

# Principles of Software Construction: Objects, Design, and Concurrency

Part 4: Et cetera

A puzzling finale: What you see is what you get?

**Charlie Garrod**

**Chris Timperley**

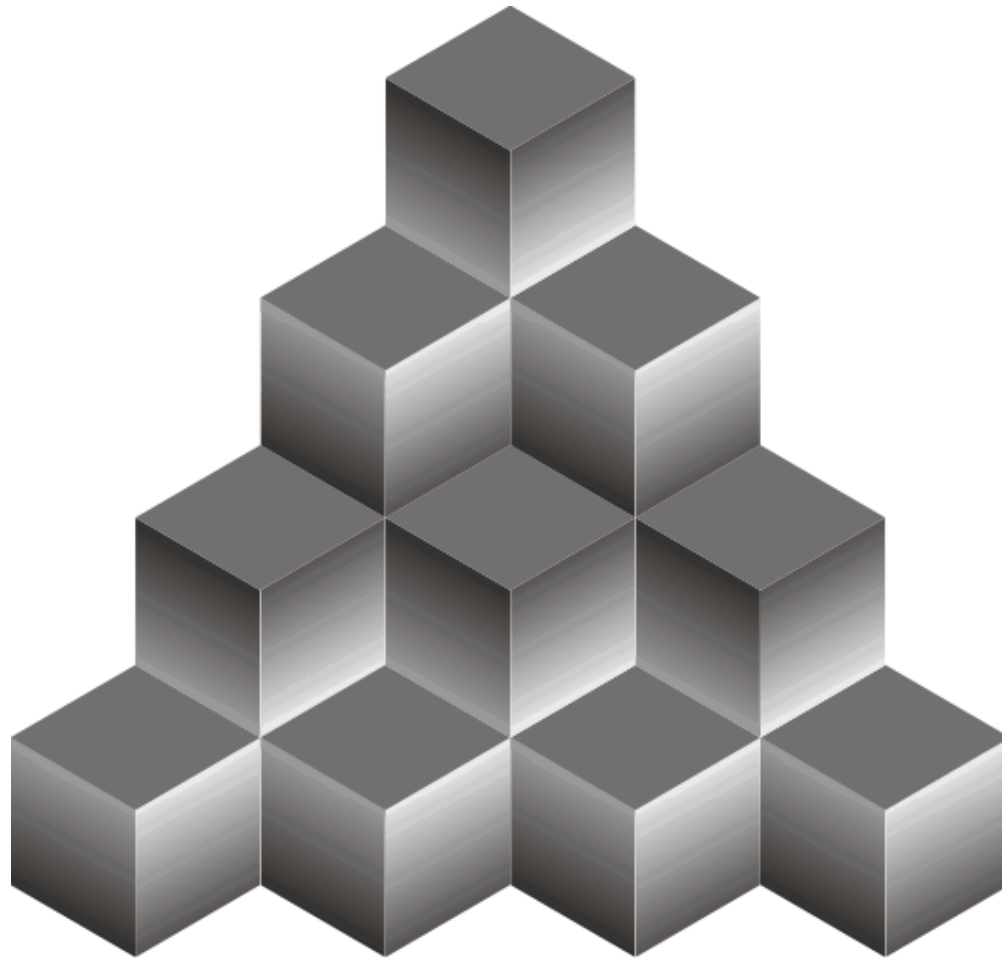


# Administrivia

- Final exam review session:
  - Saturday, Dec. 7<sup>th</sup> noon – 2:00 p.m.: Doherty Hall 1212
- Final exam:
  - Monday, Dec 9<sup>th</sup> 1:00 – 4:00 p.m.: GHC 4401
- Evaluate us: <https://cmu.smartevals.com/>
- Evaluate our TAs:  
<https://www.ugrad.cs.cmu.edu/ta/F19/feedback/>

