





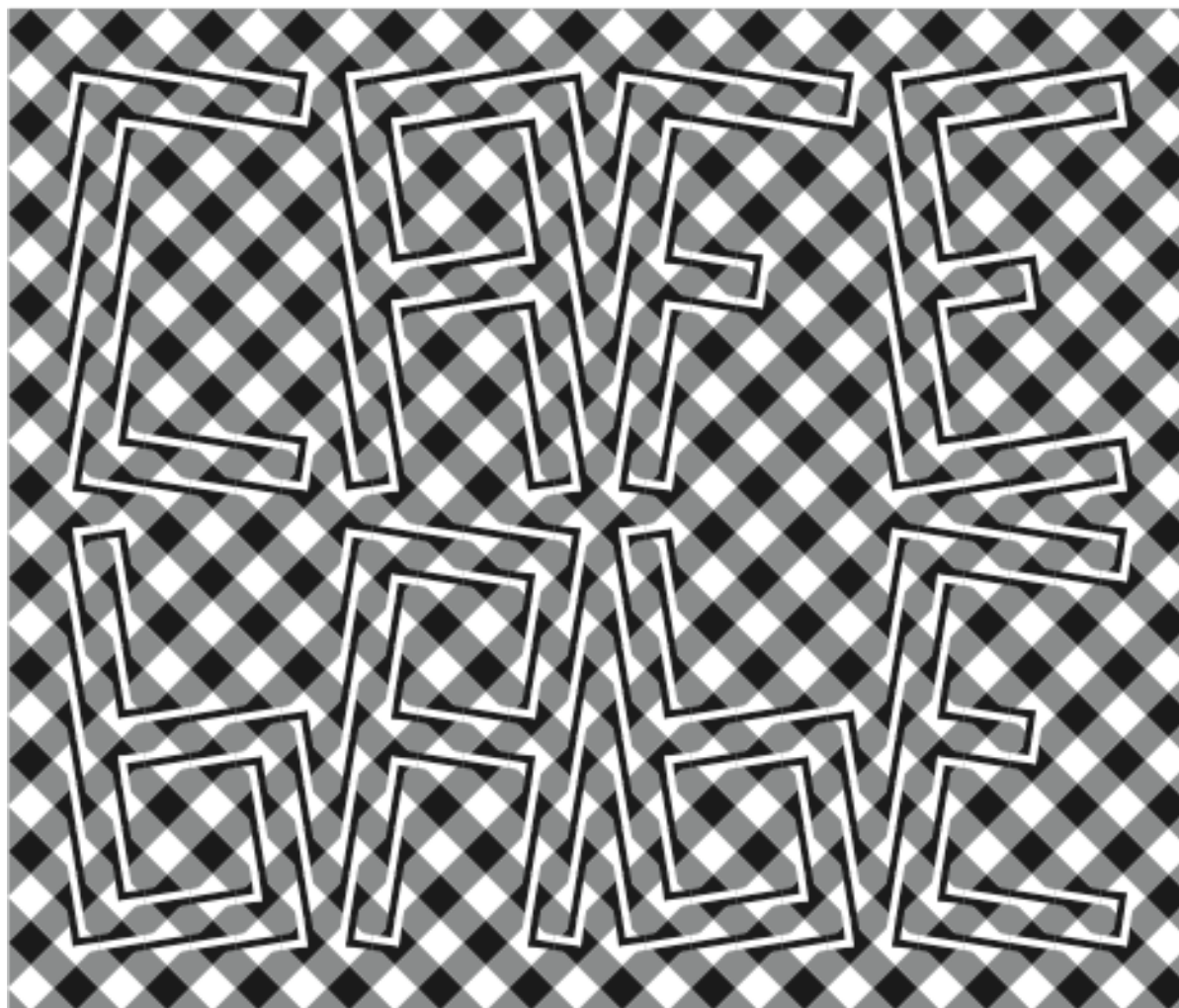
# Java Puzzlers

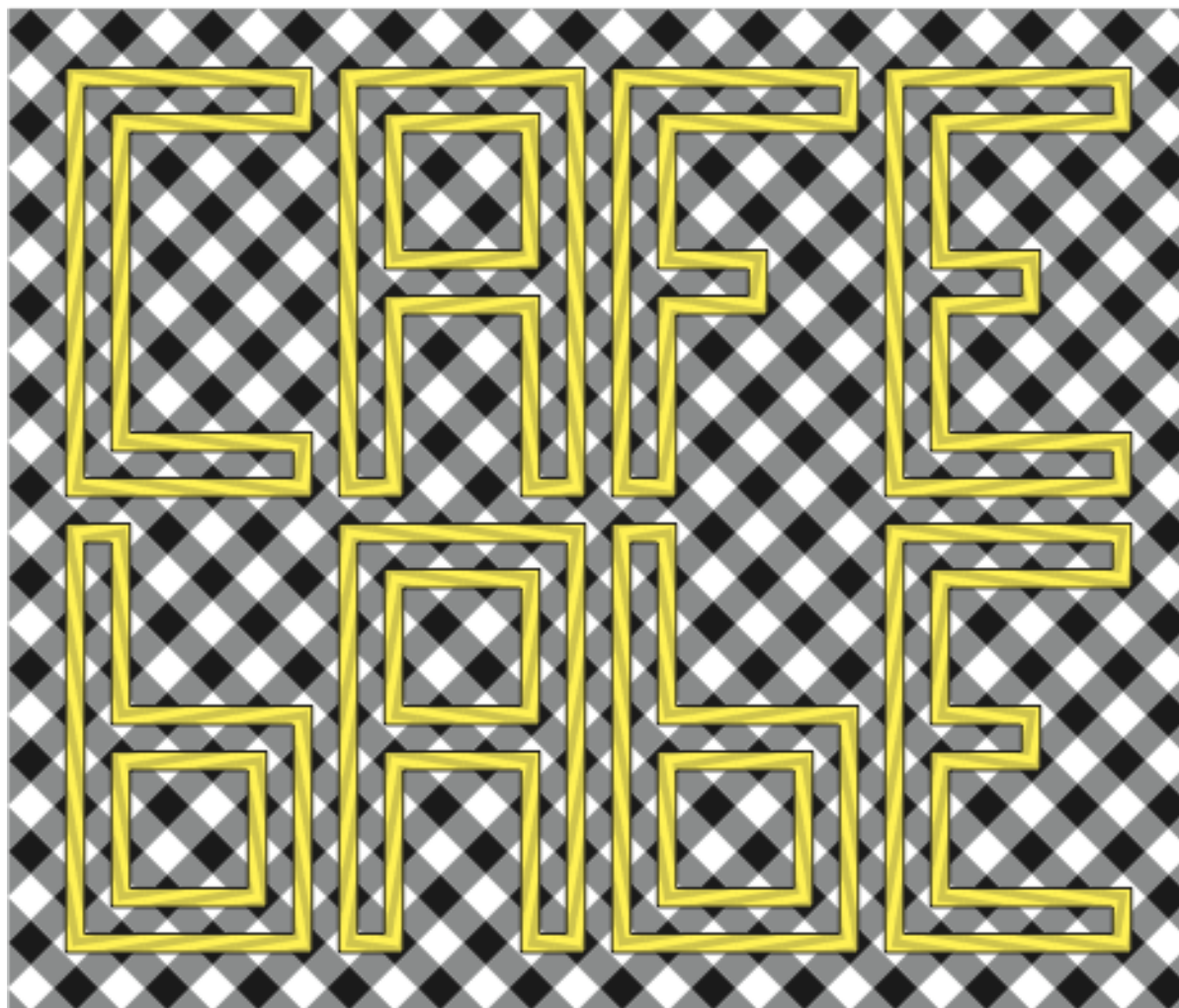
## Scrapping the Bottom of The Barrel

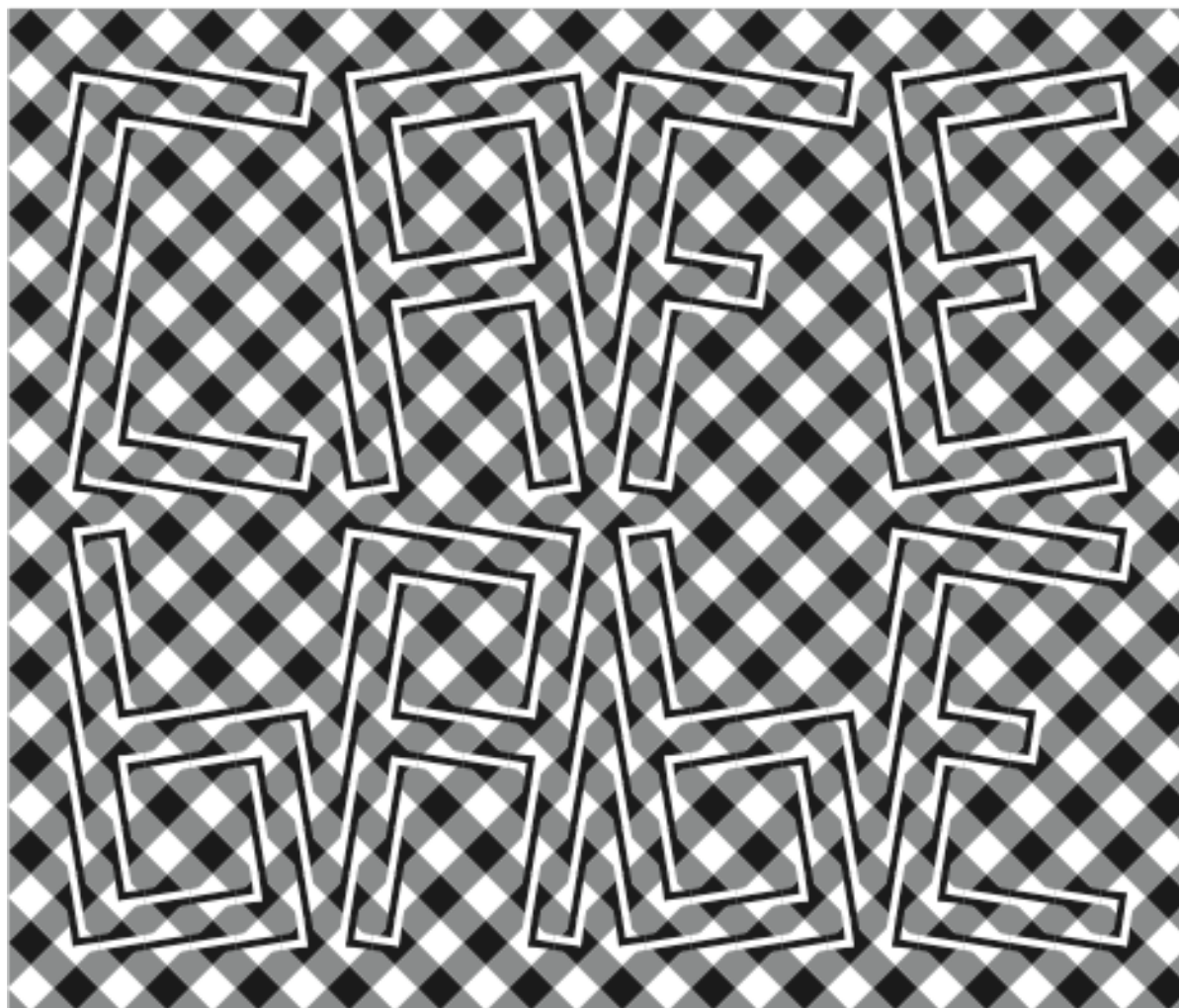
Joshua Bloch and Jeremy Manson  
May 10, 2011

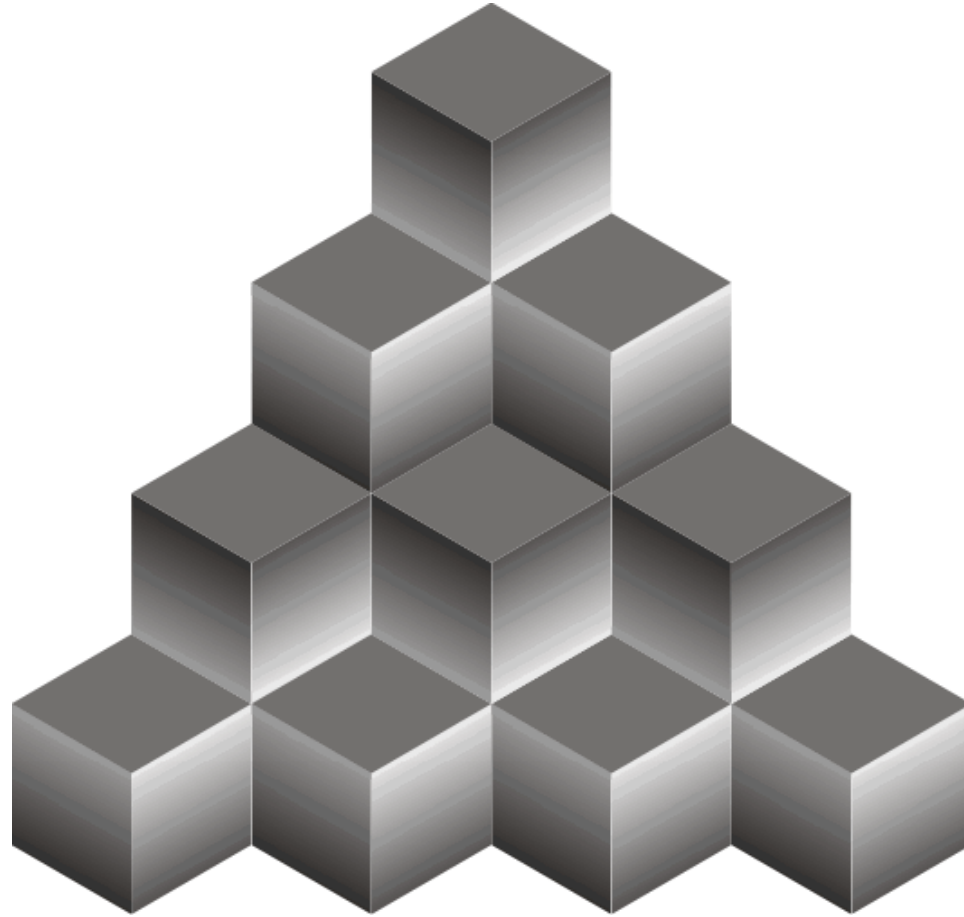


Some people say seeing is believing...



