Lab 7: In Java, Undefined Behaviors are the Exception!

Andrew Tolmach (with minor revisions by Mark P. Jones)

CS 320 Principles of Programming Languages, Spring Term 2019
Department of Computer Science, Portland State University

Learning Objectives

Upon successful completion, students will be able to:

- Write simple Java programs involving numbers, booleans, strings, arrays, classes used as record types, command line arguments, file I/O, exceptions, overloading, and packages.
- Contrast Java vs. C/C++ treatment of programmer errors including null pointers, out-of-bounds references, arithmetic errors, and uninitialized variables.

Instructions

The code for this lab is distributed with a set of subdirectories, 00-11. Each contains a Java program called `Example.java`; some directories contain other files. To compile and run the contents of each directory, `cd` to it and type

```
$ javac Example.java
$ java Example
```

We'll walk through many or all of these directories, in sequence, in the following sections of this document, some of which also include exercises for you to work through. At the very least, you should try compiling and running each example with various inputs. Some directories will have code in C or C++ as well; directions for compiling the associated files are detailed in the corresponding sections below.

Command-line String Arguments (00)

We begin with the minimal example program from the lecture, which simply echoes the command line arguments.

"00/Example.java"

```
class Example {
  public static void main(String argv[]) {
    for (int i = 0; i < argv.length; i++) {
      System.out.println(argv[i]);
5        }
6      }
7    }
```

A few things to note:

- The file contains just one top-level class definition, `Example`, whose name is the same as that of the file. In general, this is good practice, although it is legal for a file to contain multiple top-level class definitions. It is also common to define classes nested inside other classes, as we'll see later.
- Line 2 declares the top-level method for the program, called `main` just as in C/C++. Every program must contain at least one class that defines `main` with the signature used here. The `static` keyword says that the method does not have an associated object. (Indeed, the `Example` class is never intended to be instantiated.) The `public` keyword says that the method is globally visible within the program (more on this later when we discuss packages). The `void` keyword says that the function returns no value (unlike `main` in C/C++, which returns an integer status code).
- The method must have a single parameter of type array of `String`, as shown here, although the name of the argument is arbitrary. We use the name `argv` because this parameter contains essentially the same information as the `argc` and `argv` values passed to C/C++ `main` functions. Since a Java array carries its own length, there is no need for a separate count argument. Unlike in C/C++, the program name is *not* passed as the first element of the array.
- At Line 4, we use `System.out.println()` to print a string to standard output (i.e., the terminal unless redirected otherwise), followed by a new line. Unpacking this a little: `System` is a top-level library utility class (again, one that is never instantiated) that contains several useful `static` fields: `PrintStream` objects representing standard output (`out`) and standard error (`err`), and an `InputStream` object representing standard input (`in`). Static fields are instantiated just once per class; they act like global variables. Finally, class `PrintStream` defines the `println()` (non-static) method. It also defines a simpler method `print()` that omits the new line.

Converting Between Strings and Integers (01)

In this variant of the first program, we use the first command line argument as a repetition count and repeatedly print out the second command line argument.

"01/Example.java"

```java
1  class Example {
2    public static void main(String argv[]) {
3      int n = Integer.parseInt(argv[0]);
4      for (int i = 0; i < n; i++) {
5        System.out.println(Integer.toString(i) +  ": " + argv[1]);
6      }
7    }
8  }
```

To get the idea of how this program is supposed to work, try running it thus: `java Example 3 hello`

- At Line 3, we perform `String` to `int` conversion using a static method `parseInt(String s)` defined in another top-level library utility class `Integer`. (Unlike `System`, `Integer` *is* intended to be instantiated sometimes, but we're not doing so here.)

- At Line 5, we use a similar method `Integer.toString(int i)` to convert in the other direction.
- Line 5 also illustrates the use of the `+` operator to append strings. In fact, this operator is *overloaded* to work when just one argument is a `String` and the other is of some other type; in these cases, the non-string argument is internally converted to a `String` by the append operator. So a more idiomatic way to write Line 5 would be:

  ```
  System.out.println(i + " :  " + argv[1]);
  ```

  Overloading is not restricted to built-in operations like `+`; it can be used to define variants of any method that can be distinguished by having different parameter types. For example the `print()` and `println()` methods are overloaded to work on all the primitive types. The choice of which variant to call is made by the compiler based on the type of the actual argument provided by the caller.

