

Principles of Software Construction: Objects, Design, and Concurrency

Part 4: Et cetera

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

Josh Bloch

Charlie Garrod



Administrivia

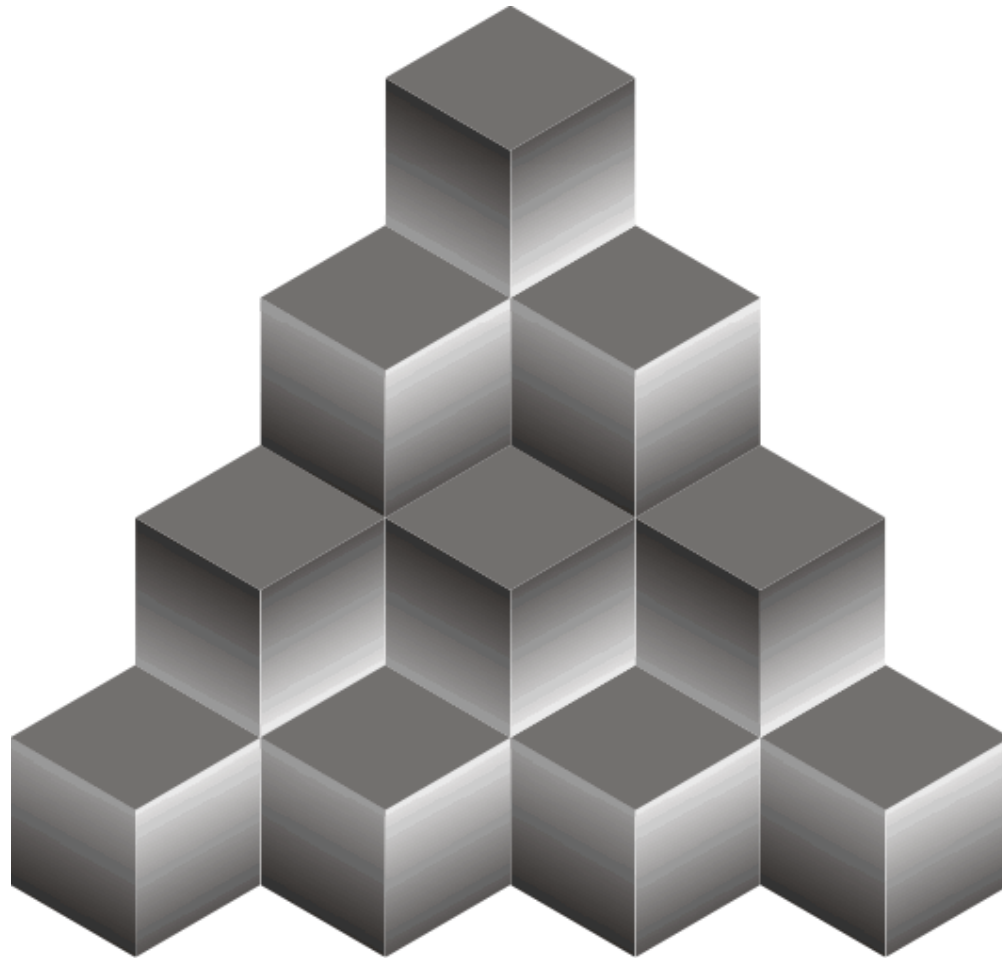
- Homework 6 due last night
- Review session Sunday, December 13th, 7:30-9:30 pm EST
 - Practice exam released today
- Final exam due 11:59 pm EST Tuesday, December 15th
 - Will be released on the evening (EST) of Monday, December 14th
 - Designed to take 3 hours
 - Open book, open notes, closed person
- Evaluate us: <https://cmu.smartevals.com/>
- Evaluate our TAs:
<https://www.ugrad.cs.cmu.edu/ta/F20/feedback/>
- (We will return at 11:54, so that you have time to evaluate us and our TAs.)

Key concepts from Tuesday

Today: A finale of puzzlers

A quick challenge: Implement binary search

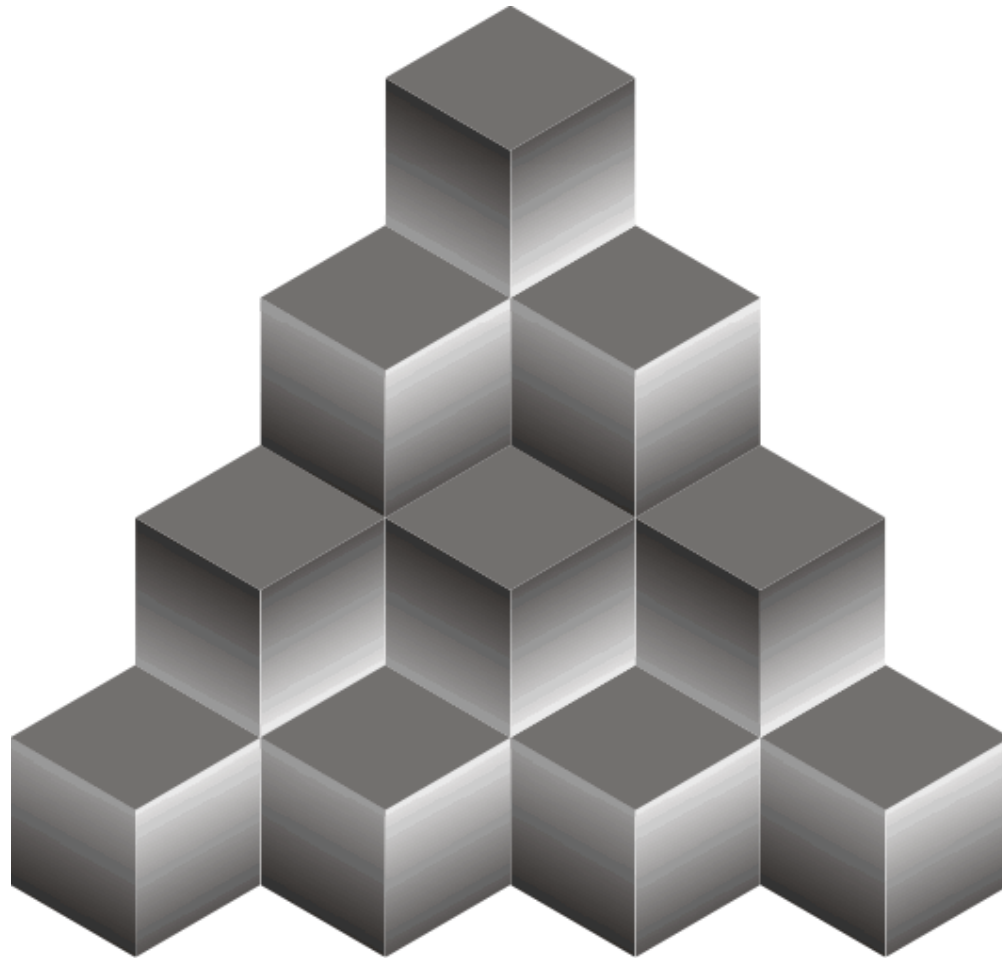
```
/**
 * Searches the specified array of ints for the specified value
 * using the binary search algorithm. If the array is not sorted,
 * the results are undefined. If the array contains multiple
 * elements with the specified value, there is no guarantee which
 * one will be found.
 *
 * @returns 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);
```



