The Continuing Adventures of *Java™ Puzzlers*: Tiger Traps

Joshua Bloch and Neal Gafter
Google Inc.

TS-1188

Introduction

- Eight more Java™ platform puzzles
  - Short program with curious behavior
  - What does it print? (multiple choice)
  - The mystery revealed
  - How to fix the problem
  - The moral
- Covers language and core libraries
- Watch out for Tiger traps!

1. “Odd Behavior”

```
public class OddBehavior {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(-2, -1, 0, 1, 2);

        boolean foundOdd = false;
        for (Iterator<Integer> it = list.iterator(); it.hasNext(); )
            foundOdd = foundOdd || isOdd(it.next());

        System.out.println(foundOdd);
    }

    private static boolean isOdd(int i) {
        return (i & 1) != 0;
    }
}
```

What Does It Print?

```
public class OddBehavior {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(-2, -1, 0, 1, 2);

        boolean foundOdd = false;
        for (Iterator<Integer> it = list.iterator(); it.hasNext(); )
            foundOdd = foundOdd || isOdd(it.next());

        System.out.println(foundOdd);
    }

    private static boolean isOdd(int i) {
        return (i & 1) != 0;
    }
}
```

(a) **`true`**

(b) **`false`**

(c) Throws exception

(d) None of the above

What Does It Print?

(a) `true`

(b) `false`

(c) Throws exception

(d) None of the above: Nothing—Infinite loop

Conditional OR operator (`||`) short-circuits iterator

Another Look

```
public class OddBehavior {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(-2, -1, 0, 1, 2);

        boolean foundOdd = false;
        for (Iterator<Integer> it = list.iterator(); it.hasNext(); )
            foundOdd = foundOdd || isOdd(it.next());

        System.out.println(foundOdd);
    }

    private static boolean isOdd(int i) {
        return (i & 1) != 0;
    }
}
```

You Could Fix It Like This…

```
public class OddBehavior {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(-2, -1, 0, 1, 2);

        boolean foundOdd = false;
        for (int i : list)
            foundOdd = foundOdd || isOdd(i);

        System.out.println(foundOdd);
    }

    private static boolean isOdd(int i) {
        return (i & 1) != 0;
    }
}
```

…But This Is Even Better

```
public class OddBehavior {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(-2, -1, 0, 1, 2);
        System.out.println(containsOdd(list));
    }

    private static boolean containsOdd(List<Integer> list) {
         for (int i : list)
            if (isOdd(i))
                return true;
         return false;
    }

    private static boolean isOdd(int i) {
        return (i & 1) != 0;
    }
}
```

The Moral

- Use for-each wherever possible
  - Nicer and safer than explicit iterator or index usage
- If you must use an iterator, make sure you call **`next()`** exactly once
- *Conditional* operators evaluate their right operand only if necessary to determine result
  - This is almost always what you want
  - If not, you can use the *logical* operators (**`&`** and **`|`**)

2. “Set List”

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

        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

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

        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

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

        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);  // List.remove(int)
        }
        System.out.println(set + " " + list);
    }
}
```

How Do You Fix It?

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

        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 in release 5.0
  - Autoboxing, varargs, generics
- Design new APIs with this in mind
  - Old rules no longer suffice
- Luckily, few existing APIs were compromised
  - Beware **`List<Integer>`**

3. “Powers of Ten”

```
public enum PowerOfTen {
    ONE(1), TEN(10),
    HUNDRED(100) {
        @Override public String toString() {
            return Integer.toString(val);
        }
    };
    private final int val;
    PowerOfTen(int val) { this.val = val;  }

    @Override public String toString() {
        return name().toLowerCase();
    }
    public static void main(String[] args) {
        System.out.println(ONE + " " + TEN + " " + HUNDRED);
    }
}
```

What Does It Print?

(a) **`ONE TEN HUNDRED`**

(b) **`one ten hundred`**

(c) **`one ten 100`**

(d) None of the above

```
public enum PowerOfTen {
    ONE(1), TEN(10),
    HUNDRED(100) {
        @Override public String toString() {
            return Integer.toString(val);
        }
    };
    private final int val;
    PowerOfTen(int val) { this.val = val;  }

