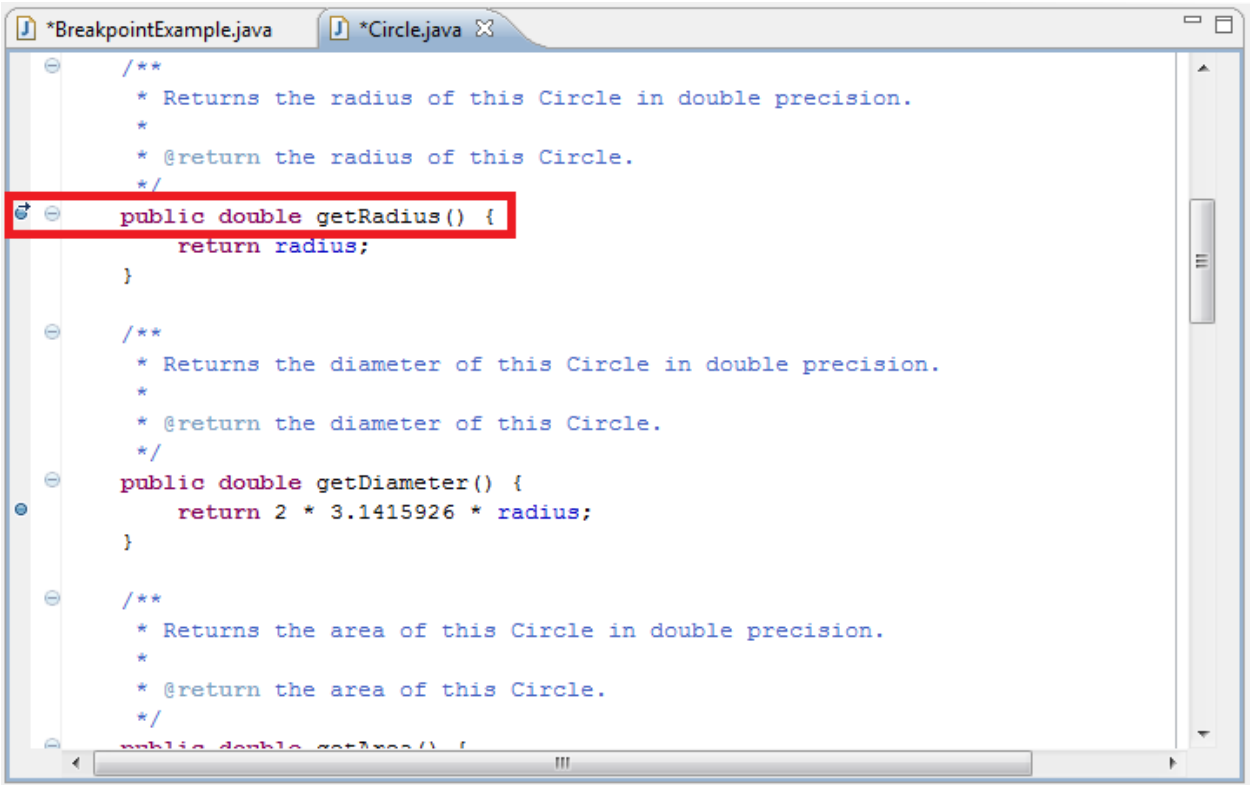
Read:
http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-running_and_debugging.htm&cp=1_3_6

