Java Debugging & Profiling Techniques

Courtesy for some of the slides:

Ethan Henry egh@klg.com

KL Group http://www.klgroup.com

Overview

This session will touch on three major topics:

- Debugging

  and

- Profiling

  plus

- Memory Leaks in Java

Debugging

- What's debugging?
- If you don't already know…
- Debugging is the process of finding errors in your software, regardless of what kind of errors they are

Types of Errors

- Program errors can be lumped into one of three broad categories:
  - syntactic errors
  - static errors
  - dynamic errors
- Syntactic errors are the easiest to find, dynamic errors are the most difficult

Syntactic Errors

- A syntactic error is one in which the program is ill-formed by the standard of the programming language
  - e.g. a keyword is misspelled
- Syntactic errors are trivial to detect and correct
  - the compiler does the detection for you
  - which is one reason compiled languages are so popular

Static Errors

- Static errors are ones in which the program is syntactically correct but semantically incorrect
- For example, substituting a ">" for a "<"
- Static errors can be very difficult to detect but are usually quite obvious
- Java's exception mechanism helps deal with these types of errors

Dynamic Errors

- The dynamic error is the most insidious kind of errors
- Dynamic errors occur in code which is both semantically and syntactically correct but where implicit assumptions have been violated
- The prevention of dynamic errors is a major focus of software engineering

Dynamic Errors

- An example of a dynamic error:
  - the most dramatic example I could find
- The Ariane 5 crash on June 4, 1996 was due to a *reuse* error
  - reuse of an inertial reference system software component caused a 64-bit integer to be incorrectly converted into a 16-bit unsigned integer, causing an exception that went uncaught and crashed the guidance system
  - http://www.eiffel.com/doc/manuals/technology/contract/ariane/index.html

Debugging Techniques

- Debugging is an art as old as programming is, so there are a lot of approaches to it
- For Java, we can divide debugging techniques into two categories:
  - “Classic”
  - Java-specific

Classic Debugging Techniques

- print statements
- assert

Jurassic Debugging

- println statements
  - System.out.println
    - connected to the standard output stream
  - System.err.println
    - connected to the standard error stream
    - both are instances of java.io.PrintStream
- helps with static & dynamic errors
- difficult to deal with in general

toString()

- The one very useful thing about using println to debug is that every object has a toString() method
  - inherited from Java.lang.Object
- This means that you can always do things like this:

```
Frame f = new Frame("Main App");
System.out.println("The frame info is
"+f);
```

toString()

- By default, toString() just prints out an object identifier, so you should override it in your own classes
- Tip: The first thing you should do in a toString method is create a StringBuffer to hold the data you’re printing

Using println

- How can you make debugging statements easier to deal with?
  - Conditional code
    - via if
    - via a pre-processor
  - a debugging class

Using println

- If use an expression that's constant at compile time, if statements act like conditional code and will be evaluated at compile time

```
static final boolean DEBUG = true; // or
   false

…

if(DEBUG) {

   System.out.println("Some message");

}
```

Using println

- Another option is to use an actual pre-processor, like in C/C++
- cpp, perl, m4, tcl - any general scripting language should work
- This is more complex, but is marginal extra work in a large project
- Advantage over using if:
  - you can completely strip out the debugging code if you want

Using println

- Another technique is to use a class that logs errors
  - typically a singleton
- For example:

```java
import java.io.*;

public final class Log {

private static Log log;
private PrintStream stream;
private boolean printing;

private Log() {
    stream = System.out;
    printing = true;
}

static {
    log = new Log();
}

public static Log getLog() {
    return log;
}
public void setPrinting(boolean b) {
    printing = b;
}
public void setStream(OutputStream os) {
    if(os instanceof PrintStream)
        stream = (PrintStream)os;
    else
        stream = new PrintStream(os);
}

public void println(String msg) {
    if(printing)
        stream.println(msg);
}

public void print(String msg) {
    if(printing)
        stream.print(msg);
}

}
```

