Debugging

Lecture 4

Based on Slides from University of Colorado Colorado Springs (author unknown) and University of Western Ontario (author unknown)

Debugging

- Debugging is a developed skill. Some things to go over, though, so they'll be concrete in our brains:
  - relation to testing
  - why debugging is hard
  - types of bugs
  - process
  - techniques
  - tools
  - avoiding bugs

Debugging and testing

- Testing and debugging go together like peas in a pod:
  - Testing finds errors; debugging localizes and repairs them.
  - Together these form the "testing/debugging cycle": we test, then debug, then repeat.
  - Any debugging should be followed by a reapplication of *all* relevant tests, particularly regression tests. This avoids (reduces) the introduction of new bugs when debugging.
  - Testing and debugging need not be done by the same people (and often should not be).

Why debugging is hard

- There may be no obvious relationship between the external manifestation(s) of an error and its internal cause(s).
- Symptom and cause may be in remote parts of the program.
- Changes (new features, bug fixes) in program may mask (or modify) bugs.
- Symptom may be due to human mistake or misunderstanding that is difficult to trace.
- Bug may be triggered by rare or difficult to reproduce input sequence, program timing (threads) or other external causes.
- Bug may depend on other software/system state, things others did to your systems weeks/months ago.

Designing for Debug/Test

- when you write code think about how you are going to test/debug it
  - lack of thought *always* translates into bugs
- write test cases when you write your code
- if something should be true assert() it
- create functions to help visualize your data
- design for testing/debugging from the start
- test early, test often
- test at abstraction boundaries

Testing vs Debugging

- **Testing**: to identify any problems before software is put to use
  - *"Testing can show the presence of bugs but can never show their absence".*
- **Debugging**: locating bugs and fixing them

Fault Injection

- many bugs only happen in the uncommon case
- make this case more common having switches that cause routines to fail
  - file open, file write, memory allocation, are all good candidates
- Have “test drivers” which test with the uncommon data. If deeply buried, test with a debugger script

Finding and Fixing Bugs

- in order to create quality software you need to find your bugs
  - testing
  - user reports
- the best bugs are those that are always reproducible

Types of bugs

- Types of bugs (gotta love em):
  - Compile time: syntax, spelling, static type mismatch.
    - Usually caught with compiler
  - Design: flawed algorithm.
    - Incorrect outputs
  - Program logic (if/else, loop termination, select case, etc).
    - Incorrect outputs
  - Memory nonsense: null pointers, array bounds, bad types, leaks.
    - Runtime exceptions
  - Interface errors between modules, threads, programs (in particular, with shared resources: sockets, files, memory, etc).
    - Runtime Exceptions
  - Off-nominal conditions: failure of some part of software of underlying machinery (network, etc).
    - Incomplete functionality
  - Deadlocks: multiple processes fighting for a resource.
    - Freeze ups, never ending processes

Common Program Errors (bugs)

- Compiler errors (syntax errors)
- Runtime errors
- Logic errors

Compiler Errors

- ***Syntax error***
  - Error in usage of Java
  - Detected by the compiler
  - A program with compilation errors cannot be run
- ***Syntax warning***
  - Warning message generated by the compiler
  - The program can be run

Compiler Errors

- Very common (but sometimes hard to understand). Examples of syntax errors:
  - Forgetting a semicolon
  - Leaving out a closing bracket }
  - Redeclaring a variable
  - Others?

Compiler Errors

- Hints to help find/fix compiler errors:
  - Compiler errors are cumulative: when you fix one, others may go away
  - Read the error messages issued by the compiler!
  - Realize that the error messages from the compiler are often (seemingly) not very helpful
    - The compiler does not know what you intended to do, it merely scans the Java code

Runtime Errors

- ***Runtime error***: program runs but gets an *exception* error message
    - Program may be terminated
- Runtime errors can be caused by
    - Program bugs
    - Bad or unexpected input
    - Hardware or software problems in the computer system

Runtime Errors

- Very common runtime errors are:
  - null reference (NullPointerException)
    - no object is referenced by the reference variable, i.e. it has the value null
  - array index out of bounds (ArrayIndexOutOfBoundsException)
  - Running out of memory
    - e.g. from creating a new object every time through an infinite loop

Runtime Errors

- Hints to help find/fix runtime errors:
  - Check the exception message for the method and line number from which it came
  - Note that the line in the code that caused the exception may ***not*** be the line with the error
    - Example: consider the code segment

```
int [] nums = new int[10];
for (int j=0; j<=10; j++)
    nums[j] = j;
```

