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

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

**Josh Bloch** **Charlie Garrod**

# Administrivia

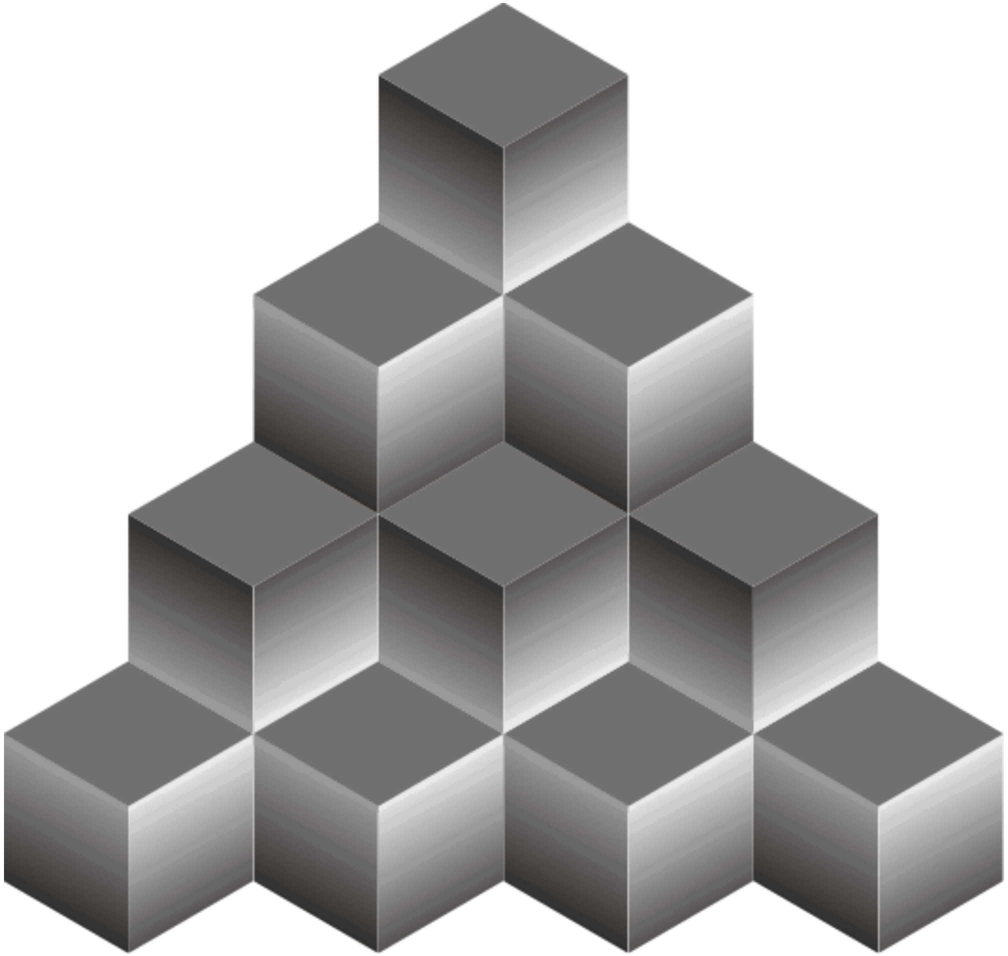
- Homework 6 due last night
- Review session Wednesday, May 12th, 7:00-9:00 pm EDT
  - Practice exam released by tomorrow
- Final exam due 11:59 pm EDT Friday, May 14th
  - Will be released on the evening (EDT) of Thursday, May 13th
  - Designed to take 3 hours
  - Open book, open notes, closed person
- Evaluate us: https://cmu.smartevals.com/
- Evaluate our TAs: https://www.ugrad.cs.cmu.edu/ta/S21/feedback/
- (We will return at 4:12, so that you have time to evaluate us and our TAs.)