Breakpoint Types

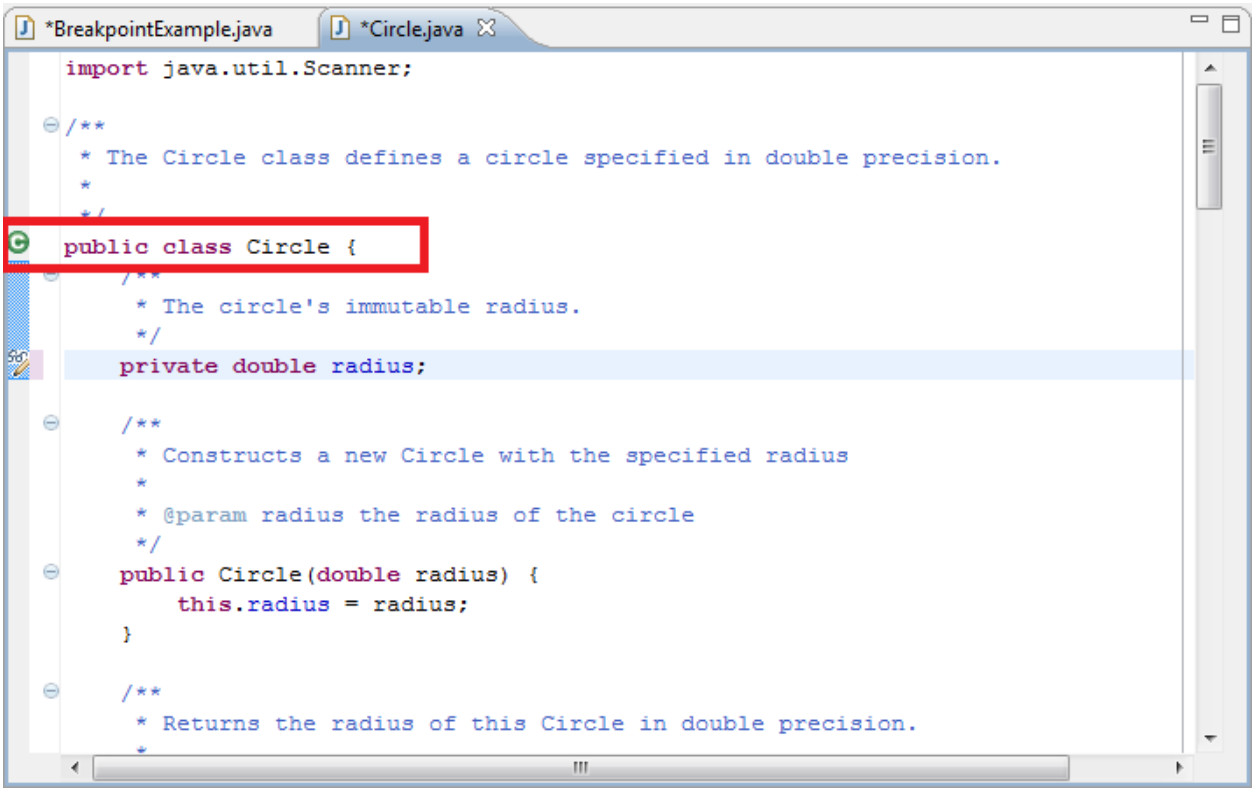
Line Breakpoint: This is the most familiar of the breakpoint types, which halts execution when a specified line in the source code is encountered during program execution. To set a line breakpoint in Eclipse, double-click in the margin to the left of the line where you want the breakpoint to be placed.



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.