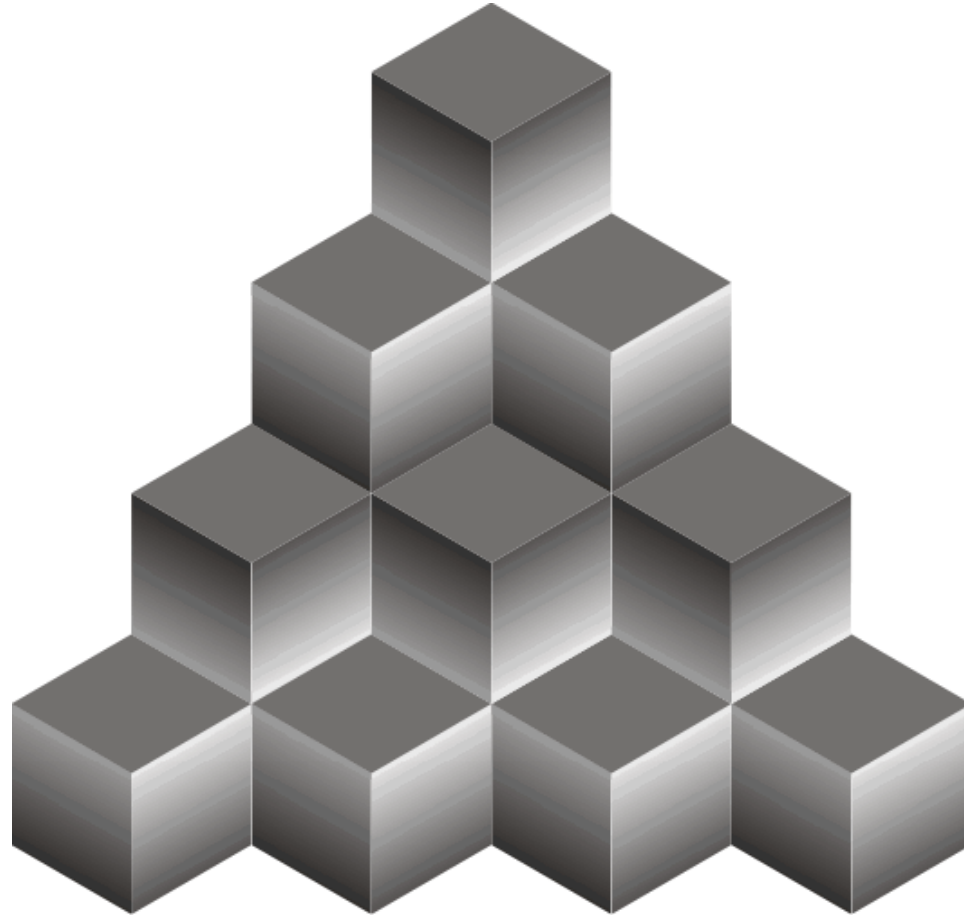


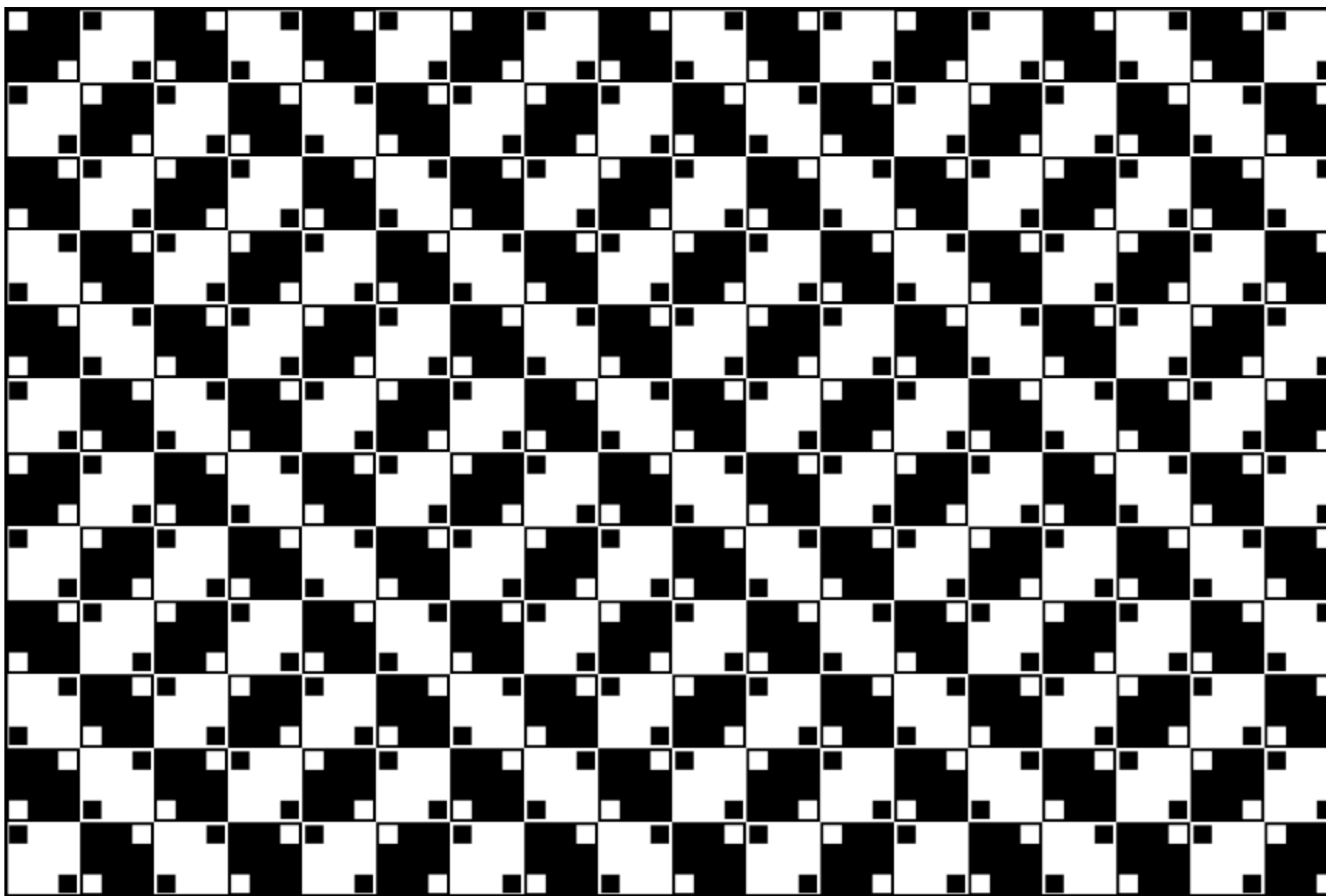


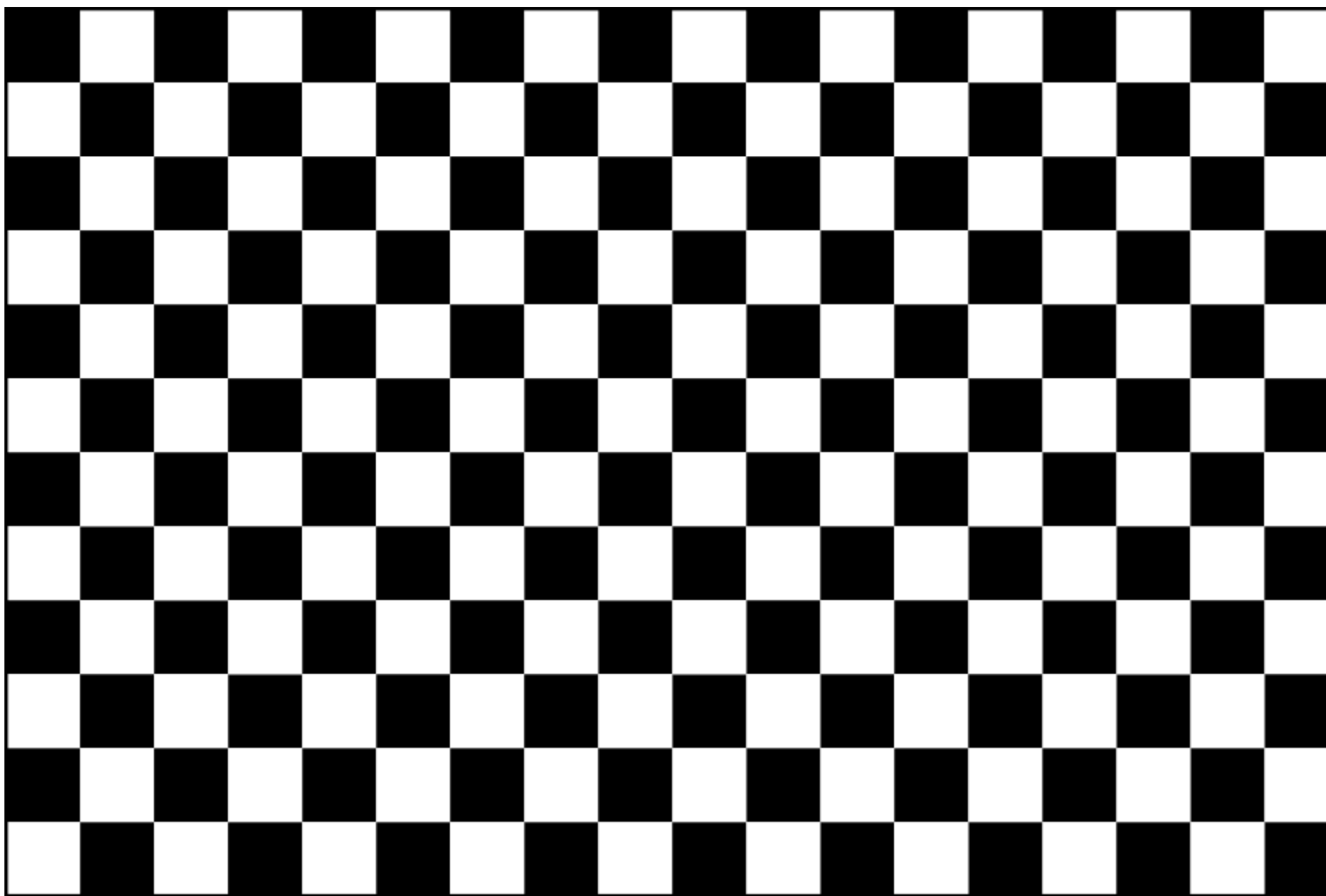


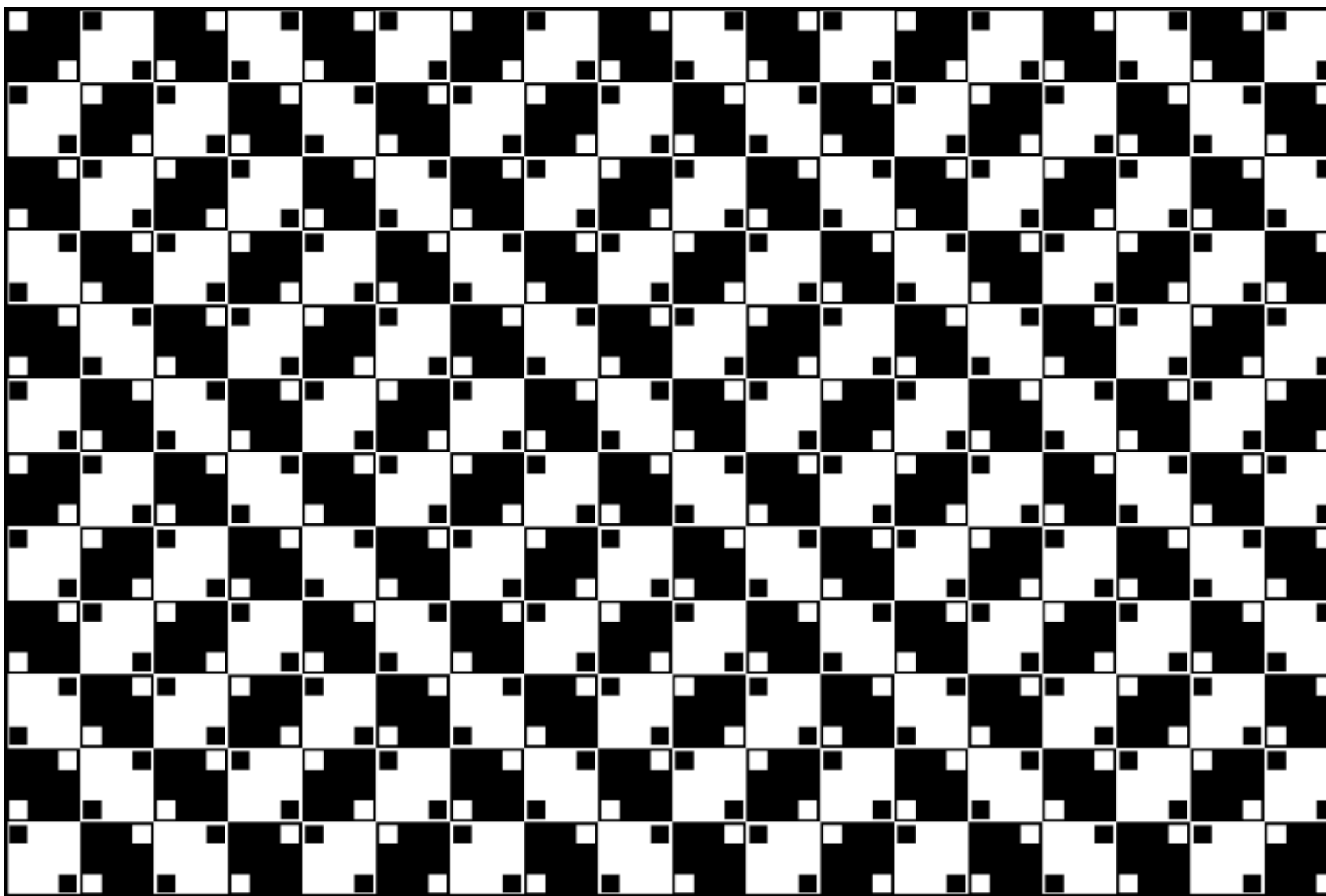












# And Now, in Code!

- **Six 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. “Time for a Change”

