
# Java Puzzlers

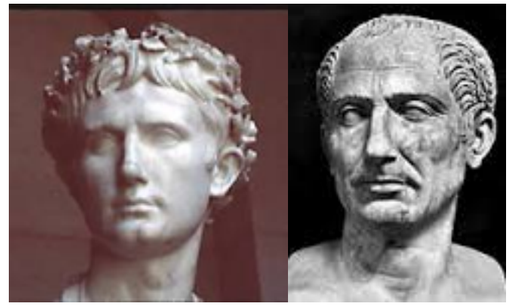
## *Something Old, Something Gnu, Something Bogus, Something Blew*

Joshua Bloch and Bob Lee
September 20, 2013

# What's All This, Then?

- **Five NEW Java programming language puzzles**
  - Short program with curious behavior
  - What does it print? (multiple choice)
  - The mystery revealed
  - How to fix the problem
  - The moral

# 1. “First Among Equals”



```
class First {
    public static void main(String[] args) {
        String[] declaration = { "I", "Came", "I", "Saw", "I", "Left" };

        Hashtable<String, Integer> firstIndex = new Hashtable<>();
        for (int i = declaration.length - 1; i >= 0; i--)
            firstIndex.put(declaration[i], i);

        int inMap = 0;
        for (String word : declaration)
            if (firstIndex.contains(word))
                inMap++;
        System.out.println(inMap);
    }
}
```

# What Does It Print?

```
class First {
    public static void main(String[] args) {
        String[] declaration = { "I", "Came", "I", "Saw", "I", "Left" };

        Hashtable<String, Integer> firstIndex = new Hashtable<>();
        for (int i = declaration.length - 1; i >= 0; i--)
            firstIndex.put(declaration[i], i);

        int inMap = 0;
        for (String word : declaration)
            if (firstIndex.contains(word))
                inMap++;
        System.out.println(inMap);
    }
}
```

(a) 0
(b) 4
(c) 6
(d) None of the above

# What Does It Print?

(a) `0`

(b) `4`

(c) `6`

(d) None of the above

The **`Hashtable.contains`** method doesn’t do what you think it does

# Another Look, Along With the Spec for `Hashtable.contains`

```
class First {
    public static void main(String[] args) {
        String[] declaration = { "I", "Came", "I", "Saw", "I", "Left" };

        Hashtable<String, Integer> firstIndex = new Hashtable<>();
        for (int i = declaration.length - 1; i >= 0; i--)
            firstIndex.put(declaration[i], i);

        int inMap = 0;
        for (String word : declaration)
            if (firstIndex.contains(word))
                inMap++;
        System.out.println(inMap);
    }
}
```

**public boolean contains(Object value)**

**Tests if some key maps into the specified value in this hashtable.**

This operation is more expensive than the containsKey method.
Note that this method is identical in functionality to containsValue,
(which is part of the Map interface in the collections framework).

# How Do You Fix It? Step 1: Declare Your Collection Properly

```
class First {
    public static void main(String[] args) {
        String[] declaration = { "I", "Came", "I", "Saw", "I", "Left" };

        Map<String, Integer> firstIndex = new Hashtable<>();
        for (int i = declaration.length - 1; i >= 0; i--)
            firstIndex.put(declaration[i], i);

        int inMap = 0;
        for (String word : declaration)
            if (firstIndex.contains(word)) // Your IDE won't let you screw up
                inMap++;
        System.out.println(inMap);
    }
}
```

**Or, if you rock it old school with emacs, javac will tell you that you screwed up**

```
First.java:13: error: cannot find symbol
            if (firstIndex.contains(word))
                           ^
  symbol:   method contains(String)
  location: variable firstIndex of type Map<String,Integer>
```

# How Do You Fix It? Step 2: Use The Method You Actually Meant To

```
class First {
    public static void main(String[] args) {
        String[] declaration = { "I", "Came", "I", "Saw", "I", "Left" };

        Map<String, Integer> firstIndex = new Hashtable<>();
        for (int i = declaration.length - 1; i >= 0; i--)
            firstIndex.put(declaration[i], i);

        int inMap = 0;
        for (String word : declaration)
            if (firstIndex.containsKey(word))
                inMap++;
        System.out.println(inMap);
    }
}
```