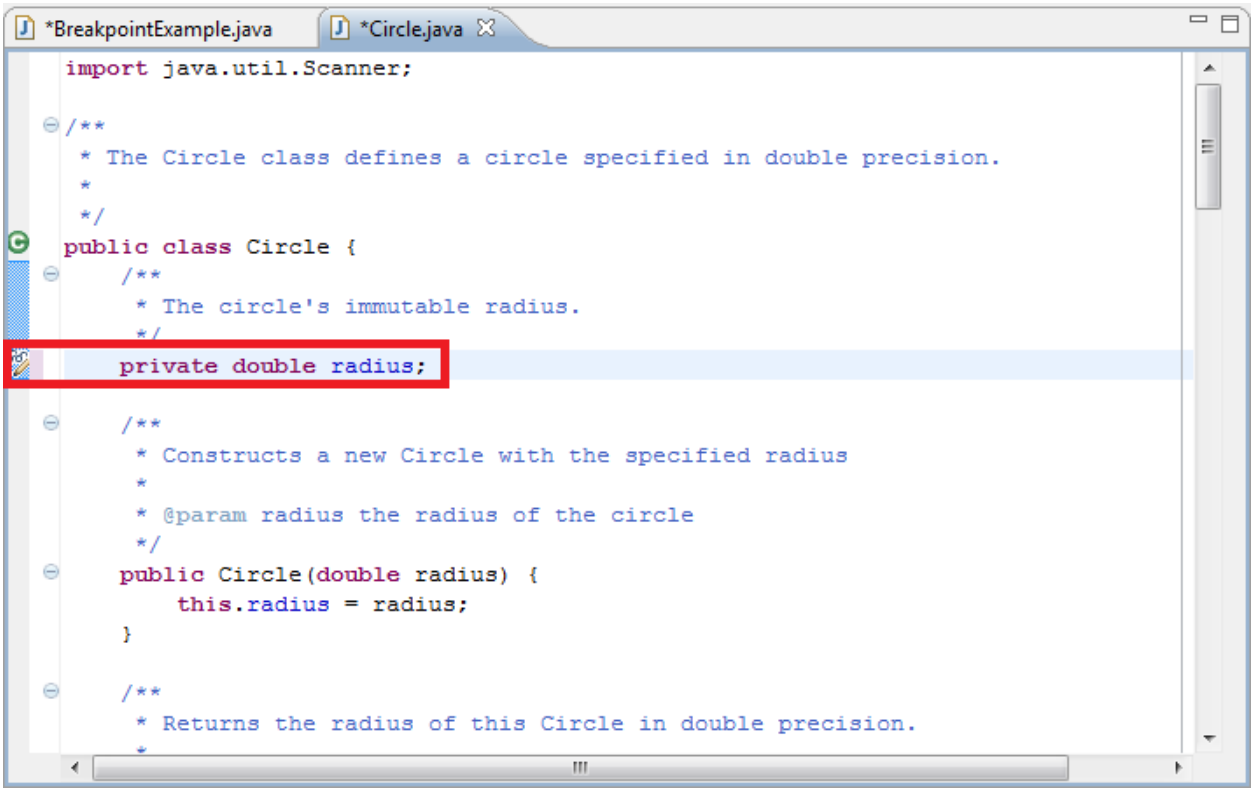


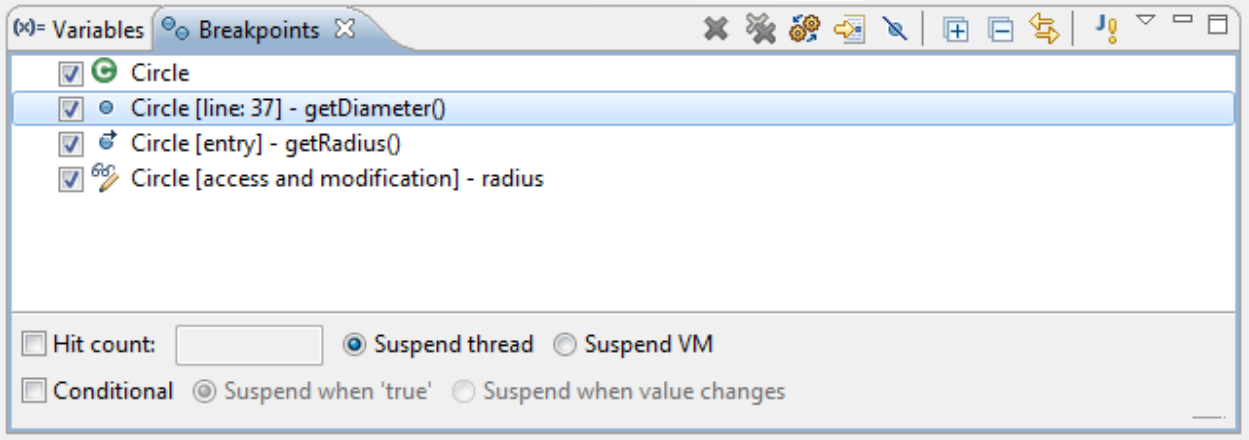
Class Load Breakpoint: This type of breakpoint is used to halt execution when a specified class is loaded. This is especially useful when classes are loaded dynamically at runtime. To set a class load breakpoint in Eclipse, double-click in the margin to the left of the line defining the class name.