Although this code works fine when given the kinds of command line arguments it expects, it isn't very robust against badly formed arguments. Before moving on to the next section, try to induce this code to behave badly. You should be able to cause at least two kinds of uncaught exceptions to occur.

Simple Exception Handling (02)

One of the things that makes Java safer than C/C++ is that it has no unchecked runtime errors: any language-level error that isn't prevented statically will cause a runtime exception to be thrown. If not explicitly caught by the program, the exception will terminate execution with an error message and a stack trace back to the exception point.

Exception values are just another kind of object in Java, belonging to some subclass of the overall `Exception` class. In the previous example, you should have been able to induce an `ArrayIndexOutOfBoundsException` by providing fewer than two command line arguments, and a `NumberFormatException` by providing a first argument that does not represent a valid integer.

In general, it is a good policy for programs to catch or prevent exceptions that might be caused by conditions other than buggy code, such as ill-formed input. In general, there are two ways to avoid getting an uncaught exception from a library routine: catch the exception, or prevent it from occurring by checking the parameters to the routine. The latter is typically preferable if we can manage it. We use both methods in this revised program.

"02/Example.java"

```
class Example {
  public static void main(String argv[]) {
    if (argv.length !=2) {
      usage();
    } else {
      int n = 0;
      try {
        n = Integer.parseInt(argv[0]);
      } catch (NumberFormatException e) {
        usage();
      }
      for (int i = 0; i < n; i++) {
        System.out.println(i + ": " + argv[1]);
      }
    }
  }
```

```
17
18      private static void usage() {
19        System.err.println("usage: java Example count string");
20        System.exit(1);
21      }
22    }
```

Here we issue an old-fashioned Unix command line tool "`usage`" message (telling the user what kinds of arguments are expected) if we don't get exactly two arguments of which the first is a valid integer count. At Line 3 we test the length of `argv` before trying to reference any of its elements. This should guarantee no out-of-bounds errors, and also checks that the user didn't provide *too many* arguments.

To deal with the potentially failing `Integer.parseInt()` call we place it inside a `try...catch` statement at Lines 7–11. The semantics of this statement are as follows: the code at Line 8, between `try` and `catch`, is executed. If no exceptions are raised during this execution, the `catch` block is ignored and the program continues at Line 12. If some exception is thrown, its value is checked against the exception class name given in the `catch` at Line 9. If the thrown exception is of the same class as (or a subclass of) `NumberFormalException`, control passes to the body of the `catch` block at Line 10, with the exception object itself bound to `e`, which is sometimes useful. In this case, the `catch` block code simply invokes `usage()` and does not make use of `e`. In general, there can be multiple `catch` clauses, corresponding to different exceptions; the first matching clause gets control. If no `catch` clause matches, the exception is *propagated* up to the next enclosing `try...catch` statement, or, if there is none, to the caller of the current method, where it might be handled.

Although handling exceptions in this way is good form, it also complicates the code considerably. So in the remainder of this lab, we will generally *not* handle exceptions explicitly, but instead assume that our input is well-formed. If it is not, the program will still halt in a timely fashion–it just won't give a nice response to the user. This is still much better than having no checks at all. In fact, the two exceptions that can happen in this code are in a subclass of `Exception` called `RunTimeException`, which Java uses to indicate that reasonable applications may not want to bother catching them.

Exercise: Loops (03)

The code in this directory is just a copy of the `Example` from directory (02). Modify it so that the command line can have one or more arguments; as before, the first argument is the count, and *all* the remaining arguments (0 or more of them) are concatenated together (with a separating space) to form the string to be echoed. For example, running

```
java Example 2 morning noon     night
```

should produce the output

```
0: morning noon night
1: morning noon night
```

There are several simple ways to do this.