JUnit Assert

```
class Money {

    private int fAmount;
    private String fCurrency;

   public Money(int amount, String currency) {
          fAmount= amount;
          fCurrency= currency;     }

   public int amount() {       return fAmount;    }

      public String currency() {       return fCurrency;    }

  public Money add(Money m) {
        return new Money(amount()+m.amount(),
  currency()); }
}
```

JUnit Assert

```
public class MoneyTest extends TestCase {
//…
   public void testSimpleAdd() {

         Money m12CHF= new Money(12, "CHF");  // (1)

         Money m14CHF= new Money(14, "CHF");

         Money expected= new Money(26, "CHF");

         Money result=  m12CHF.add(m14CHF);    // (2)

         Assert.assertTrue(expected.equals(result));     // (3)
   }
}
```

(1) Creates the objects we will interact with during the test. This testing context is commonly referred to as a test's *fixture*. All we need for the testSimpleAdd test are some Money objects.

(2) Exercises the objects in the fixture.

(3) Verifies the result

Java-Specific Techniques

- Exception Handling
- Reading Exception Messages
- Thread Dumps

Exception Handling

- An exception is “thrown” when some unexpected condition occurs at runtime
- Only instances of java.lang.Throwable (or a subclass) can be thrown by the `throw` statement
- Throwable has two subclasses:
  - java.lang.Error
  - java.lang.Exception

Errors

- Error objects are thrown by the virtual machine or Java library to indicate a serious problem
- Most applications shouldn’t try to catch or throw an Error object
- The predefined errors are:

Errors

- java.awt.AWTError
- java.lang.LinkageError
- java.lang.ClassCircularityError
- java.lang.ClassFormatError
- java.lang.ExceptionInInitializerError
- java.lang.IncompatibleClassChangeErr
  or
- java.lang.AbstractMethodError
- java.lang.IllegalAccessError
- java.lang.InstantiationError
- java.lang.NoSuchFieldError
- java.lang.NoSuchMethodError
- java.lang.NoClassDefFoundErro
  r
- java.lang.UnsatisfiedLinkError
- java.lang.VerifyError
- java.lang.ThreadDeath
- java.lang.VirtualMachineError
- java.lang.InternalError
- java.lang.OutOfMemoryError
- java.lang.StackOverflowError
- java.lang.UnknownError

Exceptions

- An exception indicates some sort of problem that a typical application might want to deal with
- There are two types of exceptions:
  - checked exceptions
  - unchecked exceptions

Checked Exceptions

- Direct subclasses of java.lang.Exception represent exceptions that the developer *must* deal with
- These exceptions can only be thrown if declared in the `throws` clause of the method
- Any code calling a method that throws a regular exception *must* catch it and deal with it

Checked Exceptions

- Some examples of checked exceptions are:
  - java.lang.ClassNotFoundException
    - thrown by Class.forName()
  - java.io.IOException
    - thrown by many I/O operations
  - java.lang.IllegalAccessException
    - thrown by Class.newInstance()
  - java.lang.InterruptedException
    - thrown by sleep and wait

Unchecked Exceptions

- Unchecked exceptions need not be declared in the `throws` clause of a method and do not need to be caught
- These are exceptions that are too frequent to check for every time
- Unchecked exceptions are derived from java.lang.RuntimeException instead of java.lang.Exception
  - RuntimeException is derived from Exception though

Unchecked Exceptions

- Some example of unchecked exceptions:
  - java.lang.ArithmeticException
  - java.lang.ClassCastException
  - java.lang.IllegalArgumentException
  - java.lang.NumberFormatException
  - java.lang.ArrayIndexOutOfBoundsException
  - java.lang.StringIndexOutOfBoundsException
  - java.lang.NegativeArraySizeException
  - java.lang.NullPointerException

Handling Exceptions

- Exceptions are dealt with via the `throw`, `try`, `catch` and `finally` keywords
- The code that encounters a problem creates an exception by instantiating some exception object and throwing it

```
throw new SomeBadException(“some
message”);
```