If you pay \$2.00 for a gasket that costs \$1.10, how much change do you get?



```
public class Change {  
    public static void main(String args[]) {  
        System.out.println(2.00 - 1.10);  
    }  
}
```

# 1. “A Change is Gonna Come”

If you pay \$2.00 for a gasket that costs \$1.10, how much change do you get?



```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```

# What Does It Print?

- (a) 0.9
- (b) 0.90
- (c) 0.89999999999999999999
- (d) None of the above

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```



# What Does It Print?

(a) 0.9

(b) 0.90

(c) 0.899999999999999999

(d) None of the above

0.89999999999999999911182158029987476766109466552734375

We used the wrong `BigDecimal` constructor

# Another Look

## The specification says this:

```
public BigDecimal(double val)
```

Translates a double into a `BigDecimal` with the exact decimal representation of the double's binary floating-point value.

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```

# How Do You Fix It?

Prints 0.90

```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal("2.00");
        BigDecimal cost = new BigDecimal("1.10");
        System.out.println(payment.subtract(cost));
    }
}
```

# The Moral

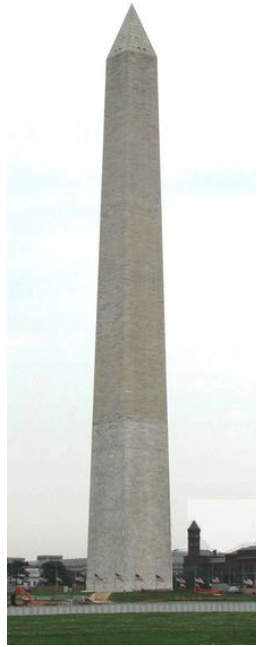
- Avoid `float` and `double` where exact answers are required
  - For example, when dealing with money
- Use `BigDecimal`, `int`, or `long` instead
- Use `new BigDecimal(String)`, not `new BigDecimal(double)`
- `BigDecimal.valueOf(double)` is better, but not perfect; use it for non-constant values.
- **For API designers**
  - **Make it easy to do the commonly correct thing**
  - **Make it possible to do exotic things**

## 2. “Size Matters”

```
public class Size {
    private enum Sex { MALE, FEMALE }

    public static void main(String[] args) {
        System.out.print(size(new HashMap<Sex, Sex>()) + " ");
        System.out.print(size(new EnumMap<Sex, Sex>(Sex.class)));
    }

    private static int size(Map<Sex, Sex> map) {
        map.put(Sex.MALE, Sex.FEMALE);
        map.put(Sex.FEMALE, Sex.MALE);
        map.put(Sex.MALE, Sex.MALE);
        map.put(Sex.FEMALE, Sex.FEMALE);
        Set<Map.Entry<Sex, Sex>> set =
            new HashSet<Map.Entry<Sex, Sex>>(map.entrySet());
        return set.size();
    }
}
```



*Thanks to Chris Dennis, via Alex Miller*

# What Does It Print?

```
public class Size {
    private enum Sex { MALE, FEMALE }

    public static void main(String[] args) {
        System.out.print(size(new HashMap<Sex, Sex>()) + " ");
        System.out.print(size(new EnumMap<Sex, Sex>(Sex.class)));
    }

    private static int size(Map<Sex, Sex> map) {
        map.put(Sex.MALE, Sex.FEMALE);
        map.put(Sex.FEMALE, Sex.MALE);
        map.put(Sex.MALE, Sex.MALE);
        map.put(Sex.FEMALE, Sex.FEMALE);
        Set<Map.Entry<Sex, Sex>> set =
            new HashSet<Map.Entry<Sex, Sex>>(map.entrySet());
        return set.size();
    }
}
```

(a) 2 1

(b) 2 2

(c) 4 4

(d) None of the above

# What Does It Print?

(a) 2 1

(b) 2 2

(c) 4 4

(d) None of the above

Enumerating over entry sets works better for some **Map** implementations than others

# Another Look

*EnumMap entrySet iterator repeatedly returns the same Entry :(*

```
public class Size {
    private enum Sex { MALE, FEMALE }

    public static void main(String[] args) {
        System.out.print(size(new HashMap<Sex, Sex>()) + " ");
        System.out.print(size(new EnumMap<Sex, Sex>(Sex.class)));
    }

    private static int size(Map<Sex, Sex> map) {
        map.put(Sex.MALE, Sex.FEMALE);
        map.put(Sex.FEMALE, Sex.MALE);
        map.put(Sex.MALE, Sex.MALE);
        map.put(Sex.FEMALE, Sex.FEMALE);
        Set<Map.Entry<Sex, Sex>> set =
            new HashSet<Map.Entry<Sex, Sex>>(map.entrySet());
        return set.size();
    }
}
```



# WTF? Is this a bug?

- Perhaps, but it's been around since 1997
  - Josh thought it was legal when he designed Collections
  - But the spec is, at best, ambiguous on this point
- Several **Map** implementations did this
  - **IdentityHashMap**, **ConcurrentHashMap**, **EnumMap**
  - **ConcurrentHashMap** was fixed a long time ago
- Happily, Android did *not* perpetuate this behavior

# How Do You Fix It?

*Copy the entries and insert manually*

Prints 2 2

```
public class Size {
    private enum Sex { MALE, FEMALE }

    public static void main(String[] args) {
        System.out.print(size(new HashMap<Sex, Sex>()) + " ");
        System.out.print(size(new EnumMap<Sex, Sex>(Sex.class)));
    }

    private static int size(Map<Sex, Sex> map) {
        map.put(Sex.MALE, Sex.FEMALE);
        map.put(Sex.FEMALE, Sex.MALE);
        map.put(Sex.MALE, Sex.MALE);
        map.put(Sex.FEMALE, Sex.FEMALE);
        Set<Map.Entry<Sex, Sex>> set = new HashSet<Map.Entry<Sex, Sex>>();
        for (Map.Entry<Sex, Sex> e : map.entrySet())
            set.add(new AbstractMap.SimpleImmutableEntry<Sex, Sex>(e));
        return set.size();
    }
}
```

# The Moral

- Iterating over `entrySet` requires care
  - `Entry` may become invalid when `Iterator` advances
  - `EnumMap` and `IdentityHashMap` are the only JDK 6 `Map` implementations with this behavior
    - No Android `Map` implementations have this behavior
  - `new HashSet<EntryType>(map.entrySet())` idiom fails in the face of this behavior
- For API designers
  - Don't violate *the principle of least astonishment*
  - Don't worsen your API to improve performance (unless you have no choice)
    - It may seem like a good idea at the time, but you'll live to regret it

### 3. “The Match Game”

```
import java.util.regex.*;

public class Match {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("(aa|aab?)+");
        int count = 0;
        for(String s = ""; s.length() < 200; s += "a")
            if (p.matcher(s).matches())
                count++;
        System.out.println(count);
    }
}
```



# What Does It Print?

```
import java.util.regex.*;

public class Match {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("(aa|aab?)+");
        int count = 0;
        for(String s = ""; s.length() < 200; s += "a")
            if (p.matcher(s).matches())
                count++;
        System.out.println(count);
    }
}
```

- (a) 99
- (b) 100
- (c) Throws an exception
- (d) None of the above

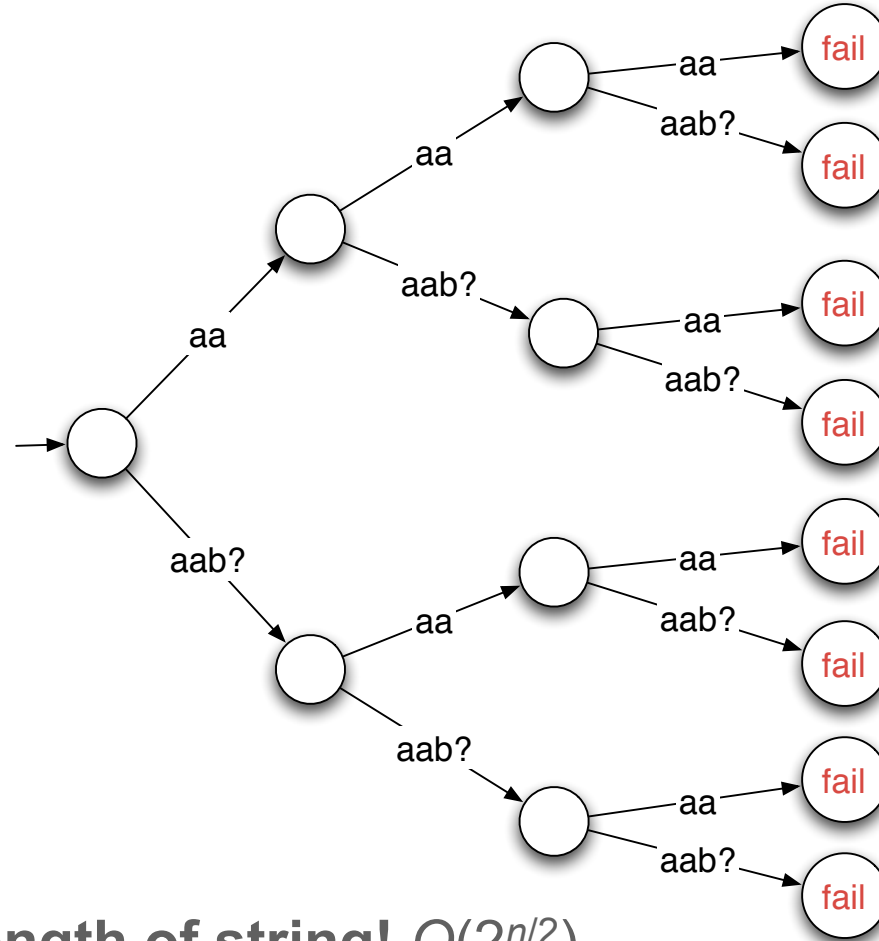
# What Does It Print?

- (a) 99
- (b) 100
- (c) Throws an exception
- (d) None of the above: runs for  $>10^{15}$  years before printing anything; the Sun won't last that long

The regular expression exhibits *catastrophic backtracking*

# What is Catastrophic Backtracking?

Here's how *matcher* tests "aaaaa" for "(aa|aab?)+"



This is **exponential** in length of string!  $O(2^{n/2})$

# How Do You Fix It?