# Key concepts from Tuesday

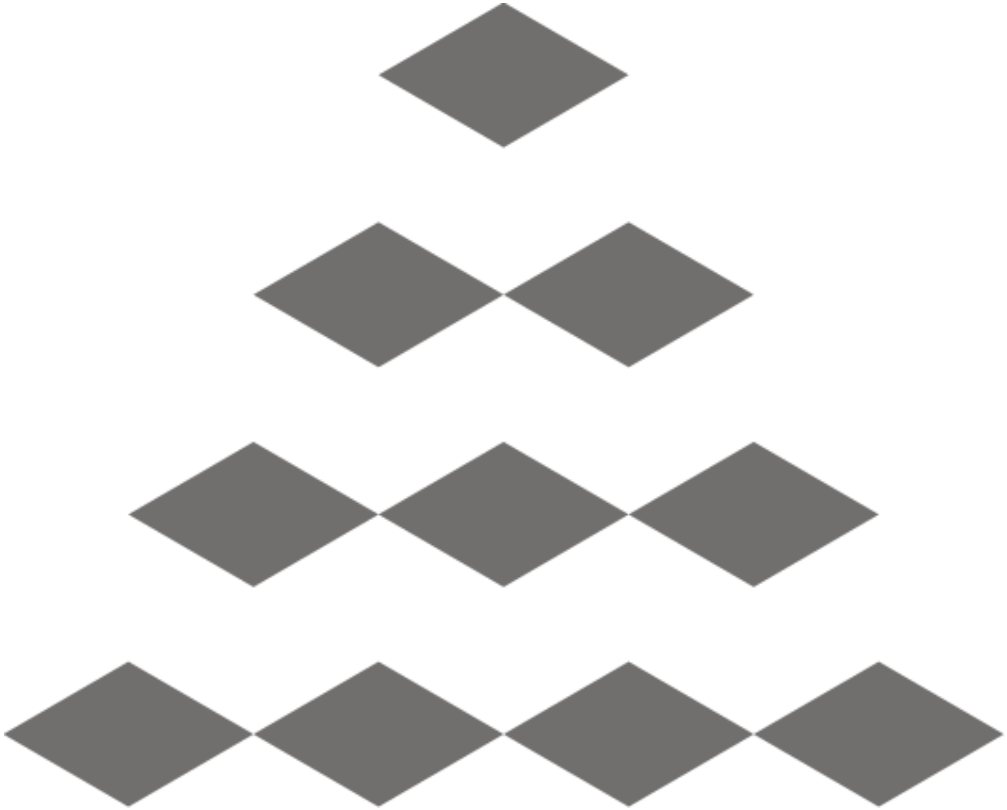
# Today:  A finale of puzzlers

# A quick challenge:  Implement