Handling Exceptions

- The exception then unwinds the stack of the thread it was thrown in, looking for a try block
- If there is a catch associated with the try block that matches the type of the exception, the catch block is executed
- After the catch block executes (or if no exception was thrown) the finally block associated with the try block is executed

Handling Exceptions

- If no handler is found, the current thread's ThreadGroup's `uncaughtException(Thread,Throwable)` method is called
  - if there is a parent ThreadGroup, pass it the exception
  - otherwise print the exception stack trace to System.err

Handling Exceptions

```
import java.io.IOException;
public class Exception {
public static void main(String args[]) {
   try {
    foo(args[0]);
    System.out.println("whew");
   }
   catch(IOException e) {
    System.out.println("doh!");
   }
   finally {
    System.out.println("finally...");
   }
   System.out.println("done!");
}
```

Handling Exceptions

```
public static void foo(String s) throws IOException {
    if(s.equals("checked")) {
     throw new IOException("Checked");
    }
    else if(s.equals("unchecked")) {
     throw new RuntimeException("Unchecked");
    }

    System.out.println("No exception");
}


}
```

Reading Exception Messages

- A typical exception stack trace looks something like this:
  - from a modified version of the ‘Fractal’ example that comes with the JDK

```
java.lang.NullPointerException
        at ContextLSystem.<init>(CLSFractal.java:319)
        at CLSFractal.init(CLSFractal.java:66)
        at sun.applet.AppletPanel.run(AppletPanel.java:287)
        at java.lang.Thread.run(Thread.java:474)
```

Reading Exception Messages

- Information shown:
  - the type of exception

```
java.lang.NullPointerException
```

  - the class & method in which the exception was thrown

```
at ContextLSystem.<init>(CLSFractal.java:319)
```

  - the full stack trace for the thread executing the code at that point

```
at CLSFractal.init(CLSFractal.java:66)
at sun.applet.AppletPanel.run(AppletPanel.java:287)
at java.lang.Thread.run(Thread.java:474)
```

Interactive Debuggers

- Free:
  - jdb
  - JBuilder Foundation (IDE)
    - http://www.borland.com/jbuilder
  - NetBeans Developer/Forte for Java (IDE)
    - http://www.netbeans.com
- Commercial:
  - Metamata Debug
    - http://www.metamata.com
  - Karmira BugSeeker
    - http://www.karmira.com

Profiling

- What is Profiling?
- What Profiling Tells You
- What Profiling Is Not
- Manual Profiling
- Profiling Techniques Overview
  - Insertion
  - Sampling
  - Instrumented VM

What is Profiling?

- Profiling is measuring an application, specifically:
  - where is the time being spent?
    - This is "classical" profiling
    - which method takes the most time?
    - which method is called the most?
  - how is memory being used?
    - what kind of objects are being created?
    - this in especially applicable in OO, GC'ed environments

What Profiling Tells You

- Basic information:
  - How much time is spent in each method? (“flat” profiling)
  - How many objects of each type are allocated?

What Profiling Tells You

- Beyond the basics:
  - Program flow (“hierarchical” profiling)
    - do calls to method A cause method B to take too much time?
  - Per-line information
    - which line(s) in a given method are the most expensive?
    - Which methods created which objects?
  - Visualization aspects
    - Is it easy to use the profiler to get to the information you’re interested in?

Profiling Techniques Overview

- Commercial profilers use one of three techniques for profiling programs:
  - insertion
  - sampling
  - instrumented virtual machine
- Why is it important to understand how a profiler works?
  - each different technique has its own pros & cons
  - different profilers may give different results

Insertion

- Multiple flavours:
  - Source code insertion
    - profiling code goes in with the source
    - easy to do
  - Object code insertion
    - profiling code goes into the .o (C++) or .class (Java) files
    - can be done statically or dynamically
    - hard to do
      - modified class loader

Insertion Pros & Cons

- Insertion Pros:
  - can be used across a variety of platforms
  - accurate (in some ways)
    - can't easily do memory profiling
- Insertion Cons:
  - requires recompilation or relinking of the app
  - profiling code may affect performance
    - difficult to calculate exact impact