    @Override public String toString() {
        return name().toLowerCase();
    }
    public static void main(String[] args) {
        System.out.println(ONE + " " + TEN + " " + HUNDRED);
    }
}
```

What Does It Print?

(a) **`ONE TEN HUNDRED`**

(b) **`one ten hundred`**

(c) **`one ten 100`**

(d) None of the above: Won't compile

```
Non-static variable val can't be referenced from static context
    return Integer.toString(val);
                            ^
```

Private members are never inherited

Another Look

```
public enum PowerOfTen {
    ONE(1), TEN(10),
    HUNDRED(100) {  // Creates static anonymous class
        @Override public String toString() {
            return Integer.toString(val);
        }
    };
    private final int val;
    PowerOfTen(int val) { this.val = val;  }

    @Override public String toString() {
        return name().toLowerCase();
    }
    public static void main(String[] args) {
        System.out.println(ONE + " " + TEN + " " + HUNDRED);
    }
}
```

How Do You Fix It?

```
public enum PowerOfTen {
    ONE(1), TEN(10),
    HUNDRED(100) {
        @Override public String toString() {
            return Integer.toString(super.val);
        }
    };
    private final int val;
    PowerOfTen(int val) { this.val = val;  }

    @Override public String toString() {
        return name().toLowerCase();
    }
    public static void main(String[] args) {
        System.out.println(ONE + " " + TEN + " " + HUNDRED);
    }
}
```

The Moral

- Nest-mates can use each others' private members
- But private members are never inherited
- Constant-specific enum bodies define static anonymous classes
- Compiler diagnostics can be confusing

4. “Testy Behavior”

```
import java.lang.reflect.*;

@interface Test { }
public class Testy {
   @Test public static void test()  { return;                  }
   @Test public static void test2() { new RuntimeException(); }
   public static void main(String[] args) throws Exception {
       for (Method m : Testy.class.getDeclaredMethods()) {
           if (m.isAnnotationPresent(Test.class)) {
               try {
                   m.invoke(null);
                   System.out.print("Pass ");
               } catch (Throwable ex) {
                   System.out.print("Fail ");
               }
           }
       }
   }
}
```

What Does It Print?

(a) **`Pass Fail`**

(b) **`Pass Pass`**

(c) It varies

(d) None of the above

```
import java.lang.reflect.*;

@interface Test { }
public class Testy {
   @Test public static void test()  { return;                  }
   @Test public static void test2() { new RuntimeException(); }
   public static void main(String[] args) throws Exception {
       for (Method m : Testy.class.getDeclaredMethods()) {
           if (m.isAnnotationPresent(Test.class)) {
               try {
                   m.invoke(null);
                   System.out.print("Pass ");
               } catch (Throwable ex) {
                   System.out.print("Fail ");
               }
           }
       }
   }
}
```

What Does It Print?

(a) **`Pass Fail`**

(b) **`Pass Pass`**

(c) It varies

(d) None of the above: In fact, nothing!

The program contains two bugs, both subtle

Another Look

```
import java.lang.reflect.*;

@interface Test { }  // By default, annotations are discarded at runtime
public class Testy {
   @Test public static void test()  { return;                }
   @Test public static void test2() { new RuntimeException(); }  // Oops !
   public static void main(String[] args) throws Exception {
       for (Method m : Testy.class.getDeclaredMethods()) {
           if (m.isAnnotationPresent(Test.class)) {
               try {
                   m.invoke(null);
                   System.out.print("Pass");
               } catch (Throwable ex) {
                   System.out.print("Fail ");
               }
           }
       }
   }
}
```

How Do You Fix It?

```
import java.lang.reflect.*;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)@interface Test { }
public class Testy {
   @Test public static void test()  { return;                          }
   @Test public static void test2() { throw new RuntimeException(); }
   public static void main(String[] args) throws Exception {
       for (Method m : Testy.class.getDeclaredMethods()) {
           if (m.isAnnotationPresent(Test.class)) {
               try {
                   m.invoke(null);
                   System.out.print("Pass ");
               } catch (Throwable ex) {
                   System.out.print("Fail ");
               }
           }
       }
   }
}
```

The Moral

- By default, annotations are discarded at runtime
  - If you need annotations at runtime, use **`@Retention(RetentionPolicy.RUNTIME)`**
  - If you want them omitted from class file, use **`@Retention(RetentionPolicy.SOURCE)`**
- No guarantee on order of reflected entities
- Don’t forget to throw your exceptions

5. “What the Bleep?”