binary search

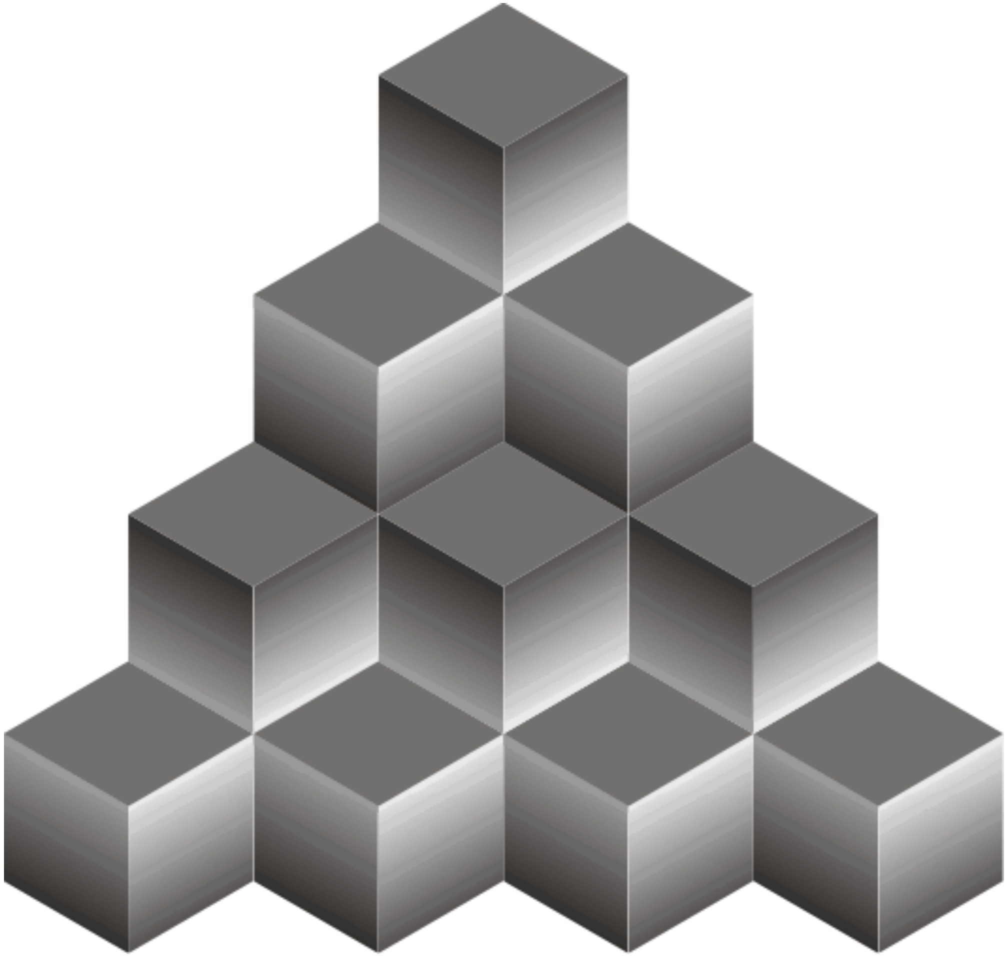
```
/**
 * Searches the specified array of ints for the specified value