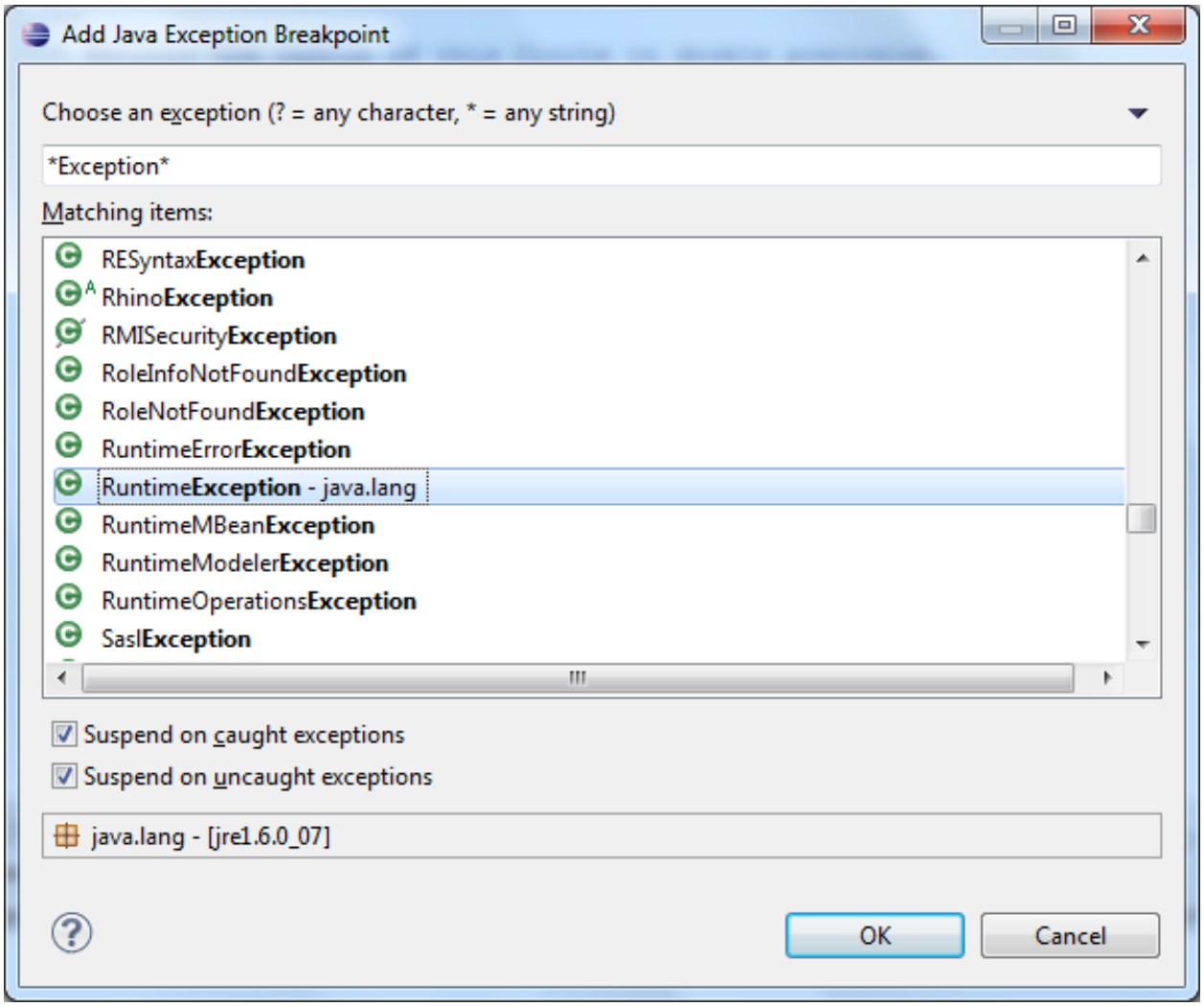
Watchpoint: This type of breakpoint is used to halt execution when a specified field is read or modified. This is especially useful because it allows you to watch a field without needing to add a breakpoint everywhere that it is read or modified throughout the code. To set a watchpoint in Eclipse, double-click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





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.