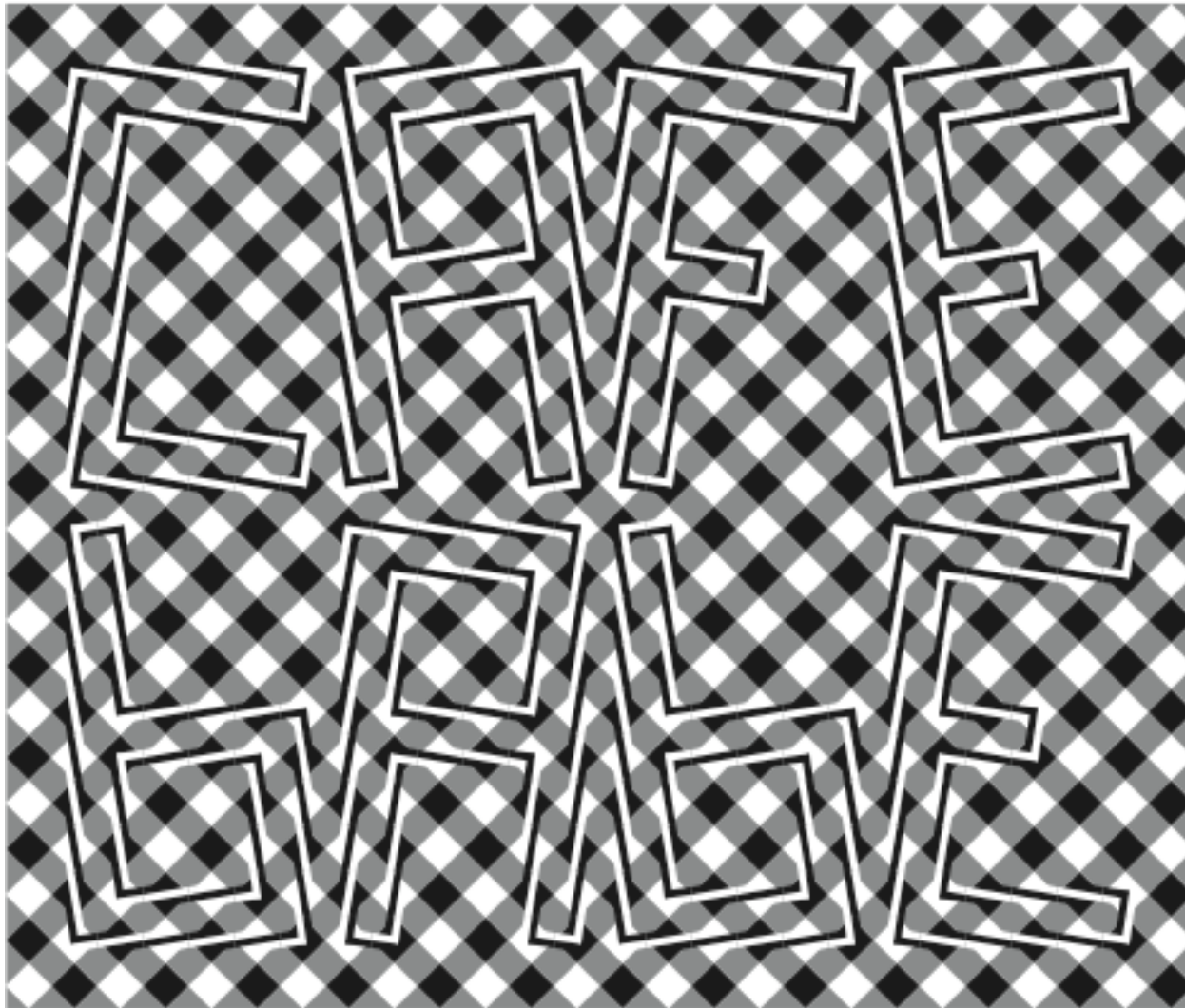
Logvinenko 1999



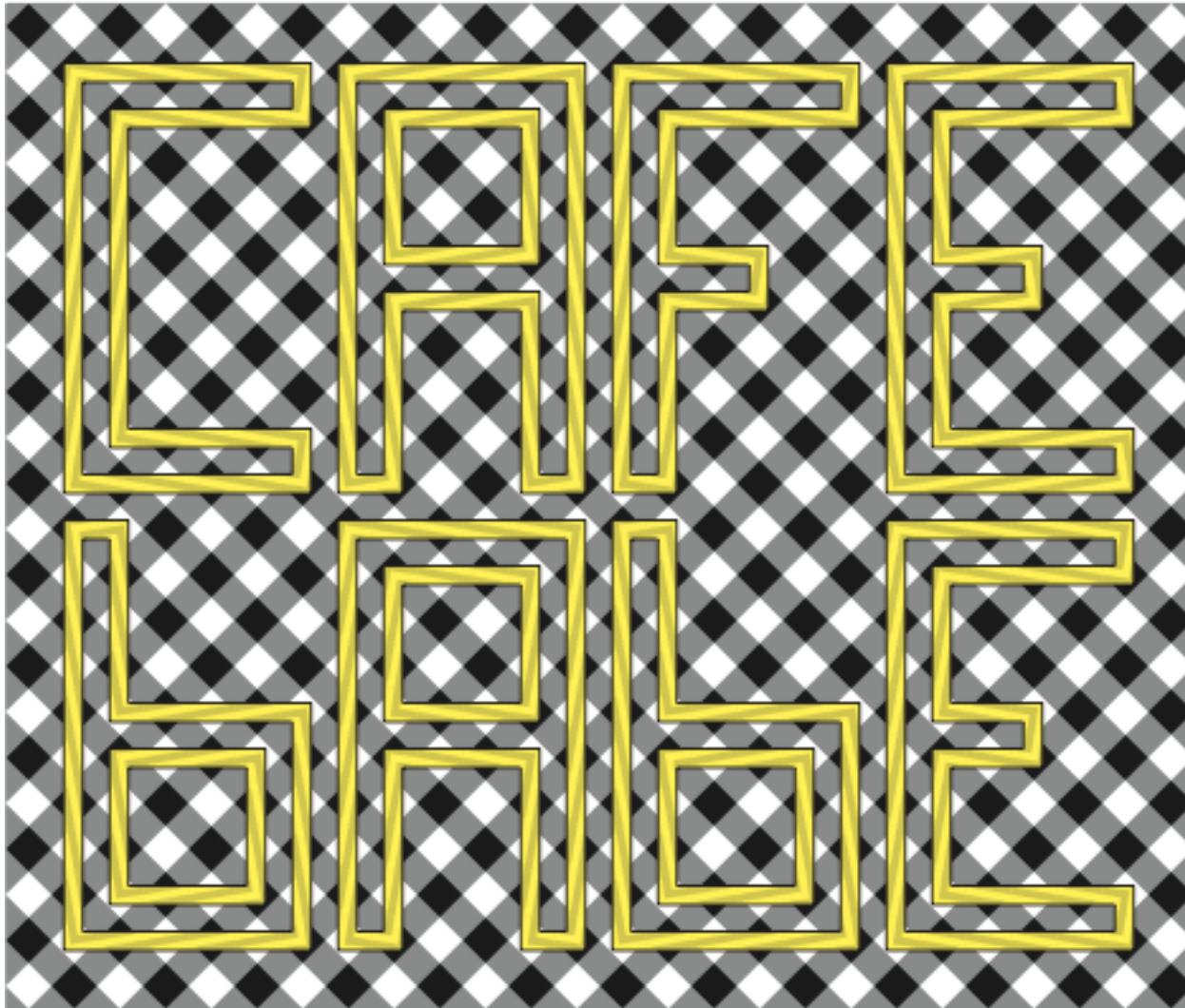
Logvinenko 1999



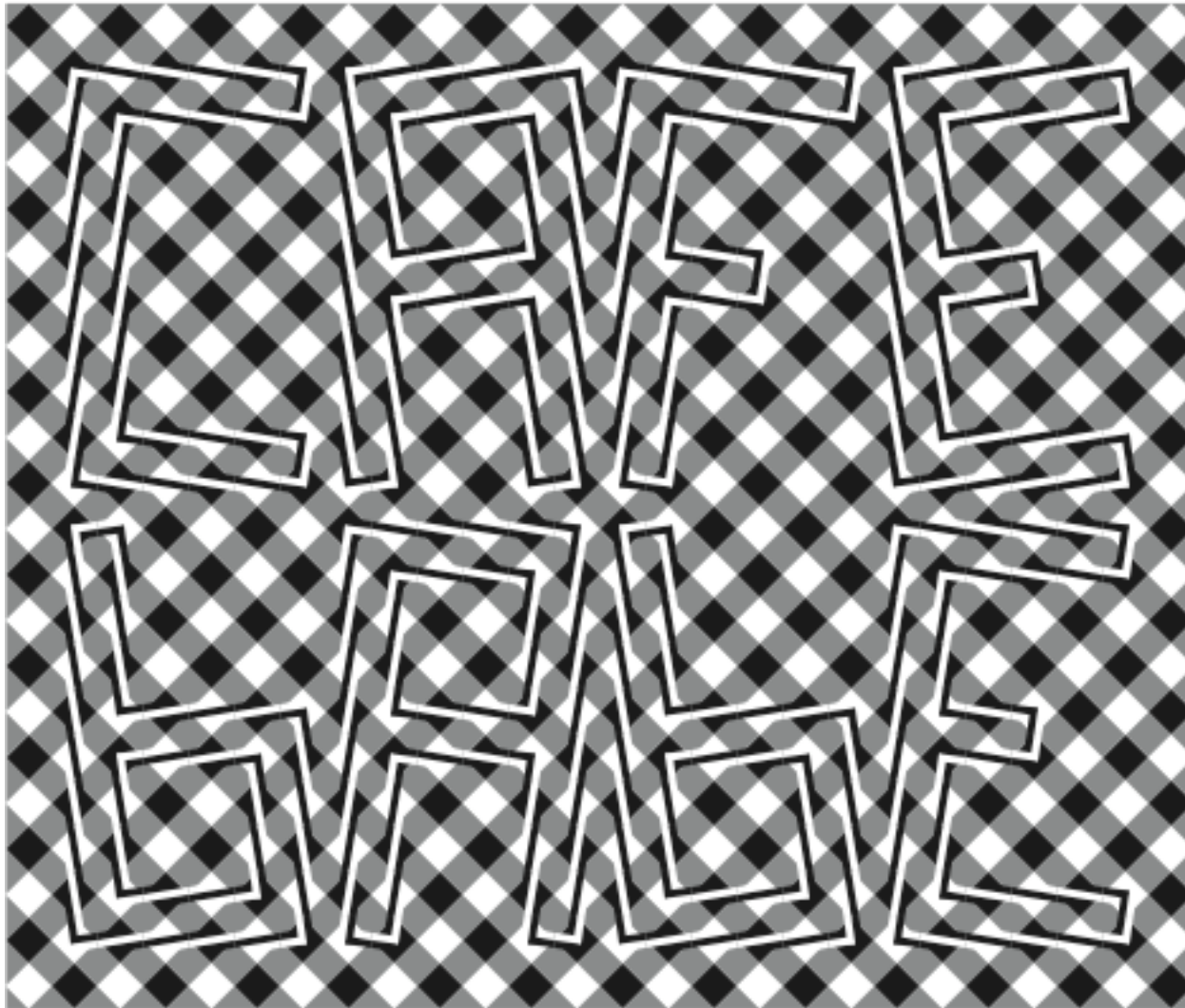
Logvinenko 1999



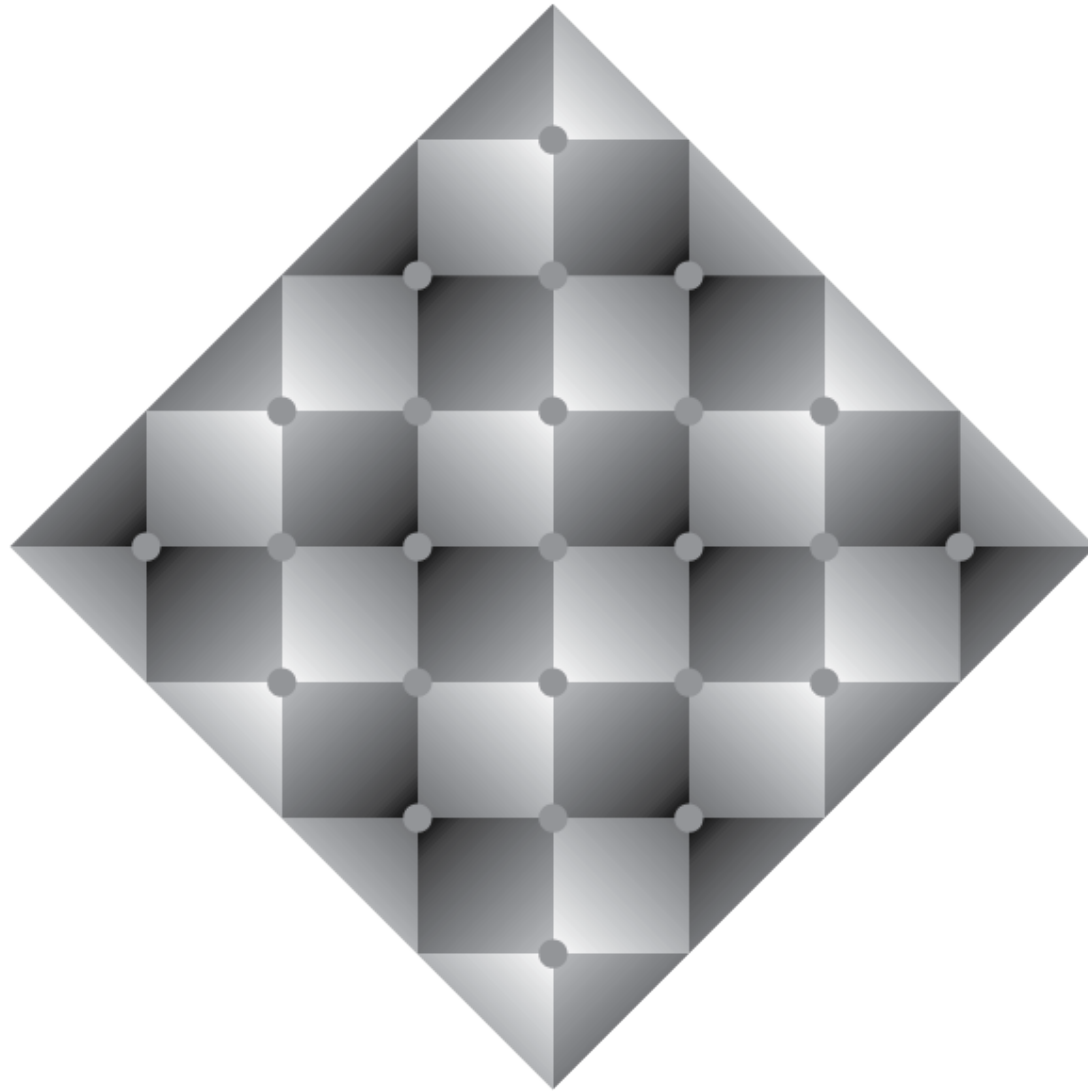
Fraser 1908



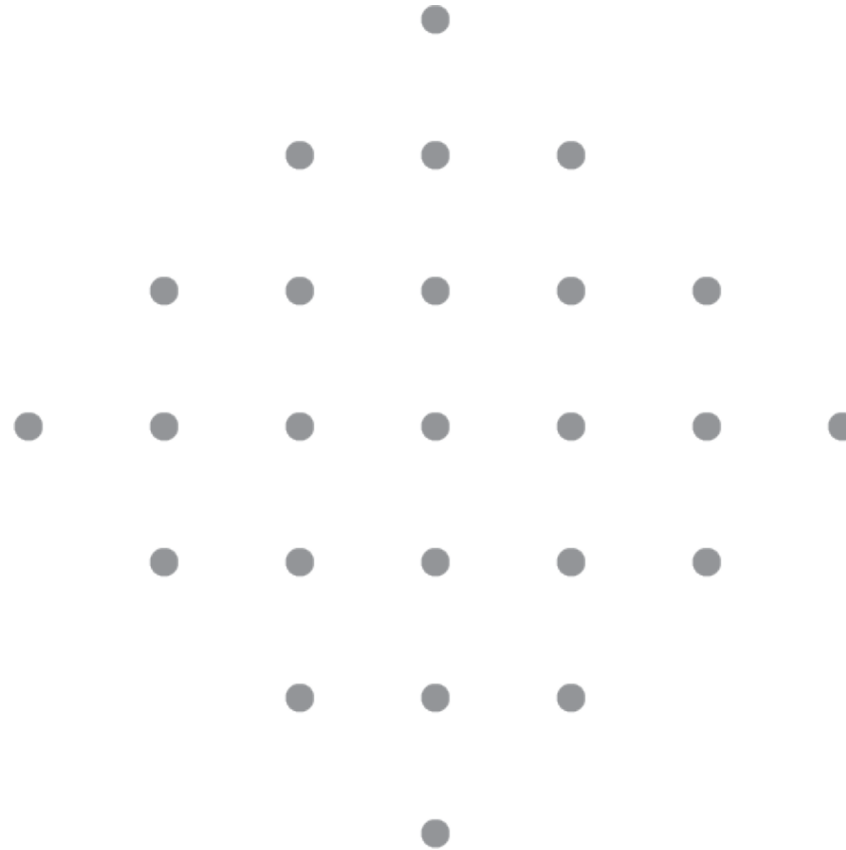
Fraser 1908



