Lab 2: Debugging with Eclipse and Trees

Objectives

- Debugging java projects with Eclipse
- Getting familiar with arrays in Java programming language
- Getting familiar with the class `Arrays` and the standard Java APIs
- Extending your understanding of trees
- Coding a more complex problem about trees

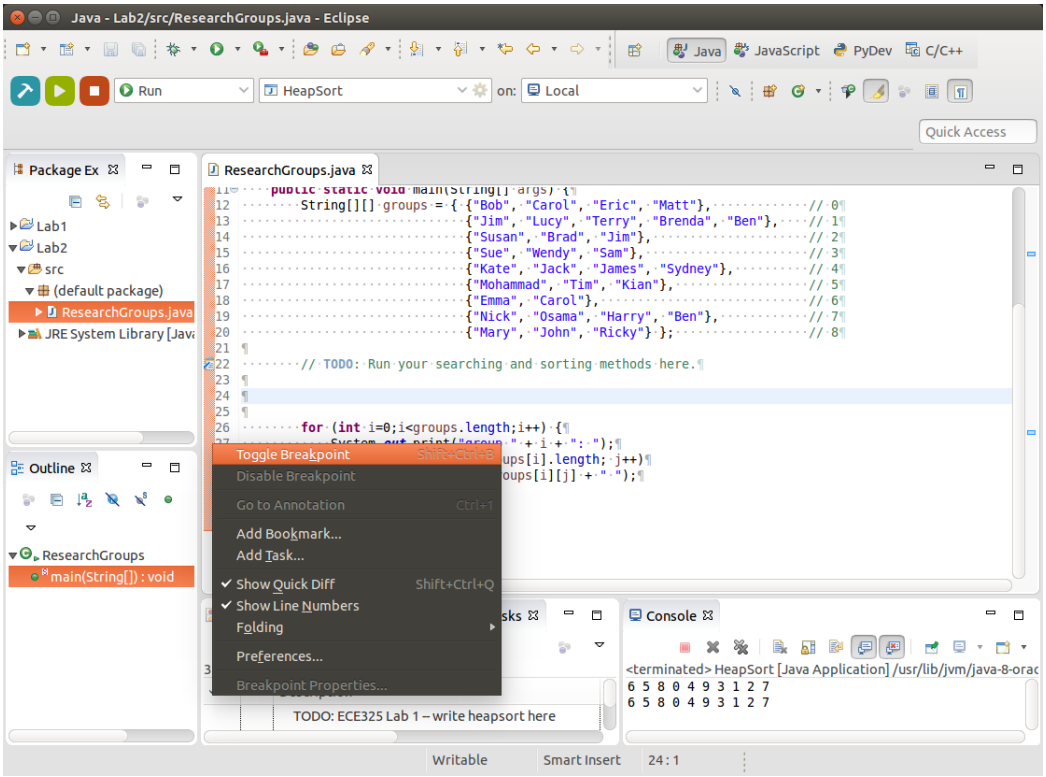
Source files

- ResearchGroups.java
- RedBlackTree.java

1 Introduction

1.1 Debugging Java Projects with Eclipse

In this lab we try to walk through debugging and its implementation in the Eclipse IDE. Debugging allows you to run a program interactively while watching the source code and the variables during the execution. By *breakpoints* in the source code, you specify where the execution of the program should stop. To stop the execution only if a field is read or modified, you can specify *watchpoints*. Eclipse allows you to start a Java program in debugging mode. It also has a special *Debug* perspective, which gives you a preconfigured set of views.