# Key concepts from Tuesday

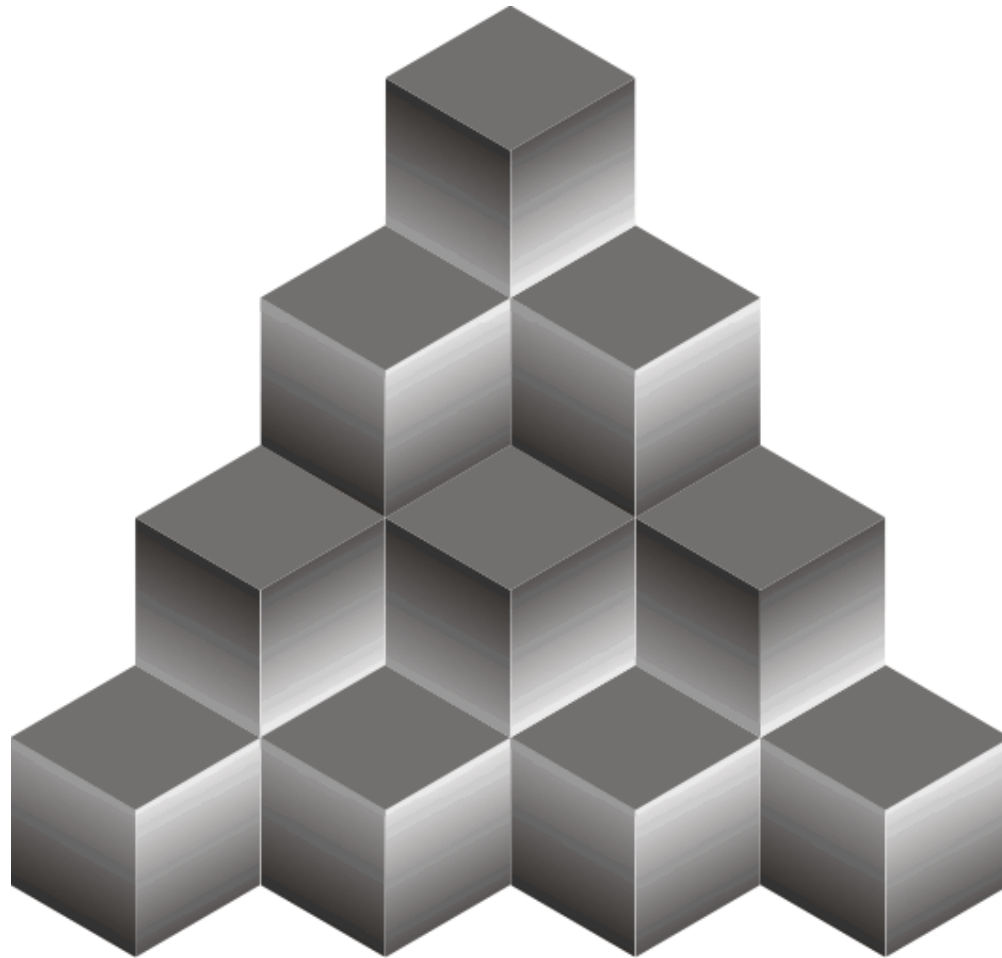
# Today: A finale of puzzlers



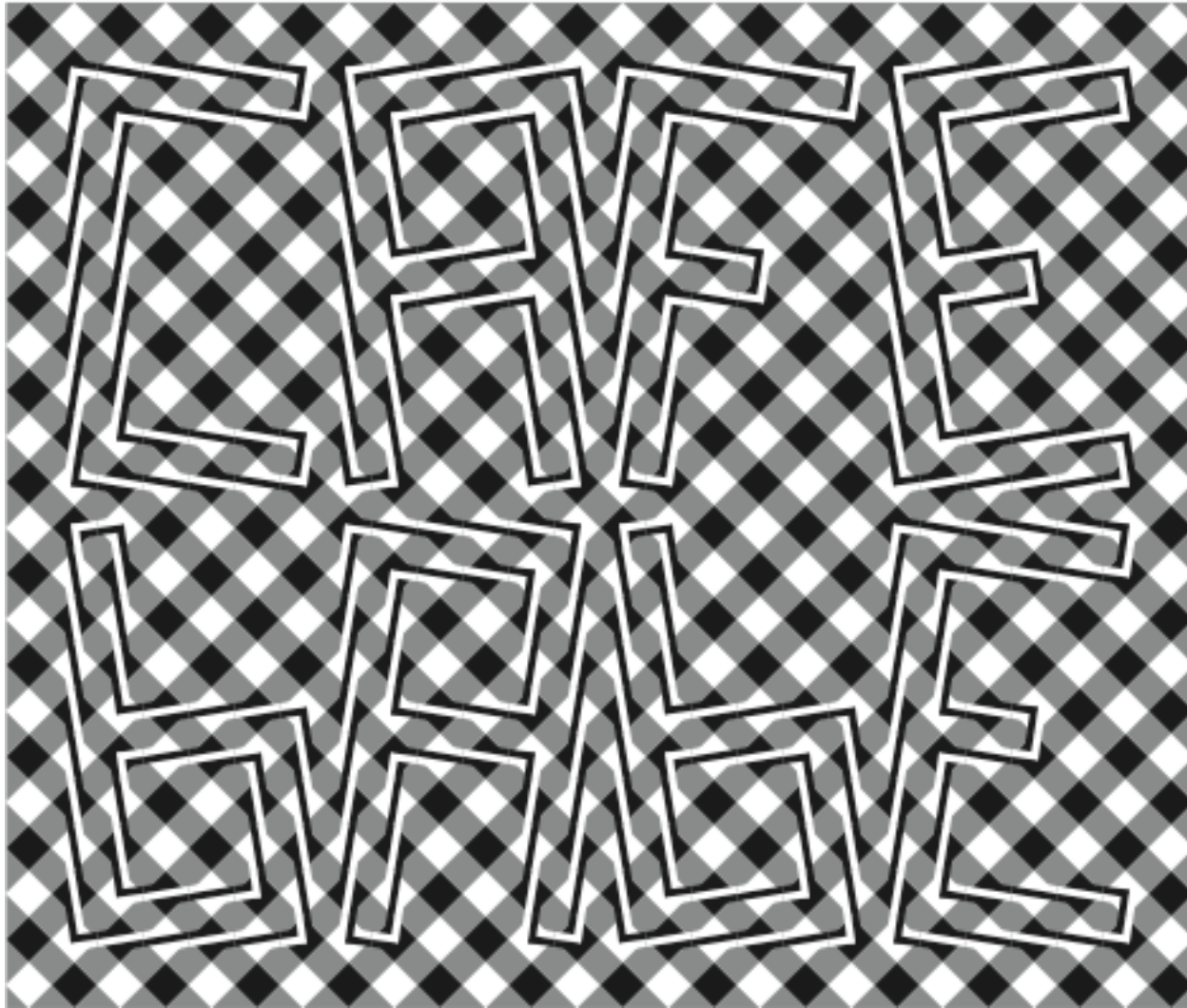
**Logvinenko 1999**



**Logvinenko 1999**

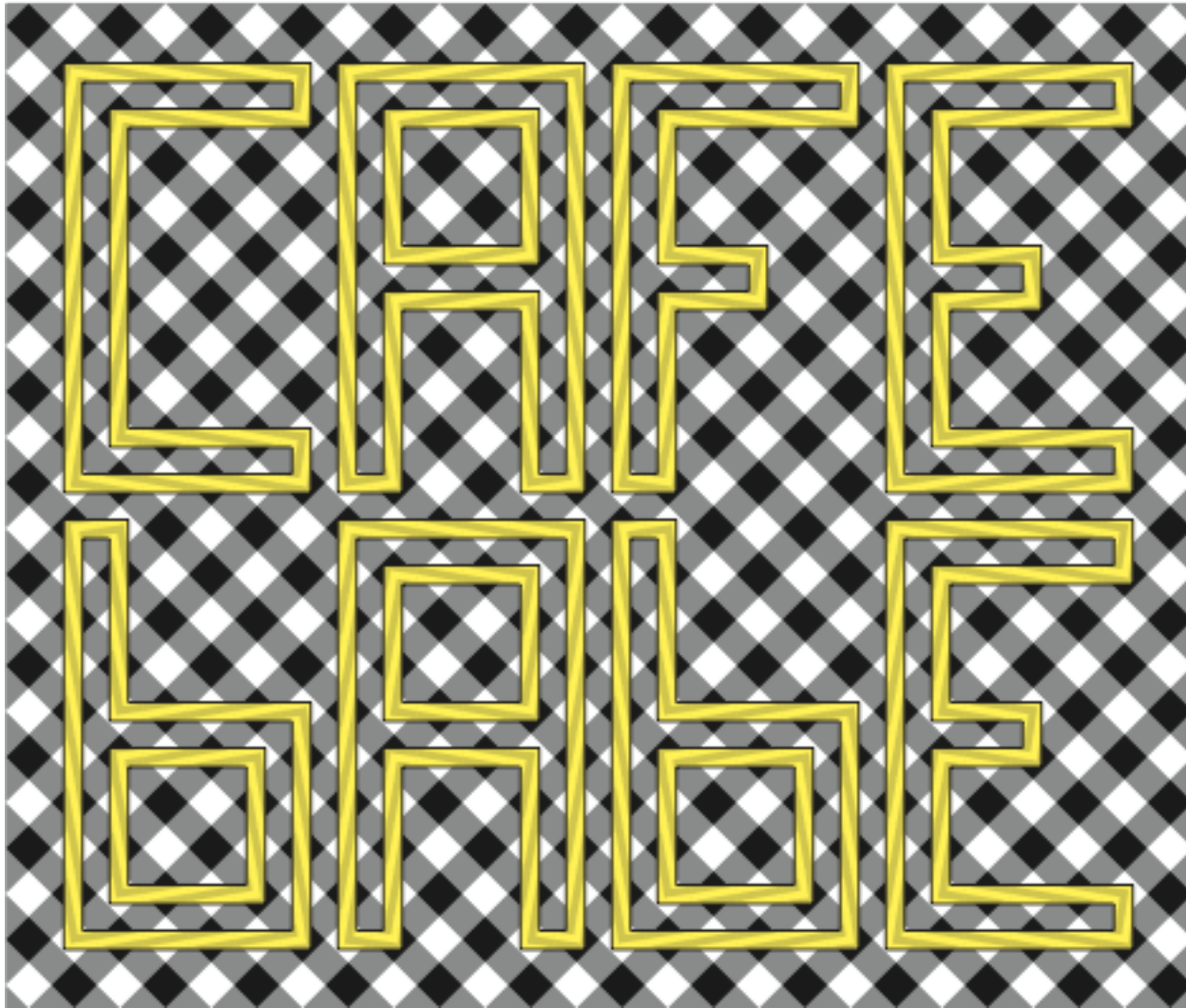


