Shivaram Lingamneni
CS 106A

Handout #39
February 8, 2010

Debugging Tactics

This handout is intended as a companion to the debugging handout, which discusses general debugging ideas and provides a tour of the Eclipse interactive debugger. This handout covers some Java-specific issues you will likely encounter as you work on the programming assignments. It also describes a debugging technique that is an alternative to the interactive debugger.

Introduction

When you try to solve any kind of problem, mistakes are inevitable. Programming is no exception. The good news is that programming mistakes (in particular, Java programming mistakes) can be divided fairly neatly into three categories:

1. *Compile-time errors*. These are those red Xs and squiggly red lines that you see in Eclipse. Compile-time errors arise, essentially, when Java isn't sure what you want it to do. For example, if you have an `int` variable called `count`, but misspell it and write

   ```
   conut++;
   ```

   Java won't know what variable it should add 1 to. Thus, it refuses to run your code until you resolve the issue. While compile-time errors can be annoying, they are generally the easiest category of mistakes to fix.

2. *"Silent" run-time bugs*. These happen when your code compiles and runs normally—but then fails to do what you expect. In these situations, you have to analyze your code until you find the part of it that doesn't accurately represent the behavior you want. These can be the most difficult mistakes to fix.

3. *Runtime exceptions*. These happen when your code compiles and begins running, but then Java realizes that it is about to do something inherently wrong. An example is division by zero. The following is a perfectly reasonable line of Java:

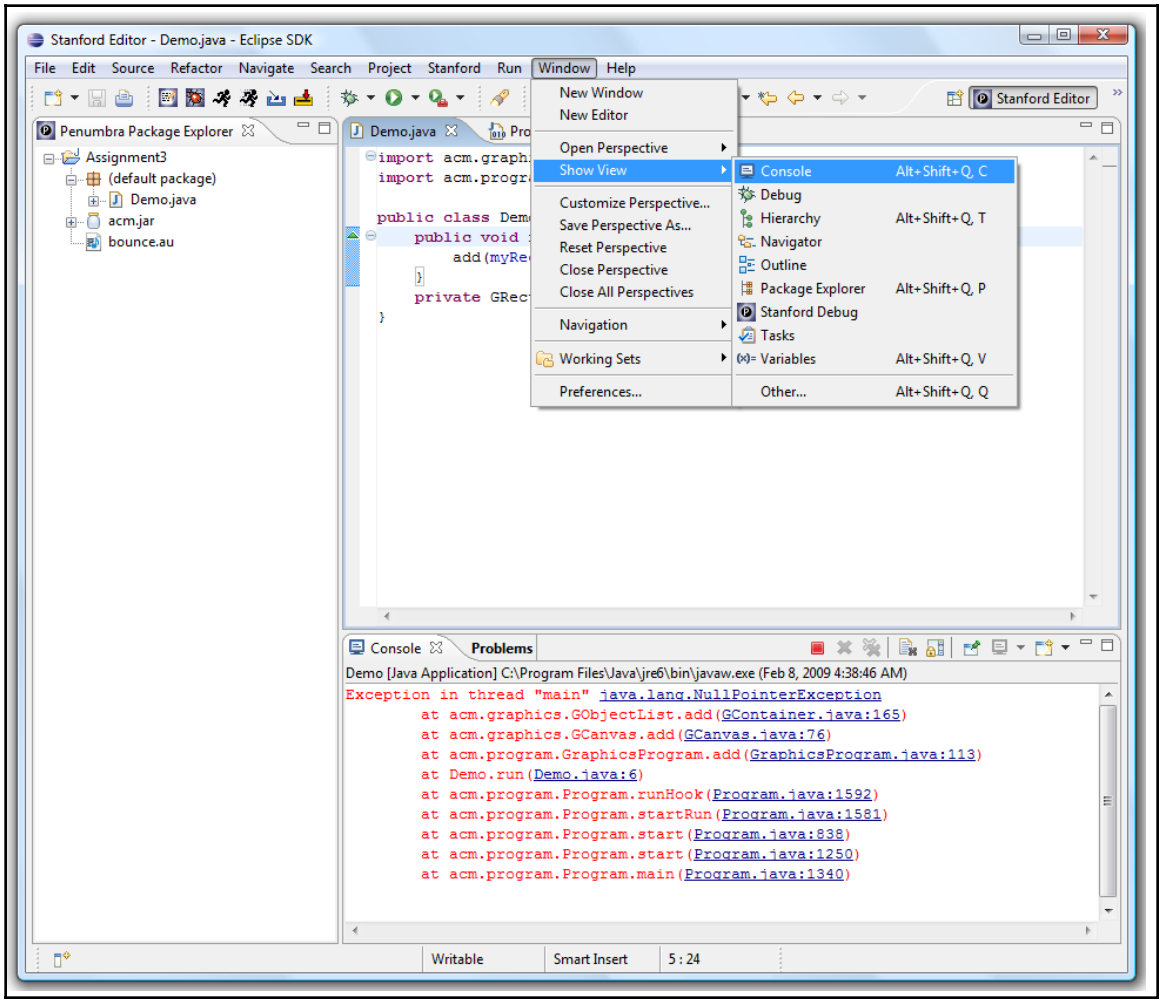
   ```
   int quotient = a / b;
   ```

   but what should happen if when the line executes, `b` has the value 0? Certainly it wouldn't be good for Java to leave quotient unchanged, or assign it some arbitrary value, and continue running your code—this would lead to highly confusing behavior. Instead, in this and similar situations, Java produces a special kind of error event/message called a **runtime exception** and stops your program. This handout covers how to recognize and deal with runtime exceptions.

How to read a stack trace

When Java throws a runtime exception, the execution of your program will typically stop, and an error message will be printed to the Java console. The Java console typically shows up in Eclipse as a pane on the bottom right; if it's not visible, you can make it show up by going to **Window->Show View->Console**, as shown in Figure 1 at the top of the next page. You can think of the Java console as a sort of miniature `ConsoleProgram`, to which your program can send text output. In general, if the execution of your program seems to stop, or your program is behaving strangely, check the console for an exception. The text describing the exception will display in red.

Figure 1. Screen image of console window



Consider the following code:

```
import acm.graphics.*;
import acm.program.*;

public class Demo extends GraphicsProgram {
   public void run() {
      add(myRect);
   }
   private GRect myRect;
}
```

You can probably already see the bug in this program. In **`run()`**, **`myRect`** is used uninitialized; in other words, we never create a new **`GRect`** and assign it to the variable **`myRect`**. Thus, our call to **`add()`** is trying to draw a nonexistent object on the screen, and the result is an exception.

This program produces the output in the Eclipse console window shown in Figure 1. This list of method names is called a **stack trace.** It describes the sequence of method calls that resulted in the exception—our **`run()`** method called **`add(myRect)`**, which called a different **`add()`** method, which called yet another, at which point the exception was

generated ("thrown" in Java parlance). Stack traces can be disorienting because frequently, many of the methods and classes they refer to will be unknown to you. For example, you may not be familiar with the **`runHook()`** method of the **`Program`** class, or know what a **`GObjectList`** is. Fear not! Generally, all this incomprehensible information will be irrelevant to your bug.

Here's a simple rule for reading stack traces: *Go to the highest line in your own code*. As we look down from the top of the stack trace, we see that we didn't write **`GContainer`**, or **`GCanvas`**, or **`GraphicsProgram`**—but we did write the **`run()`** method of the **`Demo`** class. That's the place to begin looking for the cause of the bug. In this case, the line

```
at Demo.run(Demo.java:6)
```

takes us directly to the culprit. It won't always be so easy. The code in the highest line may not itself be buggy; instead, it may be in a correctly implemented method that was called with incorrect arguments, or that was called while the instance variables were in an incorrect state. In that case, go down the trace one line at a time until you can isolate and fix the mistake.