Reading from a File (04)

Since the amount of input we can get from the command line is limited, we next consider reading values from a file. This code in this directory reads a set of double-precision floating point numbers from a named file and computes and prints their mean, variance, and (population) standard deviation. (Google if you don't know how these quantities are defined.) The format of the file is a single integer count n followed by n double-precision numbers; adjacent numbers must be separated by at least one whitespace character. The file `data` gives an example of a legal input file.

The code to read a named file into an array of doubles is encapsulated into a routine `readDoubleArray`.

"04/Example.java"

```java
    private static double[] readDoubleArray(String filename) throws IOException {
      FileReader f = new FileReader(filename);
      Scanner sc = new Scanner(f);
      int n = sc.nextInt();
      double[] ds = new double[n];
      for (int i = 0; i < n; i++) {
        ds[i] = sc.nextDouble();
      }
      f.close();
      return ds;
    }
  }
```

This code makes use of utility classes `FileReader`, which describes a file open for reading, and `Scanner`, which (among other things) can read space-separated strings and convert them to numbers using the `nextInt()` and `nextDouble()` methods.

To access these classes, we need to understand a bit about Java *packages*. A package is a top-level collection of classes (quite similar to a C++ `namespace`) that form a separate name space. That is, the same class can be defined in two different packages without causing a naming conflict. The Java standard library is divided into several packages. For example, `FileReader` lives in the `java.io` package and `Scanner` in the `java.util` package. To access these classes, we can *import* some or all of the packages into our program. The lines at the top of the file

"04/Example.java"

```java
  import java.io.*;
  import java.util.*;
```

make all the classes in these two packages visible. We also use two routines from the `Math` utility class; like the `System` and `Integer` classes we used before, this lives in a package `java.lang` which is imported by default.

Both the `FileReader` constructor and `Scanner` methods can raise exceptions (e.g., if the file name does not exist, there are not enough numbers in the file, or one of the numbers has the wrong format). Recall that our general policy in this lab is not to worry about exception handling. But the `FileReader` exceptions are subclasses of a class `IOException`, which is *not* a subclass of `RunTimeException`. Java really expects these exceptions to be caught by the application; because we don't want to do that, we must add a clause to the signature of every dynamically enclosing method (in this case `readDoubleArray` and `main`) warning that it might raise the exception.

We also introduce a new form of `for` loop within the body of method `mean`:

"04/Example.java"

```java
      for (double d : ds) {
        sum += d;
      }
```

(There is a similar example in method `variance`). The body of the loop executes once for each value `d` in the array `ds`. This form is available not just for arrays, but for objects of any class that defines a suitable way of *iterating* through its elements; this should remind yout of the `for` loop construct that we've seen already in Python, which behaves in a similar way.

Exercise: Computing a histogram (05)

Write a program `Example.java` to compute data to construct a histogram of double-precision values read from a file. A histogram is a bar chart in which the length of each bar gives the number of items that fall into a certain range of values, usually called a *bin*. You won't actually be drawing a bar chart, but instead will print out the size of each bin.

Your program should take four command line arguments:

- The name of a file containing an array of floating point numbers in the same format as in example `04`.
- An integer b giving the number of bins to sort into.
- A double min of the range of values of interest.
- A double max of the range of values of interest.

You can assume (without checking) that $b > 0$ and $min \leq max$.

Divide the range of values of interest into b equal-sized bins. Count the number of values from the file that fall into each bin. (Count a number falling on the edge between two bins as belonging to the bin containing larger values.) Also count the number of values that are completely below or above the range.

For example, given this test file `data1`:

"05/data1"

```
14
18.0 1.0 1.5 1.9 2.5 3.5 4.5 5.5 2.5 3.5 4.5 5.5 6.5 -2.0
```

the output of `java Example data1 5 0.0 10.0` should be

```
x < 0.0: 1
0.0 <= x < 2.0: 3
2.0 <= x < 4.0: 4
4.0 <= x < 6.0: 4
6.0 <= x < 8.0: 1
8.0 <= x < 10.0: 0
x >= 10.0: 1
```

and given test file `data2` in the directory, the output of `java Example data2 5 -10.0 10.0` should be

```
x < -10.0: 0
-10.0 <= x < -6.0: 2022
-6.0 <= x < -2.0: 1946
-2.0 <= x < 2.0: 1994
2.0 <= x < 6.0: 2053
6.0 <= x < 10.0: 1985
x >= 10.0: 0
```

Notes:

- Reuse the `readDoubleArray` method from `04/`.
- Java has a `Double.parseDouble()` method analogous to `Integer.parseInt()`.
- To keep life simple, don't worry about doing detailed error checking on the command line parameters.
- Don't worry about behavior due to imprecision in floating point computations, which may make your output look ugly and cause strange borderline behavior. Floating point computation is not the focus of this exercise!