    - The exception will be at the line

```
    nums[j] = j;
```

      but the error is in the *previous* line

Logic Errors

- **Logic error**: program runs but results are not correct

- Logic errors can be caused by:
  - incorrect algorithms

Logic Errors

- Very common logic errors are:
  - using == instead of the *equals* method
  - infinite loops
  - misunderstanding of operator precedence
  - starting or ending at the wrong index of an array
    - If index is invalid, you would get an exception
  - misplaced parentheses (so code is either inside a block when it shouldn't be, or vice versa)

Logic Errors

- Be careful of where you declare variables!
  - Keep in mind the scope of variables
    - Instance variables?
    - Formal parameters?
    - Local variables?
  - Example:

```
private int numStudents; // an attribute, to be
                         // initialized in some method
…
public void someMethod(){
   int numStudents = …; // not the attribute!
    …
}
```

The ideal debugging process

- A debugging algorithm for software engineers:
  - Identify test case(s) that reliably show existence of fault (when possible)
  - Isolate problem to small fragment(s) of program
  - Correlate incorrect behavior with program logic/code error
  - Change the program (and check for other parts of program where same or similar program logic may also occur)
  - Regression test to verify that the error has really been removed - without inserting new errors
  - Update documentation when appropriate

(Not all these steps need be done by the same person!)

General Advice

- try to understand as much of what is happening as possible
- “it compiles” is **NOT** the same as “it works”
- when in doubt, ask. Then test the answer!
- Error messages are generally just a vague hint and can be misleading.
- Don’t always trust the “comments/documents”, they can be out-of-date.

Debugging Strategies

- Trace your code by hand
- Use breakpoints to trace the code and inspect values
- Add main method to the class
- Add print statements to your code

Tracing by Hand

- Tracing by hand
  - Good starting point with small programs or simple methods
  - Problem: sometimes you do what you think the computer will do, but that is not what it actually does
    - Example: you may write that 9/5 is 1.8, but it is really 1
- Hint: draw diagrams of reference variables and what object(s) they are pointing to!

Debuggers

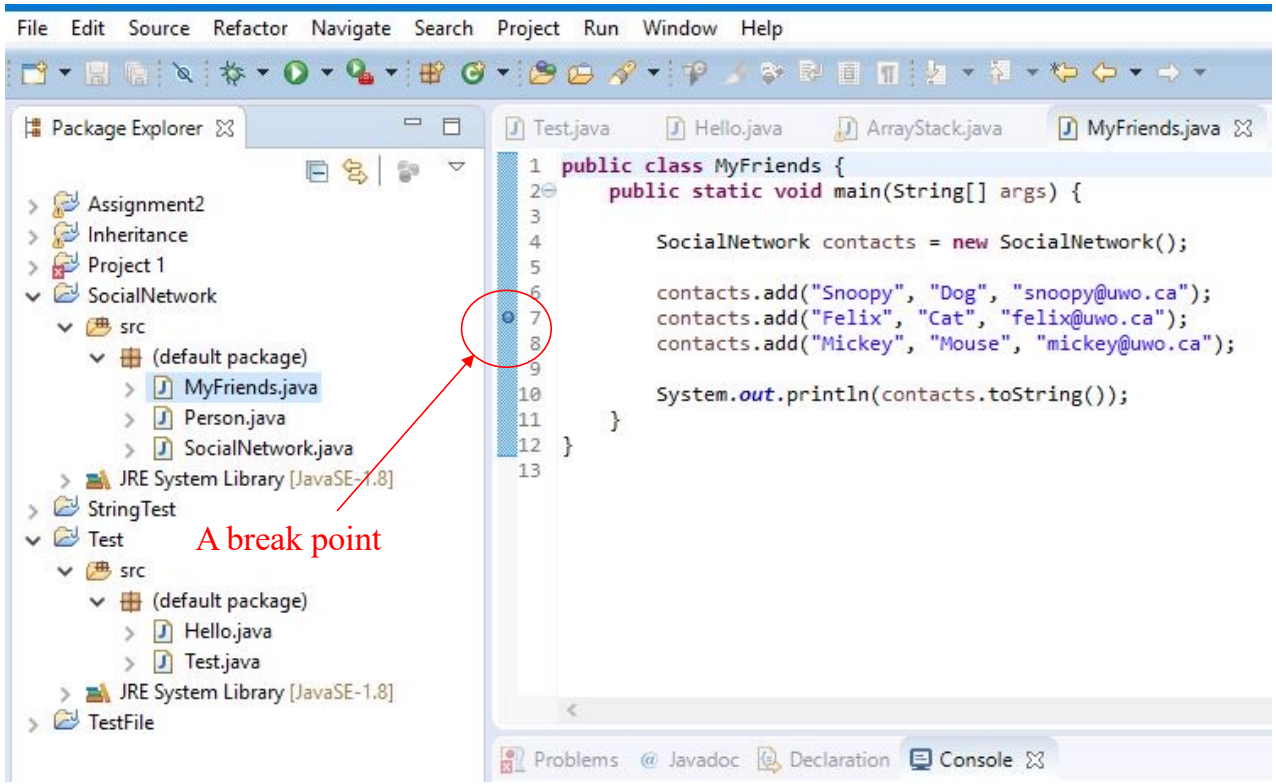
- All Integrated Development Environments have an ***interactive debugger*** feature
  - You can single-step step through your code (one statement at a time)
  - You can see what is stored in variables
  - You can set breakpoints
  - You can "watch" a variable or expression during execution