Null pointer exceptions

The **`NullPointerException`** (NPE) is probably the runtime exception you'll encounter the most often. It is caused when you attempt to use an object that has not been initialized, or has otherwise been set equal to **`null`**. Recall that just as an uninitialized **`int`** variable gets set to a default value of 0, an uninitialized object variable gets set to a default value of **`null`**. However, **`null`** is not an object and cannot be used as such; attempts to do so will result in NPEs.

What does "use" mean in this context? Here are some things that will cause a **`NullPointerException`**:

1. Calling any method of a **`null`** object. This is definitely the most common cause. For example:

   ```
   GRect myRect;
   myRect.setSize(10, 10);
   ```

   will produce an NPE, because the **`setSize`** method cannot be invoked on a **`null`** object.

2. Attempting to index or read the length of a **`null`** array. For example:

   ```
   int[] myInts;

   public void run() {
      if (myInts.length == 0) println("myInts is empty");
   }
   ```

   throws an NPE from the attempt to read the length field of **`myInts`**.

In general, once you've found the line at which an NPE is occurring, look for a period—that indicates a method call or a field access on a variable which may be null. If you don't see a period, look for square brackets.

Note that assigning to a **`null`** variable will not produce a **`NullPointerException`**. After all, if it did, you would never be able to initialize any object variable! If a

NullPointerException has been generated on a line with an equals sign in it, the cause is on the right-hand side of the equals sign.

Debugging with println

When debugging a program that manifests incorrect runtime behavior, frequently you'll want to do one of two things:

1. *Find out which lines of your code are actually being executed*. When you write code with complicated conditional statements (such as **if** statements and **while** loops), it may not be clear what's actually happening when you click **Run**. If, for example, some **GObject** on your screen is supposed to vanish, and it's not vanishing, you may want to find out if execution is ever reaching your attempt to **remove()** it.
2. *Inspect the value of a variable*. Particularly in the case of instance variables, it can be unclear how a variable changes over time (although minimizing the confusion should definitely be a design goal for you!). Frequently when debugging, you'll think to yourself, "What's the value of that variable here?"

The interactive debugger provides ways to do both of these things, but it's not always the best way. Depending on the situation, it may be overkill, providing too much information. At other times, configuring it to get the behavior you want can be difficult and confusing, especially if you're new to programming.

There is a simple, intuitive way to do these things—in fact, you may have already discovered it yourself when working on **ConsolePrograms**. In a **ConsoleProgram**, you have access to a method called **println()** that prints output on the screen. The line:

```
println(count);
```

prints the value of the variable **count**. The line:

```
println("here");
```

prints **here** to the screen—every time it is "hit" during the execution of your program. So if you stick it in a **while** loop, you'll get a **here** on the screen for every iteration of the loop. And if you don't see any **here**s, then you'll know that the condition was initially **false**, and that the loop was never entered at all.

In a **GraphicsProgram**, the output of **println()** goes to the Eclipse console, which means that you can use much the same strategy there. You can't, however, use the **println()** directly when you are in the context of a class you've defined that is not itself a subclass of **Program**—you'll get a compile-time error if you try. But there's hope: the entirely analogous method **System.out.println()** will work from any point in any Java program, and its output will go to the Java console (the same place exception stack traces go to). And **System.out.println()** accepts more than just numbers and strings. Any Java object can be fed into it (but not all objects will produce equally informative messages).

Two caveats with **System.out.println()** deserve mention. First, all output from it looks the same, unless you label it. For example, if your code is sprinkled with the call

```
System.out.println(count)
```

you'll end up seeing a bunch of numbers, and it may not be clear which calls generated which numbers. A good pattern is to give each **println** a prefix naming its origin:

```
System.out.println("in for loop, count is " + count)
```

Secondly, if you find yourself adding **println** after **println** without making much progress, it may be time to switch to the interactive debugger. The two techniques have complementary strengths. The debugger is good for tracing through complicated logic, and when you have some idea of what the bug is and you need to work out the specifics. **println()** is good for initial exploration of a problem, or when you can point at a few places in your code and say "I want to know what's happening there, there, and there."