 * using the binary search algorithm. If the array is not sorted,
 * the results are undefined. If the array contains multiple
 * elements with the specified value, there is no guarantee which
 * one will be found.
 *
 * @returns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



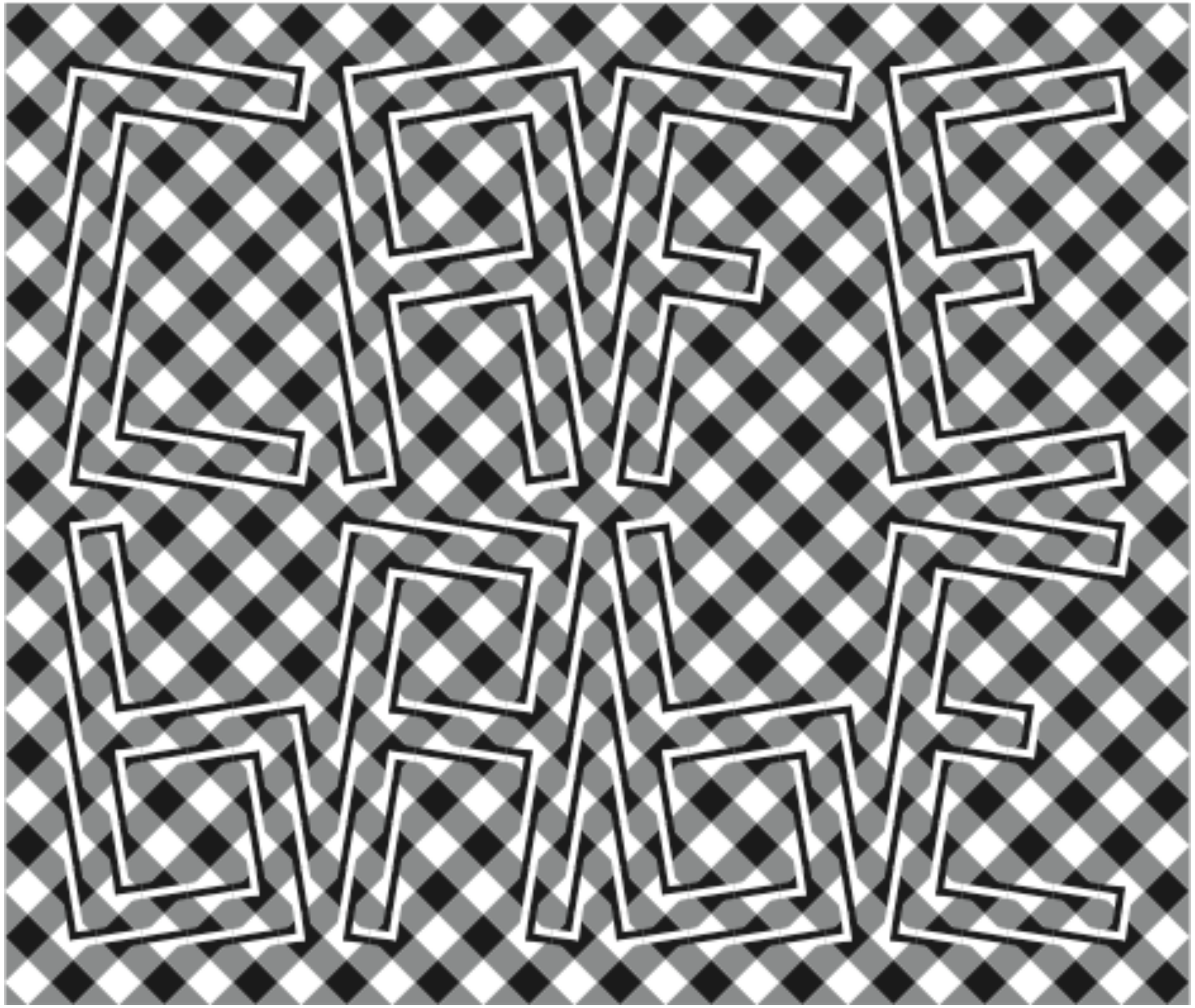
**Logvinenko 1999**



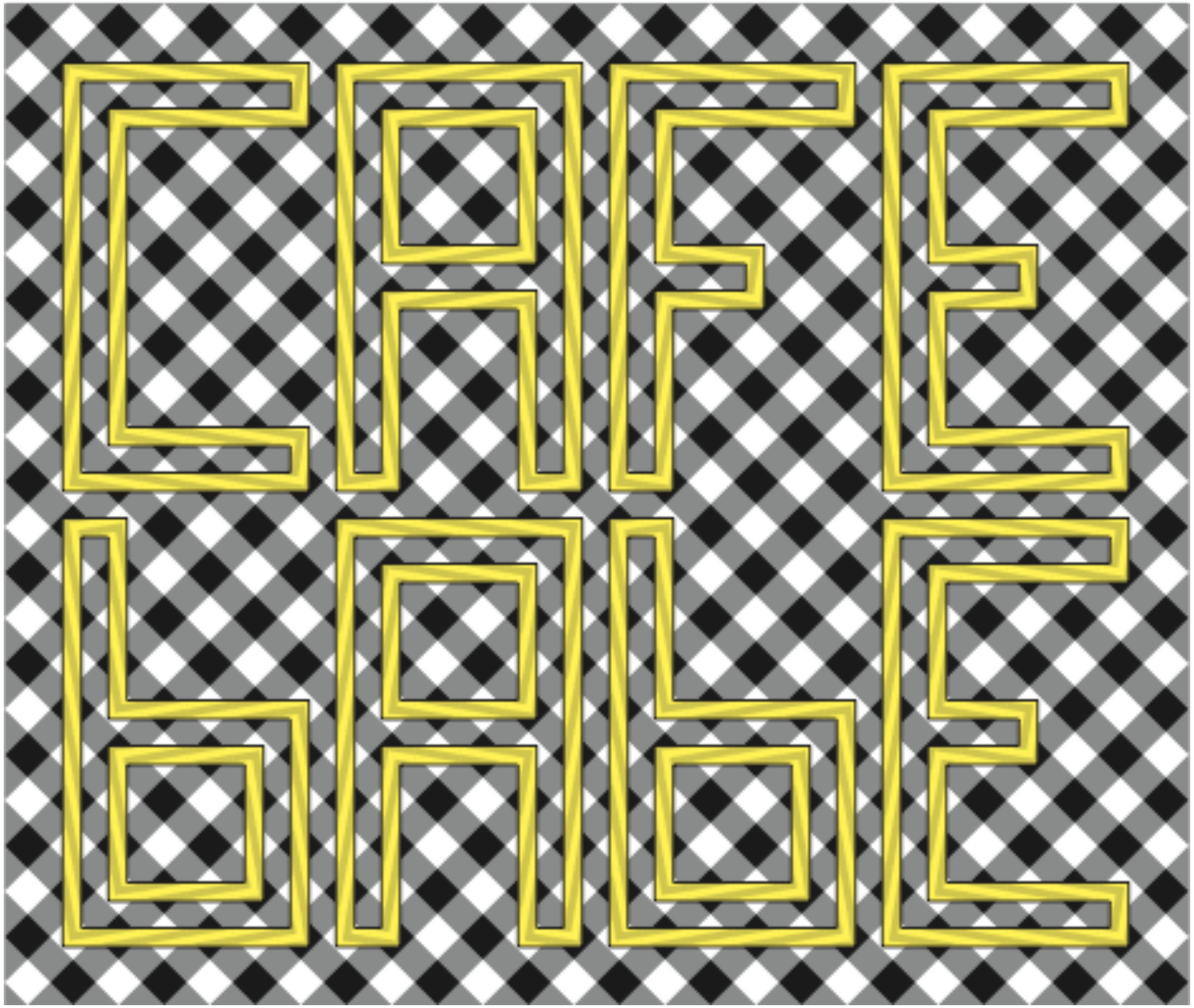