Prints 6

# The Moral

- Always declare collections using their interface types
  - e.g., `Map<String, Integer> = new HashMap<>();`
- For API Designers
  - Don’t violate *the principle of least astonishment*
- And by the way, **`Hashtable`** is almost never the right implementation to use
  - For serial use, prefer **`HashMap`**; for concurrent use, prefer **`ConcurrentHashMap`**

# 2. “Proxy Fight”

```
import java.io.*;

public class Cat implements Serializable {
     Object writeReplace() { return new Proxy(); }

     class Proxy implements Serializable {
         Object readResolve() { return new Cat(); }
     }

     public static void main(String[] args) throws Exception {
         ByteArrayOutputStream b = new ByteArrayOutputStream();
         Cat original = new Cat();
         new ObjectOutputStream(b).writeObject(original);
         Cat copyCat = (Cat) new ObjectInputStream(
                 new ByteArrayInputStream(b.toByteArray())).readObject();
         System.out.println(copyCat.equals(original));
     }
}
```

# What Does It Print?

```
import java.io.*;

public class Cat implements Serializable {
     Object writeReplace() { return new Proxy(); }

     class Proxy implements Serializable {
         Object readResolve() { return new Cat(); }
     }

     public static void main(String[] args) throws Exception {
         ByteArrayOutputStream b = new ByteArrayOutputStream();
         Cat original = new Cat();
         new ObjectOutputStream(b).writeObject(original);
         Cat copyCat = (Cat) new ObjectInputStream(
                 new ByteArrayInputStream(b.toByteArray())).readObject();
         System.out.println(copyCat.equals(original));
     }
}
```

(a) **`true`**
(b) **`false`**
(c) Throws an exception
(d) None of the above

# What Does It Print?

(a) **`true`**

(b) **`false`**

(c) Throws exception: **`ClassCastException`**, deep in the bowels of deserialization

(d) None of the above

```
Exception in thread "main" ClassCastException:
     cannot assign instance of Cat$Proxy to field Cat$Proxy.this$0 of type Cat in instance of Cat$Proxy
            at java.io.ObjectStreamClass$FieldReflector.setObjFieldValues(ObjectStreamClass.java:2089)
            at java.io.ObjectStreamClass.setObjFieldValues(ObjectStreamClass.java:1261)
            at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:1997)
            at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1915)
            at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1798)
            at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1348)
            at java.io.ObjectInputStream.readObject(ObjectInputStream.java:370)
            at Cat.main(Cat.java:15)
```

**`readResolve`** / **`writeReplace`** fails for circular references

# Another Look

*Nonstatic nested classes contain an implicit reference to their enclosing instance*

```
import java.io.*;

public class Cat implements Serializable {
     Object writeReplace() { return new Proxy(); }

     class Proxy implements Serializable { // Nonstatic nested class
         Object readResolve() { return new Cat(); }
     }

     public static void main(String[] args) throws Exception {
         ByteArrayOutputStream b = new ByteArrayOutputStream();
         Cat original = new Cat();
         new ObjectOutputStream(b).writeObject(original);
         Cat copyCat = (Cat) new ObjectInputStream(
                 new ByteArrayInputStream(b.toByteArray())).readObject(); // Exc!
         System.out.println(copyCat.equals(original));
     }
}
```

# What Does the Serialization Spec Have to Say About All This?

- “Note - The readResolve method is not invoked on the object until the object is fully constructed, so any references to this object in its object graph will not be updated to the new object nominated by readResolve. However, during the serialization of an object with the writeReplace method, all references to the original object in the replacement object's object graph are replaced with references to the replacement object. Therefore in cases where an object being serialized nominates a replacement object whose object graph has a reference to the original object, deserialization **will** result in an incorrect graph of objects. Furthermore, if the reference types of the object being read (nominated by writeReplace) and the original object are not compatible, the construction of the object graph **will** raise a ClassCastException”
- Simply put: **readResolve / writeReplace don’t work with circular references**

# How Do You Fix It?

```
import java.io.*;

public class Cat implements Serializable {
     Object writeReplace() { return new Proxy(); }

     static class Proxy implements Serializable { // bye bye, circular reference
          Object readResolve() { return new Cat(); }
     }

     public static void main(String[] args) throws Exception {
          ByteArrayOutputStream b = new ByteArrayOutputStream();
          Cat original = new Cat();
          new ObjectOutputStream(b).writeObject(original);
          Cat copyCat = (Cat) new ObjectInputStream(
                    new ByteArrayInputStream(b.toByteArray())).readObject();
          System.out.println(copyCat.equals(original));
     }
}
```