Objects and References (06)

The Java program here defines an extremely simple class `P` containing a single integer field, and illustrates how it can be manipulated.

"06/Example.java"

```java
3      static class P {
4        int a;
5        P (int a) { this.a = a; }
6      }
7
8      static void twiddle(P x, P y) {
9        P z = x;
10       x   = y;
11       y   = z;
12     }
13
14     static void swizzle(P x, P y) {
15       int z = x.a;
16       x.a   = y.a;
17       y.a   = z;
18     }
19
20     static public void main(String argv[]) {
21       P p0 = new P(0);
22       P p1 = new P(1);
23       System.out.println (p0.a + " " + p1.a);
24       twiddle(p0, p1);
25       System.out.println (p0.a + " " + p1.a);
26       swizzle(p0, p1);
27       System.out.println (p0.a + " " + p1.a);
28     }
```

We define `P` as a *nested* class within `Example`. Because we are going to use `P` from within static methods (indeed, `Example` has no non-static methods and is never going to be instantiated as an object), we need to declare `P` as `static` as well.

Line 5 defines a constructor for `P` that takes the (sole) field value `a` as parameter and uses it to initialize the field. When we use classes as records like this we almost always need to provide a constructor, since Java's default constructor does not initialize fields.

Within the constructor (and any other member method) we can refer to the fields of P (here just a) directly by an unqualified name, unless that name is hidden by another binding. That's what happens in this constructor: the parameter is also named a, and it hides the field name. We could think up a different, non-conflicting, name for the parameter, but instead we use a standard idiom to reference the field using a *qualified* name this.a. The variable this always points to the object associated with the current method.

Suppose we want to write a method that in some sense "swaps" its two arguments (both of type P): twiddle and swizzle seem to be two attempts to do that. But neither one manages to swap the values of p0 and p1 in main. Examine the output of the program and try to figure how their behavior.

Because Java arguments are always passed by value, twiddle receives copies of p0 and p1 into x and y and exchanges them locally with the aid of temporary z. Since the effects of this manipulation are visible only within the function, the overall effect of twiddle is to do nothing at all. It certainly does not change the values of p0 or p1.

In fact, we know that *no* method can change the values of main's local variables. So it is fundamentally impossible to write a swap-like operation in Java. This is unlike C++, which would let us pass references to the locals, or C, which would let us take pointers to them and pass those.

However, although p0 and p1 will continue to point to the same objects no matter what method we call, it *is* possible to change the values *within* these objects. That's because a value of type P is in fact a pointer to the location containing the contents of P's fields. Method swizzle takes advantage of this to exchange the contents of the (sole) field of its arguments. Thus, a method *can* have observable effects on the heap seen by its caller when the method returns.

Note that swizzle relies on its ability to write the a field of its arguments. If we wish to prevent that, we can mark the field declaration as final, which means that the field can only be written once (in this case by the constructor) and thereafter remains constant. Local variables can also be marked final, and it is generally good practice to do so whenever possible. (The const qualifier in C/C++ has a somewhat similar intent, but differs in many details.)

Exercise: C vs. Java (07)

The C program example.c creates a linked list of a points and then sums the x values of the first b of them, where a and b are the command line arguments. Compile the C program as

```
gcc -o example example.c
```

and run it as ./example a b. Investigate the behavior of this program, in particular when $b > a$.

Note that the link and point structures are handled differently by this program. Links are always heap-allocated, and we refer to them by (explicit) pointers. To make this more convenient, we use a C type abbreviation

"07/example.c"
```
typedef struct link *Link;
```

to define Link to mean the same thing as struct link *. C doesn't have a new operator or constructors, but the function newLink serves a similar purpose.