In fact, one technique can transition into the other quite naturally. Maybe you've added several calls to **println()**, each inspecting a variable; at the time of the fourth **println()**, the variable has the value you expect, but at the time of the fifth, its value is incorrect. A natural way to proceed is to set a breakpoint on the fourth **println**, then step through the code from there, observing how the variable's value changes line-by-line.

Have fun with **System.out.println()**! Remember to comment out or delete your calls to it before you submit your code. Even if Java console output doesn't detract from the functionality of your program, it's bad style to leave it in.

While we're on the subject of debugging techniques, here's an important one: Google knows about your error messages! If Eclipse is complaining about "unreachable code," or you see **ClassCastException** pop up in your console, entering the message into Google (possibly adding the keyword "Java") will generally prove informative. Messages that are specific to your code (for example, if the message contains a variable name) generally won't work as well, though. Similarly, Googling the name of any common (or less common) Java class will take you to its **javadoc** page, which describes the class's function and lists all methods that it supports. Here's a link to the **javadoc** page for the **String** class (it's also the first Google hit for "Java string"):

```
http://java.sun.com/j2se/1.3/docs/api/java/lang/String.html
```

Appendix: Other runtime exceptions you may encounter

1. ArrayIndexOutOfBoundsException

This happens when you try to access an element of an array that isn't present. For example, here we create an array of 6 strings, then try to access the 7th string (which is at index 6—recall that the 6 elements are at indices 0, 1, 2, 3, 4, and 5).

```
String[] array = new String[6];
System.out.println(array[6]);
```

Eclipse displays the exception in its console window, like this:

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 6
    at Demo.run(Demo.java:12)
    at acm.program.Program.runHook(Program.java:1592)
    at acm.program.Program.startRun(Program.java:1581)
    at acm.program.Program.start(Program.java:838)
    at acm.program.Program.start(Program.java:1250)
    at acm.program.Program.main(Program.java:1340)
```

Notice that the exception message includes the offending index (6).

2. `IndexOutOfBoundsException`

This exception is exactly the same thing, except thrown by an object such as an **`ArrayList`** instead of a regular array. It may come with some debugging information:

```
ArrayList<String> arr = new ArrayList<String>();
String str = "element 0 is " + arr.get(0);

Exception in thread "main"
    java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
    at java.util.ArrayList.RangeCheck(Unknown Source)
    at java.util.ArrayList.get(Unknown Source)
    at Demo.run(Demo.java:10)
    at acm.program.Program.runHook(Program.java:1592)
    at acm.program.Program.startRun(Program.java:1581)
    at acm.program.Program.start(Program.java:838)
    at acm.program.Program.start(Program.java:1250)
    at acm.program.Program.main(Program.java:1340)
```

The stack trace reports the illegal index (0) and the size of the **`ArrayList`** at the time it was indexed (also 0).

3. `StringIndexOutOfBoundsException`

Again, exactly the same—only this time it's specific to indices in strings.

```
String str = readLine();
if (str.charAt(0) == 'A') println("the string begins with an A");
```

If the user enters an empty string, there is no first character to read, and:

```
Exception in thread "main"
    java.lang.StringIndexOutOfBoundsException:
    String index out of range: 0
    at java.lang.String.charAt(Unknown Source)
    at Demo.run(Demo.java:12)
    at acm.program.Program.runHook(Program.java:1592)
    at acm.program.Program.startRun(Program.java:1581)
    at acm.program.Program.start(Program.java:838)
    at acm.program.Program.start(Program.java:1250)
    at acm.program.Program.main(Program.java:1340)
```

It isn't just **`charAt()`** that throws this exception—**`substring()`** will also produce it if you give it a bad range.

4. `NumberFormatException`

This means that you tried to turn a **`String`** into an **`int`** or a **`double`** (using **`Integer.parseInt()`** or **`Double.parseDouble()`**), but the **`String`** was of the wrong format (it didn't look like "-1" or "2.71828"). As with the previously described exceptions, the exception message should include some debugging information (in this case, the invalid **`String`**).

5. `ArithmeticException`

This is produced by integer divisions by zero.