```
public class Bleep {
    String name = "Bleep";
    void setName(String name) {
        this.name = name;
    }
    void backgroundSetName() throws InterruptedException {
        Thread t = new Thread() {
            @Override public void run() { setName("Blat"); }
        };
        t.start();
        t.join();
        System.out.println(name);
    }
    public static void main(String[] args) throws InterruptedException {
        new Bleep().backgroundSetName();
    }
}
```

What Does It Print?

(a) **`Bleep`**

(b) **`Blat`**

(c) It varies

(d) None of the above

```
public class Bleep {
    String name = "Bleep";
    void setName(String name) {
        this.name = name;
    }
    void backgroundSetName() throws InterruptedException {
        Thread t = new Thread() {
            @Override public void run() { setName("Blat"); }
        };
        t.start();
        t.join();
        System.out.println(name);
    }
    public static void main(String[] args) throws InterruptedException {
        new Bleep().backgroundSetName();
    }
}
```

What Does It Print?

(a) **`Bleep`**

(b) **`Blat`**

(c) It varies

(d) None of the above

`Bleep.setName` isn't getting called

Another Look

```
public class Bleep {
    String name = "Bleep";
    void setName(String name) { // Does this look familiar?
        this.name = name;
    }
    void backgroundSetName() throws InterruptedException {
        Thread t = new Thread() {
            // Invokes Thread.setName (shadowing)
            @Override public void run() { setName("Blat"); }
        };
        t.start();
        t.join();
        System.out.println(name);
    }
    public static void main(String[] args) throws InterruptedException {
        new Bleep().backgroundSetName();
    }
}
```

How Do You Fix It?

```
public class Bleep {
    String name = "Bleep";
    void setName(String name) {
        this.name = name;
    }
    void backgroundSetName() throws InterruptedException {
        Thread t = new Thread(new Runnable() {
            public void run() { setName("Blat"); }
        });
        t.start();
        t.join();
        System.out.println(name);
    }
    public static void main(String[] args) throws InterruptedException {
        new Bleep().backgroundSetName();
    }
}
```

The Moral

- Don’t extend **`Thread`**
  - Use new **`Thread(Runnable)`** instead
- Often the Executor Framework is better still
  - Much more flexible
  - See **`java.util.concurrent`** for more information
- Beware of shadowing

6. “Beyond Compare”

```
public class BeyondCompare {
   public static void main(String[] args) {
       Object o = new Integer(3);
       System.out.println(new Double(3).compareTo(o) == 0);
   }
}
```

What Does It Print?

```
public class BeyondCompare {
    public static void main(String[] args) {
        Object o = new Integer(3);
        System.out.println(new Double(3).compareTo(o) == 0);
    }
}
```

(a) **`true`**

(b) **`false`**

(c) Throws exception

(d) None of the above

What Does It Print?

(a) **`true`**

(b) **`false`**

(c) Throws exception

(d) None of the above: Won't compile (it did in 1.4)

```
compareTo(Double) in Double cannot be applied to (Object)
        System.out.println(new Double(3).compareTo(o) == 0);
                           ^
```

The **`Comparable`** interface was generified in 5.0

Another Look

```
public class BeyondCompare {
   public static void main(String[] args) {
       Object o = new Integer(3);
       System.out.println(new Double(3).compareTo(o) == 0);
   }
}

// Interface Comparable was generified in release 5.0
public interface Comparable<T> {
    int compareTo(T t);  // Was Object
}

public class Double extends Number
                    implements Comparable<Double>
```

How Do You Fix It?

```
// Preserves 1.4 semantics
public class BeyondCompare {
   public static void main(String[] args) {
       Object o = new Integer(3);
       System.out.println(
           new Double(3).compareTo((Double) o) == 0);
   }
}
```

```
// Fixes the underlying problem
public class BeyondCompare {
   public static void main(String[] args) {
       Double d = 3.0;
       System.out.println(Double.valueOf(3).compareTo(d) == 0);
   }
}
```

The Moral

- Binary compatibility is preserved at all costs
- Source compatibility broken for good cause (rare)
  - **`Comparable<T>`** alerts you to errors at compile time
- Take compiler diagnostics seriously
  - Often there is an underlying problem

7. “Fib O’Nacci”

```
public class Fibonacci {
    private static final int LENGTH = 7;
    public static void main(String[] args) {
        int[] fib = new int[LENGTH];
        fib[0] = fib[1] = 1;  // First 2 Fibonacci numbers
        for (int i = 2; i < LENGTH; i++)
            fib[i] = fib[i - 2] + fib[i - 1];

        System.out.println(Arrays.asList(fib));
    }
}
```

What Does It Print?