"07/example.c"
```
Link newLink(struct point p, Link n) {
  Link link = malloc(sizeof(*link));
  link->p = p;
  link->n = n;
19      return link;
20    }
```

Points, on the other hand, are handled directly as `struct` values, not pointers. One such `struct point` is embedded in each `struct link`. So assuming (for illustration) that each `int` value and each pointer occupies 4 bytes, a `struct point` occupies 8 bytes and a `struct link` occupies 12 bytes.

"07/example.c"

```
35      for (int i = 0; i < a; i++) {
36        list = newLink((struct point) {.x=i,.y=i},list);
37      }
```

We use a compound literal `(struct point) {.x=i,.y=i}` to create and initialize a point in Line 36. We then pass it is the first argument to `newLink`; C handles assignment and argument passing of `struct`s by copying their contents (i.e., 8 bytes in this case).

The decision to handle these two types in two different ways in this program is somewhat arbitrary, but not entirely so. We are forced to represent links indirectly via pointers, because otherwise we cannot use them to build a list. (Consider trying to contain the body of a link within the body of a link within...) We could represent points indirectly too, but there is no particular necessity to do so, and holding the contents of the point directly within the link saves the space and time overheads of adding another layer of indirection.

Write a program `Example.java` that mimics this C program as closely as you can, being sure to continue defining two distinct classes `Point` and `Link`. Because Java uses references for all object values, you won't be able to replicate the memory layout of the C program exactly.

By the way, what is the behavior of your Java program when $b > a$? Does Java do a fundamentally better job of handling this situation than C did?

Exercise: Object Arrays (08)

In this directory, write a program `Example.java` that behaves as follows.

- The program takes two command line arguments, a filename f and an integer size n (you can assume $n > 0$).
- It creates an array of n points, where points are represented as objects of this class:

  ```
  static class P {
    double x;
    double y;
    P (double x, double y) {this.x = x; this.y = y;}
  }
  ```

- It initializes each element of the array to contain the point $(0.0, 0.0)$.
- It opens file f and reads and executes commands from it until the end of file is reached. Each command consists of an operation name followed by one or more space separated parameters whose meaning depends on the operation.
  - `PRINT` i prints a comma-separated pair of doubles representing the coordinates of the point in array location i, where i is an integer with $0 \leq i < n$.

- `SET` i x y updates array location i to contain the point (x, y), where x and y are doubles and i is an integer with $0 \leq i < n$.
- `COPY` i j updates array location j to contain the point (x, y) where array location i currently contains point (x, y) and i, j are integers with $0 \leq i, j < n$.

If you find this description confusing (perhaps that's "the point"?!) it should help to know that given any size $n \geq 3$, the expected output for the sample data file

"08/data"

```
PRINT 1
SET 1 1.0 1.0
PRINT 1
PRINT 2
SET 2 2.0 2.0
COPY 2 1
PRINT 1
PRINT 2
SET 2 2.5 2.5
PRINT 1
PRINT 2
```

is

```
0.0,0.0
1.0,1.0
0.0,0.0
2.0,2.0
2.0,2.0
2.0,2.0
2.5,2.5
```

Helpful implementation hints:

- The `Scanner` class has methods `String next()` to fetch the next white-space delimited string from the file, and `boolean hasNext()` to check whether there are any more items in the file.
- To test whether two `Strings` $s1$ and $s2$ contain the same sequence of characters, use a method call `s1.equals(s2)` rather than `s1 == s2`. (You should be able to make a good guess about why the latter does not consistently work.)

Declaration and Initialization (09)

This function tries to use the value of a variable that has been declared but not initialized.

"09/Example.java"

```
  static int f(int a) {
    int b;
    int c;
    if (a < 100) {
      c = 10;
    } else {
```

```
 9         c = 20;
10       }
11       return a + b + c;
12     }
```

What is the difference between the behavior of the C++ and Java versions of the example program? (To compile and run the C++ version, type `g++ -o example example.cpp` and then `./example` *n*.)