Sampling

- In sampling, the processor or VM is monitored and at regular intervals an interrupt executes and saves a “snapshot” of the processor state
- This data is then compared with the program’s layout in memory to get an idea of where the program was at each sample

Sampling Pros & Cons

- Sampling Pros:
  - no modification of app is necessary
- Sampling Cons:
  - a definite time/accuracy trade-off
    - a high sample rate is accurate, but takes a lot of time
  - more…

Sampling Pros & Cons

- Sampling Cons:
  - very small methods will almost always be missed
    - if a small method is called frequently and you have are unlucky, small but expensive methods may <u>never</u> show up
  - sampling cannot easily monitor memory usage

Instrumented VM

- Another way to collect information is to instrument the Java VM
- Using this technique each and every VM instruction can be monitored
    - highly accurate

Instrumented VM Pros&Cons

- Pros:
  - The most accurate technique
  - Can monitor memory usage data as well as time data
  - Can easily be extended to allow remote profiling
- Cons:
  - The instrumented VM is platform-specific

Instrumented VMs

- Java 2 (JDK 1.2 and higher)
  - information can be accessed through JVMPI
  - the Java Virtual Machine Profiling Interface
    - http://java.sun.com/products/jdk/1.3/docs/guide/jvmpi/
- JProbe Profiler
  - http://www.javaperformancetuning.com/tools/jprobe/