```
import java.util.regex.*;

public class Match {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(" (aab?)+");
        int count = 0;
        for(String s = ""; s.length() < 200; s += "a")
            if (p.matcher(s).matches())
                count++;
        System.out.println(count);
    }
}
```

Prints 99

The regex " (aab?)+ " matches exactly the same strings as " (aa|aab?)+ " but doesn't exhibit catastrophic backtracking



# The Moral

- **To avoid catastrophic backtracking, ensure there's only one way to match each string**
- This goes way beyond Java
  - Affects most regular expression systems
  - Enables denial-of-service attacks
- Since regexes provide grouping information, they can't just be compiled into optimal DFAs
  - Matcher must try all combinations of subpatterns
- Just because you can express it concisely doesn't mean computation is fast

## 4. “That Sinking Feeling”

```
abstract class Sink<T> {
    abstract void add(T... elements);
    void addUnlessNull(T... elements) {
        for (T element : elements)
            if (element != null)
                add(element);
    }
}

public class StringSink extends Sink<String> {
    private final List<String> list = new ArrayList<String>();
    void add(String... elements) {
        list.addAll(Arrays.asList(elements));
    }
    public String toString() { return list.toString(); }
    public static void main(String[] args) {
        Sink<String> ss = new StringSink();
        ss.addUnlessNull("null", null);
        System.out.println(ss);
    }
}
```



# What Does It Print?

```
abstract class Sink<T> {  
    abstract void add(T... elements);  
    void addUnlessNull(T... elements) {  
        for (T element : elements)  
            if (element != null)  
                add(element);  
    }  
}
```

```
public class StringSink extends Sink<String> {  
    private final List<String> list = new ArrayList<String>();  
    void add(String... elements) {  
        list.addAll(Arrays.asList(elements));  
    }  
    public String toString() { return list.toString(); }  
    public static void main(String[] args) {  
        Sink<String> ss = new StringSink();  
        ss.addUnlessNull("null", null);  
        System.out.println(ss);  
    }  
}
```

- (a) [null]
- (b) [null, null]
- (c) NullPointerException
- (d) None of the above

# What Does It Print?

- (a) `[null]`
- (b) `[null, null]`
- (c) `NullPointerException`
- (d) None of the above: `ClassCastException`

Varargs and generics don't get along very well.

# Another Look

*Stack trace is very confusing!*

```
abstract class Sink<T> {
    abstract void add(T... elements);
    void addUnlessNull(T... elements) {
        for (T element : elements)
            if (element != null)
                add(element);           // (2)
    }
}

public class StringSink extends Sink<String> { // (3) Throws ClassCastException!
    private final List<String> list = new ArrayList<String>();
    void add(String... elements) {
        list.addAll(Arrays.asList(elements));
    }
    public String toString() { return list.toString(); }
    public static void main(String[] args) {
        Sink<String> ss = new StringSink();
        ss.addUnlessNull("null", null);    // (1)
        System.out.println(ss);
    }
}
```

# What's Going On Under the Covers?

*Varargs, erasure, and a bridge method :(*

```
abstract class Sink<T> {
    abstract void add(T... elements);    // Really Object[]
    void addUnlessNull(T... elements) {  // Really Object[]
        for (T element : elements)
            if (element != null)
                add(element); // Creates Object[]
    }
}

public class StringSink extends Sink<String> {
    private final List<String> list = new ArrayList<String>();
    /** Synthetic bridge method - not present in source! */
    void add(Object[] a) { // Overrides abstract method
        add((String[]) a); // Really throws ClassCastException!
    }
    void add(String... elements) { list.addAll(Arrays.asList(elements)); }
    public String toString() { return list.toString(); }
    public static void main(String[] args) {
        Sink<String> ss = new StringSink();
        ss.addUnlessNull("null", null); // Creates String[]
        System.out.println(ss);
    }
}
```

# The Compiler Tried to Warn You!

```
javac StringSink.java
```

Note: **StringSink.java uses unchecked or unsafe operations.**

Note: Recompile with `-Xlint:unchecked` for details.

And hopefully:

```
javac -Xlint:unchecked StringSink.java
```

```
StringSink.java:8: warning: [unchecked] unchecked generic array creation of  
type T[] for varargs parameter
```

```
    add(element);
```

```
      ^
```

# How Do You Fix It?

*Replace varargs and arrays with collections*

```
abstract class Sink<T> {
    abstract void add(Collection<T> elements);
    void addUnlessNull(Collection<T> elements) {
        for (T element : elements)
            if (element != null)
                add(Collections.singleton(element));
    }
}

public class StringSink extends Sink<String> {
    private final List<String> list = new ArrayList<String>();
    void add(Collection<String> elements) {
        list.addAll(elements); // No Arrays.asList
    }
    public String toString() { return list.toString(); }
    public static void main(String[] args) {
        Sink<String> ss = new StringSink();
        ss.addUnlessNull(Arrays.asList("null", null));
        System.out.println(ss);
    }
}
```

Prints [null]



# The Moral

- Varargs provide a *leaky abstraction*
  - Arrays leak through the veneer of varargs
- Generics and arrays don't get along well
  - So generics and varargs don't get along well
- **Prefer collections to arrays**
  - Especially in APIs
- **Don't ignore compiler warnings**
  - Ideally, eliminate them by fixing the code. If that's not possible:
    - *Prove* that there's no real problem and write down your proof in a comment
    - Disable the warning locally with a `@SuppressWarnings` annotation

## 5. “Glommer Pile”

```
public class Glommer<T> {  
    String glom(Collection<?> objs) {  
        String result = "";  
        for (Object o : objs)  
            result += o;  
        return result;  
    }  
  
    int glom(List<Integer> ints) {  
        int result = 0;  
        for (int i : ints)  
            result += i;  
        return result;  
    }  
  
    public static void main(String args[]) {  
        List<String> strings = Arrays.asList("1", "2", "3");  
        System.out.println(new Glommer().glom(strings));  
    }  
}
```



# What Does It Print?

```
public class Glommer<T> {
    String glom(Collection<?> objs) {
        String result = "";
        for (Object o : objs)
            result += o;
        return result;
    }

    int glom(List<Integer> ints) {
        int result = 0;
        for (int i : ints)
            result += i;
        return result;
    }

    public static void main(String args[]) {
        List<String> strings = Arrays.asList("1", "2", "3");
        System.out.println(new Glommer().glom(strings));
    }
}
```

- (a) 6
- (b) 123
- (c) Throws an exception
- (d) None of the above

# What Does It Print?

- (a) 6
- (b) 123
- (c) Throws an exception: `ClassCastException`
- (d) None of the above

Raw types are dangerous

# Another Look

*Raw type discards **all** generic type information, causing incorrect overload resolution*

```
public class Glommer<T> {
    String glom(Collection<?> objs) {
        String result = "";
        for (Object o : objs)
            result += o;
        return result;
    }

    int glom(List<Integer> ints) {
        int result = 0;
        for (int i : ints)
            result += i;
        return result;
    }

    public static void main(String args[]) {
        List<String> strings = Arrays.asList("1", "2", "3");
        System.out.println(new Glommer().glom(strings));
    }
}
```

# The Compiler Tried to Warn You!