Use breakpoints to trace the code and inspect values

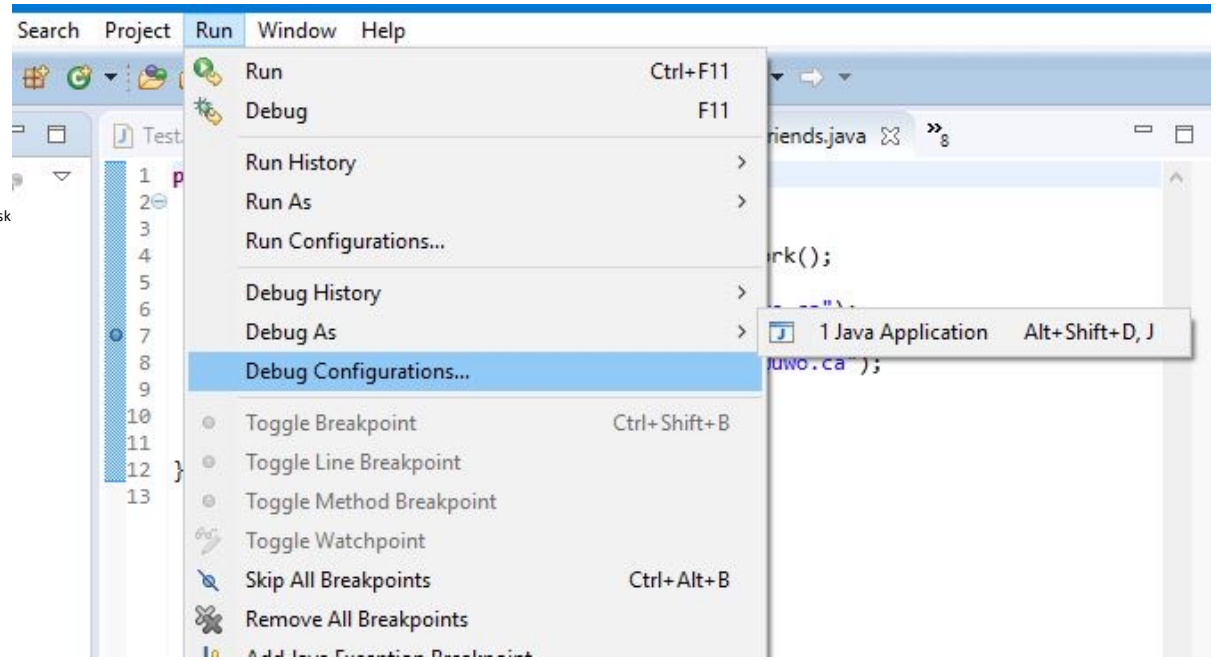
Introduction to Eclipse's Debugger

Debugging a Program

1. Add breakpoints: double-click the blue bar on the left side of ***Edit*** window or right click on the bar and select “toggle breakpoint”. A blue dot indicates a breakpoint. To remove a break point, double click the breakpoint.

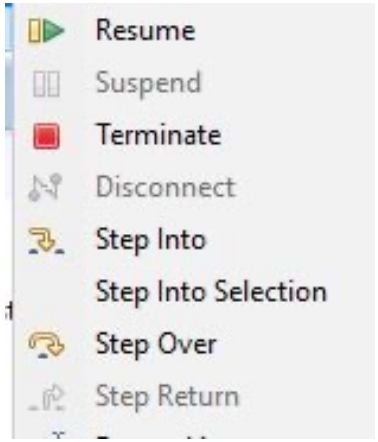


2. Select ***Run->Debug as...->Java Application*** to start the debugger.



If ask

4. Click on Run and then try the debug commands to see what they do and see how the values of the variables change in the ***Variable*** window and what the outputs are in the ***Console*** window.



Resume resume the execution of a paused program.

Suspend temporarily pause the execution of a program.

Terminate end the current debug session.