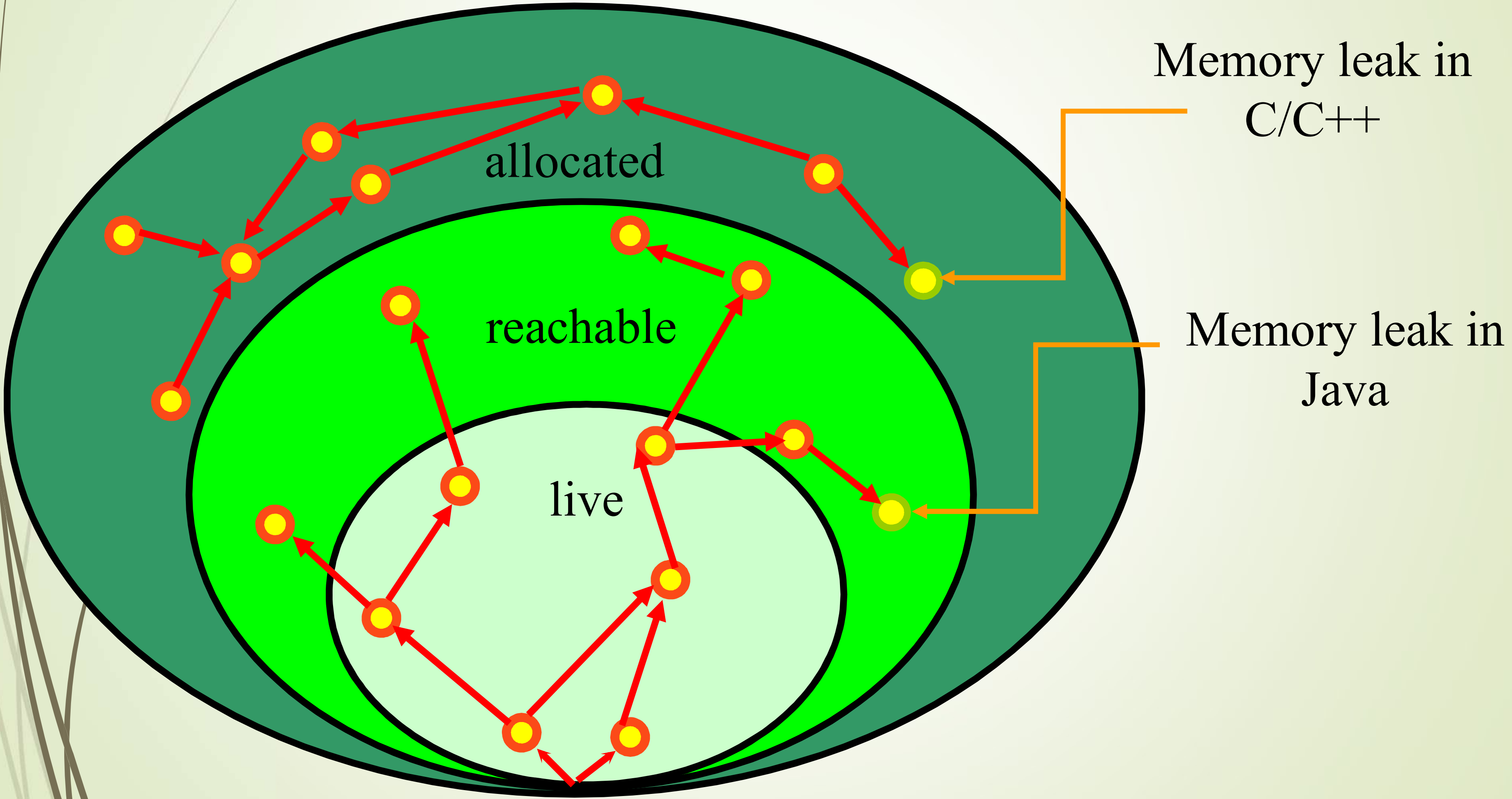
Memory Leaks

- Java doesn't have "memory leaks" like a C or C++ program, but…
- Some Java programs exhibit classic "memory leak" behavior:
  - their memory usage grows, unbounded, over time
- Memory leaks usually also lead to performance problems once the process starts getting swapped out
- https://stackify.com/memory-leaks-java/

What is a Memory Leak?

- Allocated
  - exists on the heap
- Reachable
  - a path exists from some root to it
- Live
  - program may use it along some future execution path

What is a memory leak?



Memory Leaks

- So, how can you identify & track down memory leaks?
- This is almost impossible to do without a memory analysis tool
- Some tools:
  - Jprobe (http://www.javaperformancetuning.com/tools/jprobe/)
  - OptimizeIt

Debugging in NetBeans IDE

Use Case

What is debugging?

- Run through the code with the interpreter.
- Allows to see whether the execution path is as expected.

Using the Debugger

We can use the debugger:

- to verify if a programm behaves as we except.
- to identify the nature of a runtime-error.
- to force the programm entering a given state.

Debugging in NetBeans IDE

Howto

Fist steps

- Open a project in NetBeans IDE.
- Define the *Breakpoints* at some interesting points.
- Start the Debugger

Debugging mode

- Runs the code with the Java virtual machine.
- Everything that is possible in normal mode, should also be possible in debugging mode.
- Execution stops for user action at the defined breakpoints.

Breakpoints I

Defining Breakpoints

Definition (Breakpoint)

A Breakpoint is an indication for the Java Debugger. They are ignored by the Java virtual machine when running in normal mode. However when running in debug mode, the execution stops at every breakpoint and waits for a user action before it continues to the next break points.

Actions

When the execution of the programm stops at a breakpoint following actions are possible:

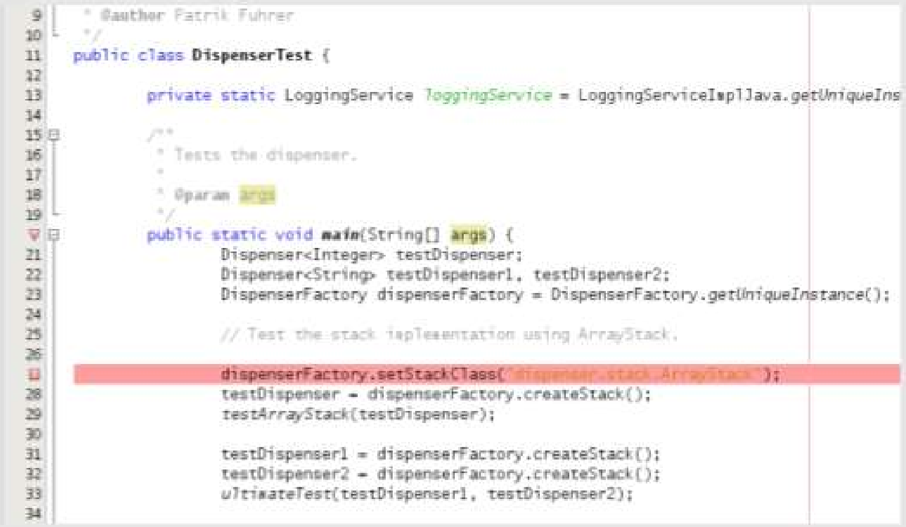
- Inspection or modification of dynamic variables.
- Step-by-step execution.
- Go to the next breakpoint (or to the end of the application if there are no more breakpoints).

