Principles of Software Construction:
I/O and reflection

Josh Bloch Charlie Garrod

School of
Computer Science

Administrivia

- Homework 4c due **tonight**
- Homework 5 coming out tomorrow
- Midterm next Thursday in class
- Midterm review next Wednesday 7-9pm HH B103

Collections Puzzler: “Set List”

BEETHOVEN
PLEASE PLEASE ME
SAW HER STANDING
FROM ME TO YOU
TASTE OF HONEY
BOYS
SHE LOVES YOU
TWIST AND SHOUT
NEMS ENTERPRISES LTD.

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer>  set =  new LinkedHashSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }

        System.out.println(set + " " + list);
    }
}
```

What Does It Print?

(a) [-3, -2, -1] [-3, -2, -1]
(b) [-3, -2, -1] [-2, 0, -2]
(c) It varies
(d) None of the above

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer>  set =  new LinkedHashSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }

        System.out.println(set + " " + list);
    }
}
```

What Does It Print?

a. `[-3, -2, -1] [-3, -2, -1]`

b. `[-3, -2, -1] [-2, 0, 2]`

c. **Throws exception**

d. **None of the above**

Autoboxing + overloading = confusion

Another look

We're getting wrong overloading of `remove` *on the list*

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer>  set =  new LinkedHashSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);   // Invokes Set.remove(E)
            list.remove(i);  // Invokes List.remove(int)
        }

        System.out.println(set + " " + list);
    }
}
```

How do you fix it?

Force the desired overloading with a cast

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer>  set =  new LinkedHashSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove((Integer) i);
        }

        System.out.println(set + " " + list);
    }
}
```

The moral

- Avoid ambiguous overloadings
  - Harder to avoid after Java 5
  - Autoboxing, generics, varargs
- Design APIs with this in mind
  - Old rules no longer suffice
- Luckily, few existing APIs were compromised
  - Beware `List<Integer>`
- **`Overload with care!`**

Key concepts from Tuesday…

- Frameworks are like APIs but different
  - They generally *have* APIs
  - But they “drive,” not you
- Designing frameworks is tricky
  - All the challenges of API design and more
- Whitebox vs. blackbox frameworks

Outline

I. I/O – history, critique, and advice
II. A brief introduction to reflection

A brief, sad history of I/O in Java

Release, Year	Changes
JDK 1.0, 1996	`java.io.InputStream/OutputStream` – byte-based
JDK 1.1, 1997	`java.io.Reader/Writer` – char-based wrappers
J2SE 1.4, 2002	`java.nio.Channel/Buffer` – "Flexible" + select/poll, mmap
J2SE 5.0, 2004	`java.util.Scanner`, `String.printf/format` – Formatted
Java 7, 2011	`java.nio.file Path/Files` – file systems `java.nio.AsynchronousFileChannel` - *Real* async I/O
Java 8, 2014	`Files.lines` – lambda/stream integration
3d party, 2014	`com.squareup.okio.Buffer` – "Modern"

A Rogue's Gallery of cats

Thanks to Tim Bloch for cat-herding

cat 1: StreamCat

```
/**
 * Reads all lines from a text file and prints them.
 * Uses Java 1.0-era (circa 1996) Streams to read the file.
 */
public class StreamCat {
    public static void main(String[] args) throws IOException {
        DataInputStream dis = new DataInputStream(
                new FileInputStream(args[0]));

        // Don't do this! DataInputStream.readLine is DEPRECATED!
        String line;
        while ((line = dis.readLine()) != null)
            System.out.println(line);
    }
}
```

cat 2: ReaderCat

```
/**
 * Reads all lines from a text file and prints them.
 * Uses Java 1.1-era (circa 1997) Streams to read the file.
 */
public class ReaderCat {
    public static void main(String[] args) throws IOException {
        try (BufferedReader rd = new BufferedReader(
                new FileReader(args[0]))) {
            String line;
            while ((line = rd.readLine()) != null) {
                System.out.println(line);
                // you could also wrap System.out in a PrintWriter
            }
        }
    }
}
```

cat 3: NioCat

```
/**
 * Reads all lines from a text file and prints them.
 * Uses nio FileChannel and ByteBuffer.
 */
public class NioCat {
    public static void main(String[] args) throws IOException {
        ByteBuffer buf = ByteBuffer.allocate(512);
        try (FileChannel ch = FileChannel.open(Paths.get(args[0]),
                StandardOpenOption.READ)) {
            int n;
            while ((n = ch.read(buf)) > -1) {
                System.out.print(new String(buf.array(), 0, n));
                buf.clear();
            }
        }
    }
}
```