**Logvinenko 1999**

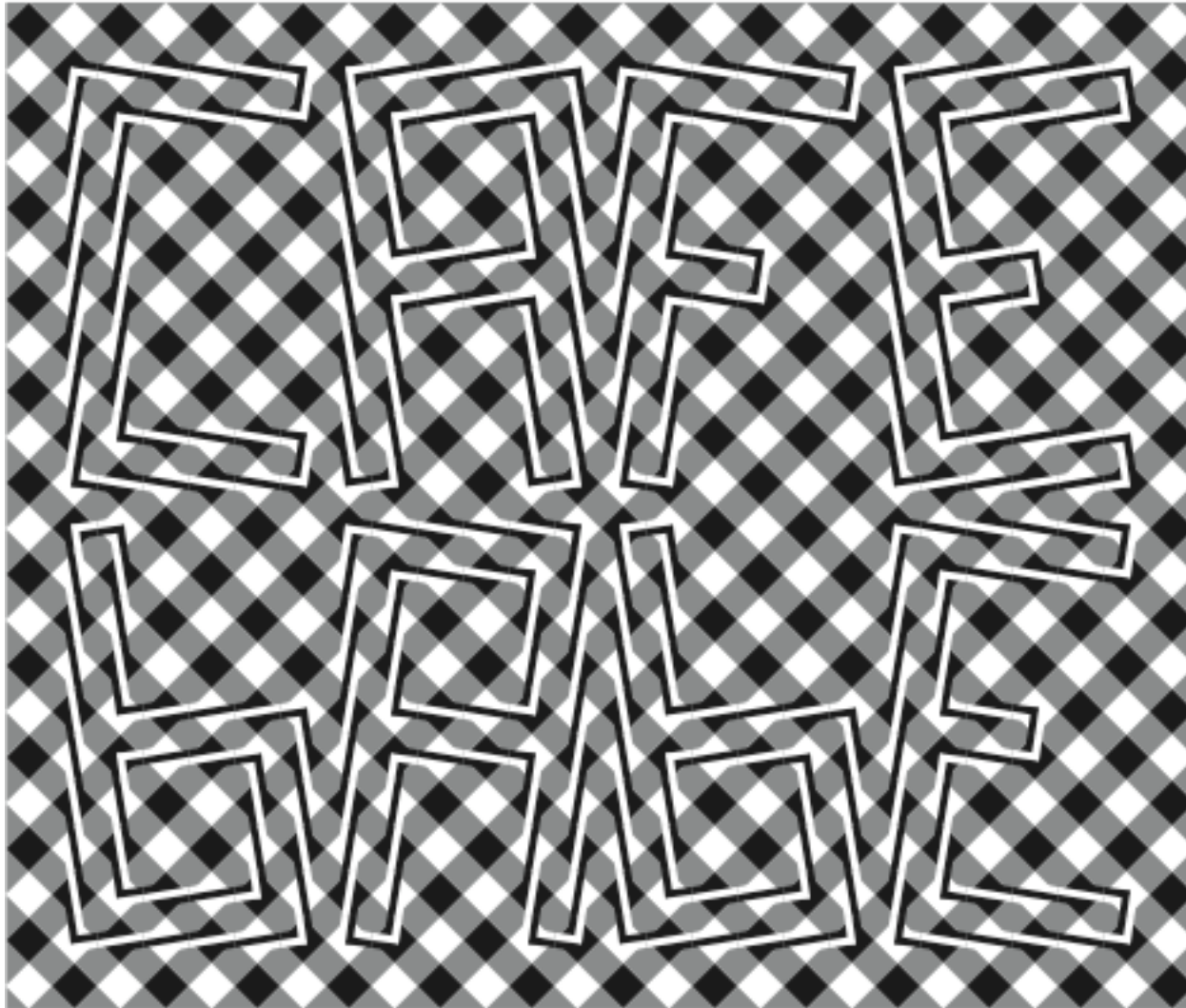


Fraser 1908

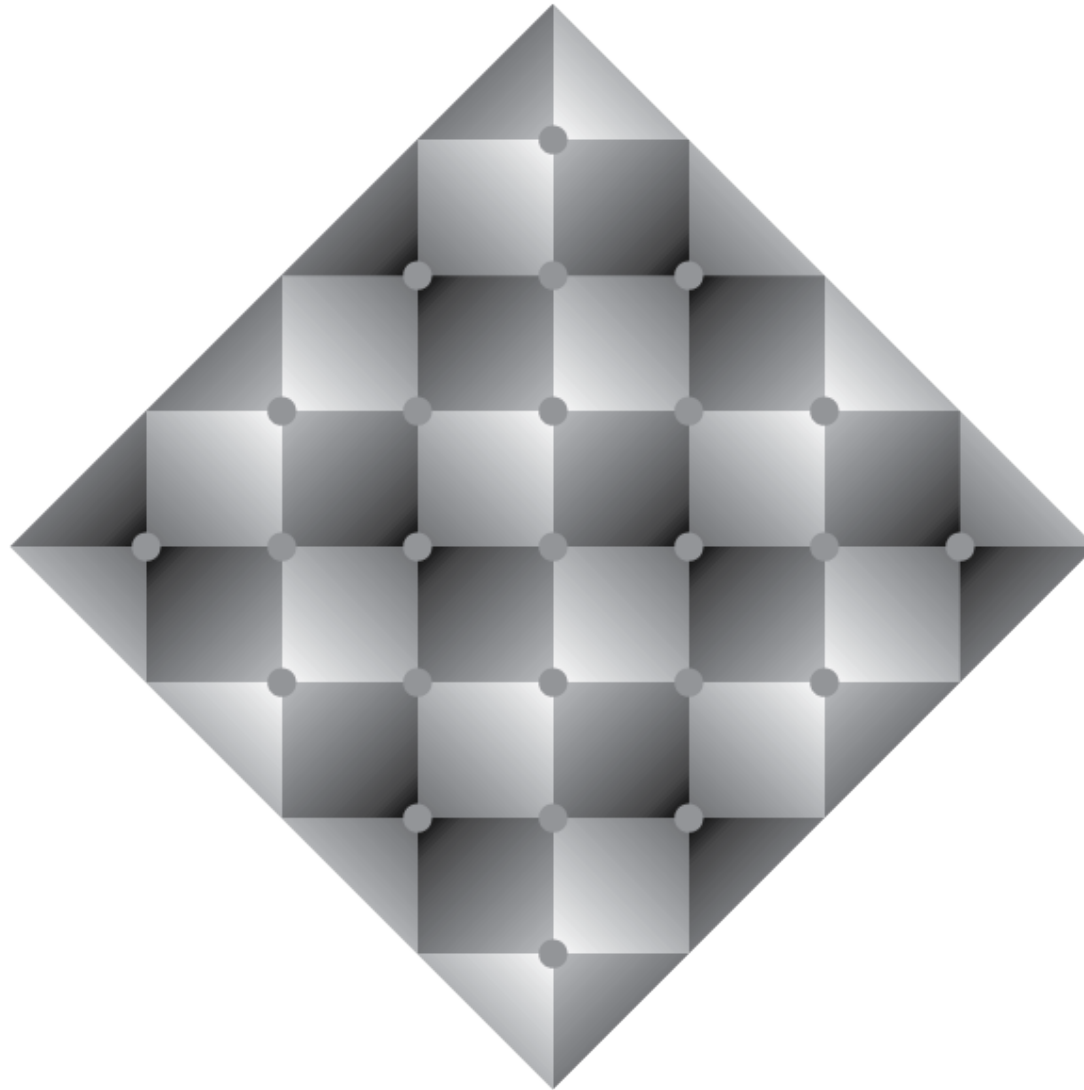




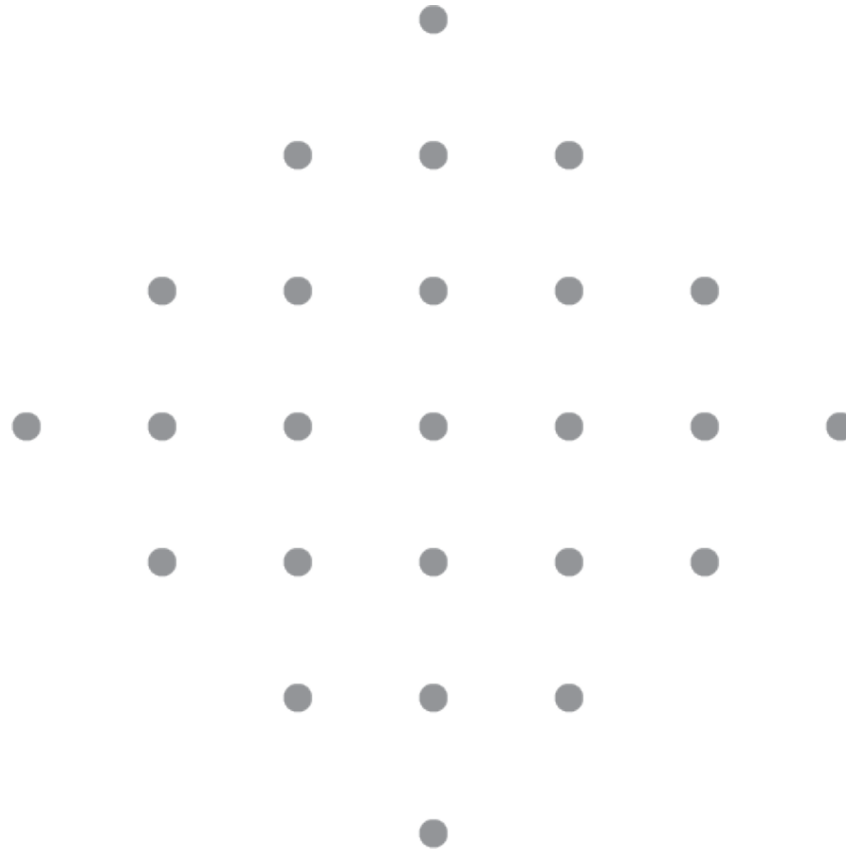
Fraser 1908



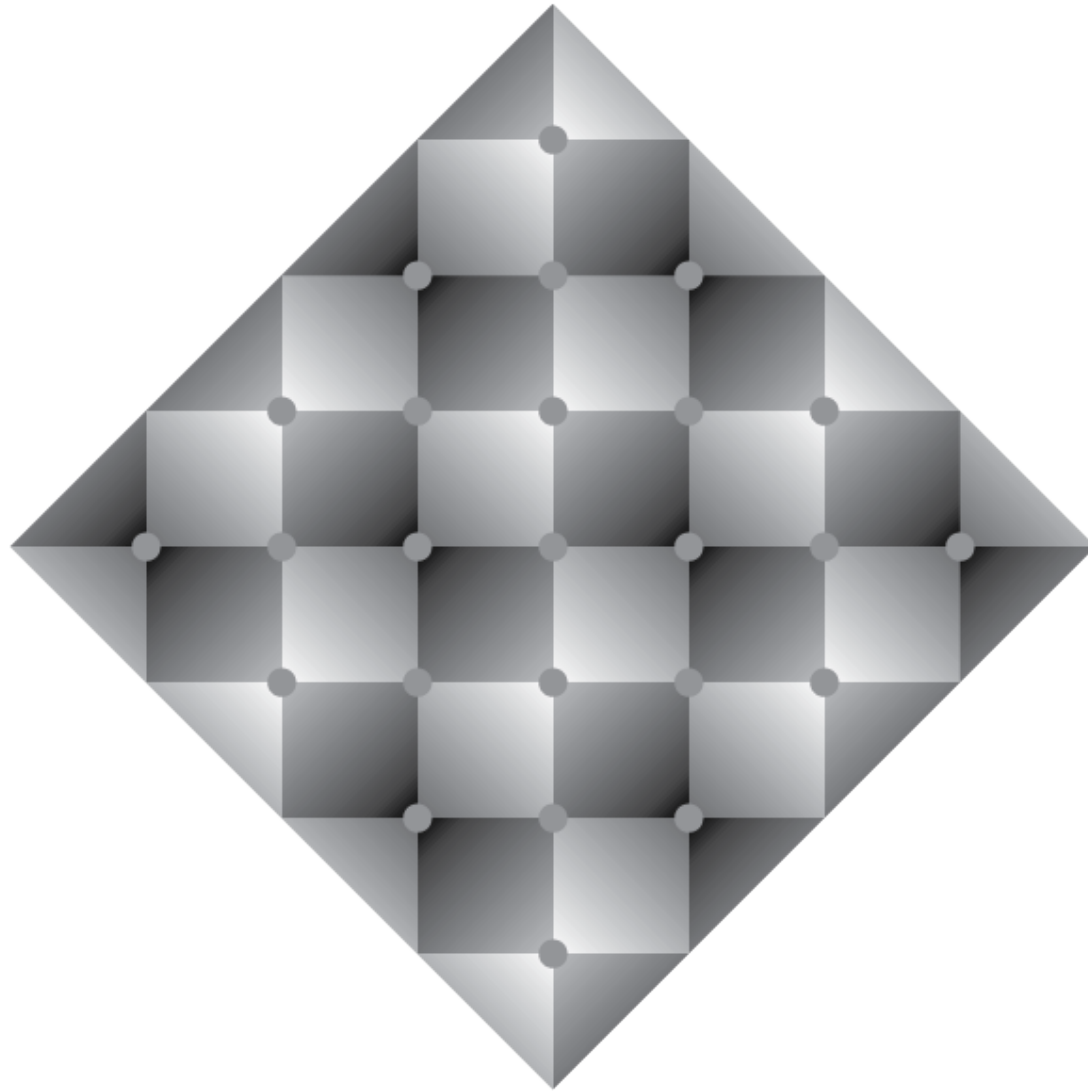