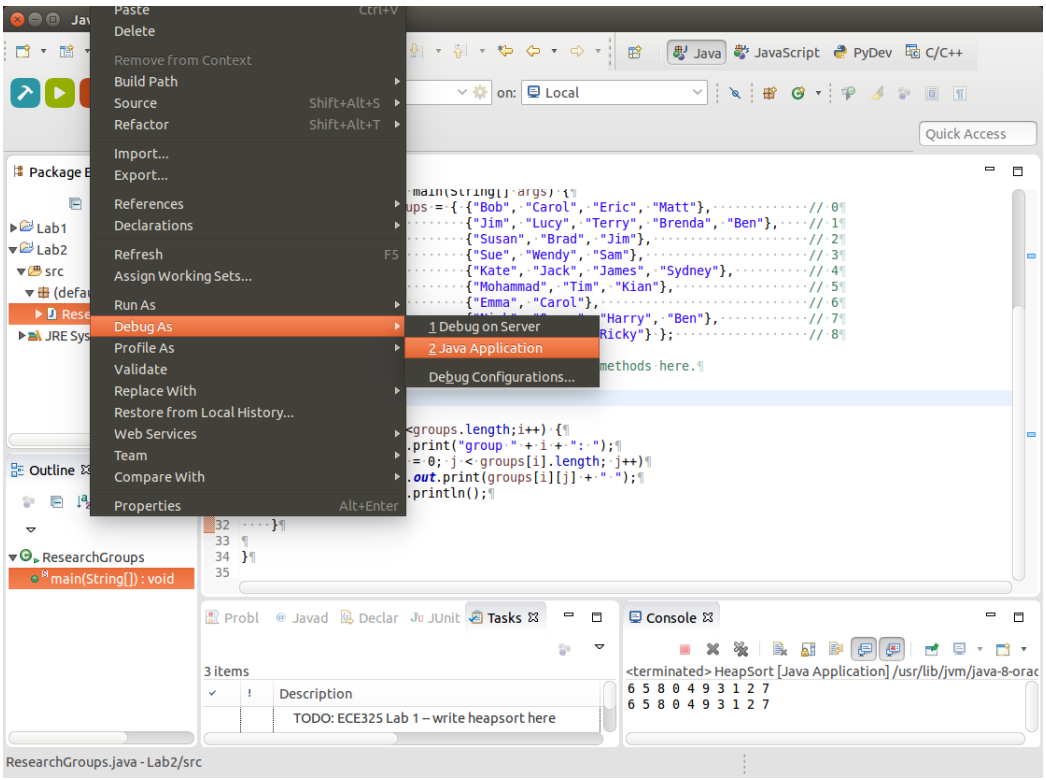


1.2 Setting Breakpoints

To set a breakpoint in your source code you can simply double click on the small left margin in of your source code editor, or right click on this part and select *Toggle Breakpoint*.

1.3 Starting the Debugger

To debug your application, select a Java file which can be executed, right-click on it and select *Debug As => Java Application*.



NOTE: If you have not defined any breakpoints, this will run your program as normal. To debug the program, you need to define breakpoints first.

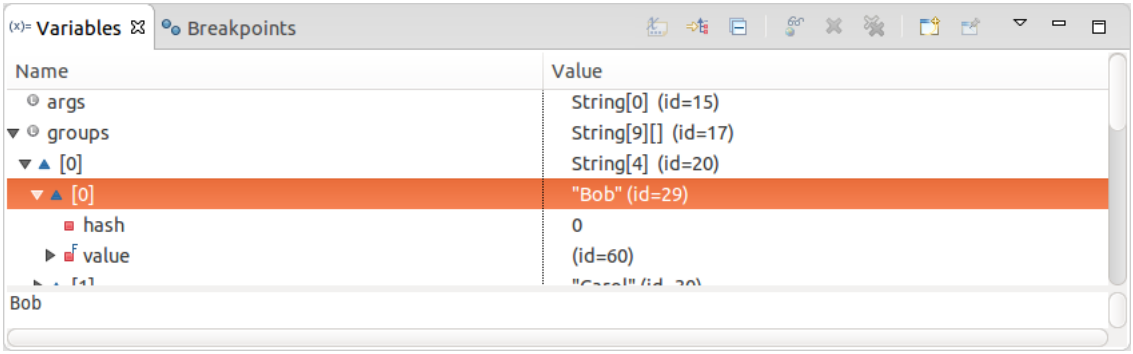
1.4 Controlling the Program Execution

Eclipse provides toolbar buttons for controlling the execution of the program you are debugging. Typically, it is easier to use the corresponding shortcut keys to control this execution. You can use the F5, F6, F7 and F8 key to step through your coding. The meaning of these keys is explained in the following table.

Key	Description
F5	Executes the currently selected line and goes to the next line in your program. If the selected line is a method call the debugger steps into the associated code.
F6	Steps over the call, i.e. it executes a method without stepping into it in the debugger.
F7	Steps out to the caller of the currently executed method. This finishes the execution of the current method and returns to the caller of this method.
F8	Resume the execution of the program code until it reaches the next breakpoint or watchpoint.