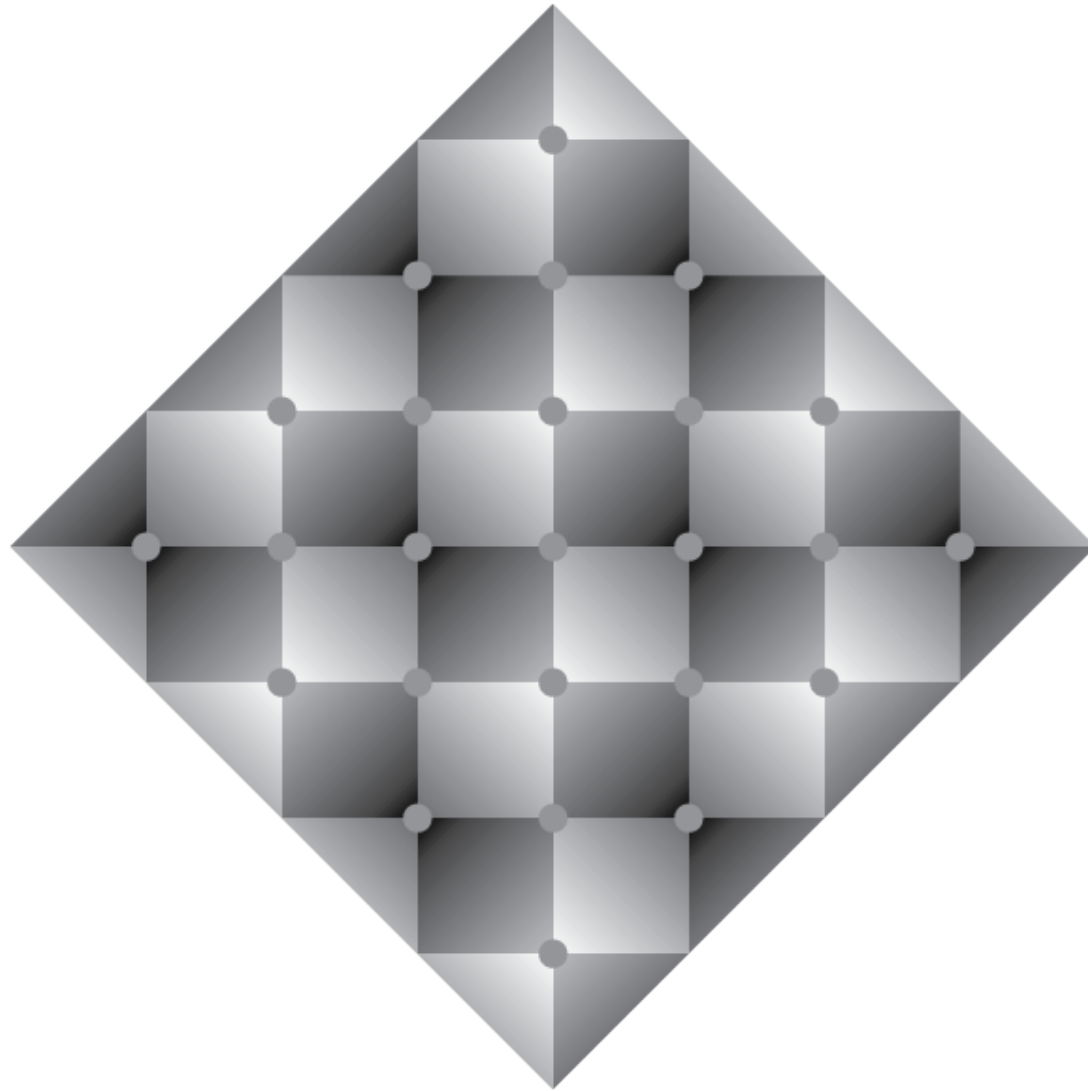
Fraser 1908



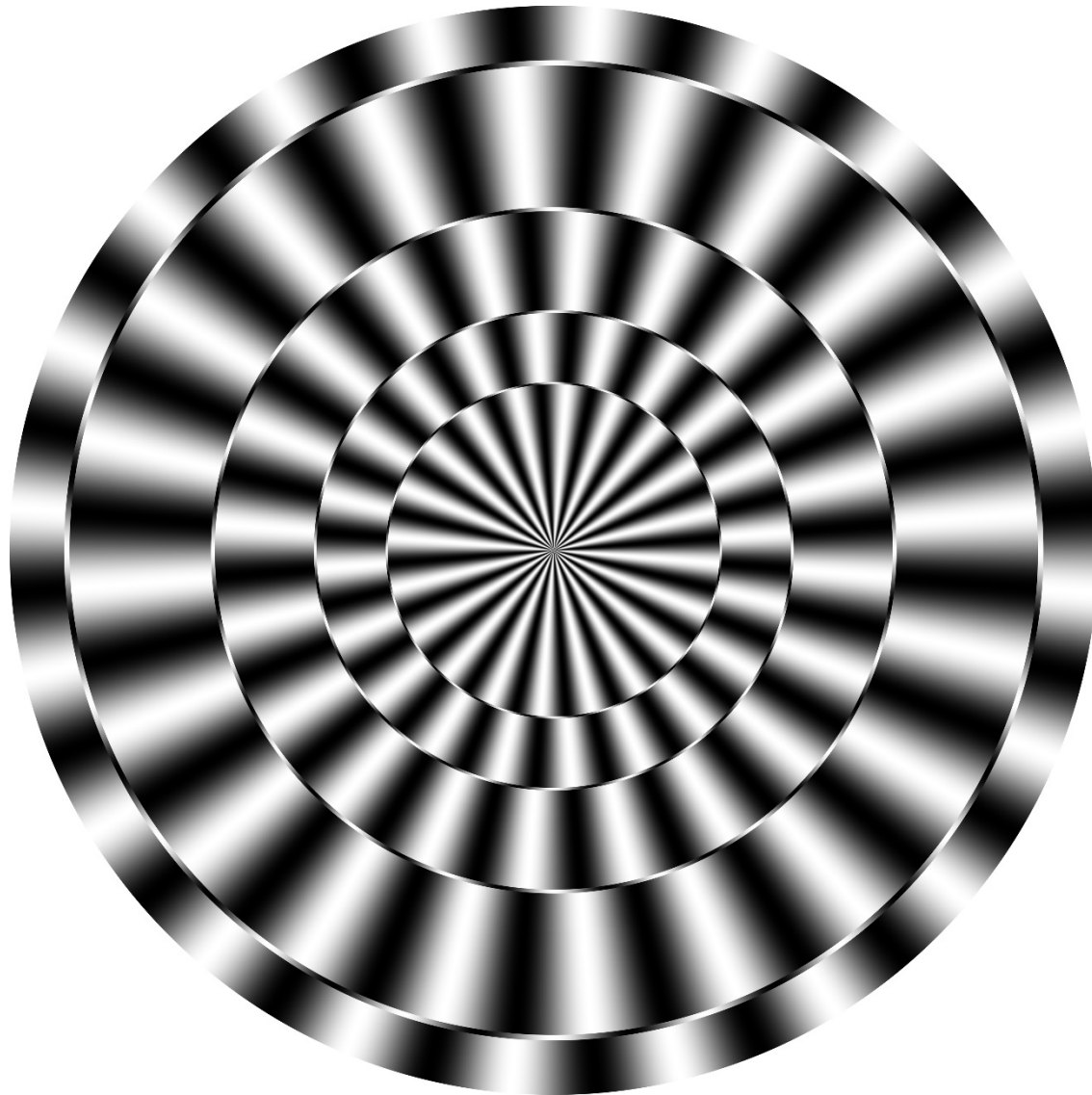
Todorovic 1997



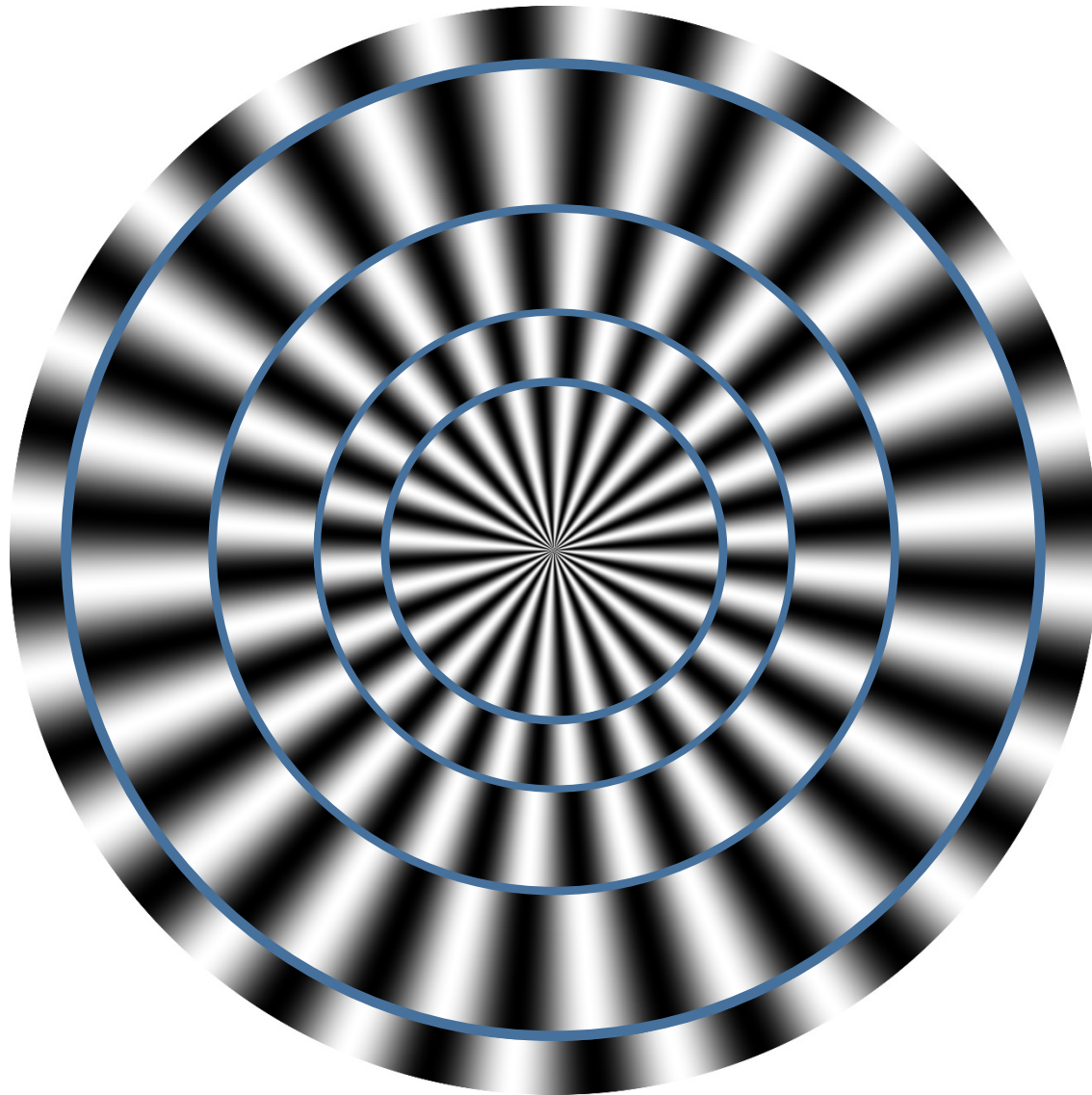
Todorovic 1997



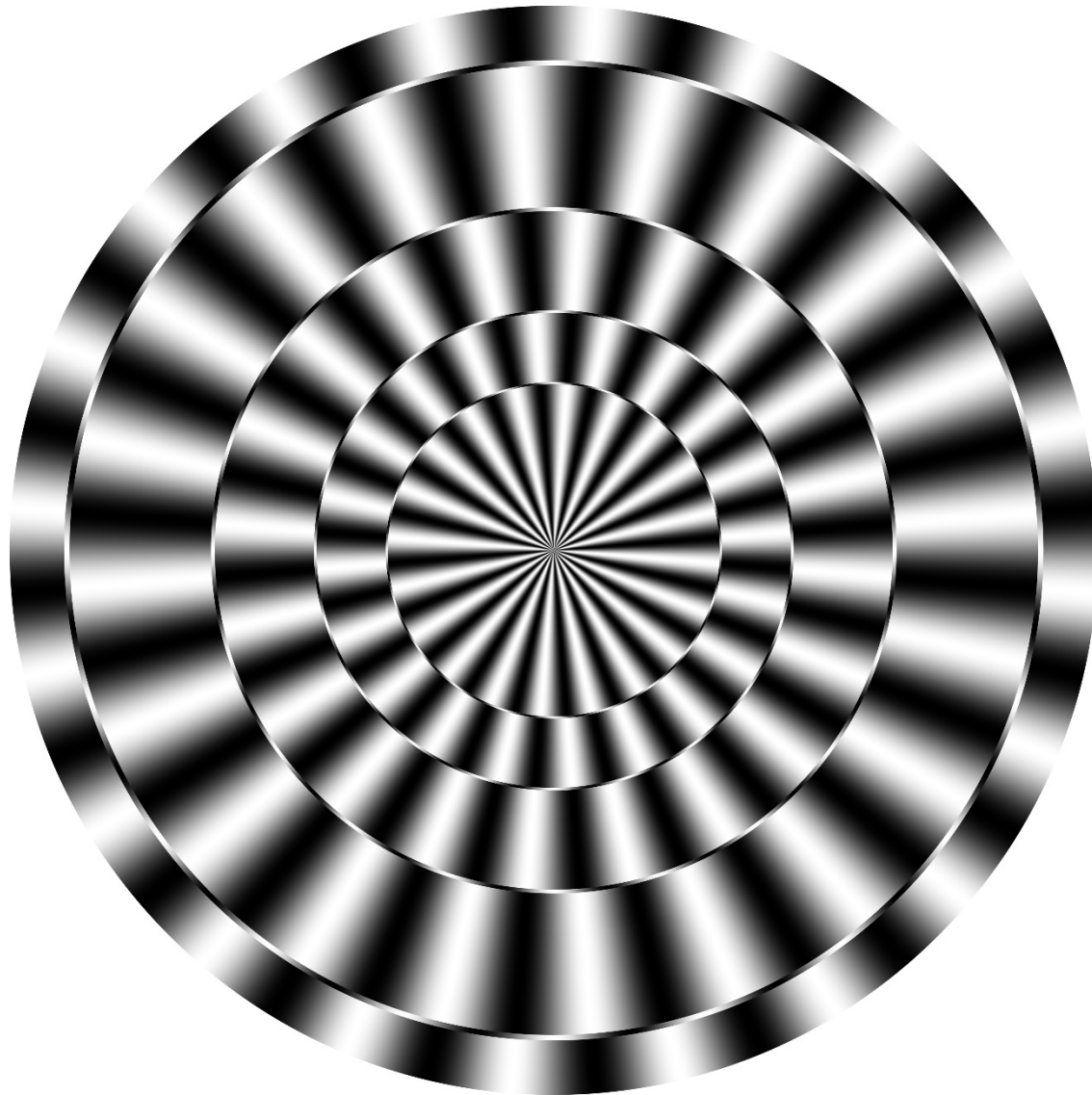
Todorovic 1997



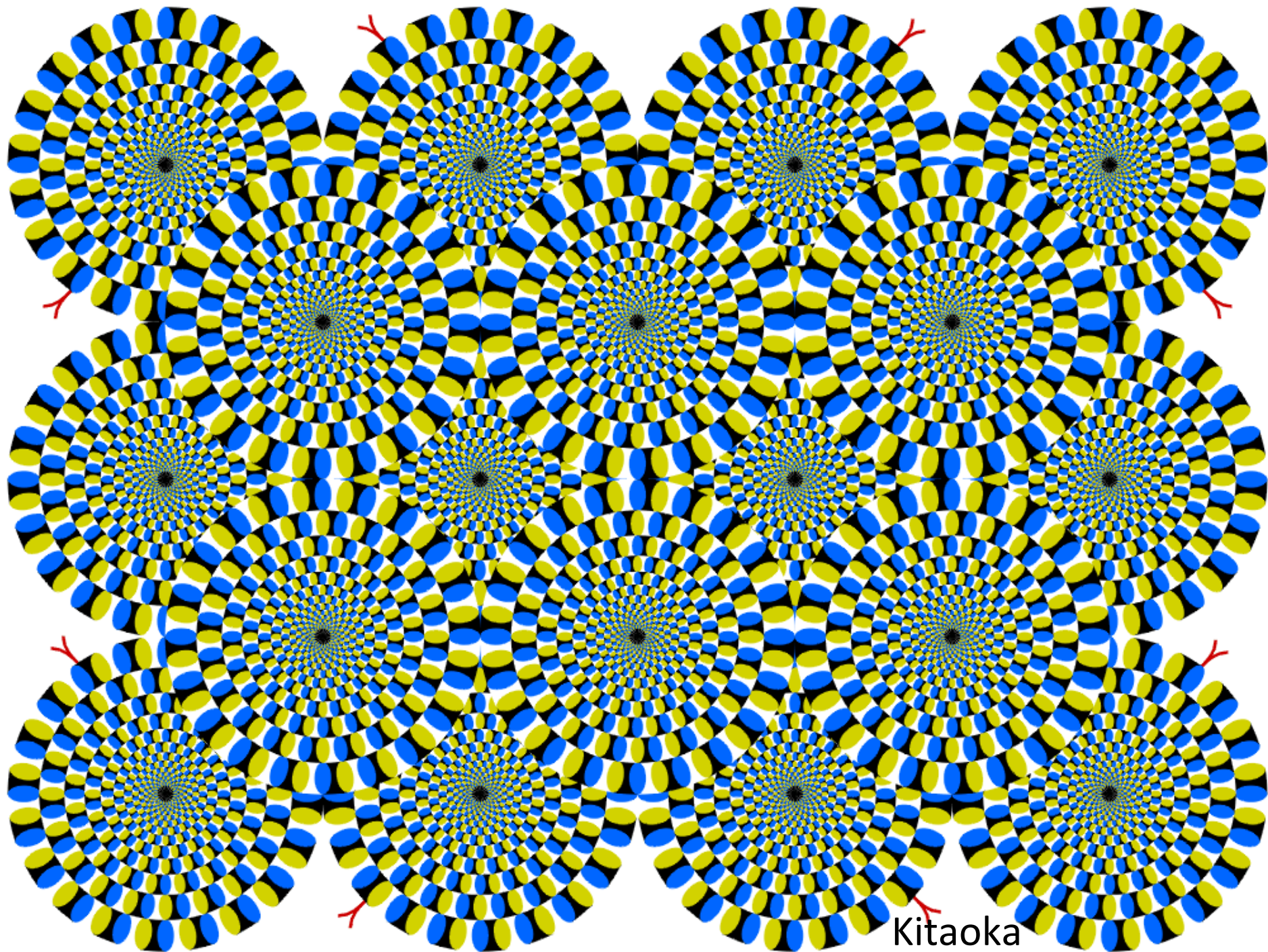
