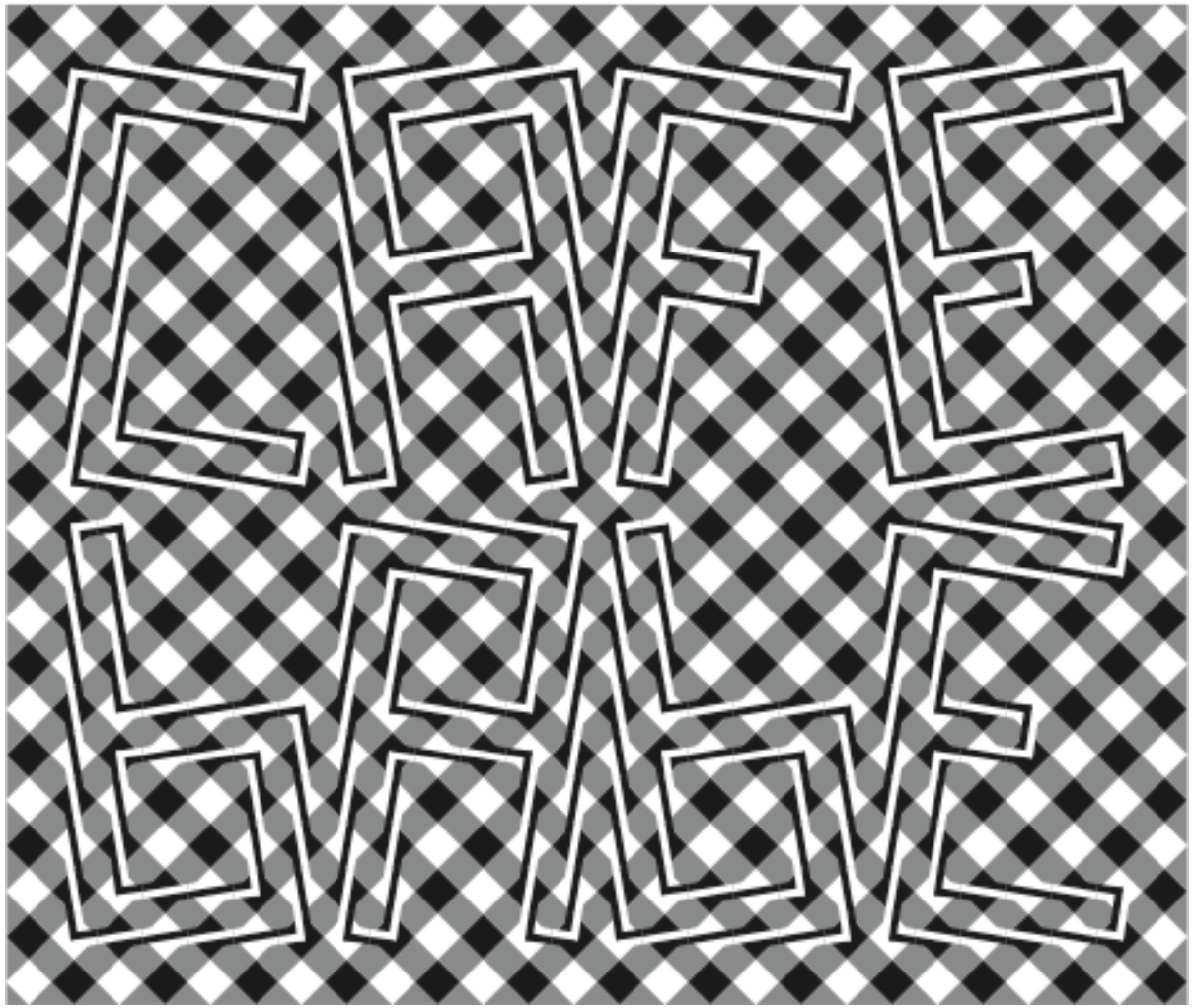

Google™
11
I/O

Java Puzzlers
Scraping the Bottom of The Barrel