1.5 Evaluating Variables in the Debugger

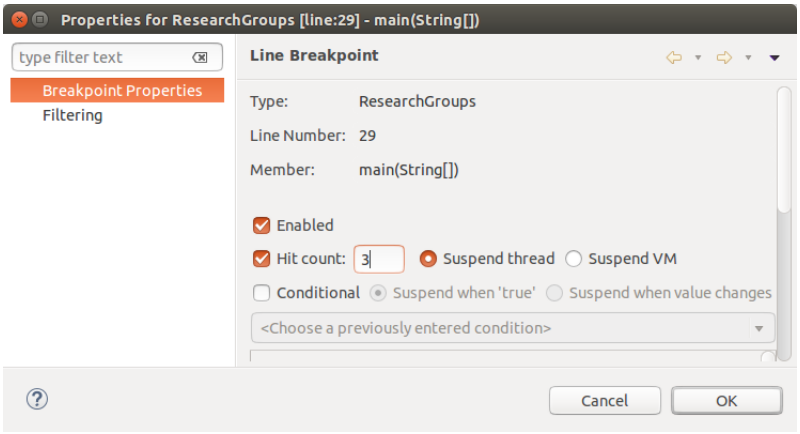
The *Variables* view displays fields and local variables from the current executing stack. Please note you need to run the debugger to see the variables in this view.



1.6 Breakpoints Properties

After setting a breakpoint you can check the properties of it by right-click => *Breakpoint Properties*. In the breakpoint properties you can define a condition that restricts the activation of this breakpoint.

- You can for example specify that a breakpoint should only become active after it has reached 3 or more times via the *Hit Count* property.
- You can also create a conditional expression. The execution of the program only stops at the breakpoint if the condition becomes true. This mechanism can also be used for additional logging, as the code that specifies the condition is executed every time the program reaches that point.



1.7 Different Types of Breakpoints

- *Method breakpoint*: A method breakpoint is defined by double-clicking in the left margin of the editor next to the method header. You can configure it if you want to stop the program before entering or after leaving the method.
- *A class load breakpoint*: It stops when the class is loaded. To set a class load breakpoint, right-click on a class in the *Outline* view and choose the *Toggle Class Load Breakpoint*option.

2 Deliverable 1 -- Arrays and Debugging

Consider the class in ResearchGroups.java. This class contains the name of nine research group members.

Step 1:

You need to write new method called `searchMember` to search for a member and report the following information:

- His/her name's existence in the list (yes or not)
- His/her group number (some members are in more than one group)
- Whether he/she is a group leader (the first person listed in each group is indicated as its leader)

NOTE: You can use `equal` method from `Array` class to check equality of names.

Step 2:

In this step, you are required to write another method called `sortGroups` which is able to sort groups in terms of their length in ascending order. Luckily you just wrote heap sort in Lab 1, so work out how to reuse it. (Hint: consider hashing that maps objects into an integer)

DEMO this deliverable to the lab instructor.

Step 3:

Set a breakpoint in the `searchMember` method. Debug your program and follow the execution of the variables in the method.

Step 4:

Follow the variable you have used in searching loop as a counter by setting a breakpoint. Fix a hit count to activate the breakpoint after a certain number of iterations.

Step 5:

Add a breakpoint for class loading. Debug your program again and verify that the debugger stops when your class is loaded.

3 Deliverable 2 -- Red Black Tree / Heap Tree

Let's try something harder. It is where you really need to understand how to use debugging facilities. When things go wrong if you cannot use a debugger, you are DEAD.

Case 1 -- Red Black Tree

The difficulties: - insertion as a binary search tree - tree balancing by rotation

Finish the methods in *RedBlackTree.java*. Read black tree is a more modern data structure than perhaps we are used to using. It becomes popular in the 1980's and quickly replaces earlier ideas, such as simple binary trees in most applications. A description can be found at http://en.wikipedia.org/wiki/Red-black_tree.

Hint: Watch out for these NIL nodes -- they are not `null`; they are black.

Note 1: Write carefully, you are going to reuse this code later!

Note 2: If you can't remember what a binary search tree is -- STOP. And go and revise.

NOTE 3: THIS IS A VERY DEMANDING ASSIGNMENT. IT IS EXPECTED THAT MANY STUDENTS WILL NOT PRODUCE A PERFECT SOLUTION. HOWEVER, JUST BECAUSE YOUR SOLUTION IS NOT PERFECT THAT DOES NOT MEAN THAT YOU SHOULD NOT HAND IT IN AND GET MARKS! PARTIAL MARKS ARE AVAILABLE FOR REASONABLE ATTEMPTS.

Case 2 -- Heap Tree

The difficulties: - Searching for sibling nodes (breadth first searching)

You have already known what is a heap, and how to use it to sort arrays. The heap is saved in an linear array, where indices show the hierarchy of the heap. However, we should bear in mind that it is actually a binary tree. Therefore, let's try write it as a tree: open the *HeapTree.java*, finish all TODOs, and makes it run correctly.

Remember this is only a lab -- do not write a heap in such style in your work.

DEMO this deliverable to the lab instructor.