Kitaoka 2020



Kitaoka 2020



Kitaoka 2020



Kitaoka

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 and ErrorProne
 - 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!");
```

Prints 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!");  
    }  
}
```

Prints 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) `[-20000000000, 0, 20000000000]`

(b) `[20000000000, 0, -20000000000]`

(c) `[-20000000000, 20000000000, 0]`

(d) None of the above: Unspecified;

In practice, `[20000000000, -20000000000, 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) It varies
- (d) None of the above

What does it print?

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

What does it print?

- (a) true
- (b) false
- (c) It varies
- (d) None of the above

The finally is processed after the try.

Another look

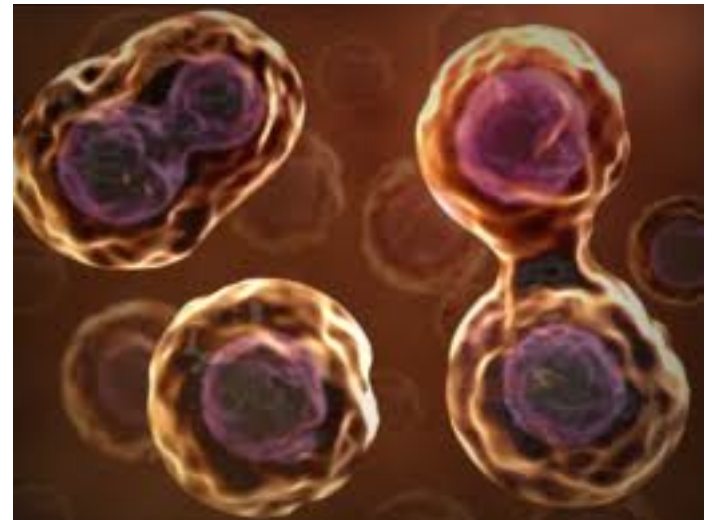
```
class Indecisive {  
    public static void main(String[] args) {  
        System.out.println(decision());  
    }  
  
    static boolean decision() {  
        try {  
            return true;  
        } finally {  
            return false;  
        }  
    }  
}
```