Fraser 1908



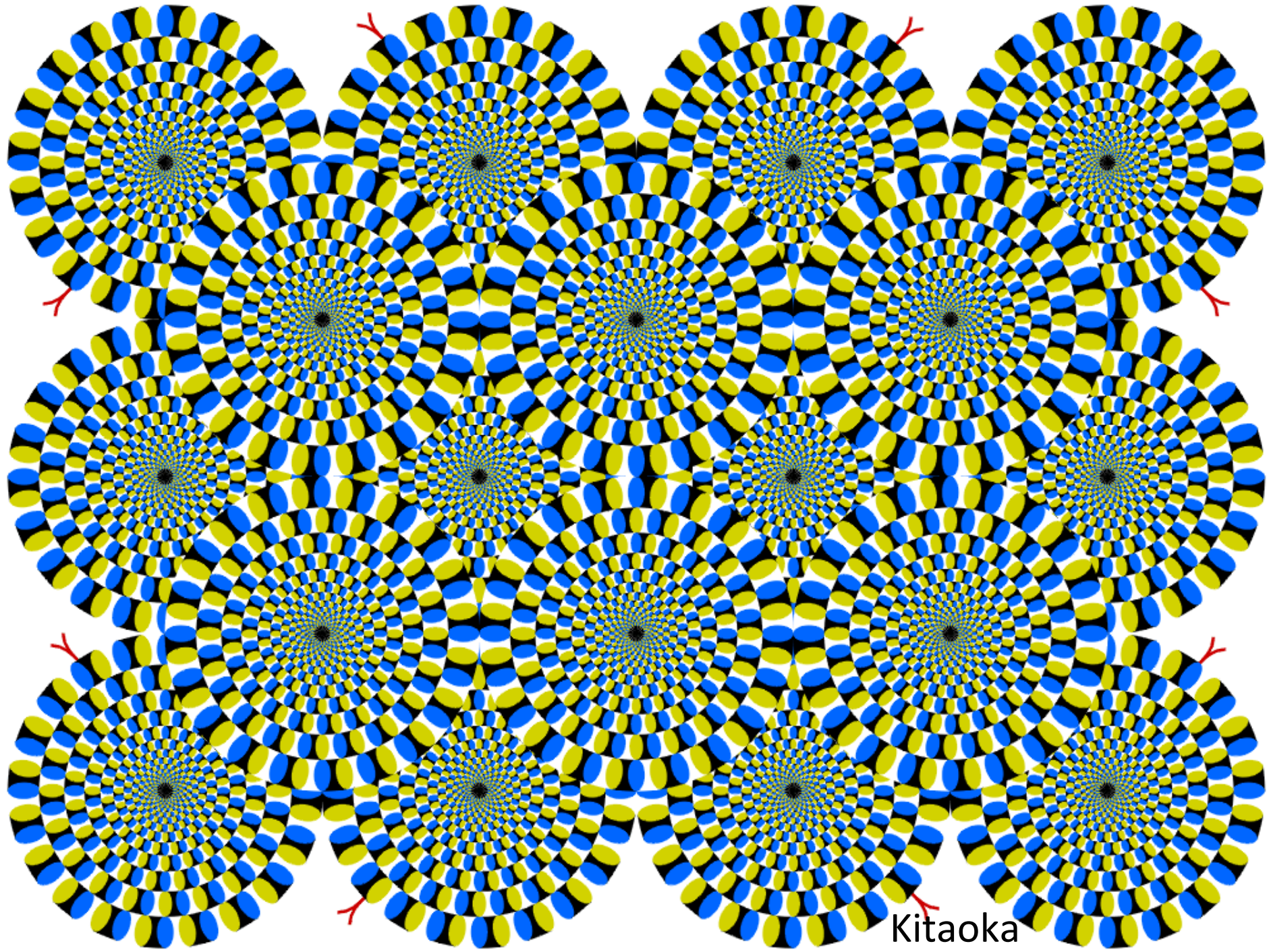
**Todorovic 1997**



**Todorovic 1997**



**Todorovic 1997**



Kitaoka

# A quick challenge: Implement binary search

```
/**
 * Searches the specified array of ints for the specified value
 * using the binary search algorithm. If the array is not sorted,
 * the results are undefined. If the array contains multiple
 * elements with the specified value, there is no guarantee which
 * one will be found.
 *
 * @returns the index of the search key if it is in the array;
 * otherwise ~(insertion point). (Or for you, -1 is fine.)
 */
public static int binarySearch(int[] a, int key);
```

A correct binary search solution?