**Logvinenko 1999**



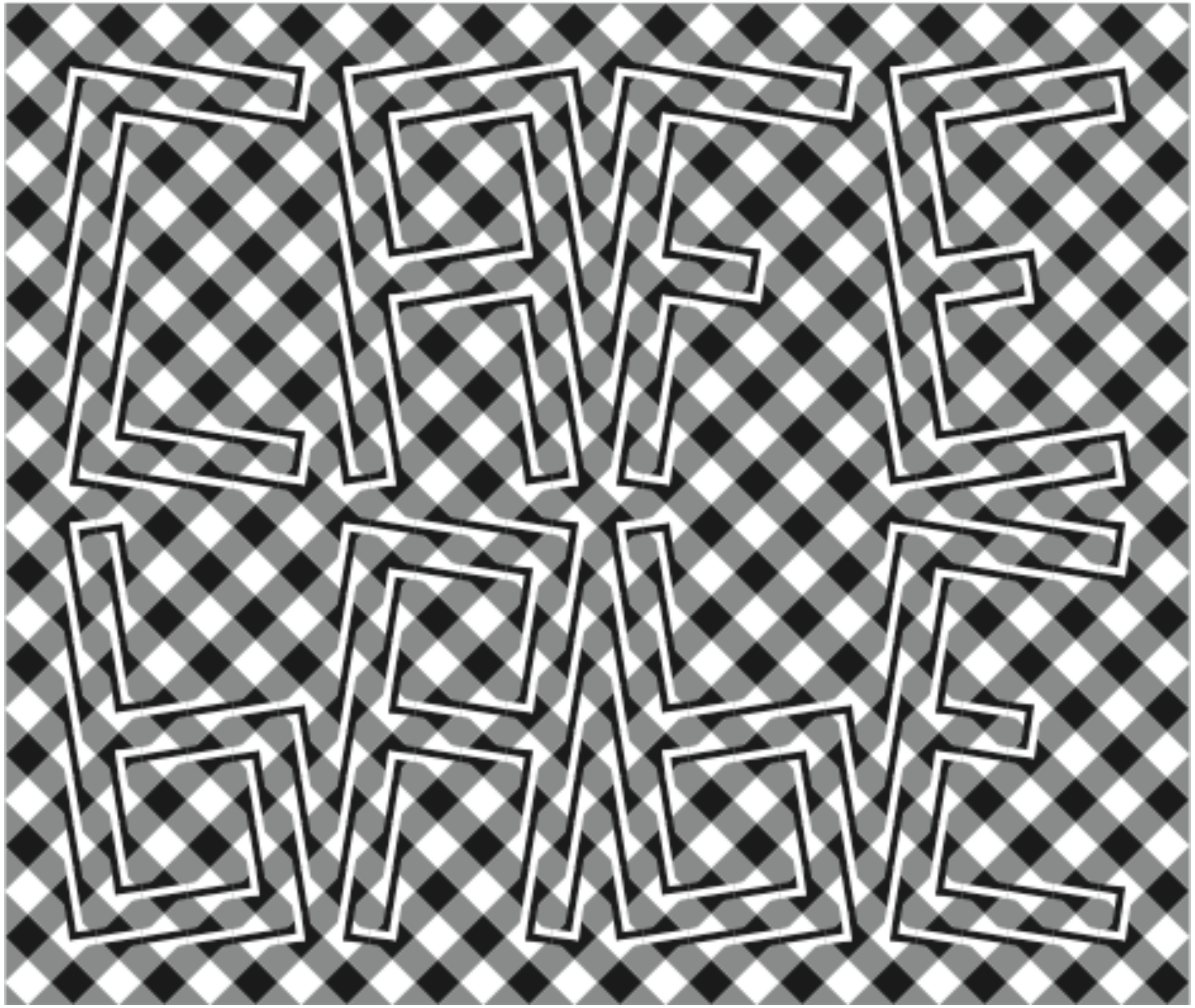
**Logvinenko 1999**



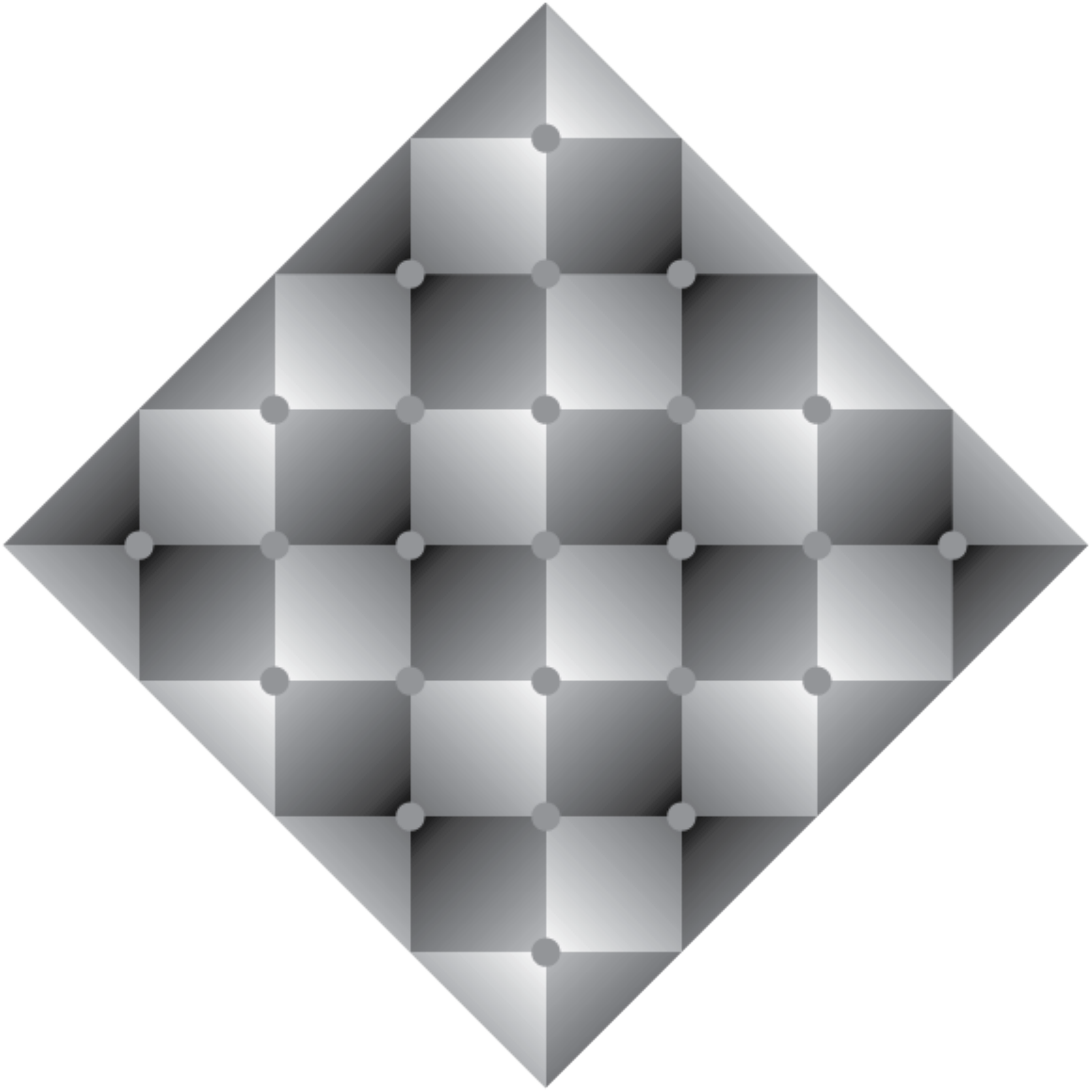