Prints `false`

# The Moral

- **`readResolve`/ `writeReplace`** don’t work in the presence of circular references
- The serialization proxy pattern doesn’t work if the proxy refers to the original object
  - (Unless the proxy type is compatible with the original type)
- Always make your serialization proxy classes static
- For system designers
  - Provide symmetric behavior for serialization and deserialization

# 3.“Creation Miss”

```
public class Creation {
    enum Man {
        ADAM(Woman.EVE);
        final Woman wife;
        Man(Woman wife) { this.wife = wife; }
    }

    enum Woman {
        EVE(Man.ADAM);
        final Man husband;
        Woman(Man husband) { this.husband = husband; }
    }

    public static void main(String[] args) {
        boolean adamHasEve = Man.ADAM.wife.equals(Woman.EVE);
        boolean eveHasAdam = Woman.EVE.husband.equals(Man.ADAM);
        System.out.println(adamHasEve + " " + eveHasAdam);
    }
}
```

# What Does It Print?

```
public class Creation {
    enum Man {
        ADAM(Woman.EVE);
        final Woman wife;
        Man(Woman wife) { this.wife = wife; }
    }

    enum Woman {
        EVE(Man.ADAM);
        final Man husband;
        Woman(Man husband) { this.husband = husband; }
    }

    public static void main(String[] args) {
        boolean adamHasEve = Man.ADAM.wife.equals(Woman.EVE);
        boolean eveHasAdam = Woman.EVE.husband.equals(Man.ADAM);
        System.out.println(adamHasEve + " " + eveHasAdam);
    }
}
```

(a) **`true true`**
(b) **`true false`**
(c) **`false true`**
(d) None of the above

# What Does It Print?

(a) `true true`

(b) `true false`

(c) `false true`

(d) None of the above: `NullPointerException`

`Man` and `Woman` have a circular class initialization dependency

# Another Look

*Let's analyze the class initialization in detail*

```
public class Creation {
    enum Man {
        ADAM(Woman.EVE); // 2. Param eval causes Woman init, 7. Initializes Man.ADAM
        final Woman wife;
        Man(Woman wife) { this.wife = wife; }  // 6. Correctly stores Woman.EVE
    }

    enum Woman {
        EVE(Man.ADAM);  // 3. Causes recursive Man init, 5. Initializes Woman.EVE
        final Man husband;
        Woman(Man husband) { this.husband = husband; }  // 4. Stores null
    }

    public static void main(String[] args) {
        boolean adamHasEve = Man.ADAM.wife.equals(Woman.EVE); // 1. Man init,8. true
        boolean eveHasAdam = Woman.EVE.husband.equals(Man.ADAM); // 9. Game over
        System.out.println(adamHasEve + " " + eveHasAdam);
    }
}
```

# How Do You Fix It?

*Break the circularity by manually initializing one of the two classes*

```
public class Creation {
    enum Man {
        ADAM(Woman.EVE);
        final Woman wife;
        Man(Woman wife) {
            this.wife = wife;
            wife.husband = this; // Tell my (lucky) wife that I'm her husband
        }
    }

    enum Woman {
        EVE; // No explicit constructor
        Man husband; // Sadly, not final
    }

    public static void main(String[] args) {
        boolean adamHasEve = Man.ADAM.wife.equals(Woman.EVE);
        boolean eveHasAdam = Woman.EVE.husband.equals(Man.ADAM);
        System.out.println(adamHasEve + " " + eveHasAdam);
    }
}
```

Prints `true true`

# The Moral

- Circular dependencies in class initialization are dangerous
  - Spec mandates “plowing on through”
  - As a result, classes can see one another in uninitialized states
- Enums are particularly prone to this problem
- Keep class initialization as simple as possible
- Simulate or step through initialization if you have any doubts
- To break a circularity, finish initialization of one class manually or lazily

# 4. “What The F?”

Consider the following variable declarations:

```
short a = 0xFFFF;
short b = 0xFFFF_FFFF;
```