```
public class Fibonacci {
    private static final int LENGTH = 7;
    public static void main(String[] args) {
        int[] fib = new int[LENGTH];
        fib[0] = fib[1] = 1;  // First 2 Fibonacci numbers
        for (int i = 2; i < LENGTH; i++)
            fib[i] = fib[i - 2] + fib[i - 1];

        System.out.println(Arrays.asList(fib));
    }
}
```

(a) `[1, 1, 2, 3, 5, 8, 13]`

(b) Throws exception

(c) It varies

(d) None of the above

What Does It Print?

(a) `[1, 1, 2, 3, 5, 8, 13]`

(b) Throws exception

(c) It varies: Depends on hashcode `[[I@ad3ba4]`

(d) None of the above

`Arrays.asList` only works on arrays of object refs

Another Look

```
public class Fibonacci {
    private static final int LENGTH = 7;
    public static void main(String[] args) {
        int[] fib = new int[LENGTH];
        fib[0] = fib[1] = 1;  // First 2 Fibonacci numbers
        for (int i = 2; i < LENGTH; i++)
            fib[i] = fib[i - 2] + fib[i - 1];

        // Idiom only works for arrays of object references
        System.out.println(Arrays.asList(fib));
    }
}
```

How Do You Fix It?

```
public class Fibonacci {
    private static final int LENGTH = 7;
    public static void main(String[] args) {
        int[] fib = new int[LENGTH];
        fib[0] = fib[1] = 1;  // First 2 Fibonacci numbers
        for (int i = 2; i < LENGTH; i++)
            fib[i] = fib[i - 2] + fib[i - 1];

        System.out.println(Arrays.toString(fib));
    }
}
```

The Moral

- Use varargs sparingly in your APIs
  - It can hide errors and cause confusion
  - This program wouldn't compile under 1.4
- **`Arrays.asList`** printing idiom is obsolete
  - use **`Arrays.toString`** instead
  - Prettier, safer, and more powerful
- A full complement of array utilities added in 5.0
  - **`equals`**, **`hashCode`**, **`toString`** for all array types
- **`Integer`** is not the same as **`int`**

8. “Parsing Is Such Sweet Sorrow”

```
public class Parsing {
   /**
    * Returns Integer corresponding to s, or null if s is null.
    * @throws NumberFormatException if s is nonnull and
    *         doesn't represent a valid integer
    */
   public static Integer parseInt(String s) {
       return (s == null) ?
                (Integer) null : Integer.parseInt(s);
   }

   public static void main(String[] args) {
       System.out.println(parseInt("-1") + " " +
                          parseInt(null) + " " +
                          parseInt("1"));
   }
}
```

What Does It Print?

(a) `-1 null 1`
(b) `-1 0 1`
(c) Throws exception
(d) None of the above

```
public class Parsing {
   /**
    * Returns Integer corresponding to s, or null if s is null.
    * @throws NumberFormatException if s is nonnull and
    *         doesn't represent a valid integer
    */
   public static Integer parseInt(String s) {
       return (s == null) ?
                (Integer) null : Integer.parseInt(s);
   }

   public static void main(String[] args) {
       System.out.println(parseInt("-1") + " " +
                          parseInt(null) + " " +
                          parseInt("1"));
   }
}
```

What Does It Print?

(a) `-1 null 1`

(b) `-1 0 1`

(c) Throws exception: `NullPointerException`

(d) None of the above

Program attempts to auto-unbox `null`

Another Look

```
public class Parsing {
   /**
    * Returns Integer corresponding to s, or null if s is null.
    * @throws NumberFormatException if s is nonnull and
    *         doesn't represent a valid integer.
    */
   public static Integer parseInt(String s) {
       return (s == null) ?  // Mixed-type computation: Integer and int
               (Integer) null : Integer.parseInt(s);
   }

   public static void main(String[] args) {
       System.out.println(parseInt("-1") + " " +
                          parseInt(null) + " " +
                          parseInt("1"));
   }
}
```

How Do You Fix It?

