Lab 2: Debugging with IDEs and Prefix Tree

Objectives

- Debugging java projects with IDEs
- 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: prefix tree (trie)
- Full mark: 40 points

Source files

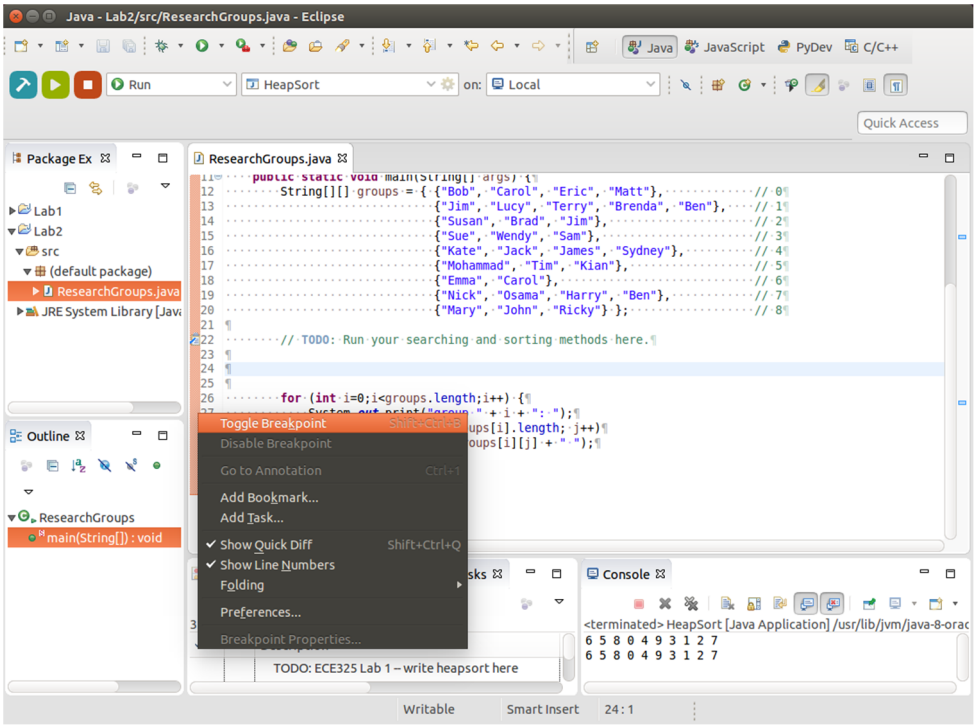
- `ResearchGroups.java`
- `PrefixTree.java`

1 Introduction

1.1 Debugging Java Projects with Eclipse

In this lab we try to walk through debugging and its implementation in IDs. Debugging process and features are the same in almost all the IDEs however, we use Eclipse in this document for the explanation. (You can use your IDE of preference)

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.

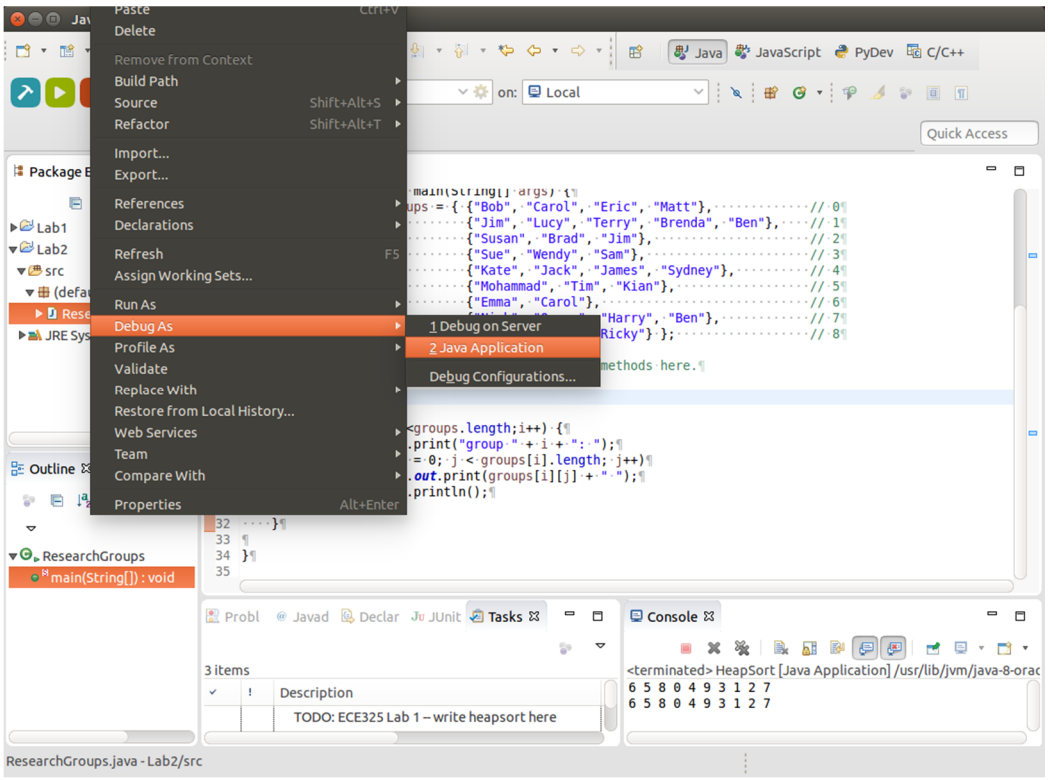


1.2 Setting Breakpoints

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.