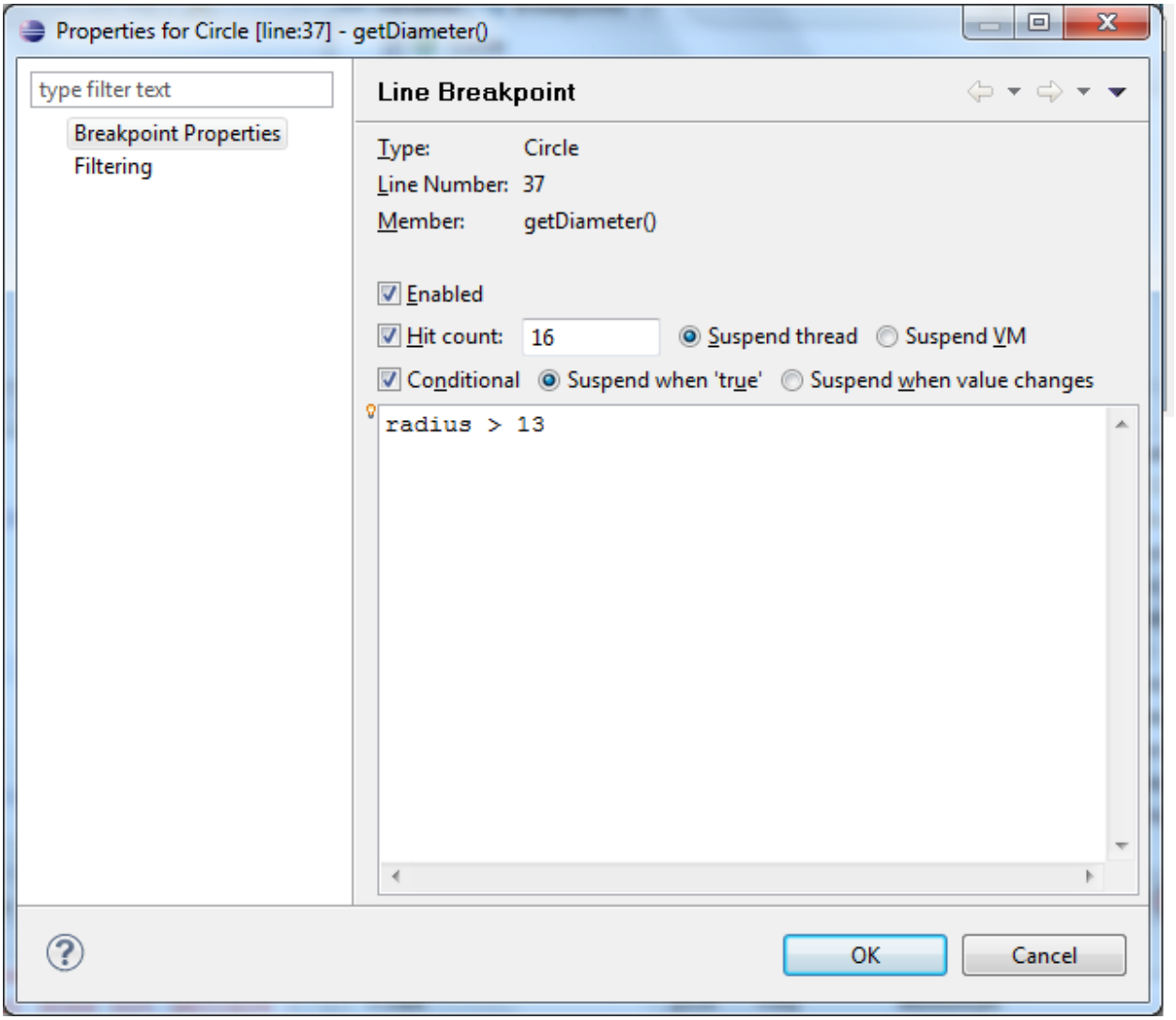


Breakpoint Properties

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 (Think list in Python but strictly typed and statically sized). 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. I am not interested in you solving these issues with your eyes. I want detailed descriptions of how you used the debugger. **Screenshots of the debug windows showing your approach are required**