# A correct binary search solution?

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return ~(low + 1); // key not found.
}
```

# Integer overflows for large values of low and high:

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return ~(low + 1); // key not found.
}
```

# One possible fix

- Avoid overflow, using signed ints:

```
int mid = (low + high) / 2;
```

```
int mid = low + ((high - low) / 2);
```

# Lessons

- Keep it simple
- Use all the tools you know:
  - A good IDE
  - Static analysis tools like SpotBugs
  - Verification tools for critical code
  - Unit tests and regression testing
  - Assert statements for known invariants
  - Code review for all code intended for other developers or users
  - Continuous integration testing for any project with multiple developers

# “A Big Delight in Every Byte”

```
class Delight {  
    public static void main(String[] args) {  
        for (byte b = Byte.MIN_VALUE;  
            b < Byte.MAX_VALUE; b++) {  
            if (b == 0x90)  
                System.out.print("Joy! ");  
        }  
    }  
}
```



# What Does It Print?

```
class Delight {  
    public static void main(String[] args) {  
        for (byte b = Byte.MIN_VALUE;  
            b < Byte.MAX_VALUE; b++) {  
            if (b == 0x90)  
                System.out.print("Joy! ");  
        }  
    }  
}
```

- (a) Joy!**
- (b) Joy! Joy!**
- (c) Nothing**
- (d) None of the above**

# What Does It Print?

- (a) Joy!
- (b) Joy! Joy!
- (c) Nothing
- (d) None of the above

Program compares a `byte` with an `int`;  
`byte` is *promoted* with surprising results

## Another Look

*bytes are signed; range from -128 to 127*

```
class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
            b < Byte.MAX_VALUE; b++) {
            if (b == 0x90) // (b == 144)
                System.out.print("Joy! ");
        }
    }
}
```

```
// (byte)0x90 == -112
// (byte)0x90 != 0x90
```



## You Could Fix it Like This...

- Cast int to byte

```
if (b == (byte)0x90)
    System.out.println("Joy!");
```

- Or convert byte to int, suppressing sign extension with mask

```
if ((b & 0xff) == 0x90)
    System.out.println("Joy!");
```

## ...But This is Even Better

```
public class Delight {  
    private static final byte TARGET = 0x90; // Won't compile!  
    public static void main(String[] args) {  
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)  
            if (b == TARGET)  
                System.out.print("Joy!");  
    }  
}
```

Delight.java:2: possible loss of precision

found : int

required: byte

```
private static final byte TARGET = 0x90; // Won't compile!  
                                ^
```

# The Best Solution, Debugged

```
public class Delight {  
    private static final byte TARGET = (byte) 0x90; // Fixed  
    public static void main(String[] args) {  
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)  
            if (b == TARGET)  
                System.out.print("Joy!");  
    }  
}
```

# The Moral

- **byte values are signed** ☹️
- Be careful when mixing primitive types
- **Compare like-typed expressions**
  - Cast or convert one operand as necessary
  - Declared constants help keep you in line
- For language designers
  - Don't violate principle of least astonishment
  - Don't make programmers' lives miserable

# “Strange Saga of a Sordid Sort”

```
public class SordidSort {  
    static final Integer BIG    = 2_000_000_000;  
    static final Integer SMALL = -2_000_000_000;  
    static final Integer ZERO  = 0;  
  
    public static void main(String args[]) {  
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};  
        Arrays.sort(arr, (i1, i2) -> i1 - i2);  
        System.out.println(Arrays.toString(arr));  
    }  
}
```



# What does it print?

```
public class SordidSort {
    static final Integer BIG    = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) -> i1 - i2);
        System.out.println(Arrays.toString(arr));
    }
}
```

- (a) [-2000000000, 0, 2000000000]
- (b) [2000000000, 0, -2000000000]
- (c) [-2000000000, 2000000000, 0]
- (d) None of the above

What does it print?

(a) [-2000000000, 0, 2000000000]

(b) [2000000000, 0, -2000000000]

(c) [-2000000000, 2000000000, 0]

(d) None of the above: Unspecified;

In practice, [2000000000, -2000000000, 0]

Comparator is broken!

It relies on `int` subtraction

`int` too small to hold difference of 2 arbitrary `ints`

# Another Look

```
public class SordidSort {
    static final Integer BIG    = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) -> i1 - i2);
        System.out.println(Arrays.toString(arr));
    }
}
```

Subtraction overflows.



# A possible fix?

```
public class SordidSort {
    static final Integer BIG    = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) ->
            i1 < i2 ? -1 : (i1 == i2 ? 0 : 1));
        System.out.println(Arrays.toString(arr));
    }
}
```

## ...Another bug!

```
public class SordidSort {
    static final Integer BIG    = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) ->
            i1 < i2 ? -1 : (i1 == i2 ? 0 : 1));
        System.out.println(Arrays.toString(arr));
    }
}
```

Unspecified behavior

`==` checks for identity, not equality, of object references!

## You could fix it like this...

```
public class SordidSort {
    static final Integer BIG    = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) ->
            i1 < i2 ? -1 : (i1 > i2 ? 1 : 0));
        System.out.println(Arrays.toString(arr));
    }
}
```

Prints [-2000000000, 0, 2000000000]

Works, but fragile!

## ...But this is better

```
public class SordidSort {
    static final Integer BIG    = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, Integer::compareTo);
        System.out.println(Arrays.toString(arr));
    }
}
```

Prints [-2000000000, 0, 2000000000]

# Moral (1 of 2)

- ints aren't integers
  - Think about overflow
- The comparison technique  $(i1, i2) \rightarrow i1 - i2$  requires  $|i1 - i2| \leq \text{Integer.MAX\_VALUE}$ 
  - For example: all values non-negative
- Don't write overly clever code
- Use standard idioms
  - But beware; some idioms are broken

## Moral (2 of 2)

- `ints` aren't Integers
  - Think about identity vs. equality
  - Think about null
- For language designers
  - Don't violate the principle of least astonishment
  - Don't insist on backward compatibility

# “Indecision”

```
class Indecisive {  
    public static void main(String[] args) {  
        System.out.println(decision());  
    }  
  
    static boolean decision() {  
        try {  
            return true;  
        } finally {  
            return false;  
        }  
    }  
}
```



What does it print?

(a) true

(b) false

(c) None of the above



# What does it print?

- (a) true
- (b) false
- (c) None of the above
- (d) Who cares?!?

# What does it print?

(a) true

(b) false

(c) None of the above

The finally is processed after the try.