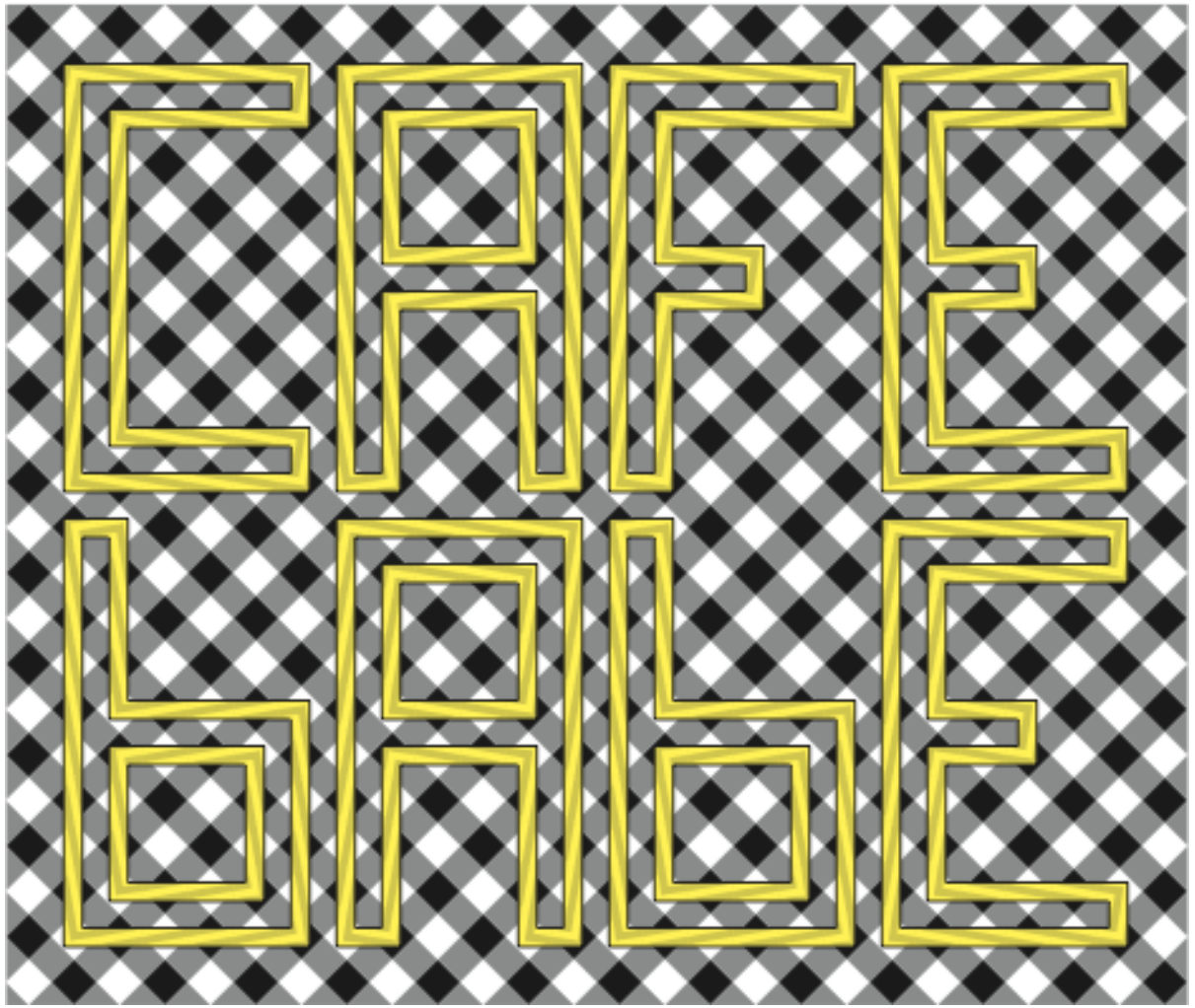
Joshua Bloch and Jeremy Manson
May 10, 2011