```
javac Glommer.java
```

```
Note: Glommer.java uses unchecked or unsafe operations.
```

```
Note: Recompile with -Xlint:unchecked for details.
```

And hopefully:

```
javac -Xlint:unchecked Glommer.java
```

```
Glommer.java:20: warning: [unchecked] unchecked call to  
glom(List<java.lang.Integer>) as a member of the raw type Glommer  
    System.out.println(new Glommer().glom(strings));  
                                ^
```

# You Could Fix it Like This...

*Specify a type parameter for Glommer*

```
public class Glommer<T> {
    String glom(Collection<?> objs) {
        String result = "";
        for (Object o : objs)
            result += o;
        return result;
    }

    int glom(List<Integer> ints) {
        int result = 0;
        for (int i : ints)
            result += i;
        return result;
    }

    public static void main(String args[]) {
        List<String> strings = Arrays.asList("1", "2", "3");
        System.out.println(new Glommer<Random>().glom(strings));
    }
}
```

Prints 123

# Or you Could Fix it Like This...

*Remove extraneous type parameter from Glommer*

```
public class Glommer { // Look Ma, no type parameter!
    String glom(Collection<?> objs) {
        String result = "";
        for (Object o : objs)
            result += o;
        return result;
    }

    int glom(List<Integer> ints) {
        int result = 0;
        for (int i : ints)
            result += i;
        return result;
    }

    public static void main(String args[]) {
        List<String> strings = Arrays.asList("1", "2", "3");
        System.out.println(new Glommer().glom(strings));
    }
}
```

Prints 123



# But This is even better

*Make Glommer a static utility class*

```
public class Glommer {  
    static String glom(Collection<?> objs) {  
        String result = "";  
        for (Object o : objs)  
            result += o;  
        return result;  
    }  
  
    static int glom(List<Integer> ints) {  
        int result = 0;  
        for (int i : ints)  
            result += i;  
        return result;  
    }  
  
    public static void main(String args[]) {  
        List<String> strings = Arrays.asList("1", "2", "3");  
        System.out.println(glom(strings)); // No Glommer instance!  
    }  
}
```

Prints 123

# The Moral

- **Never use raw types in new code!**
- Raw types lose **all** generic type information
  - Can break overload resolution
- **Do not ignore compiler warnings, even when they're indecipherable**
  - Unchecked warnings mean automatically generated casts can fail at runtime

## 6. “It’s Elementary” (2004; 2010 remix)



