Principles of Software Construction: Objects, Design, and Concurrency

Generics, I/O, and reflection

Josh Bloch  Charlie Garrod  Darya Melicher
Administrivia

- Homework 4b due this Thursday, October 18th
```java
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "+ pig == dog);
    }
}
```

Java puzzlers: “Animal Farm” (2005)
What does it print?

```java
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: " + (pig == dog));
    }
}
```

(a) Animals are equal: true
(b) Animals are equal: false
(c) It varies
(d) None of the above
What does it print?

(a) Animals are equal: true
(b) Animals are equal: false
(c) It varies
(d) None of the above: false

The + operator binds tighter than ==
Another look

```java
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "
                              + pig == dog);
    }
}
```
You could try to fix it like this...

```java
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: " + (pig == dog));
    }
}
```

**Prints** Animals are equal: false
But this is much better

```java
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: " + pig.equals(dog));
    }
}
```

*Prints Animals are equal: true*
The moral

• Use parens, not spacing, to express intent
• Use parens whenever there is any doubt
• Don’t depend on interning of string constants
• Use `.equals`, not `==` for object references
Key concepts from Tuesday...

This is actually the conclusion from last lecture, which I forgot to go over

• It takes a lot of work to make something that appears obvious
  – Coherent, unified vision
  – Willingness to listen to others
  – Flexibility to accept change
  – Tenacity to resist change
  – Good documentation!

• It’s worth the effort!
  – A solid foundation can last two+ decades
Outline

I. Generics – better late than never
II. I/O – history, critique, and advice
III. A brief introduction to reflection
Parametric polymorphism (a.k.a. generics)

- *Parametric polymorphism* is the ability to define a type generically, allowing static type-checking without fully specifying the type
  - e.g.:
    
    ```java
    public class Frequency {
        public static void main(String[] args) {
            Map<String, Integer> m = new TreeMap<>();
            for (String word : args) {
                Integer freq = m.get(word);
                m.put(word, (freq == null ? 1 : freq + 1));
            }
            System.out.println(m);
        }
    }
    ```
A generic implementation of pairs

```java
public class Pair<E> {
    private final E first, second;
    public Pair(E first, E second) {
        this.first = first;
        this.second = second;
    }
    public E first() { return first; }
    public E second() { return second; }
}
```

• Better client code:
  ```java
  Pair<String> p = new Pair<>("Hello", "world");
  String result = p.first();
  ```
Some Java Generics details

• Can have multiple type parameters
  – e.g., Map<String, Integer>

• Generics are type invariant
  – ArrayList<String> is a subtype of List<String>
  – List<String> is not a subtype of List<Object>

• Generic type info is erased (i.e. compile-time only)
  – Cannot use instanceof to check generic type

• Cannot create Generic arrays
  Pair<String>[] foo = new Pair<String>[42]; // won't compile
Generic array creation is illegal

// won't compile
List<String>[] stringLists = new List<String>[1];
List<Integer> intList = Arrays.asList(42);
Object[] objects = stringLists;
objects[0] = intList;
String s = stringLists[0].get(0); // Would be type-safe
Generic design advice: Prefer lists to arrays

// Fails at runtime
Object[] oArray = new Long[42];
oArray[0] = "I don't fit in"; // Throws ArrayStoreException

// Won't compile
List<Object> ol = new ArrayList<Long>(); // Incompatible type
ol.add("I don't fit in");
Wildcard types provide API flexibility

- List<String> is not a subtype of List<Object>
  - i.e., generic types are invariant
  - But **wildcard types provide inheritance on generics**

- How wildcard types are read
  - List<?> is a “list of some type”
  - List<? extends Animal> is “list of some subtype of animal”
  - List<? Super Animal> is “list of some supertype of animal”

- Subtyping relations
  - List<String> is a subtype of List<? extends Object>
  - List<Object> is a subtype of List<? super String>
  - List<Anything> is a subtype of List<?>

- Wildcards are technically known as **variance annotations**
Wildcards in the java.util.Collection API

```java
public interface Collection<E> ... {
    boolean    add(E e);
    boolean    addAll(Collection<? extends E> c);
    boolean    remove(Object e);
    boolean    removeAll(Collection<?> c);
    boolean    contains(Object e);
    boolean    containsAll(Collection<?> c);
    void         clear();
    int         size();
    boolean    isEmpty();
    Iterator<E> iterator();
    Object[]    toArray()
    <T> T[]      toArray(T[] a);
    ...
}
```
An inflexible API without wildcards

• Suppose you want to add bulk methods to Stack<E>:
  void pushAll(Collection<E> src);
  void popAllInto(Collection<E> dst);

• Problem:
  – It should be fine to push a Long onto a Stack<Number>:
    Collection<Long> numbers = …;
    Stack<Number> numberStack = …;
    for (Long n : numbers) {
      numberStack.push(n);
    }
  – This API prevents pushAll(Collection<Long>) onto a Stack<Number>
Generic design advice: Use your PECS

• PECS: Producer extends, Consumer super
  – For a T producer, use Foo<? extends T>
  – For a T consumer, use Foo<? super T>
  – Mnemonic only works for input parameters
Use your PECS

- Suppose you want to add bulk methods to Stack\(<E>\):
  
  void pushAll(Collection\(<E>\) src);

  void popAllInto(Collection\(<E>\) dst);
Use your PECS

• Suppose you want to add bulk methods to Stack\<E\>:

  void pushAll(Collection\<? extends E\> src);
  – src is an E producer

  void popAllInto(Collection\<? super E\> dst);
  – dst is an E consumer
Outline

I. Generics – better late than never
II. I/O – history, critique, and advice
III. A brief introduction to reflection
A brief, sad history of I/O in Java

<table>
<thead>
<tr>
<th>Release, Year</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 1.0, 1996</td>
<td>java.io.InputStream/OutputStream – byte-based</td>
</tr>
<tr>
<td>JDK 1.1, 1997</td>
<td>java.io.Reader/Writer – char-based wrappers</td>
</tr>
<tr>
<td>J2SE 5.0, 2004</td>
<td>java.util.Scanner, String_printf/format – Formatted</td>
</tr>
<tr>
<td>Java 7, 2011</td>
<td>java.nio.file Path/Files – file systems</td>
</tr>
<tr>
<td></td>
<td>java.nio.AsynchronousFileChannel - Real async I/O</td>
</tr>
<tr>
<td>Java 8, 2014</td>
<td>Files.lines – lambda/stream integration</td>
</tr>
</tbody>
</table>
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
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
  - Support for parallelism in Java 9
- Async – java.nio.AsynchronousFileChannel
- Many niche APIs, e.g. memory mapped files, line numbering
  - Search them out as needed
- Consider Okio if third party API allowed
  - Very powerful, very fast, high-quality API
Outline

I. Generics – better late than never
II. I/O – history, critique, and advice
III. 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);
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.));
}
public class SortBenchmark implements Benchmark {
    private int[] a;

    @Override 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);
    }

    @Override public void run(int numReps) {
        for (int i = 0; i < numReps; i++) {
            int[] tmp = a.clone();
            Arrays.sort(tmp);
        }
    }
}
Demo – RunBenchmark
Conclusion

• Generics provide API flexibility with type safety
• 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
  – A more powerful option that hides the reflection: ServiceLoader