Google I/O 11

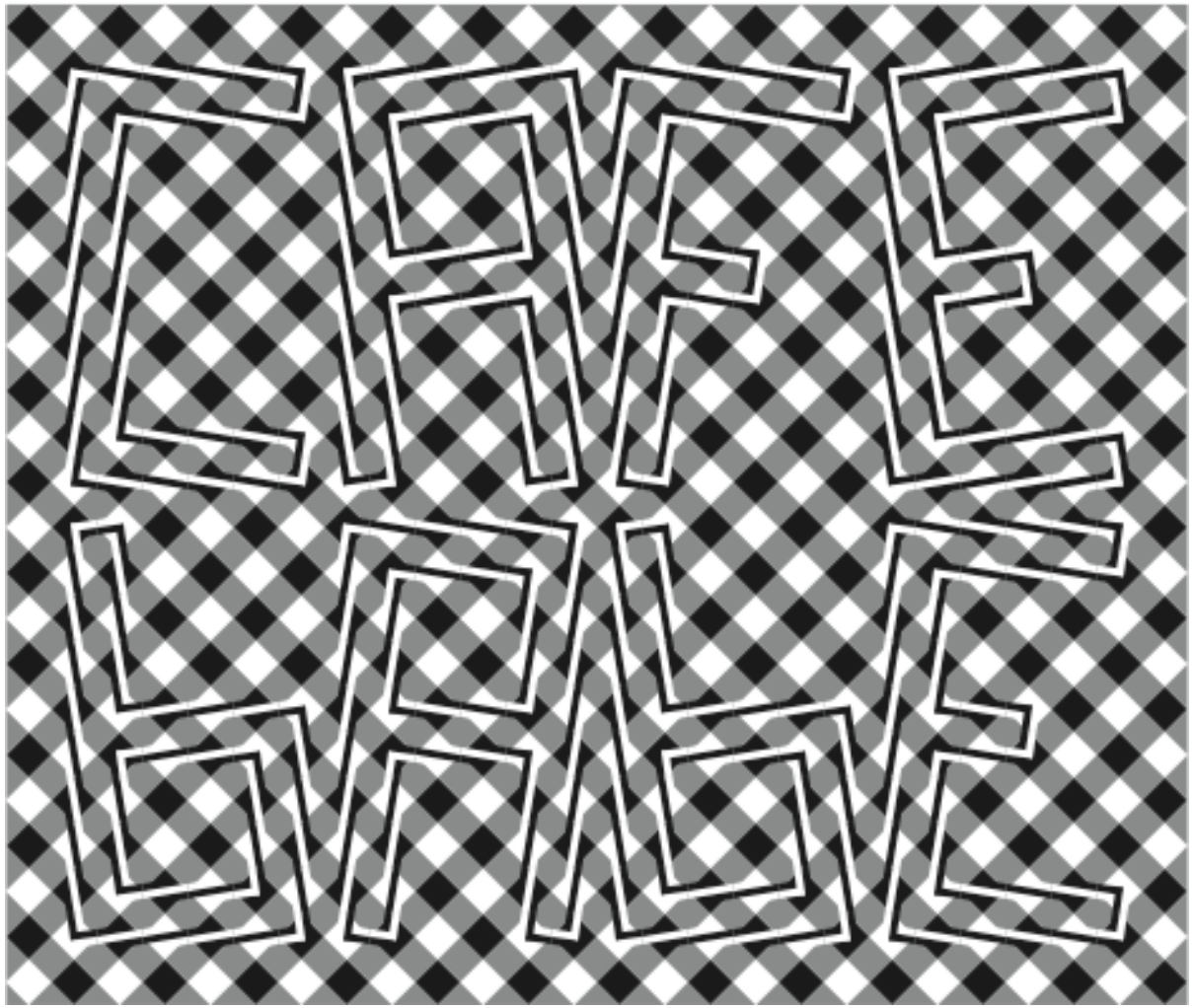
Some people say seeing is believing...



