Testing and Debugging

Program Errors

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

Compiler Errors

- ***Syntax errors***
  - Errors 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

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 errors***: 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 (very rare)

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)

    We will talk later about exceptions.
  - 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 errors**: program runs but results are not correct

- Logic errors can be caused by:
  - Incorrect algorithms. These errors are the most difficult to fix. It is very important that you spend sufficient time designing your algorithms and making sure they are correct before you implement them in Java.

Logic Errors

- 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
  - Example:

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

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

Hints for Success

- When writing code:
  - Make sure your algorithm is correct 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

Debugging Strategies

- Trace your code by hand
- Add main method to the class
- Add print statements to your code
- Use a debugger

Tracing by Hand

- Tracing by hand
  - Good starting point for 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 `int i = 9/5;` assigns to `i` the value 1.8, but it is really 1
- Hint: draw diagrams of reference variables and what object(s) they are pointing to.

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 harness (main method) right in the class, rather than creating another class that is just a test program?

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

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

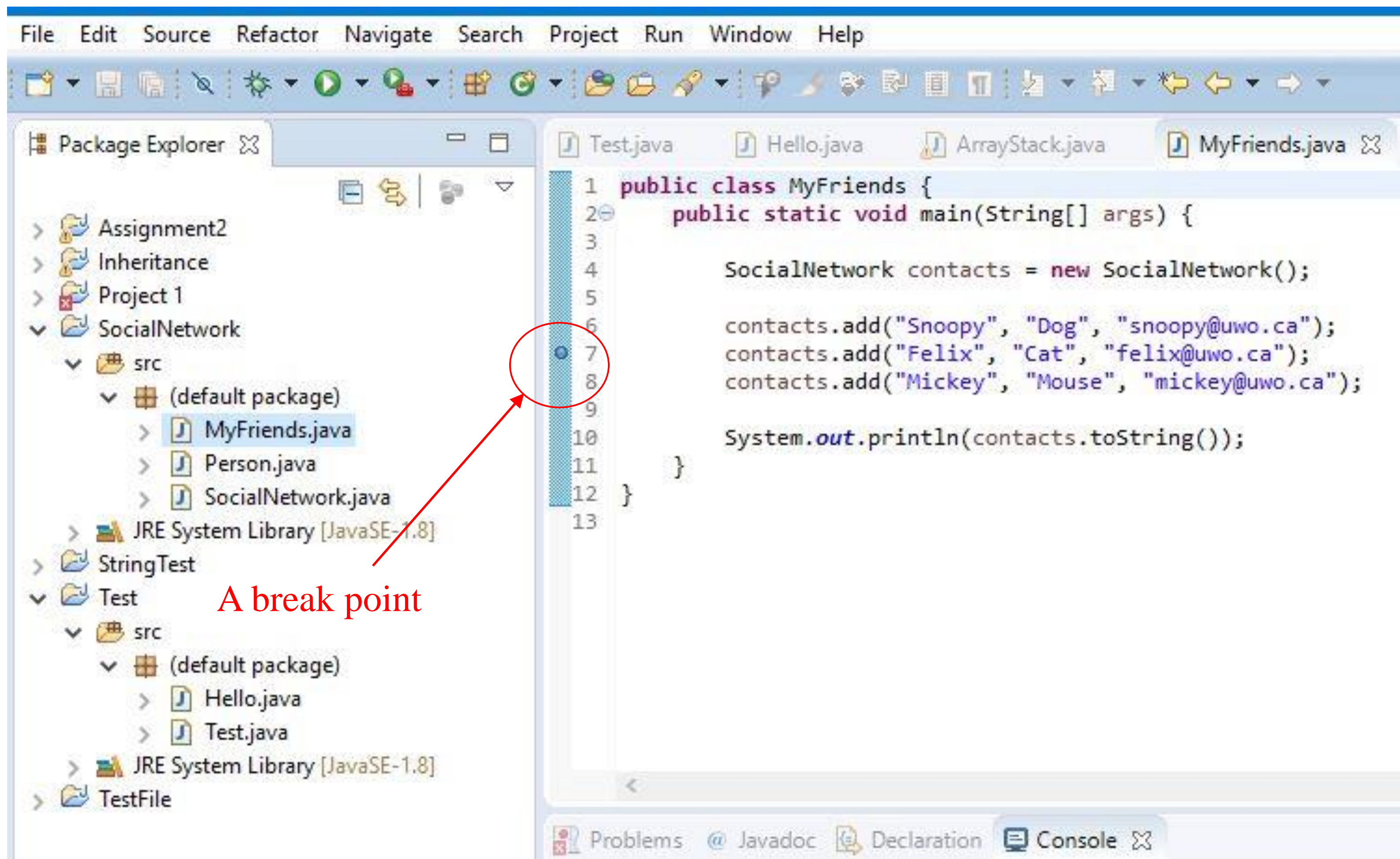
Defensive Programming

- 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 (we will cover this topoc later)