Now change the Java program so that `b` is omitted from the sum returned at Line 10. Notice that the program now compiles, even though `c` is not initialized where it was declared, but is only written to later. Java performs a *static analysis* called to check that every variable is written to before it is read on every possible execution path. This property is called *definite assignment*.

Like almost any static analysis, the definite assignment property is a conservative approximation of the runtime behavior of the program. Change the program again so that Line 8 reads `if (a >= 100)` rather than `else` and try compiling it. Even though this latest version is logically equivalent to the previous one, Java's analysis fails to realize that `c` is still definitely assigned on every path to its use at Line 10.

Towards the end of the course, we'll see why a static analysis of this kind can never be perfect; it will always need to reject some programs that would in fact behave OK. Could a smarter compiler detect that this *particular* program is OK and allow compilation to continue? Yes, but then it wouldn't be a valid Java compiler! The precise rules followed by the analysis are part of the Java language definition. This guarantees that porting code to a different Java compiler won't suddenly make it compile when it didn't before (or vice-versa).

Division by zero (10)

Here we have Java and C++ functions that are intended to have identical behavior.

"10/Example.java"

```
    static int f(int a,int b) {
      return a*b/b;
    }
```

To start, compile the C++ program using

```
g++ -O0 -o example example.cpp
```

What happens when the second command line argument, *b*, is 0? Both C++ and Java programs should raise exceptions (although this is a little misleading: the C++ exception is not so readily catchable as the Java one).

Now recompile the C++ program with a higher level of optimization:

```
g++ -O2 -o example example.cpp
```

and repeat the experiment where *b* is 0. There should be no exception this time; instead `f` returns a definite value; namely the value of *a*. What on earth is going on here?

The answer lies in the concept of "undefined behaviors" (UB) in C/C++. The specification of these languages say any program that attempts to divide by zero is erroneous. In other words, function `f` is valid code only as long as we pass in a non-zero value for *b*. Moreover, the C/C++ system is not required to check and report this condition at run time (although it might—and the `-O0` version effectively does). In fact, something even stronger is true about UB: if it occurs, the compiled code is allowed to do *anything at* *all*! So it is fine for it to return the value of a, or the value 42, or to seg fault, or (if the computer is suitably equipped) to launch the missiles that start WW3.

What is the point of this flexibility? Simple: It allows the compiler to perform optimizations that preserve the observable behavior of the code in the normal, non-UB case, but might change it in the UB case. In function `f`, the C++ compiler observes that multiplying by `b` and then dividing by `b` is mathematically equivalent to doing nothing at all—except if `b` is zero, when the result is mathematically undefined. Since it is permitted to ignore the UB possibility, the compiler goes ahead and removes both the multiply and the divide operation from the compiled code, which hence simply returns `a`!

A similar spirit is behind C/C++'s treatment of other UBs, such as referencing an array element out of bounds. The efficient thing to do is to simply compute the memory address where the element *would* live (if the array were big enough) and read or write from that address, no matter what actually lives there. The results are typically quite unpredictable, but that is OK from C/C++'s point of view, because the program has performed a UB, so all bets are off.

Note that no Java compiler will ever do anything like this: Java has no undefined behaviors—indeed, its definition tries to explicitly pin down every possible behavior for any (successfully compiled) program. In this case, the language clearly specifies that `f` should throw an `ArithmeticException` if `b` is 0. Programmers must be able to depend on this behavior because they may have included an exception handler that they expect to execute in this circumstance.

Exercise: Arithmetic Overflow (11)

Once again, we have Java and C++ versions of the same program. Observe what happens when you pass the command line argument a = 2000000000 to the Java program, the C++ program compiled with -O0, and the C++ program compiled with -O2.

Use the ideas of UB developed in the last example to explain the behavior you see here. Put your answer in a short file `answer.txt` in this directory.

Useful information:

- The signed `int` type (in Java and in our C++ installation) occupies 32 bits and has range

$$[-2147483648, 2147483647].$$

- Java semantics say that signed integer overflow "wraps around" to a negative number—i.e., the overflow is ignored.
- C++ semantics say that signed integer overflow is a UB. (Incidentally, *unsigned* integer overflow is defined to wrap around, but that is irrelevant here.)