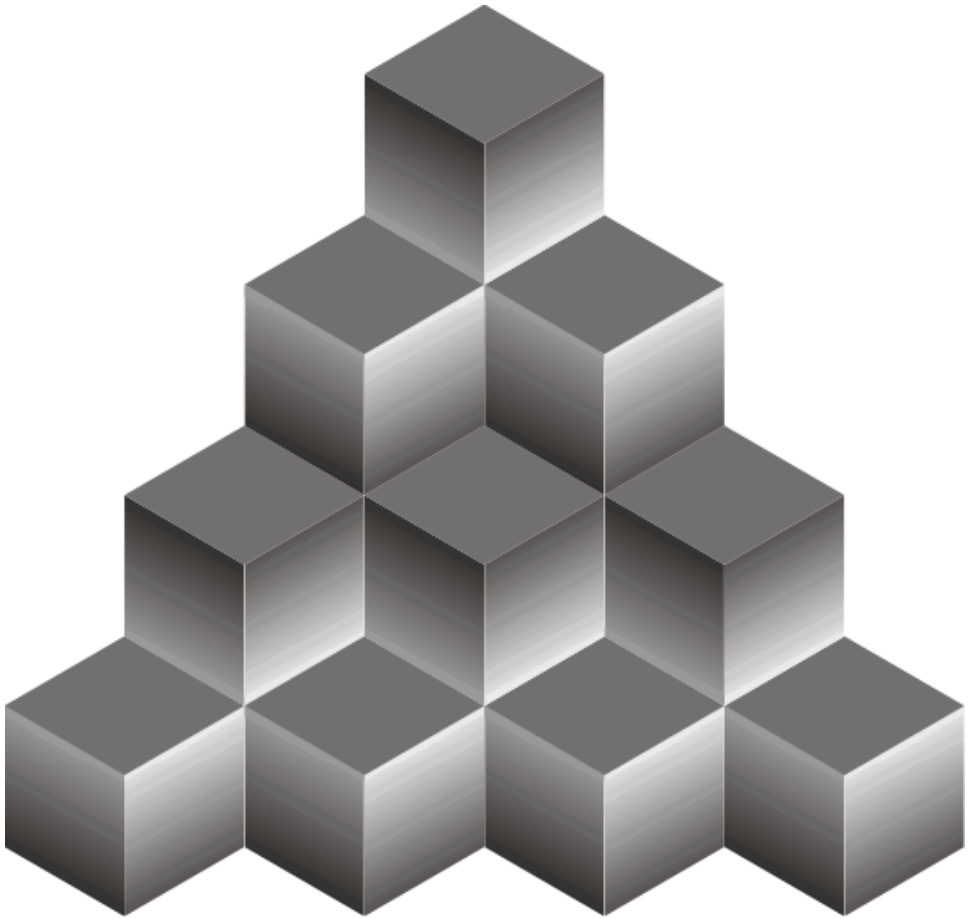
Fraser 1908



Fraser 1908

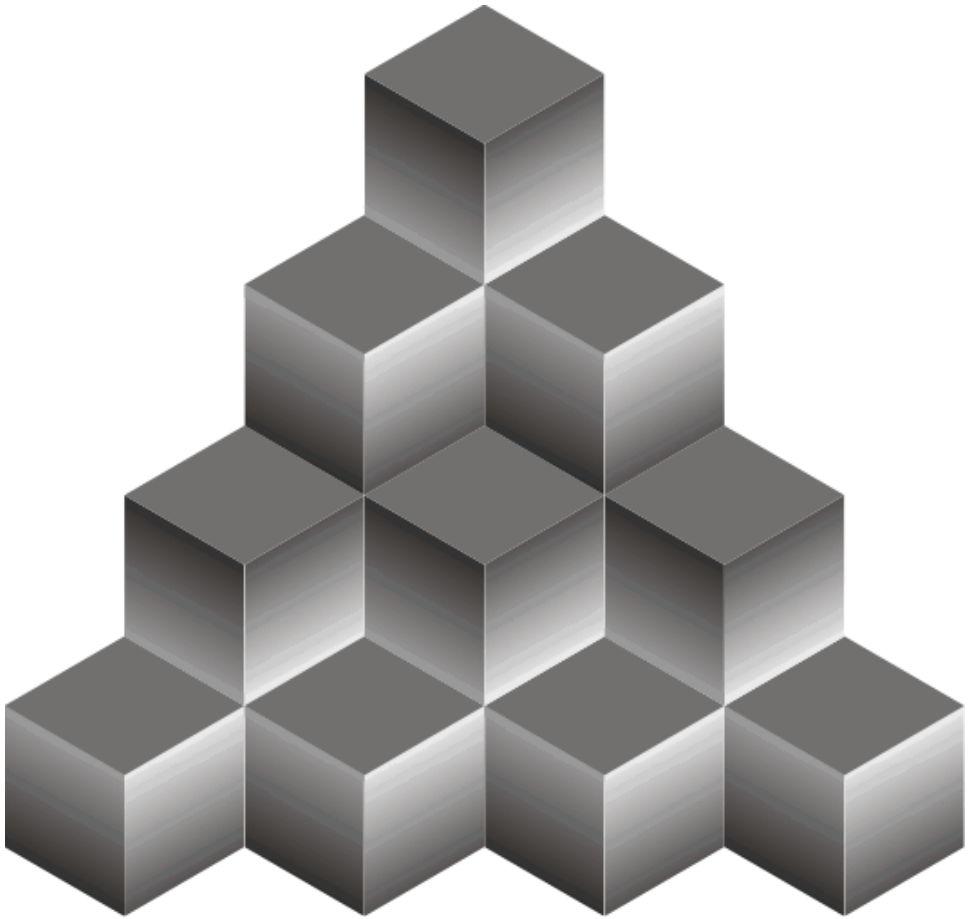


Fraser 1908

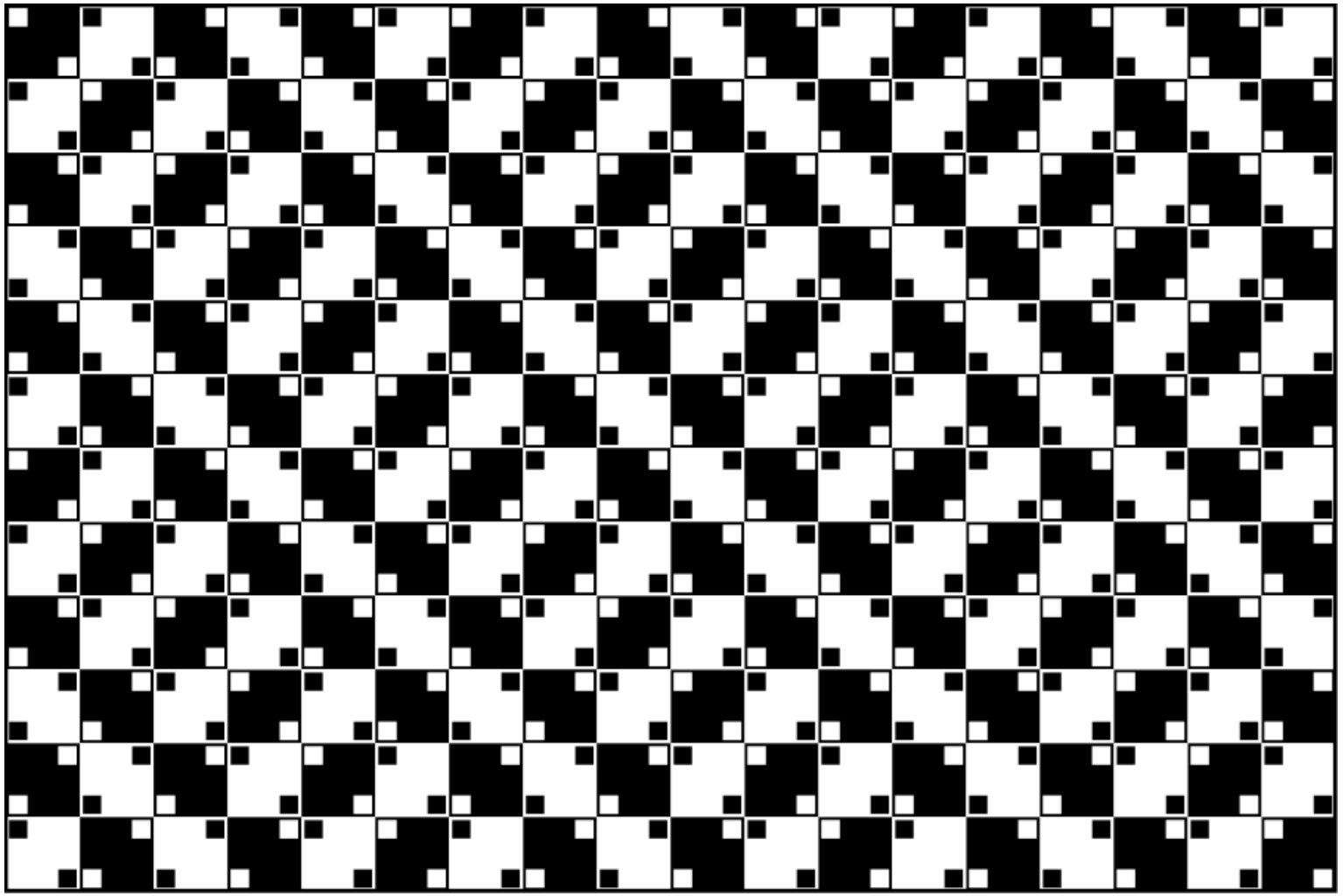


Logvinenko 1999