Step Into execute a single statement or step into a method.

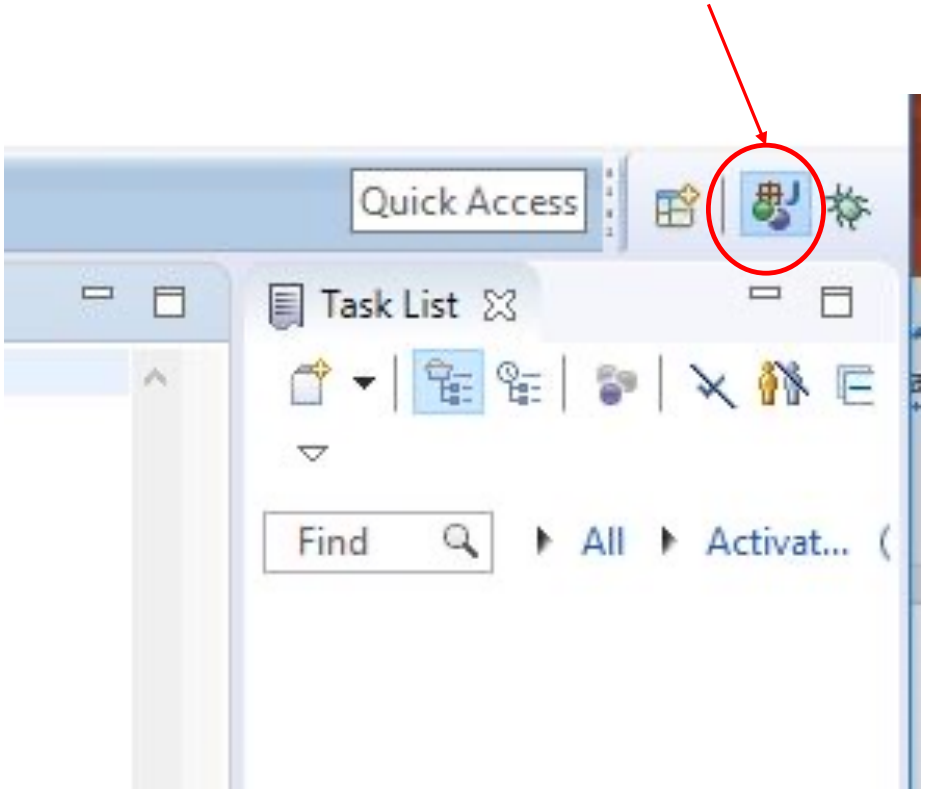
Step Into Selection While debugger is stopped on a break point, put cursor on a method you want to step into

Step Over execute a single statement. If the statement contains a call to method, the entire method is executed without stepping into the method.

Step Return execute all the statements in the current method and returns to the caller.

5. Switch Eclipse from ***Debug Perspective*** back to ***Java Perspective***.

- Click ***on the Java Perspective button***



Adding a main Method

- Adding a main method to the class
  - Conventionally placed at the end of the class code, after all the other methods
  - What are the advantages of having the test driver (main method) right in the class, rather than creating another class that is just a test program?

Bug hunting with print

- Weak form of debugging, but still common
- How bug hunting with print can be made more useful:
  - print variables other than just those you suspect.
  - print valuable statements (not just “hi\n”).
  - use exit() to concentrate on a part of a program.
  - move print through a program to track down a bug.

Debugging with print (continued)

Building debugging with print into a program (more common/valuable):

- print messages, variables/test results in useful places throughout program.
- use a ‘debug’ or ‘debug_level’ global flag to turn debugging messages on or off, or change “levels”
- possibly use a source file preprocessor (#ifdef) to insert/remove debug statements.
- Often part of “regression testing” so automated scripts can test output of many things at once.

Using Print Statements

- Using print statements
  - Insert System.out.println() statements at key locations
    - To show values of significant variables
    - To show how far your code got before there was a problem
  - In the print statement, it's a good idea to specify
    - The location of the trace (what method)
    - The variable name as well as its value

Defensive Programming

- Write *robust programs*
  - Include checking for exceptional conditions; try to think of situations that might reasonably happen, and check for them
    - Examples: files that don't exist, bad input data
- Generate appropriate error messages, and either allow the user to reenter the data or exit from the program
- Throw exceptions
  - Can aid in finding errors or in avoiding errors
  - Example: invalid arguments (IllegalArgumentException)

Hints for Success

- When writing code:
  - Understand the algorithm before you start coding!
  - Start small!
    - Write and test first simpler methods (e.g. getters, setters, toString)
    - Then write and test each of the more complex methods individually
- Check your code first by a preliminary hand trace
- *Then* try running it