```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.print(12345 + 54321);  
        System.out.print(" ");  
        System.out.print(01234 + 43210);  
    }  
}
```

The Periodic Table of the Elements

1 <b>H</b> Hydrogen 1.00794																	2 <b>He</b> Helium 4.00260																																																																																																																																																																																																																																																																																																																																											
3 <b>Li</b> Lithium 6.941	4 <b>Be</b> Beryllium 9.012182															5 <b>B</b> Boron 10.811	6 <b>C</b> Carbon 12.0107	7 <b>N</b> Nitrogen 14.00643	8 <b>O</b> Oxygen 15.9994	9 <b>F</b> Fluorine 18.9984032	10 <b>Ne</b> Neon 20.1797																																																																																																																																																																																																																																																																																																																																							
11 <b>Na</b> Sodium 22.989770	12 <b>Mg</b> Magnesium 24.3050															13 <b>Al</b> Aluminum 26.981538	14 <b>Si</b> Silicon 28.0855	15 <b>P</b> Phosphorus 30.973761	16 <b>S</b> Sulfur 32.066	17 <b>Cl</b> Chlorine 35.4527	18 <b>Ar</b> Argon 39.948																																																																																																																																																																																																																																																																																																																																							
19 <b>K</b> Potassium 39.0983	20 <b>Ca</b> Calcium 40.078	21 <b>Sc</b> Scandium 44.955910	22 <b>Ti</b> Titanium 47.867	23 <b>V</b> Vanadium 50.9415	24 <b>Cr</b> Chromium 51.9961	25 <b>Mn</b> Manganese 54.938049	26 <b>Fe</b> Iron 55.845	27 <b>Co</b> Cobalt 58.933200	28 <b>Ni</b> Nickel 58.6934	29 <b>Cu</b> Copper 63.546	30 <b>Zn</b> Zinc 65.39	31 <b>Ga</b> Gallium 69.723	32 <b>Ge</b> Germanium 72.61	33 <b>As</b> Arsenic 74.92160	34 <b>Se</b> Selenium 78.96	35 <b>Br</b> Bromine 79.904	36 <b>Kr</b> Krypton 83.80																																																																																																																																																																																																																																																																																																																																											
37 <b>Rb</b> Rubidium 85.4678	38 <b>Sr</b> Strontium 87.62	39 <b>Y</b> Yttrium 88.90585	40 <b>Zr</b> Zirconium 91.224	41 <b>Nb</b> Niobium 92.90638	42 <b>Mo</b> Molybdenum 95.94	43 <b>Tc</b> Technetium (98)	44 <b>Ru</b> Ruthenium 101.07	45 <b>Rh</b> Rhodium 102.90550	46 <b>Pd</b> Palladium 106.42	47 <b>Ag</b> Silver 107.8682	48 <b>Cd</b> Cadmium 112.411	49 <b>In</b> Indium 114.818	50 <b>Sn</b> Tin 118.710	51 <b>Sb</b> Antimony 121.760	52 <b>Te</b> Tellurium 127.60	53 <b>I</b> Iodine 126.90447	54 <b>Xe</b> Xenon 131.29																																																																																																																																																																																																																																																																																																																																											
55 <b>Cs</b> Cesium 132.90545	56 <b>Ba</b> Barium 137.327	57 <b>La</b> Lanthanum 138.9055	58 <b>Ce</b> Cerium 140.9076	59 <b>Pr</b> Praseodymium 140.9076	60 <b>Nd</b> Neodymium 144.24	61 <b>Pm</b> Promethium (145)	62 <b>Sm</b> Samarium 150.36	63 <b>Eu</b> Europium 151.964	64 <b>Gd</b> Gadolinium 157.25	65 <b>Tb</b> Terbium 158.92534	66 <b>Dy</b> Dysprosium 162.50	67 <b>Ho</b> Holmium 164.93032	68 <b>Er</b> Erbium 167.26	69 <b>Tm</b> Thulium 168.93421	70 <b>Yb</b> Ytterbium 173.04	71 <b>Lu</b> Lutetium 174.967	72 <b>Hf</b> Hafnium 178.49																																																																																																																																																																																																																																																																																																																																											
87 <b>Fr</b> Francium (223)	88 <b>Ra</b> Radium (226)	89 <b>Ac</b> Actinium (227)	90 <b>Th</b> Thorium 232.0381	91 <b>Pa</b> Protactinium 231.03588	92 <b>U</b> Uranium 238.0289	93 <b>Np</b> Neptunium (237)	94 <b>Pu</b> Plutonium (244)	95 <b>Am</b> Americium (243)	96 <b>Cm</b> Curium (247)	97 <b>Bk</b> Berkelium (247)	98 <b>Cf</b> Californium (251)	99 <b>Es</b> Einsteinium (252)	100 <b>Fm</b> Fermium (257)	101 <b>Md</b> Mendelevium (258)	102 <b>No</b> Nobelium (259)	103 <b>Lr</b> Lawrencium (262)	104 <b>Rf</b> Rutherfordium (261)																																																																																																																																																																																																																																																																																																																																											
																		105 <b>Db</b> Dubnium (262)	106 <b>Sg</b> Seaborgium (263)	107 <b>Bh</b> Bohrium (262)	108 <b>Hs</b> Hassium (265)	109 <b>Mt</b> Meitnerium (266)	110 <b>Ds</b> Darmstadtium (271)	111 <b>Rg</b> Roentgenium (272)	112 <b>Cn</b> Copernicium (285)	113 <b>Nh</b> Nihonium (286)	114 <b>Fl</b> Flerovium (289)	115 <b>Mc</b> Moscovium (290)	116 <b>Lv</b> Livermorium (293)	117 <b>Ts</b> Tennessine (294)	118 <b>Og</b> Oganesson (294)	119 <b>Uu</b> Ununennium (295)	120 <b>Uub</b> Unbibium (296)	121 <b>Uut</b> Untrium (297)	122 <b>Uuq</b> Unquadium (298)	123 <b>Uubk</b> Unbikium (299)	124 <b>Uuhk</b> Unhikium (300)	125 <b>Uufl</b> Unflavium (301)	126 <b>Uuql</b> Unquadium (302)	127 <b>Uubk</b> Unbikium (303)	128 <b>Uuhk</b> Unhikium (304)	129 <b>Uufl</b> Unflavium (305)	130 <b>Uuql</b> Unquadium (306)	131 <b>Uubk</b> Unbikium (307)	132 <b>Uuhk</b> Unhikium (308)	133 <b>Uufl</b> Unflavium (309)	134 <b>Uuql</b> Unquadium (310)	135 <b>Uubk</b> Unbikium (311)	136 <b>Uuhk</b> Unhikium (312)	137 <b>Uufl</b> Unflavium (313)	138 <b>Uuql</b> Unquadium (314)	139 <b>Uubk</b> Unbikium (315)	140 <b>Uuhk</b> Unhikium (316)	141 <b>Uufl</b> Unflavium (317)	142 <b>Uuql</b> Unquadium (318)	143 <b>Uubk</b> Unbikium (319)	144 <b>Uuhk</b> Unhikium (320)	145 <b>Uufl</b> Unflavium (321)	146 <b>Uuql</b> Unquadium (322)	147 <b>Uubk</b> Unbikium (323)	148 <b>Uuhk</b> Unhikium (324)	149 <b>Uufl</b> Unflavium (325)	150 <b>Uuql</b> Unquadium (326)	151 <b>Uubk</b> Unbikium (327)	152 <b>Uuhk</b> Unhikium (328)	153 <b>Uufl</b> Unflavium (329)	154 <b>Uuql</b> Unquadium (330)	155 <b>Uubk</b> Unbikium (331)	156 <b>Uuhk</b> Unhikium (332)	157 <b>Uufl</b> Unflavium (333)	158 <b>Uuql</b> Unquadium (334)	159 <b>Uubk</b> Unbikium (335)	160 <b>Uuhk</b> Unhikium (336)	161 <b>Uufl</b> Unflavium (337)	162 <b>Uuql</b> Unquadium (338)	163 <b>Uubk</b> Unbikium (339)	164 <b>Uuhk</b> Unhikium (340)	165 <b>Uufl</b> Unflavium (341)	166 <b>Uuql</b> Unquadium (342)	167 <b>Uubk</b> Unbikium (343)	168 <b>Uuhk</b> Unhikium (344)	169 <b>Uufl</b> Unflavium (345)	170 <b>Uuql</b> Unquadium (346)	171 <b>Uubk</b> Unbikium (347)	172 <b>Uuhk</b> Unhikium (348)	173 <b>Uufl</b> Unflavium (349)	174 <b>Uuql</b> Unquadium (350)	175 <b>Uubk</b> Unbikium (351)	176 <b>Uuhk</b> Unhikium (352)	177 <b>Uufl</b> Unflavium (353)	178 <b>Uuql</b> Unquadium (354)	179 <b>Uubk</b> Unbikium (355)	180 <b>Uuhk</b> Unhikium (356)	181 <b>Uufl</b> Unflavium (357)	182 <b>Uuql</b> Unquadium (358)	183 <b>Uubk</b> Unbikium (359)	184 <b>Uuhk</b> Unhikium (360)	185 <b>Uufl</b> Unflavium (361)	186 <b>Uuql</b> Unquadium (362)	187 <b>Uubk</b> Unbikium (363)	188 <b>Uuhk</b> Unhikium (364)	189 <b>Uufl</b> Unflavium (365)	190 <b>Uuql</b> Unquadium (366)	191 <b>Uubk</b> Unbikium (367)	192 <b>Uuhk</b> Unhikium (368)	193 <b>Uufl</b> Unflavium (369)	194 <b>Uuql</b> Unquadium (370)	195 <b>Uubk</b> Unbikium (371)	196 <b>Uuhk</b> Unhikium (372)	197 <b>Uufl</b> Unflavium (373)	198 <b>Uuql</b> Unquadium (374)	199 <b>Uubk</b> Unbikium (375)	200 <b>Uuhk</b> Unhikium (376)	201 <b>Uufl</b> Unflavium (377)	202 <b>Uuql</b> Unquadium (378)	203 <b>Uubk</b> Unbikium (379)	204 <b>Uuhk</b> Unhikium (380)	205 <b>Uufl</b> Unflavium (381)	206 <b>Uuql</b> Unquadium (382)	207 <b>Uubk</b> Unbikium (383)	208 <b>Uuhk</b> Unhikium (384)	209 <b>Uufl</b> Unflavium (385)	210 <b>Uuql</b> Unquadium (386)	211 <b>Uubk</b> Unbikium (387)	212 <b>Uuhk</b> Unhikium (388)	213 <b>Uufl</b> Unflavium (389)	214 <b>Uuql</b> Unquadium (390)	215 <b>Uubk</b> Unbikium (391)	216 <b>Uuhk</b> Unhikium (392)	217 <b>Uufl</b> Unflavium (393)	218 <b>Uuql</b> Unquadium (394)	219 <b>Uubk</b> Unbikium (395)	220 <b>Uuhk</b> Unhikium (396)	221 <b>Uufl</b> Unflavium (397)	222 <b>Uuql</b> Unquadium (398)	223 <b>Uubk</b> Unbikium (399)	224 <b>Uuhk</b> Unhikium (400)	225 <b>Uufl</b> Unflavium (401)	226 <b>Uuql</b> Unquadium (402)	227 <b>Uubk</b> Unbikium (403)	228 <b>Uuhk</b> Unhikium (404)	229 <b>Uufl</b> Unflavium (405)	230 <b>Uuql</b> Unquadium (406)	231 <b>Uubk</b> Unbikium (407)	232 <b>Uuhk</b> Unhikium (408)	233 <b>Uufl</b> Unflavium (409)	234 <b>Uuql</b> Unquadium (410)	235 <b>Uubk</b> Unbikium (411)	236 <b>Uuhk</b> Unhikium (412)	237 <b>Uufl</b> Unflavium (413)	238 <b>Uuql</b> Unquadium (414)	239 <b>Uubk</b> Unbikium (415)	240 <b>Uuhk</b> Unhikium (416)	241 <b>Uufl</b> Unflavium (417)	242 <b>Uuql</b> Unquadium (418)	243 <b>Uubk</b> Unbikium (419)	244 <b>Uuhk</b> Unhikium (420)	245 <b>Uufl</b> Unflavium (421)	246 <b>Uuql</b> Unquadium (422)	247 <b>Uubk</b> Unbikium (423)	248 <b>Uuhk</b> Unhikium (424)	249 <b>Uufl</b> Unflavium (425)	250 <b>Uuql</b> Unquadium (426)	251 <b>Uubk</b> Unbikium (427)	252 <b>Uuhk</b> Unhikium (428)	253 <b>Uufl</b> Unflavium (429)	254 <b>Uuql</b> Unquadium (430)	255 <b>Uubk</b> Unbikium (431)	256 <b>Uuhk</b> Unhikium (432)	257 <b>Uufl</b> Unflavium (433)	258 <b>Uuql</b> Unquadium (434)	259 <b>Uubk</b> Unbikium (435)	260 <b>Uuhk</b> Unhikium (436)	261 <b>Uufl</b> Unflavium (437)	262 <b>Uuql</b> Unquadium (438)	263 <b>Uubk</b> Unbikium (439)	264 <b>Uuhk</b> Unhikium (440)	265 <b>Uufl</b> Unflavium (441)	266 <b>Uuql</b> Unquadium (442)	267 <b>Uubk</b> Unbikium (443)	268 <b>Uuhk</b> Unhikium (444)	269 <b>Uufl</b> Unflavium (445)	270 <b>Uuql</b> Unquadium (446)	271 <b>Uubk</b> Unbikium (447)	272 <b>Uuhk</b> Unhikium (448)	273 <b>Uufl</b> Unflavium (449)	274 <b>Uuql</b> Unquadium (450)	275 <b>Uubk</b> Unbikium (451)	276 <b>Uuhk</b> Unhikium (452)	277 <b>Uufl</b> Unflavium (453)	278 <b>Uuql</b> Unquadium (454)	279 <b>Uubk</b> Unbikium (455)	280 <b>Uuhk</b> Unhikium (456)	281 <b>Uufl</b> Unflavium (457)	282 <b>Uuql</b> Unquadium (458)	283 <b>Uubk</b> Unbikium (459)	284 <b>Uuhk</b> Unhikium (460)	285 <b>Uufl</b> Unflavium (461)	286 <b>Uuql</b> Unquadium (462)	287 <b>Uubk</b> Unbikium (463)	288 <b>Uuhk</b> Unhikium (464)	289 <b>Uufl</b> Unflavium (465)	290 <b>Uuql</b> Unquadium (466)	291 <b>Uubk</b> Unbikium (467)	292 <b>Uuhk</b> Unhikium (468)	293 <b>Uufl</b> Unflavium (469)	294 <b>Uuql</b> Unquadium (470)	295 <b>Uubk</b> Unbikium (471)	296 <b>Uuhk</b> Unhikium (472)	297 <b>Uufl</b> Unflavium (473)	298 <b>Uuql</b> Unquadium (474)	299 <b>Uubk</b> Unbikium (475)	300 <b>Uuhk</b> Unhikium (476)	301 <b>Uufl</b> Unflavium (477)	302 <b>Uuql</b> Unquadium (478)	303 <b>Uubk</b> Unbikium (479)	304 <b>Uuhk</b> Unhikium (480)	305 <b>Uufl</b> Unflavium (481)	306 <b>Uuql</b> Unquadium (482)	307 <b>Uubk</b> Unbikium (483)	308 <b>Uuhk</b> Unhikium (484)	309 <b>Uufl</b> Unflavium (485)	310 <b>Uuql</b> Unquadium (486)	311 <b>Uubk</b> Unbikium (487)	312 <b>Uuhk</b> Unhikium (488)	313 <b>Uufl</b> Unflavium (489)	314 <b>Uuql</b> Unquadium (490)	315 <b>Uubk</b> Unbikium (491)	316 <b>Uuhk</b> Unhikium (492)	317 <b>Uufl</b> Unflavium (493)	318 <b>Uuql</b> Unquadium (494)	319 <b>Uubk</b> Unbikium (495)	320 <b>Uuhk</b> Unhikium (496)	321 <b>Uufl</b> Unflavium (497)	322 <b>Uuql</b> Unquadium (498)	323 <b>Uubk</b> Unbikium (499)	324 <b>Uuhk</b> Unhikium (500)	325 <b>Uufl</b> Unflavium (501)	326 <b>Uuql</b> Unquadium (502)	327 <b>Uubk</b> Unbikium (503)	328 <b>Uuhk</b> Unhikium (504)	329 <b>Uufl</b> Unflavium (505)	330 <b>Uuql</b> Unquadium (506)	331 <b>Uubk</b> Unbikium (507)	332 <b>Uuhk</b> Unhikium (508)	333 <b>Uufl</b> Unflavium (509)	334 <b>Uuql</b> Unquadium (510)	335 <b>Uubk</b> Unbikium (511)	336 <b>Uuhk</b> Unhikium (512)	337 <b>Uufl</b> Unflavium (513)	338 <b>Uuql</b> Unquadium (514)	339 <b>Uubk</b> Unbikium (515)	340 <b>Uuhk</b> Unhikium (516)	341 <b>Uufl</b> Unflavium (517)	342 <b>Uuql</b> Unquadium (518)	343 <b>Uubk</b> Unbikium (519)	344 <b>Uuhk</b> Unhikium (520)	345 <b>Uufl</b> Unflavium (521)	346 <b>Uuql</b> Unquadium (522)	347 <b>Uubk</b> Unbikium (523)	348 <b>Uuhk</b> Unhikium (524)	349 <b>Uufl</b> Unflavium (525)	350 <b>Uuql</b> Unquadium (526)	351 <b>Uubk</b> Unbikium (527)	352 <b>Uuhk</b> Unhikium (528)	353 <b>Uufl</b> Unflavium (529)	354 <b>Uuql</b> Unquadium (530)	355 <b>Uubk</b> Unbikium (531)	356 <b>Uuhk</b> Unhikium (532)	357 <b>Uufl</b> Unflavium (533)	358 <b>Uuql</b> Unquadium (534)	359 <b>Uubk</b> Unbikium (535)	360 <b>Uuhk</b> Unhikium (536)	361 <b>Uufl</b> Unflavium (537)	362 <b>Uuql</b> Unquadium (538)	363 <b>Uubk</b> Unbikium (539)	364 <b>Uuhk</b> Unhikium (540)	365 <b>Uufl</b> Unflavium (541)	366 <b>Uuql</b> Unquadium (542)	367 <b>Uubk</b> Unbikium (543)	368 <b>Uuhk</b> Unhikium (544)	369 <b>Uufl</b> Unflavium (545)	370 <b>Uuql</b> Unquadium (546)	371 <b>Uubk</b> Unbikium (547)	372 <b>Uuhk</b> Unhikium (548)	373 <b>Uufl</b> Unflavium (549)	374 <b>Uuql</b> Unquadium (550)	375 <b>Uubk</b> Unbikium (551)	376 <b>Uuhk</b> Unhikium (552)	377 <b>Uufl</b> Unflavium (553)	378 <b>Uuql</b> Unquadium (554)	379 <b>Uubk</b> Unbikium (555)	380 <b>Uuhk</b> Unhikium (556)	381 <b>Uufl</b> Unflavium (557)	382 <b>Uuql</b> Unquadium (558)	383 <b>Uubk</b> Unbikium (559)	384 <b>Uuhk</b> Unhikium (560)	385 <b>Uufl</b> Unflavium (561)	386 <b>Uuql</b> Unquadium (562)	387 <b>Uubk</b> Unbikium (563)	388 <b>Uuhk</b> Unhikium (564)	389 <b>Uufl</b> Unflavium (565)	390 <b>Uuql</b> Unquadium (566)	391 <b>Uubk</b> Unbikium (567)	392 <b>Uuhk</b> Unhikium (568)	393 <b>Uufl</b> Unflavium (569)	394 <b>Uuql</b> Unquadium (570)	395 <b>Uubk</b> Unbikium (571)	396 <b>Uuhk</b> Unhikium (572)	397 <b>Uufl</b> Unflavium (573)	398 <b>Uuql</b> Unquadium (574)	399 <b>Uubk</b> Unbikium (575)	400 <b>Uuhk</b> Unhikium (576)	401 <b>Uufl</b> Unflavium (577)	402 <b>Uuql</b> Unquadium (578)	403 <b>Uubk</b> Unbikium (579)	404 <b>Uuhk</b> Unhikium (580)	405 <b>Uufl</b> Unflavium (581)	406 <b>Uuql</b> Unquadium (582)	407 <b>Uubk</b> Unbikium (583)	408 <b>Uuhk</b> Unhikium (584)	409 <b>Uufl</b> Unflavium (585)	410 <b>Uuql</b> Unquadium (586)	411 <b>Uubk</b> Unbikium (587)	412 <b>Uuhk</b> Unhikium (588)	413 <b>Uufl</b> Unflavium (589)	414 <b>Uuql</b> Unquadium (590)	415 <b>Uubk</b> Unbikium (591)	416 <b>Uuhk</b> Unhikium (592)	417 <b>Uufl</b> Unflavium (593)	418 <b>Uuql</b> Unquadium (594)	419 <b>Uubk</b> Unbikium (595)	420 <b>Uuhk</b> Unhikium (596)	421 <b>Uufl</b> Unflavium (597)	422 <b>Uuql</b> Unquadium (598)	423 <b>Uubk</b> Unbikium (599)	424 <b>Uuhk</b> Unhikium (600)	425 <b>Uufl</b> Unflavium (601)	426 <b>Uuql</b> Unquadium (602)	427 <b>Uubk</b> Unbikium (603)	428 <b>Uuhk</b> Unhikium (604)	429 <b>Uufl</b> Unflavium (605)	430 <b>Uuql</b> Unquadium (606)	431 <b>Uubk</b> Unbikium (607)	432 <b>Uuhk</b> Unhikium (608)	433 <b>Uufl</b> Unflavium (609)	434 <b>Uuql</b> Unquadium (610)	435 <b>Uubk</b> Unbikium

# What Does It Print?