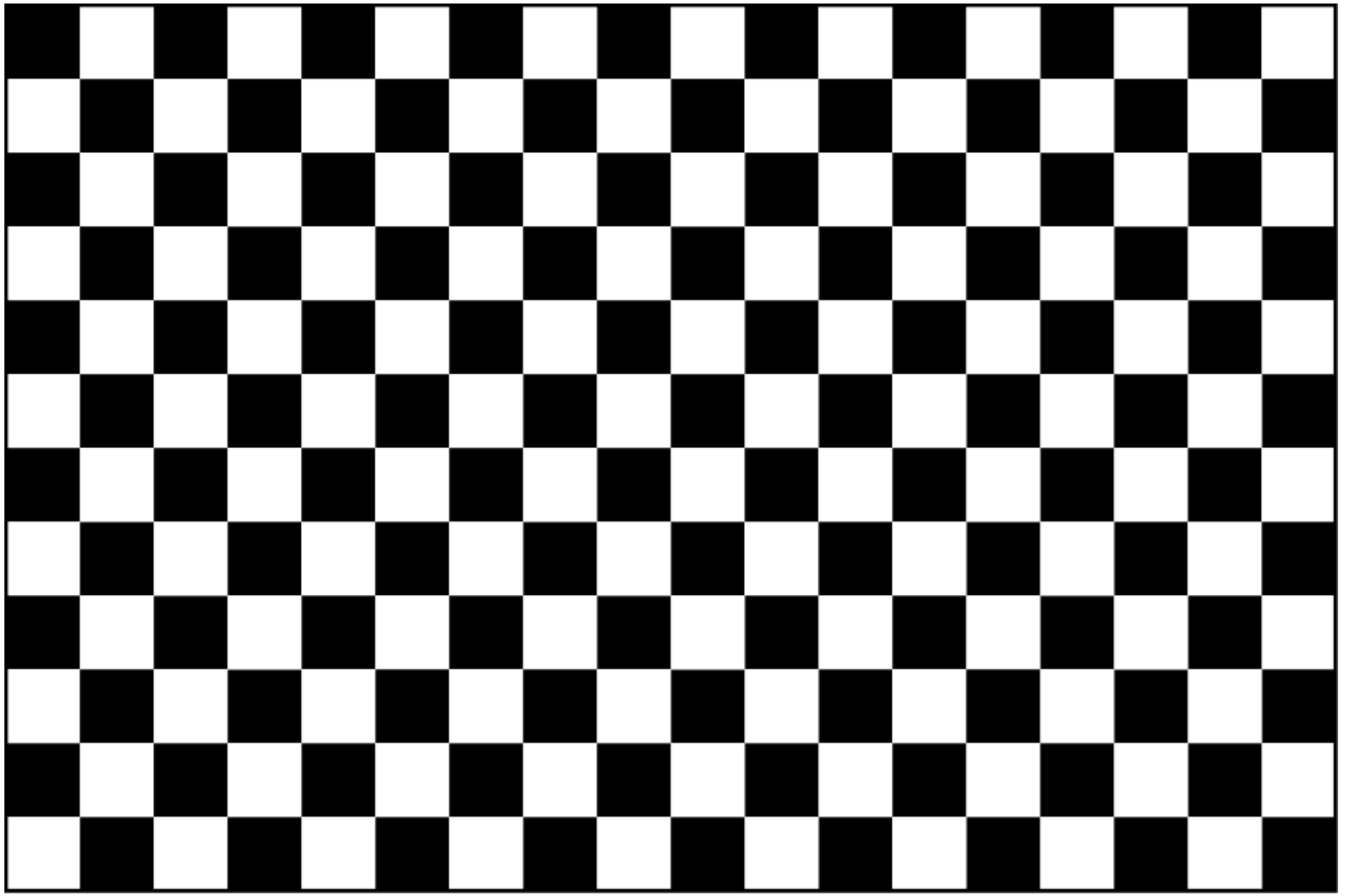
Logvinenko 1999



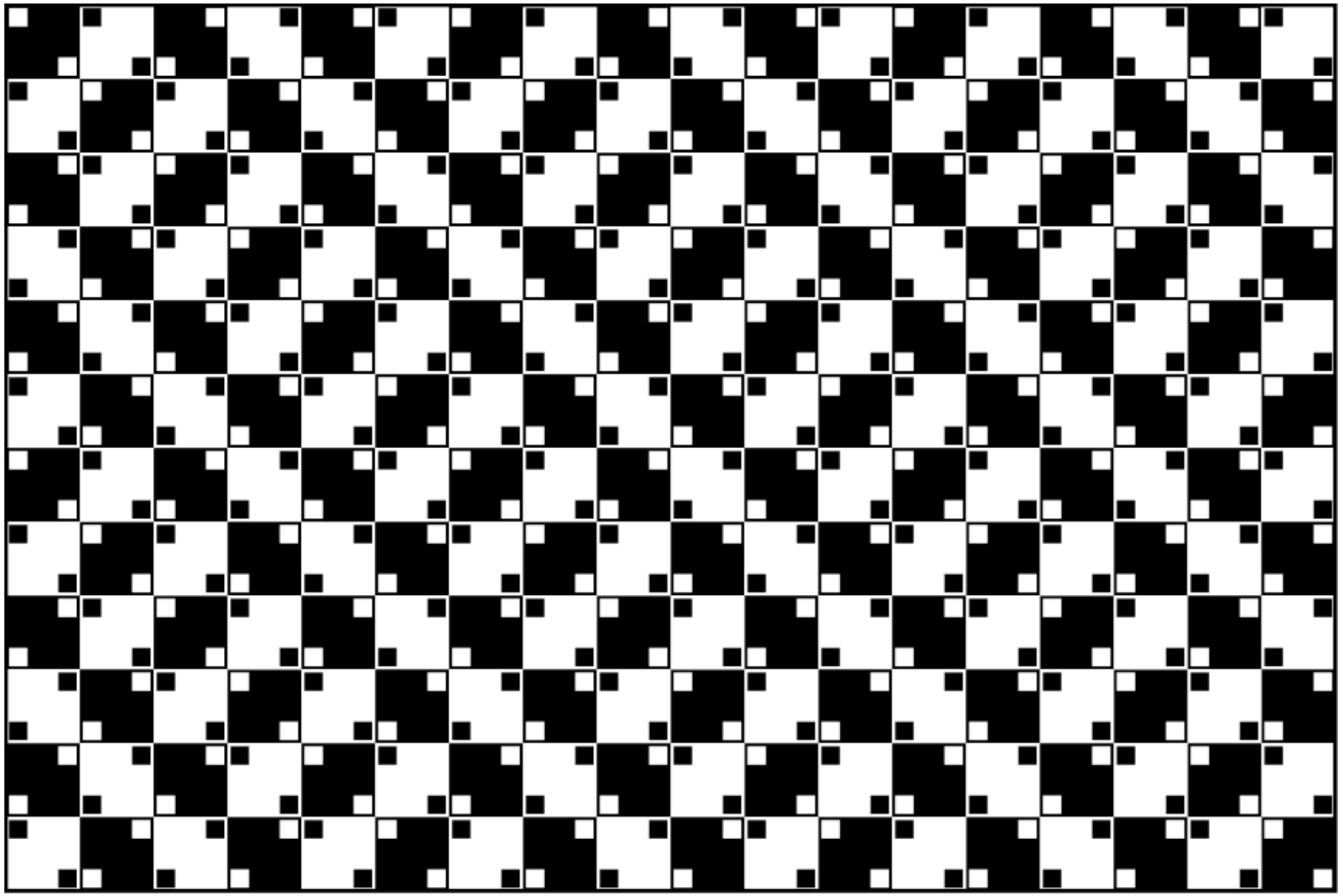
Logvinenko 1999



Kitaoka 1998



Kitaoka 1998



Kitaoka 1998

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) `0.8999999999999999`
(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) `0.8999999999999999`

(d) None of the above
`0.899999999999999911821580299874767661094665527343 75`

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?

(a) 2 1
(b) 2 2
(c) 4 4
(d) None of the above

```
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();
    }
}
```