# Another look

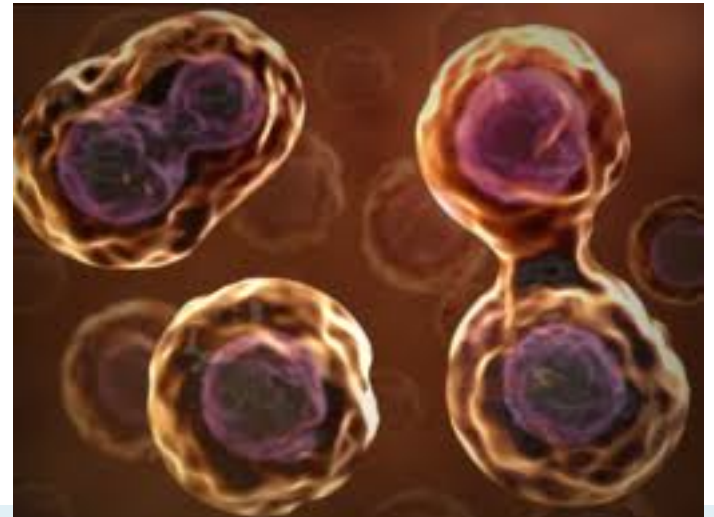
```
class Indecisive {  
    public static void main(String[] args) {  
        System.out.println(decision());  
    }  
  
    static boolean decision() {  
        try {  
            return true;  
        } finally {  
            return false;  
        }  
    }  
}
```

# The moral

- Don't rely on obscure language or library details
- Here: Avoid abrupt completion of `finally` blocks
  - Don't return or throw exception from `finally`
  - Wrap unpredictable actions with nested `try`

# “Long Division” (2004)

```
public class LongDivision {  
    private static final long MILLIS_PER_DAY  
        = 24 * 60 * 60 * 1000;  
    private static final long MICROS_PER_DAY  
        = 24 * 60 * 60 * 1000 * 1000;  
  
    public static void main(String[] args) {  
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);  
    }  
}
```



# What does it print?

```
public class LongDivision {  
    private static final long MILLIS_PER_DAY  
        = 24 * 60 * 60 * 1000;  
    private static final long MICROS_PER_DAY  
        = 24 * 60 * 60 * 1000 * 1000;  
  
    public static void main(String[] args) {  
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);  
    }  
}
```

- (a) 5
- (b) 1000
- (c) 5000
- (d) Throws an exception**

What does it print?

(a) 5

(b) 1000

(c) 5000

(d) Throws an exception

Computation overflows

# Another look

```
public class LongDivision {  
    private static final long MILLIS_PER_DAY  
        = 24 * 60 * 60 * 1000;  
    private static final long MICROS_PER_DAY  
        = 24 * 60 * 60 * 1000 * 1000; // >> Integer.MAX_VALUE  
  
    public static void main(String[] args) {  
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);  
    }  
}
```



# How do you fix it?

```
public class LongDivision {  
    private static final long MILLIS_PER_DAY  
        = 24L * 60 * 60 * 1000;  
    private static final long MICROS_PER_DAY  
        = 24L * 60 * 60 * 1000 * 1000;  
  
    public static void main(String[] args) {  
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);  
    }  
}
```

**Prints 1000**

# The moral

- When working with large numbers, watch out for overflow—it's a silent killer
- Just because variable can hold result doesn't mean computation won't overflow
- When in doubt, use **larger type**

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

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

The Periodic Table of the Elements

1 <b>H</b> Hydrogen 1.00794																	2 <b>He</b> Helium 4.003
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 14.00674	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 106.42	46 <b>Pd</b> Palladium 107.8682	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>Hf</b> Hafnium 178.49	59 <b>Ta</b> Tantalum 180.9479	60 <b>W</b> Tungsten 183.84	61 <b>Re</b> Rhenium 186.207	62 <b>Os</b> Osmium 190.23	63 <b>Ir</b> Iridium 192.217	64 <b>Pt</b> Platinum 195.078	65 <b>Au</b> Gold 196.96655	66 <b>Hg</b> Mercury 200.59	67 <b>Tl</b> Thallium 204.3833	68 <b>Pb</b> Lead 208.3833	69 <b>Bi</b> Bismuth (209)	70 <b>Po</b> Polonium (209)	71 <b>At</b> Astatine (210)	72 <b>Rn</b> Radon (222)
87 <b>Fr</b> Francium (223)	88 <b>Ra</b> Radium (226)	89 <b>Ac</b> Actinium (227)	104 <b>Rf</b> Rutherfordium (261)	105 <b>Db</b> Dubnium (263)	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>(269)</b>	111 <b>(272)</b>	112 <b>(277)</b>	113 <b>(285)</b>	114 <b>(284)</b>				
58 <b>Ce</b> Cerium 140.116	59 <b>Pr</b> Praseodymium 140.90765	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				
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)				

# What does it print?

```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.println(12345 + 54321);  
        System.out.println(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.println(12345 + 5432l);  
        System.out.println(01234 + 43210);  
    }  
}
```

**1** - the numeral one

**l** - the lowercase letter el

## Another look, continued

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

`01234` is an octal literal equal to  $1,234_8$ , which is 668

# How do you fix it?

```
public class Elementary {  
    public static void main(String[] args) {  
        System.out.println(12345 + 54321);  
        System.out.println(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 (base 8)
  - And add a comment if this is your intent

# Lessons (repeated)

- Keep it simple
- Use all the tools you know:
  - A good IDE
  - Static analysis tools like SpotBugs
  - Verification tools for critical code
  - Unit tests
  - Assert statements for known invariants
  - Code review for all code intended for other developers or users
  - Continuous integration testing for any project with multiple developers