```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.print(12345 + 54321);  
        System.out.print(" ");  
        System.out.print(01234 + 43210);  
    }  
}
```

- (a) 17777 44444
- (b) 17777 43878
- (c) 66666 44444
- (d) 66666 43878

# What Does It Print?

(a) 17777 44444

(b) 17777 43878

(c) 66666 44444

(d) 66666 43878

Program doesn't say what you think it does!  
Also, leading zeros can cause trouble.

# Another Look

```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.print(12345 + 5432l);  
        System.out.print(" ");  
        System.out.print(01234 + 43210);  
    }  
}
```

- 1** - the digit one
- l** - the lowercase letter el

## Another Look, Continued

```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.print(12345 + 54321);  
        System.out.print(" ");  
        System.out.print(01234 + 43210);  
    }  
}
```

**01234** is an octal literal equal to  $1,234_8$ , or 668

# How Do You Fix It?

```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.print(12345 + 54321); // The digit 1  
        System.out.print(" ");  
        System.out.print( 1234 + 43210); // No leading 0  
    }  
}
```

Prints 66666 44444



# The Moral

- **Always** use uppercase `el` (`L`) for long literals
  - Lowercase `el` makes the code unreadable
  - `5432L` is clearly a long, `5432l` is misleading
- Never use lowercase `el` (`l`) as a variable name
  - Not this: `List<String> l = ... ;`
  - But this: `List<String> list = ...;`
- Never precede an `int` literal with `0` unless you actually want to express it in octal
  - When you use an octal literal, always add a comment expressing your intent

# A Summary of the Traps

1. Use `new BigDecimal(String)`, not `new BigDecimal(double)`
2. Don't assume that `Map.Entry` objects in `entrySet` iteration are stable  
`new HashSet<EntryType>(map.entrySet())` idiom fails for `EnumMap`, `IdentityHashMap`
3. Beware of catastrophic backtracking when writing regular expressions
4. Generics and arrays don't mix—don't ignore compiler warnings!
5. Never use raw types in new code—they lose all generic type information
6. Always use uppercase `L` for `long` literals; never use `0` to pad `int` literal

# Lessons for API Designers

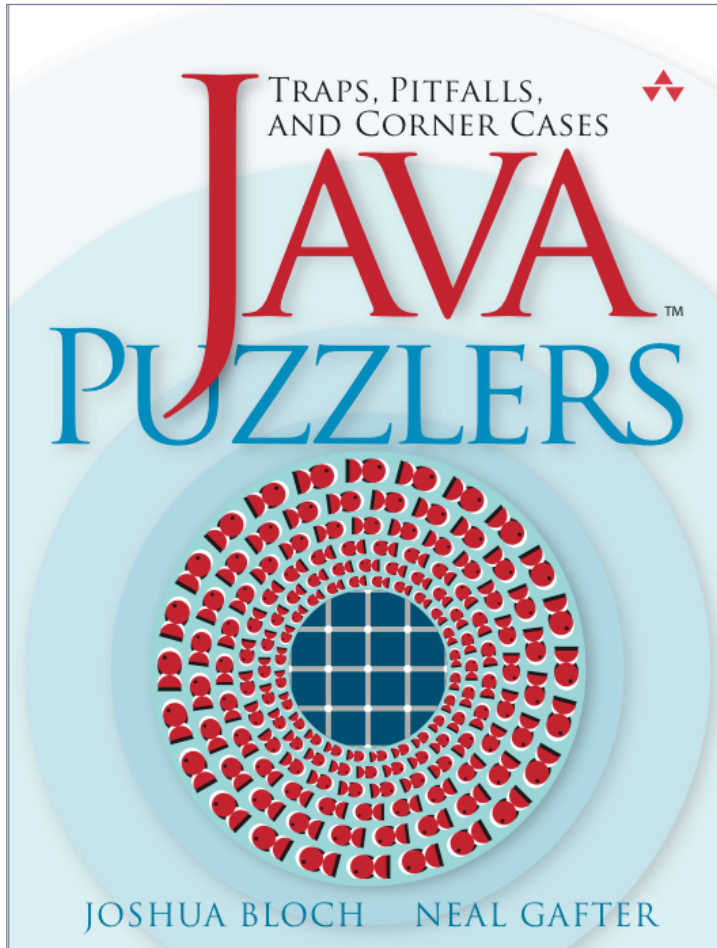
- Make it easy to do the commonly correct thing; possible to do exotic things
- Don't violate the principle of least astonishment
- Don't worsen your API to improve performance (unless you have no choice)

# Conclusion

- Java platform is 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

## Scrapping the Bottom of The Barrel

Compliments, Marriage Proposals, Hot Tips:

Hashtags: `#io2011 #JavaPuzzlers`

Web: `http://goo.gl/sV0bg`

Complaints: `/dev/null`