*Thanks to Aleksey Shipilev*

# Which of These Declarations Are legal?

```
short a = 0xFFFF;
short b = 0xFFFF_FFFF;
```

(a) **a** is legal, **b** is not
(b) **b** is legal, **a** is not
(c) Both **a** and **b** are legal
(d) Neither **a** nor **b** are legal

# Which of These Declarations Are Legal?

(a) **`a`** is legal, **`b`** is not

(b) **`b`** is legal, **`a`** is not

(c) Both **`a`** and **`b`** are legal

(d) Neither **`a`** nor **`b`** are legal

We're initializing **`short`** variables with **`int`** literals

# Another Look

```
short a = 0xFFFF;           // i.e., short a = 65535;
short b = 0xFFFF_FFFF;      // i.e., short b = -1;
```

**Short values are signed, must be between -32768 and 32767 inclusive**

# How Do You Fix It?

```
short a = (short) 0xFFFF;
short b = (short) 0xFFFF;  // We omit the misleading high-order bits
```

Both declarations are now legal

# The Moral

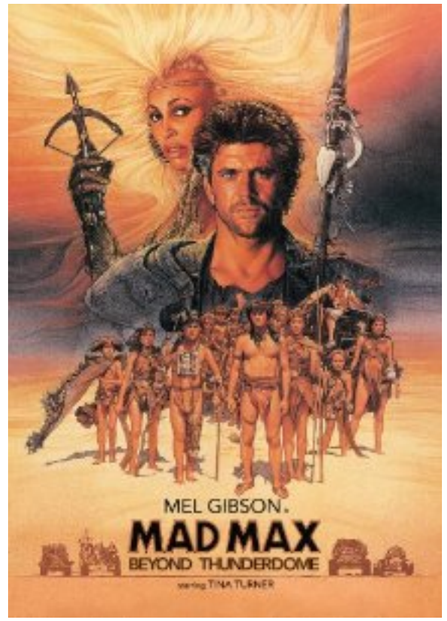
- Java lacks **`short`** (and **`byte`**) literals
  - So you're forced to use **`int`** literals when you need a **`short`** or **`byte`**
  - If the **`int`** literal value is in range for the **`short`** or **`byte`**, the compiler silently casts it for you
- Like **`int`**, the **`short`** and **`byte`** (!) types are always signed
  - To obtain a **`short`** (or **`byte`**) with high bit set, you must cast an **`int`** literal (or use a negative)
  - Hexadecimal, binary, and octal literals can be negative even without a minus sign
- **Avoid `short` and `byte`; they're painful to use, and seldom what you want**
- For Language designers
  - Provide literals for all built-in types
  - Provide unsigned primitive types

# 5. “Mad Max”

```
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.MIN_VALUE;
        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```



# What Does It Print?

(a) `0.0`
(b) `4.9E-324`
(c) Throws exception
(d) None of the above

```
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.MIN_VALUE;
        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

# What Does It Print?

(a) `0.0`

(b) **`4.9E-324`**

(c) Throws exception

(d) None of the above

**`Double.MIN_VALUE`** is very different from **`Integer.MIN_VALUE`**

# Another Look

*Integer.MIN_VALUE is most negative int; Double.MIN_VALUE is **not** most negative double*

```
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.MIN_VALUE; // Positive double w/ smallest mag: 4.9E-324

        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

# You Could Fix It Like This...

```
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.NEGATIVE_INFINITY; // Less than any other double val
        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