Breakpoints in NetBeans IDE

Defining breakpoints

Example (Usage)

- Define a Breakpoint by clicking on the corresponding line.
- The line number is replaced by a red square.
- It's not possible to define breakpoints on non executable lines (i.e. commentary lines)



```
     * @author Patrik Fuhrer
     */
public class DispenserTest {

        private static LoggingService loggingService = LoggingServiceImplJava.getUniqueIns

        /**
         * Tests the dispenser.
         *
         * @param args
         */
        public static void main(String[] args) {
                Dispenser<Integer> testDispenser;
                Dispenser<String> testDispenser1, testDispenser2;
                DispenserFactory dispenserFactory = DispenserFactory.getUniqueInstance();

                // Test the stack implementation using ArrayStack.

                dispenserFactory.setStackClass("dispenser.stack.ArrayStack");
                testDispenser = dispenserFactory.createStack();
                testArrayStack(testDispenser);

                testDispenser1 = dispenserFactory.createStack();
                testDispenser2 = dispenserFactory.createStack();
                ultimateTest(testDispenser1, testDispenser2);

```

Running the Debugger

The NetBeans IDE way

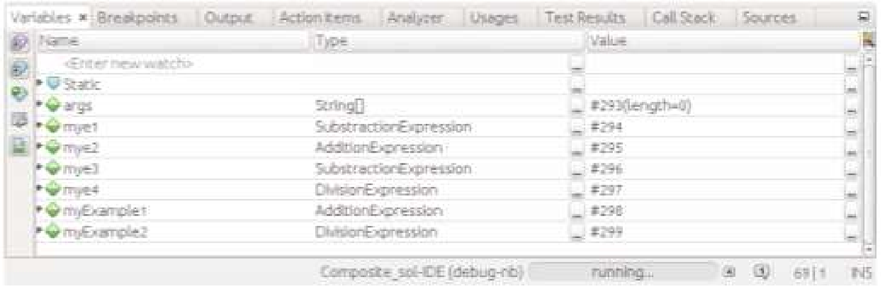
Debugging with and IDE

- It is possible (and even recommended) to do the debugging tasks directly from the IDE.
- NetBeans IDE offers a great debugging support.
- To launch the debugger, either click the debug icon in the launch bar (the 6th one), or select the debug target from the ANT script or select debug from the right click on the project.
- Once started the program starts as normal but the execution stops at the first breakpoint.

The debugging console

NetBeans IDE layout

- When the debugger is started in NetBeans IDE a new tab becomes visible.
- This tab is updated on each step of the debugger.
- It shows the value of all fields of the current class.
- Besides inspecting the value of each field, it is also possible to change them.



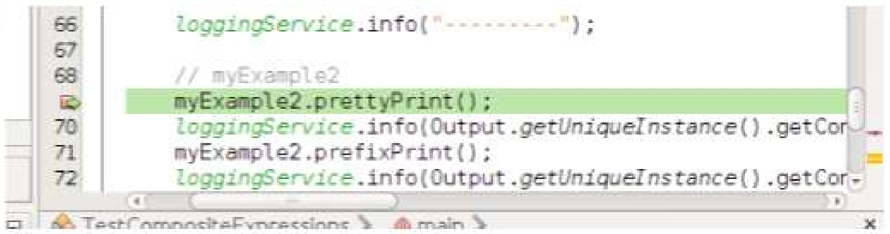
Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#293(length=0)
mye1	SubstractionExpression	#294
mye2	AdditionExpression	#295
mye3	SubstractionExpression	#296
mye4	DivisionExpression	#297
myExample1	AdditionExpression	#298
myExample2	DivisionExpression	#299