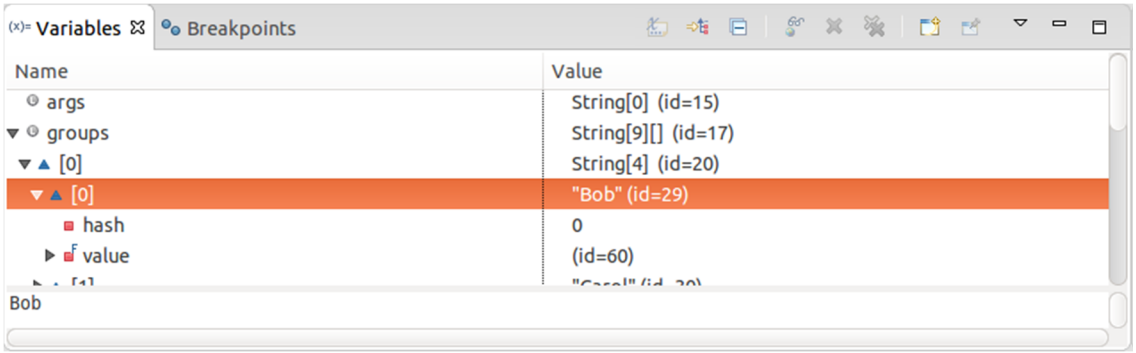
Key	Description
F5	Executes the currently selected line and goes to the next line in your program. If the selected line is a method call the debugger steps into the associated code.
F6	Steps over the call, i.e. it executes a method without stepping into it in the debugger.

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

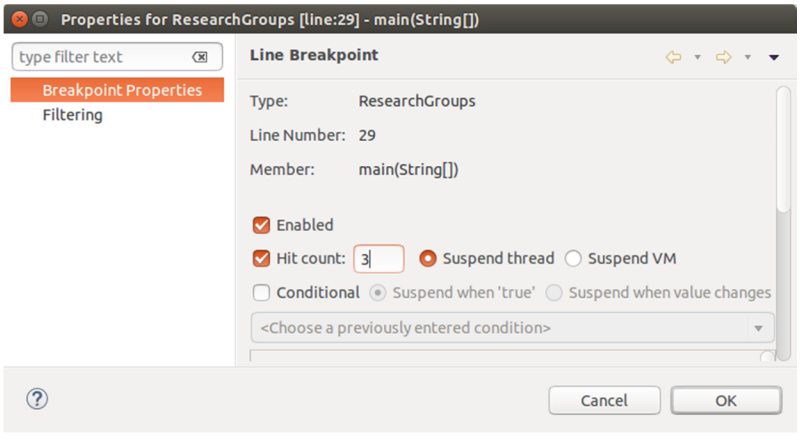
The *Variables* view displays fields and local variables from the current executing stack. Please note you need to run the debugger to see the variables in this view.



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 a 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 merge sort in Lab 1, so work out how to reuse it. (Hint: consider hashing that maps objects into an integer)

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 - Prefix Tree (trie)

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.

Prefix Tree

A **Trie** is an ordered tree structure that is mostly used to store String. The reason it is used to store String is that it has fast retrieval time. The complexity of finding a String in a Trie is O(m), where m is the length of the string. The term trie came from the word re**trie**val as it makes retrieval of a string from the collection of strings. Trie is also called as **Prefix Tree.**

In Trie the root is empty and each child node contains only one character. So the number of child nodes, a particular node can have depends upon the number of alphabets in a given language.

Each node in Trie must contain two pieces of information:

1. A boolean flag which indicates that this node is end of a word.
2. An array of size 26(depends on the language used. 26 is for English). Each index in this array points to another node.

Finish the methods (`insert`, `search` and `startsWith`) in *PrefixTree.java*.

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

Trie's visualization link: https://www.cs.usfca.edu/~galles/visualization/Trie.html

Lab 2: Debugging with Eclipse and Trees

Marking sheet

Deliverable 1 : Arrays and Debugging

searchMember

	Only check the member existence (1-2)	Report the group number, but not all of them in cases where a person is in more than one group (3-4)	All the information reported is correct (5-6)	/6

sortGroups

	Does not work, but the idea is correct (1-2)	Work and the result is reported correctly (3-4)		/4

Deliverable 2 -- Prefix Tree

Insertion into Prefix Tree

	Does not work, but the idea is correct (1-2)	Work and the result is reported correctly (3-5)	/5

Search

	Does not work, but the idea is correct (1-2)	Work and the result is reported correctly (3-5)	/5

StartWith

	Does not work, but the idea is correct (1-2)	Work and the result is reported correctly (3-5)	/5

Code quality (in both deliverables)

Naming and usage of variables (3)	Design (logic structure)			
	Correct use of indentation (1)	Use of helper functions/procedures (5)	Correct use of statements (1)	/10

Documentation (in both deliverables)

	No documentation (0)	One line comments but not using proper Javadoc (2)	Clear documentation about what the function/procedure does using javadoc style (5)	/5
			Total:	**/40**