```
public class Parsing {
   /**
    * Returns Integer corresponding to s, or null if s is null.
    * @throws NumberFormatException if s is nonnull and
    *         doesn't represent a valid integer.
    */
   public static Integer parseInt(String s) {
       return (s == null) ? null : Integer.valueOf(s);
   }

   public static void main(String[] args) {
       System.out.println(parseInt("-1") + " " +
                          parseInt(null) + " " +
                          parseInt("1"));
   }
}
```

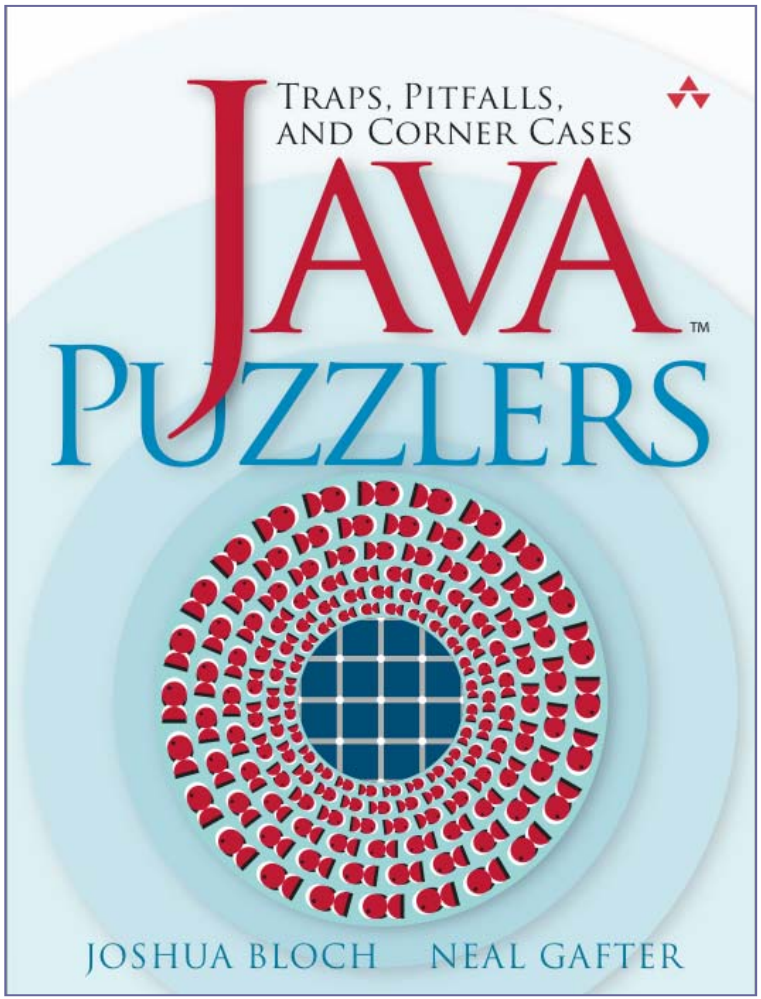
The Moral

- Mixed-type computations are confusing
- Especially true for **`?:`** expressions
- Avoid **`null`** where possible
- Auto-unboxing and **`null`** are a dangerous mix

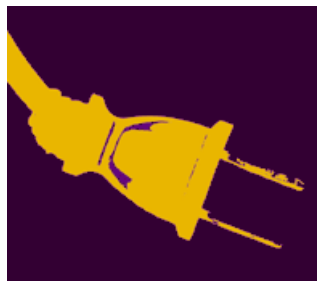
Conclusion

- Tiger is all about you, the programmer
  - Better programs with less effort
- But it adds a few sharp corners—Avoid them!
- Keep programs clear and simple
  - If you aren't sure what a program does, it probably doesn't do what you want
- Don't code like my brother

Shameless Commerce Division



- 95 Puzzles
- 52 Illusions
- Tons of fun



Send Us Your Puzzlers!

If you have a puzzler for us, send it to puzzlers@javapuzzlers.com

The Continuing Adventures of *Java™ Puzzlers*: Tiger Traps

Joshua Bloch and Neal Gafter
Google Inc.

TS-1188