- 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 <u>only submit the write up</u>

<u>Object Oriented Design Exercise</u>

Write a Temperature class that has two instance variables:

- a temperature value (a floating-point number)
- a character 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:

        DegreesC = 5(degreesF - 32)/9
        DegreesF = (9(degreesC)/5) + 32

2. **Three mutator (setter) methods:**
    a. one to set the value. Validate the following domains (min temp = -200, max temp + 200) Invalid arguments should be set to 0
    b. one to set the scale (F or C). This must also be validated. Invalid arguments should be set to Celsius. An enumeration would be a good descriptive approach

for this. Making it **public static** would be even better. Check the Radio code for examples of this and please ask questions.

c. A setter to set both temperature and scale.
d. **Enforce the following policy:** scale must be C or F, violations will be set to C. This code can only be written once.

3. **Two constructors**
    a. A no-argument constructor that sets the temp to 0 and the scale to C
    b. An overloaded constructor that accepts bot the temp and the scale. These must be validated, but you cannot duplicate the validation code.

4. **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.

5. **a suitable toString() method.**

6. Write a Driver class and demonstrate that you can create instances and call methods correctly. Demonstrate every method and include descriptive messages to accompany all of the output.

    a. Create an array of 5 Temperature objects. Instantiate the five objects with valid Temperature instances
    b. Use Java's enhanced for loop to iterate through the array, calling toString on each one. A brief description and example of this concept is included below.

Arrays of Objects

Arrays are capable of storing objects also. For example, we can create an array of Strings which is a reference type variable. However, using a String as a reference type to illustrate the concept of array of objects isn't too appropriate due to the immutability of String objects. Therefore, for

this purpose, we will use a class Student containing a single instance variable **marks**. Following is the definition of this class.

```
class Student {
    private int marks;

    public Student(int marks){
        setMarks(marks);
    }

    public void setMarks(int marks){
        if(marks ≥ 0 && marks ≤ 100)
            this.marks = marks;
    }

    public String toString(){
        return "Student marks: " + this.marks;
    }
}
```

An array of objects is created just like an array of primitive type data items in the following way.

```
Student[] students = new Students[5];
```

The above statement creates the array which can hold references to 5 Student objects. It doesn't create the Student objects themselves. They have to be created separately using the constructor of the Student class. The students array contains 5 memory spaces in which the address of 5 Student objects may be stored. If we try to access the Student objects even before creating them, run time errors would occur. For instance, the following statement throws a **NullPointerException** during runtime which indicates that ***students[0]*** isn't yet pointing to a valid Student object.

students[0].setMarks(100);

The Student objects have to be instantiated using the constructor of the Student class and their references should be assigned to the array elements in the following way.

studentArray[0] = new Student(50);

In this way, we create the other Student objects also. If each of the Student objects have to be created using a different constructor, we use a statement similar to the above several times.

However, in this particular case, we may use a for loop since all Student objects are created with the same constructor.

```
for (int i = 0; i < students.length; i++)
        students[i] = new Student(50);
```

The above for loop creates 5 Student objects and assigns their reference to the array elements. Now, a statement like the following would be valid.

students[0].setMarks(100);

Enhanced for loops find a better application here as we not only get the Student object but also we are capable of modifying it. This is because of the fact that Student is a reference type.

```
for (Student x : students )
           x.setMarks(s.nextInt()); // s is a Scanner object
```

The variable ***x*** in the header of the enhanced for loop will store a reference to the Student object and not a copy of the Student object which was the case when primitive type variables like int were used as array elements.

7. Back to the Temperature requirements. Write a collection of JUnit tests that demonstrate that each of the assigned methods performs correctly. Each test case must contain multiple cases that demonstrate ample coverage of your code , both invalid and valid arguments. Include screen shots of Eclipse showing that all tests pass.

   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: Submit ***TemperatureTest.java, Driver.java and Temperature.java*** and the screen shots of the unit tests passing . Do not submit class files.