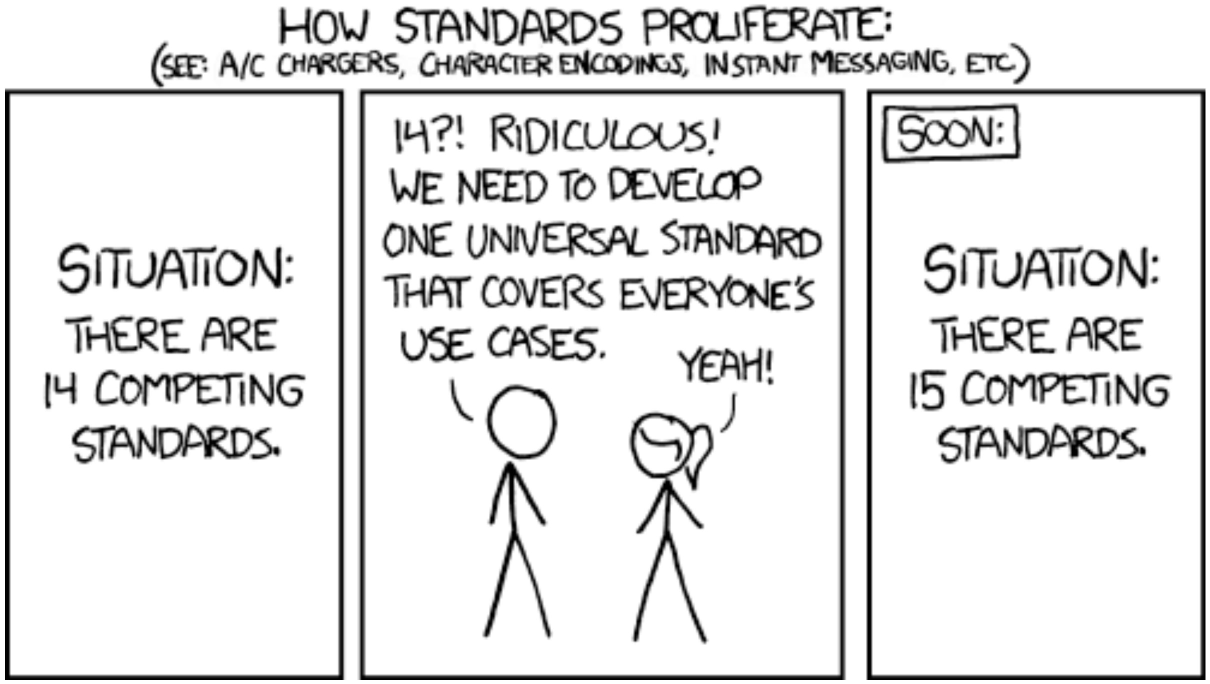
cat 4: ScannerCat

```
/**
 * Reads all lines from a text file and prints them
 * Uses Java 5 scanner.
 */
public class ScannerCat {
    public static void main(String[] args) throws IOException {
        try (Scanner s = new Scanner(new File(args[0]))) {
            while (s.hasNextLine())
                System.out.println(s.nextLine());
        }
    }
}
```

cat 5: LinesCat

```
/**
 * Reads all lines from a text file and prints them.  Uses Files,
 * Java 8-era Stream API (not IO Streams!) and method references.
 */
public class LinesCat {
  public static void main(String[] args) throws IOException {
    Files.lines(Paths.get(args[0])).forEach(System.out::println);
  }
}
```

Randall Munroe understands


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

A useful example – curl in Java

prints the contents of a URL

```
public class Curl {
    public static void main(String[] args) throws IOException {
        URL url = new URL(args[0]);
        try (BufferedReader r = new BufferedReader(
                new InputStreamReader(url.openStream(),
                StandardCharsets.UTF_8))) {
            String line;
            while ((line = r.readLine()) != null)
                System.out.println(line);
        }
    }
}
```

Java I/O Recommendations

- Everyday use – `Buffered{Reader, Writer}`
- Casual use - `Scanner`
  - Easy but not general and swallows exceptions
- Stream integration – `Files.lines`
  - No parallelism support *yet*
- Async – `java.nio.AsynchronousFileChannel`
- Many niche APIs, e.g. mem mapping, line numbering
  - Search them out as needed
- Consider Okio if third party API allowed

Outline

I. I/O – history, critique, and advice

II. A brief introduction to reflection

What is reflection?

- Operating programmatically on objects that represent linguistic entities (e.g., classes, methods)
- Allows program to work with classes that were not know (or didn't exist!) at compile time
- Quite complex – involves many APIs
- But there's a simple form
  - Involves `Class.forName` and `newInstance`

Benchmark interface

```
/** Implementations can be timed by RunBenchmark. */
public interface Benchmark {
    /**
     * Initialize the benchmark.  Passed all command line
     * arguments beyond first three.  Used to parameterize a
     * a benchmark This method will be invoked once by
     * RunBenchmark prior to timings.
     */
    void init(String[] args);

    /**
     * Performs the test being timed.
     * @param numReps the number of repetitions comprising test
     */
    void run(int numReps);
}
```

RunBenchmark program (1)

```
public class RunBenchmark {
    public static void main(String[] args) throws Exception {
        if (args.length < 3) {
            System.out.println(
"Usage: java RunBenchmark <# tests> <# reps/test> <class name> [<arg>...]");
            System.exit(1);
        }

        int numTests = Integer.parseInt(args[0]);
        int numReps = Integer.parseInt(args[1]);
        Benchmark b =
               (Benchmark) Class.forName(args[2]).newInstance();
        String[] initArgs = new String[args.length - 3];
        System.arraycopy(args, 3, initArgs, 0, initArgs.length);
```

RunBenchmark program (2)

```
        if (initArgs.length != 0)
          System.out.println("Args: " + Arrays.toString(initArgs));
        b.init(initArgs);

        for (int i = 0; i < numTests; i++) {
            long startTime = System.nanoTime();
            b.run(numReps);
            long endTime = System.nanoTime();
            System.out.printf("Run %d: %d ms.%n", i,
                Math.round((endTime - startTime)/1_000_000.));
        }
    }
}
```

Sample Benchmark

```
public class SortBench implements Benchmark {
    private int[] a;

    public void init(String[] args) {
        int arrayLen = Integer.parseInt(args[0]);
        a = new int[arrayLen];
        Random rnd = new Random(666);
        for (int i = 0; i < arrayLen; i++)
            a[i] = rnd.nextInt(arrayLen);
    }
    public void run(int numReps) {
        for (int i = 0; i < numReps; i++) {
            int[] tmp = a.clone();
            Arrays.sort(tmp);
        }
    }
}
```

Demo – RunBenchmark

Conclusion

- Java I/O is a bit of a mess
  - There are many ways to do things
  - Use readers most of the time
- Reflection is tricky, but `Class.forName` and `newInstance` go a long way