Composite_sol-IDE (debug-nb) running... 69 | 1 INS

Using the NetBeans IDE Debugger

Associated GUI Elements

- When the debugger stops on a line, this line is highlighted green.



```
    loggingService.info("---------");

    // myExample2
    myExample2.prettyPrint();
    loggingService.info(Output.getUniqueInstance().getCor
    myExample2.prefixPrint();
    loggingService.info(Output.getUniqueInstance().getCor
```

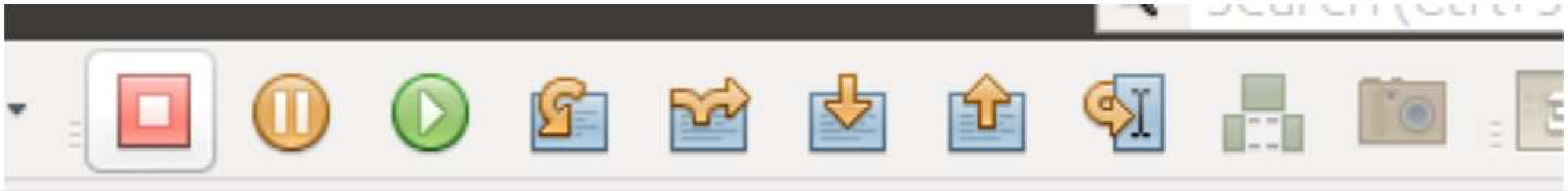
- All other debugging GUI elements relate to this line or the class containing this line.