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`

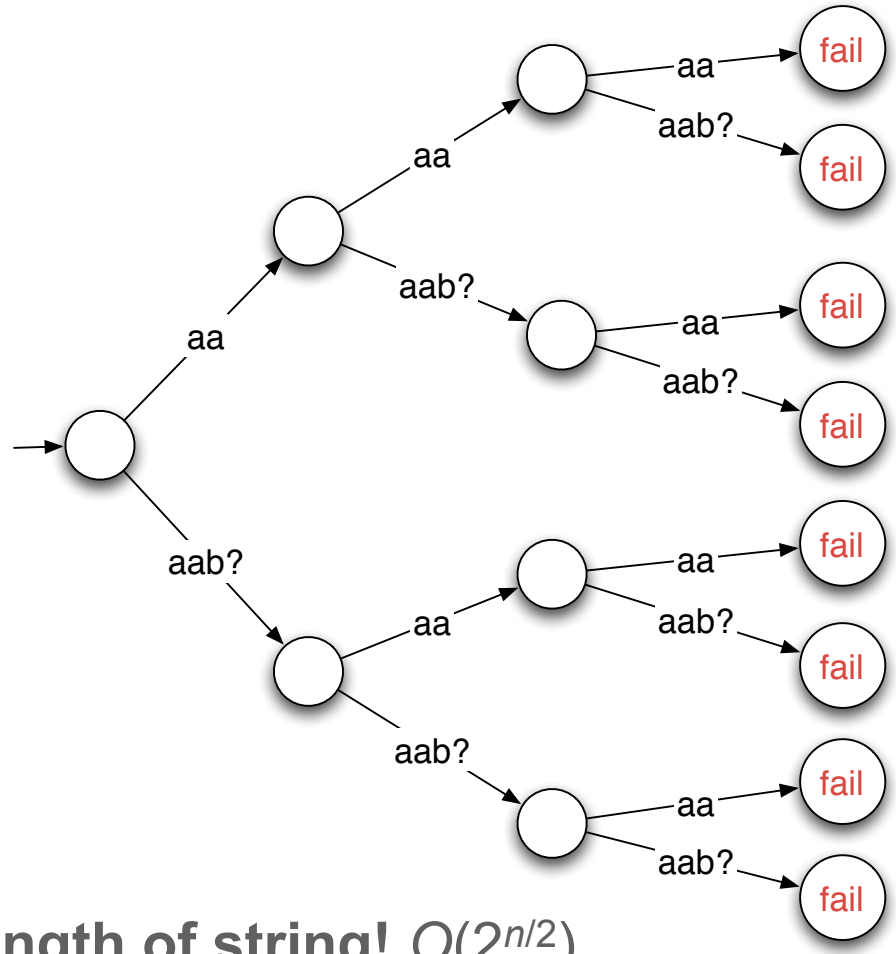
(c) Throws an exception

(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!
    }
}
```

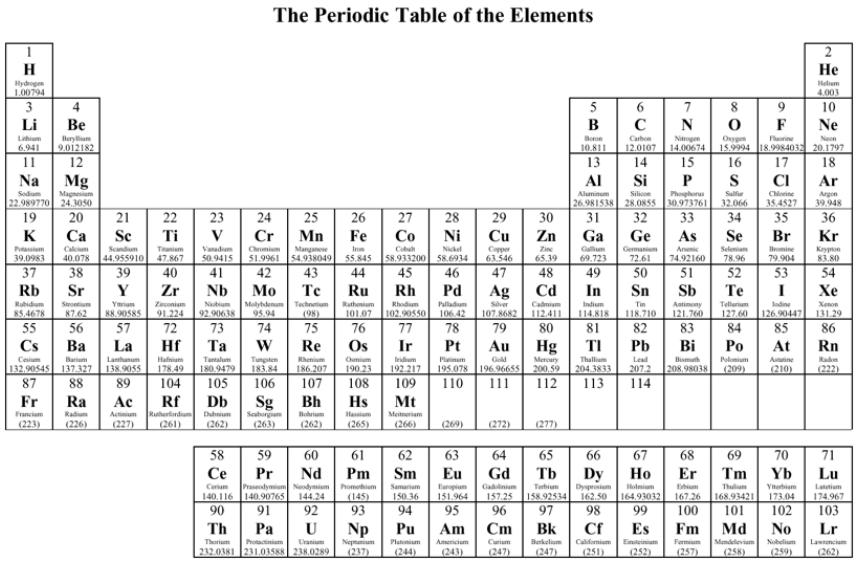
Prints 123

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);
    }
}
```



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 + 54321);
        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**

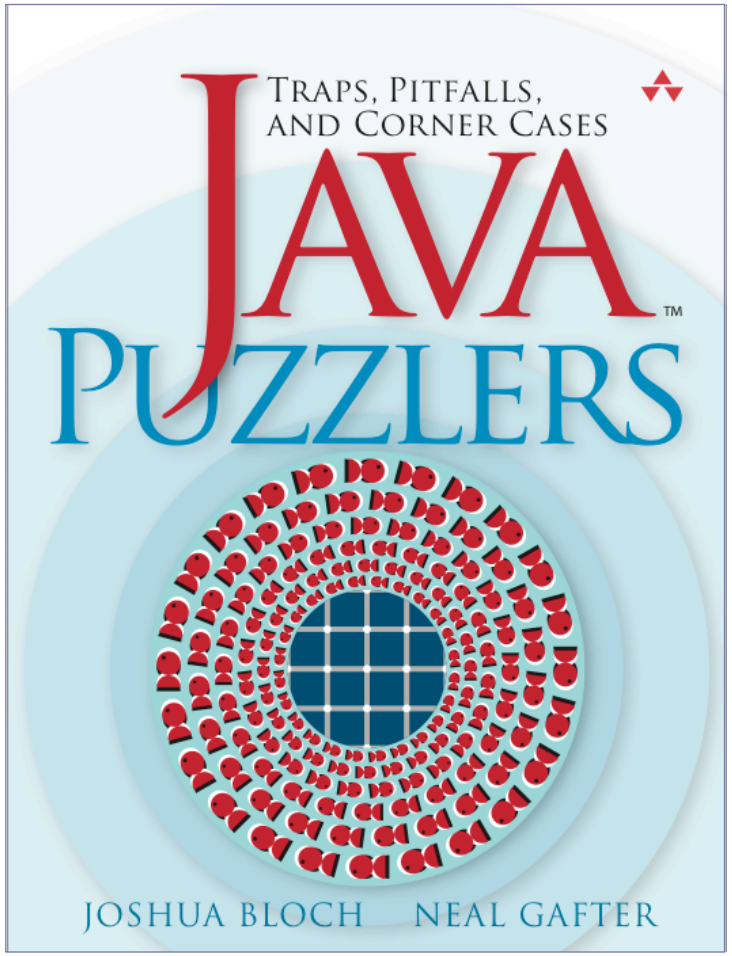
Lessons for API Designers

- 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)

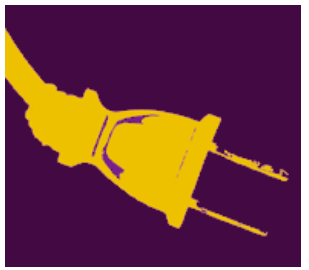
Conclusion

- 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
Scraping the Bottom of The Barrel

Compliments, Marriage Proposals, Hot Tips:

Hashtags: `#io2011 #JavaPuzzlers`
Web: `http://goo.gl/sV0bg`

Complaints: `/dev/null`

Google I/O 11


Google™
11
I/O