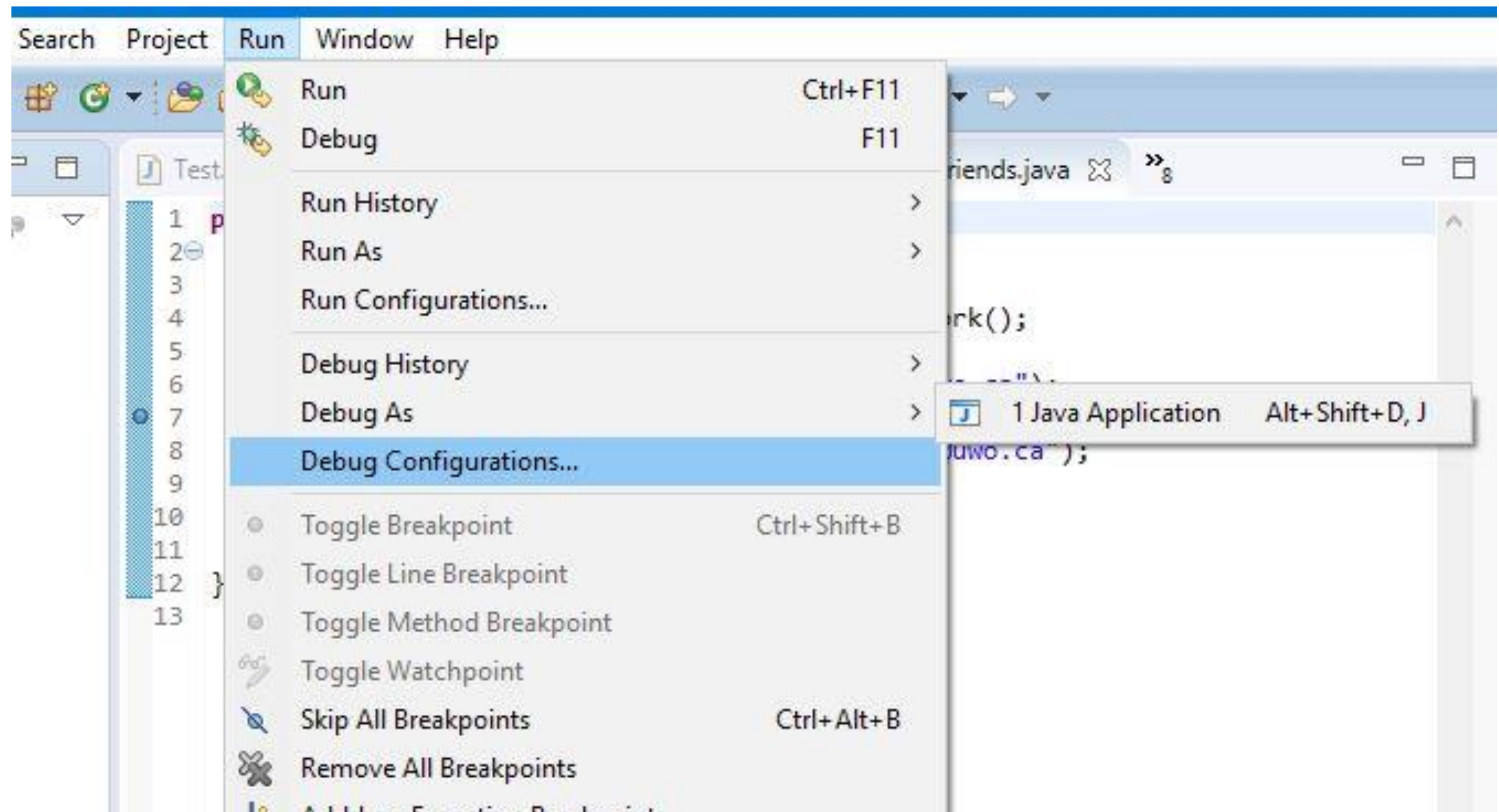
Introduction to Eclipse's Debugger

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.

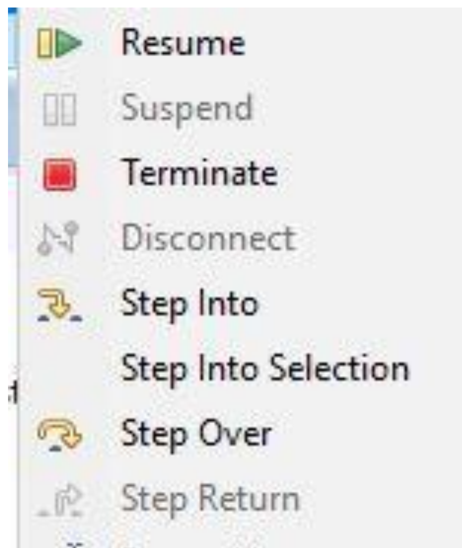


2. Select ***Run->Debug as...->Java Application*** to start the debugger.



If asked, accept to switch to debug Perspective.

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***