Prints `0.0`

# But This Is **Much** Better

```
public class Max {
    public static double max(double first, double... rest) {
        double result = first;
        for (double val : rest)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

Prints `0.0`

Shorter, Clearer, and blows up at compile time if you pass in no values

# The Moral

- The least double val is **`Double.NEGATIVE_INFINITY`**, not **`Double.MIN_VALUE`**
  - The same is true of **`Float`**
- If a method requires one or more arguments, declare with **`(T first, T... rest)`**
  - The technique generalizes to n or more values, for any n
- For API designers
  - Don’t violate the principle of least astonishment
  - Be consistent in the use of name parts (“words”)

# A Summary of the Traps

1. **Always declare collections using their interface types**
2. **Always declare serialization proxy classes static**
3. **Circular dependencies in class initialization are dangerous; to break them, finish initialization of one class manually or lazily**
4. **Avoid `short` and `byte`; they're painful to use, and seldom what you want**
5. **If a method requires 1 or more args, declare with `(T first, T... rest)`**

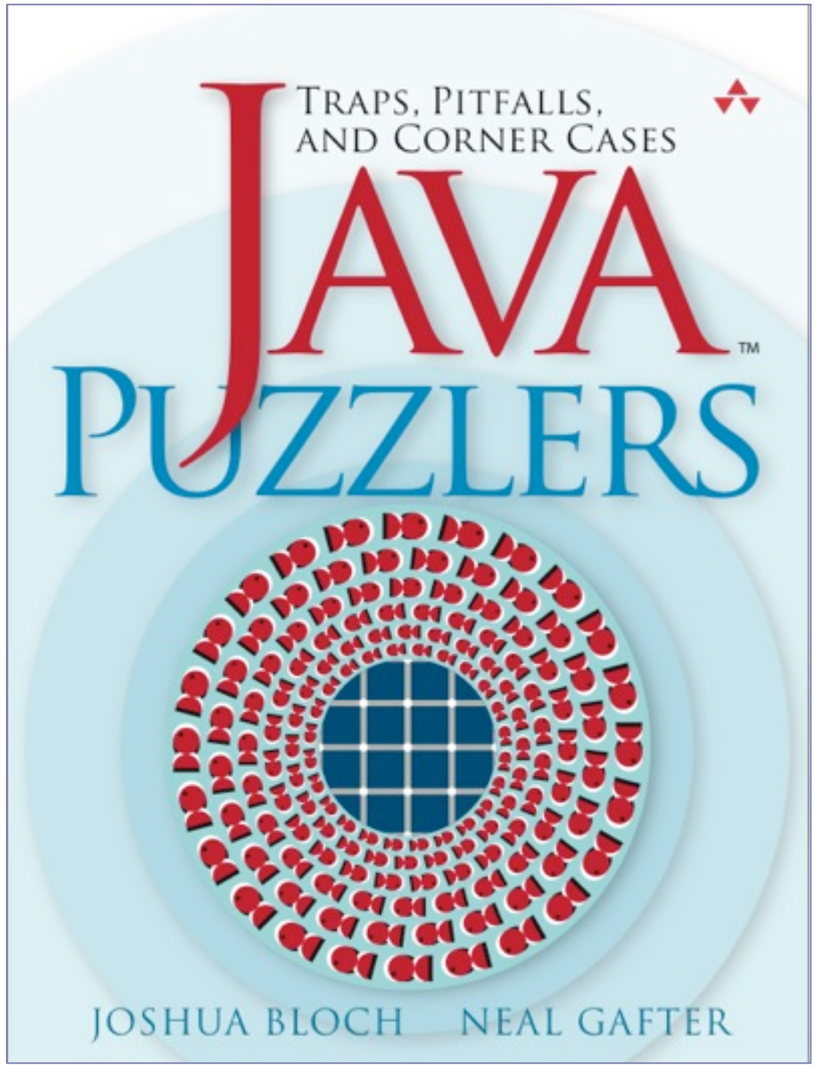
# A Summary of the Lessons for Language and API Designers

1. **Don’t violate the principle of least astonishment**
2. **Provide symmetric behavior for serialization and deserialization**
3. **Provide literals for all built-in types**
4. **Provide unsigned primitive types**
5. **Be consistent in the use of name parts (“words”)**

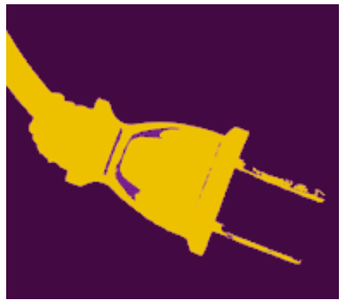
# Conclusion

- Java platform is still (as of Java 7) reasonably simple and elegant
  - But it has 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**
- Use FindBugs and a good IDE
- Don't code like my brother

# Shameless Commerce Division




- 95 Puzzles
- 52 Illusions
- Tons of fun




# Java Puzzlers

## *Something Old, Something Gnu, Something Bogus, Something Blew*

**Compliments, Valuable Goods, Hot Tips:**

Twitter: `@joshbloch @crazybob`

Hashtags: `#JavaPuzzlers #StrangeLoop #SexiestMenAlive`

**Complaints:** `/dev/null`