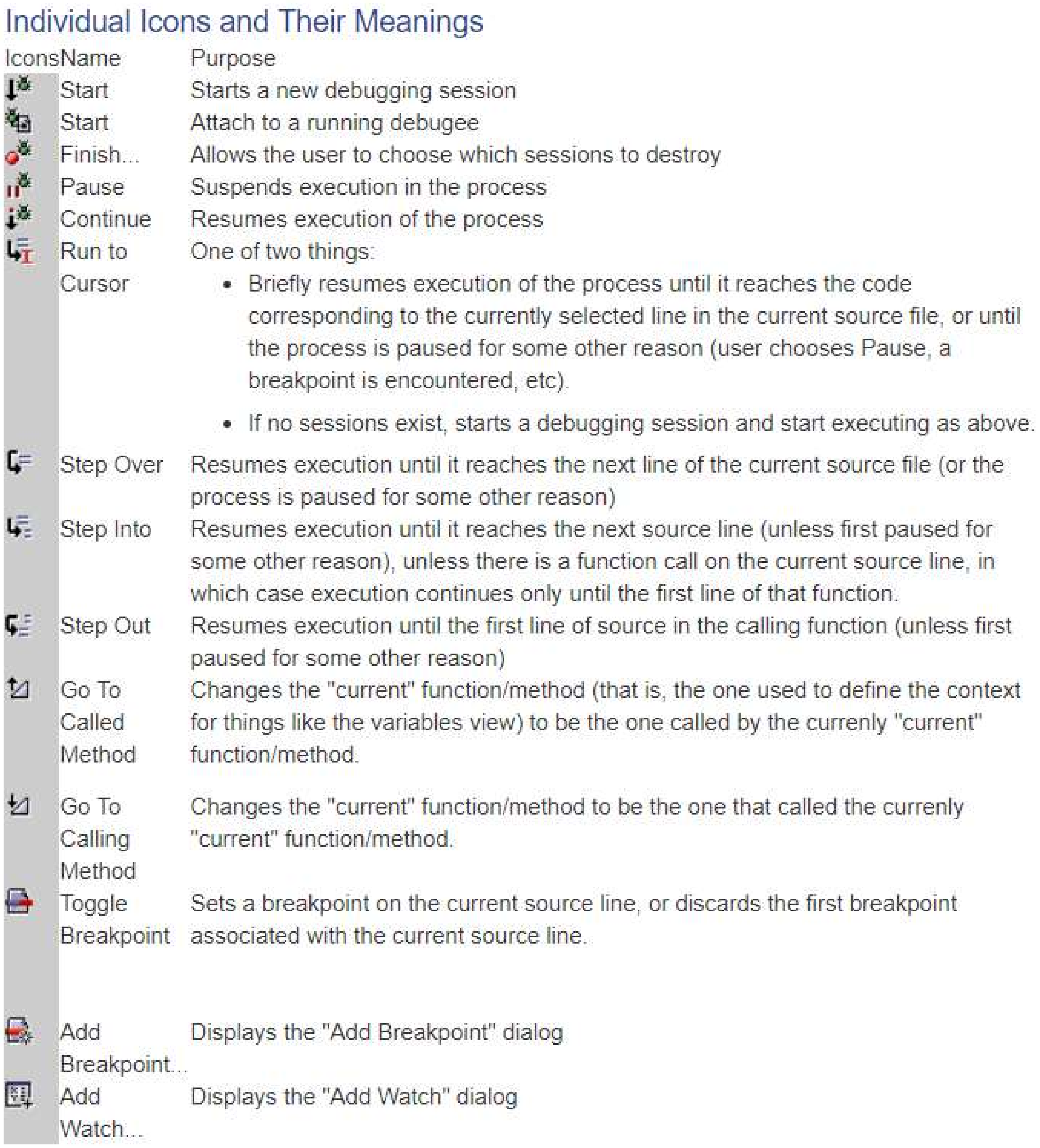
Using the NetBeans IDE Debugger

Associated GUI Elements

When everyhting seems ok to continue, NetBeans IDE offers several buttons for going on:

- Stop (the debugger)
- Pause
- Continue (to next Breakpoint)
- Step Over
- Step over Expression
- Step into
- Step out
- Run to cursor
- Apply Code Change
- Take GUI Snapshot





Individual Icons and Their Meanings

Icons	Name	Purpose
	Start	Starts a new debugging session
	Start	Attach to a running debugee
	Finish...	Allows the user to choose which sessions to destroy
	Pause	Suspends execution in the process
	Continue	Resumes execution of the process
	Run to Cursor	One of two things: • Briefly resumes execution of the process until it reaches the code corresponding to the currently selected line in the current source file, or until the process is paused for some other reason (user chooses Pause, a breakpoint is encountered, etc). • If no sessions exist, starts a debugging session and start executing as above.
	Step Over	Resumes execution until it reaches the next line of the current source file (or the process is paused for some other reason)
	Step Into	Resumes execution until it reaches the next source line (unless first paused for some other reason), unless there is a function call on the current source line, in which case execution continues only until the first line of that function.
	Step Out	Resumes execution until the first line of source in the calling function (unless first paused for some other reason)
	Go To Called Method	Changes the "current" function/method (that is, the one used to define the context for things like the variables view) to be the one called by the currenly "current" function/method.
	Go To Calling Method	Changes the "current" function/method to be the one that called the currenly "current" function/method.
	Toggle Breakpoint	Sets a breakpoint on the current source line, or discards the first breakpoint associated with the current source line.
	Add Breakpoint...	Displays the "Add Breakpoint" dialog
	Add Watch...	Displays the "Add Watch" dialog

Step Over ( ) - Executes one line of code. If the line is a call to a method, **step over** executes the entire method as a single command, without stepping into the method's code.

Step Into ( ) - Executes one line of code. If the line is a call to a method, **step into** steps into the method and stops on the first line of the method.

Step Out ( ) - Executes one line of code. However, if the source code line is in a method, it will execute all the remaining code in the method and return to the method's caller.

Run to Cursor ( ) - Executes the program to the line where the typing cursor is located.

Continue ( ) - Continues the execution of program at full speed until the next breakpoint or until the program terminates.

Example demo

- Loan calculator

Questions?