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 H Hydrogen 1.00794																	2 He Helium 4.003																												
3 Li Lithium 6.941	4 Be Beryllium 9.012182															5 B Boron 10.811	6 C Carbon 12.0107	7 N Nitrogen 14.00674	8 O Oxygen 15.9994	9 F Fluorine 18.9984032	10 Ne Neon 20.1797																								
11 Na Sodium 22.989770	12 Mg Magnesium 24.3050															13 Al Aluminum 26.981538	14 Si Silicon 28.0855	15 P Phosphorus 30.973761	16 S Sulfur 32.066	17 Cl Chlorine 35.4527	18 Ar Argon 39.948																								
19 K Potassium 39.0983	20 Ca Calcium 40.078	21 Sc Scandium 44.955910	22 Ti Titanium 47.867	23 V Vanadium 50.9415	24 Cr Chromium 51.9961	25 Mn Manganese 54.938044	26 Fe Iron 55.845	27 Co Cobalt 58.933200	28 Ni Nickel 58.6934	29 Cu Copper 63.546	30 Zn Zinc 65.39	31 Ga Gallium 69.723	32 Ge Germanium 72.61	33 As Arsenic 74.92160	34 Se Selenium 78.96	35 Br Bromine 79.904	36 Kr Krypton 83.80																												
37 Rb Rubidium 85.4678	38 Sr Strontium 87.62	39 Y Yttrium 88.90585	40 Zr Zirconium 91.224	41 Nb Niobium 92.90638	42 Mo Molybdenum 95.94	43 Tc Technetium (98)	44 Ru Ruthenium 101.07	45 Rh Rhodium 102.90550	46 Pd Palladium 106.42	47 Ag Silver 107.8682	48 Cd Cadmium 112.411	49 In Indium 114.818	50 Sn Tin 118.710	51 Sb Antimony 121.760	52 Te Tellurium 127.60	53 I Iodine 126.90447	54 Xe Xenon 131.29																												
55 Cs Cesium 132.90545	56 Ba Barium 137.327	57 La Lanthanum 138.9055	72 Hf Hafnium 178.49	73 Ta Tantalum 180.9479	74 W Tungsten 183.84	75 Re Rhenium 186.207	76 Os Osmium 190.23	77 Ir Iridium 192.217	78 Pt Platinum 195.078	79 Au Gold 196.96655	80 Hg Mercury 200.59	81 Tl Thallium 204.3833	82 Pb Lead 207.2	83 Bi Bismuth 208.98038	84 Po Polonium (209)	85 At Astatine (210)	86 Rn Radon (222)																												
87 Fr Francium (223)	88 Ra Radium (226)	89 Ac Actinium (227)	104 Rf Rutherfordium (261)	105 Db Dubnium (262)	106 Sg Seaborgium (263)	107 Bh Bohrium (262)	108 Hs Hassium (265)	109 Mt Meitnerium (266)	110	111	112	113	114																																
<table><tr><td>58 Ce Cerium 140.116</td><td>59 Pr Praseodymium 140.90765</td><td>60 Nd Neodymium 144.24</td><td>61 Pm Promethium (145)</td><td>62 Sm Samarium 150.36</td><td>63 Eu Europium 151.964</td><td>64 Gd Gadolinium 157.25</td><td>65 Tb Terbium 158.92534</td><td>66 Dy Dysprosium 162.50</td><td>67 Ho Holmium 164.93032</td><td>68 Er Erbium 167.26</td><td>69 Tm Thulium 168.93421</td><td>70 Yb Ytterbium 173.04</td><td>71 Lu Lutetium 174.967</td></tr><tr><td>90 Th Thorium 232.0381</td><td>91 Pa Protactinium 231.03588</td><td>92 U Uranium 238.0289</td><td>93 Np Neptunium (237)</td><td>94 Pu Plutonium (244)</td><td>95 Am Americium (243)</td><td>96 Cm Curium (247)</td><td>97 Bk Berkelium (247)</td><td>98 Cf Californium (251)</td><td>99 Es Einsteinium (252)</td><td>100 Fm Fermium (257)</td><td>101 Md Mendelevium (258)</td><td>102 No Nobelium (259)</td><td>103 Lr Lawrencium (262)</td></tr></table>																		58 Ce Cerium 140.116	59 Pr Praseodymium 140.90765	60 Nd Neodymium 144.24	61 Pm Promethium (145)	62 Sm Samarium 150.36	63 Eu Europium 151.964	64 Gd Gadolinium 157.25	65 Tb Terbium 158.92534	66 Dy Dysprosium 162.50	67 Ho Holmium 164.93032	68 Er Erbium 167.26	69 Tm Thulium 168.93421	70 Yb Ytterbium 173.04	71 Lu Lutetium 174.967	90 Th Thorium 232.0381	91 Pa Protactinium 231.03588	92 U Uranium 238.0289	93 Np Neptunium (237)	94 Pu Plutonium (244)	95 Am Americium (243)	96 Cm Curium (247)	97 Bk Berkelium (247)	98 Cf Californium (251)	99 Es Einsteinium (252)	100 Fm Fermium (257)	101 Md Mendelevium (258)	102 No Nobelium (259)	103 Lr Lawrencium (262)
58 Ce Cerium 140.116	59 Pr Praseodymium 140.90765	60 Nd Neodymium 144.24	61 Pm Promethium (145)	62 Sm Samarium 150.36	63 Eu Europium 151.964	64 Gd Gadolinium 157.25	65 Tb Terbium 158.92534	66 Dy Dysprosium 162.50	67 Ho Holmium 164.93032	68 Er Erbium 167.26	69 Tm Thulium 168.93421	70 Yb Ytterbium 173.04	71 Lu Lutetium 174.967																																
90 Th Thorium 232.0381	91 Pa Protactinium 231.03588	92 U Uranium 238.0289	93 Np Neptunium (237)	94 Pu Plutonium (244)	95 Am Americium (243)	96 Cm Curium (247)	97 Bk Berkelium (247)	98 Cf Californium (251)	99 Es Einsteinium (252)	100 Fm Fermium (257)	101 Md Mendelevium (258)	102 No Nobelium (259)	103 Lr 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 (reprised)

- Keep it simple
- Use all the tools you know:
 - A good IDE
 - Static analysis tools like SpotBugs and ErrorProne
 - 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