**Fraser 1908**



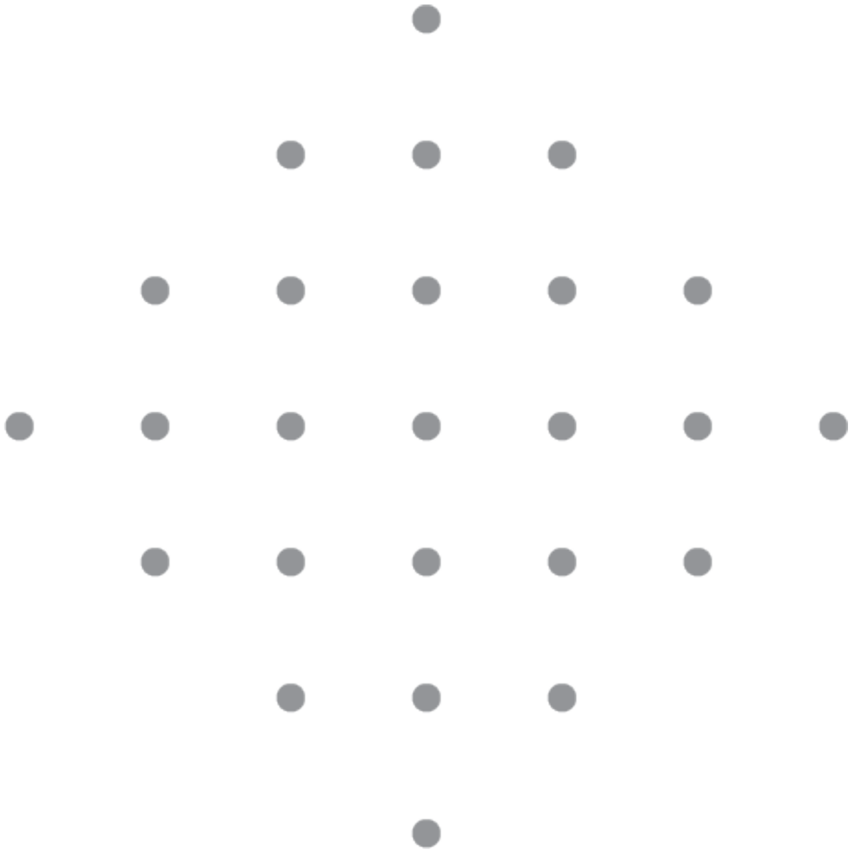
**Fraser 1908**



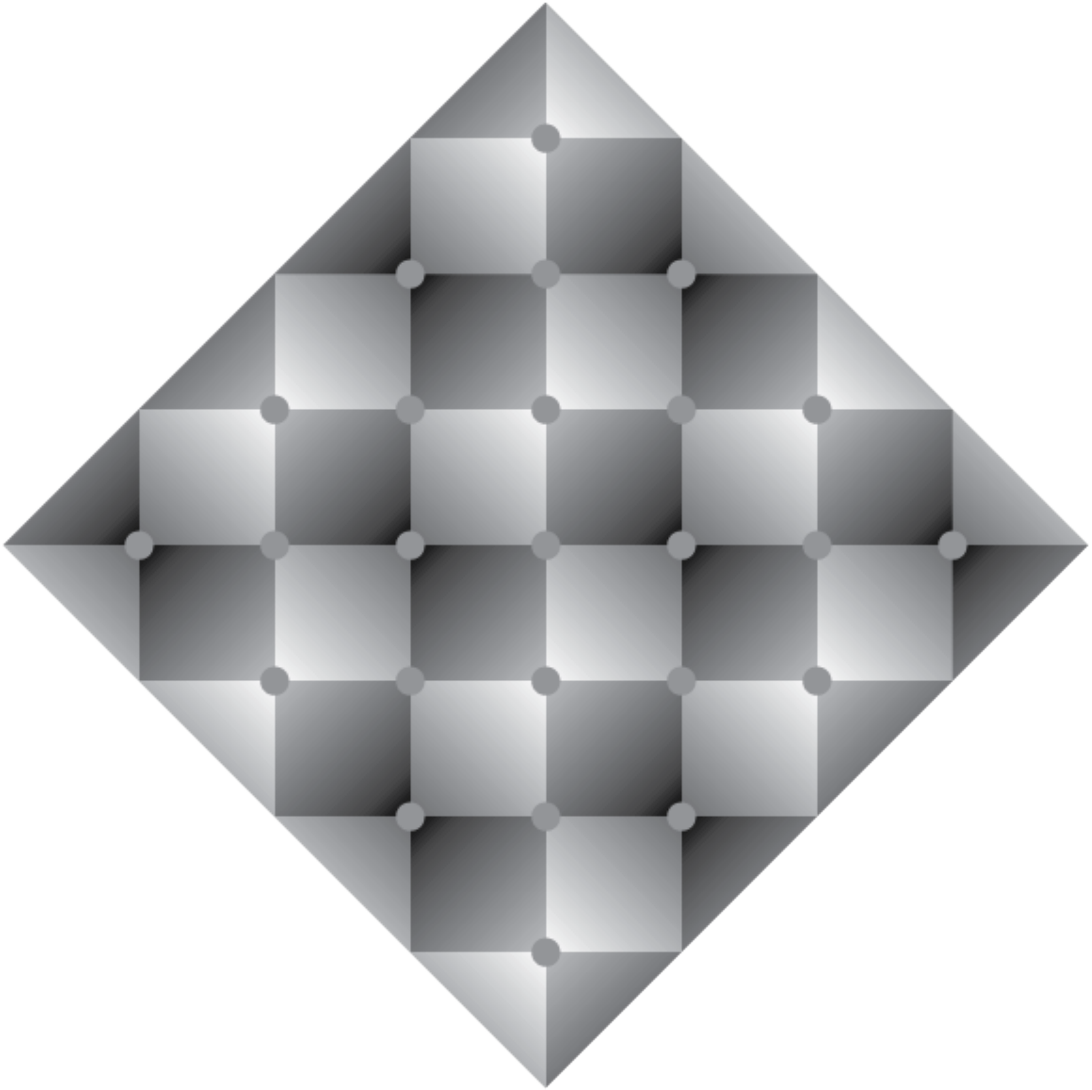
**Fraser 1908**



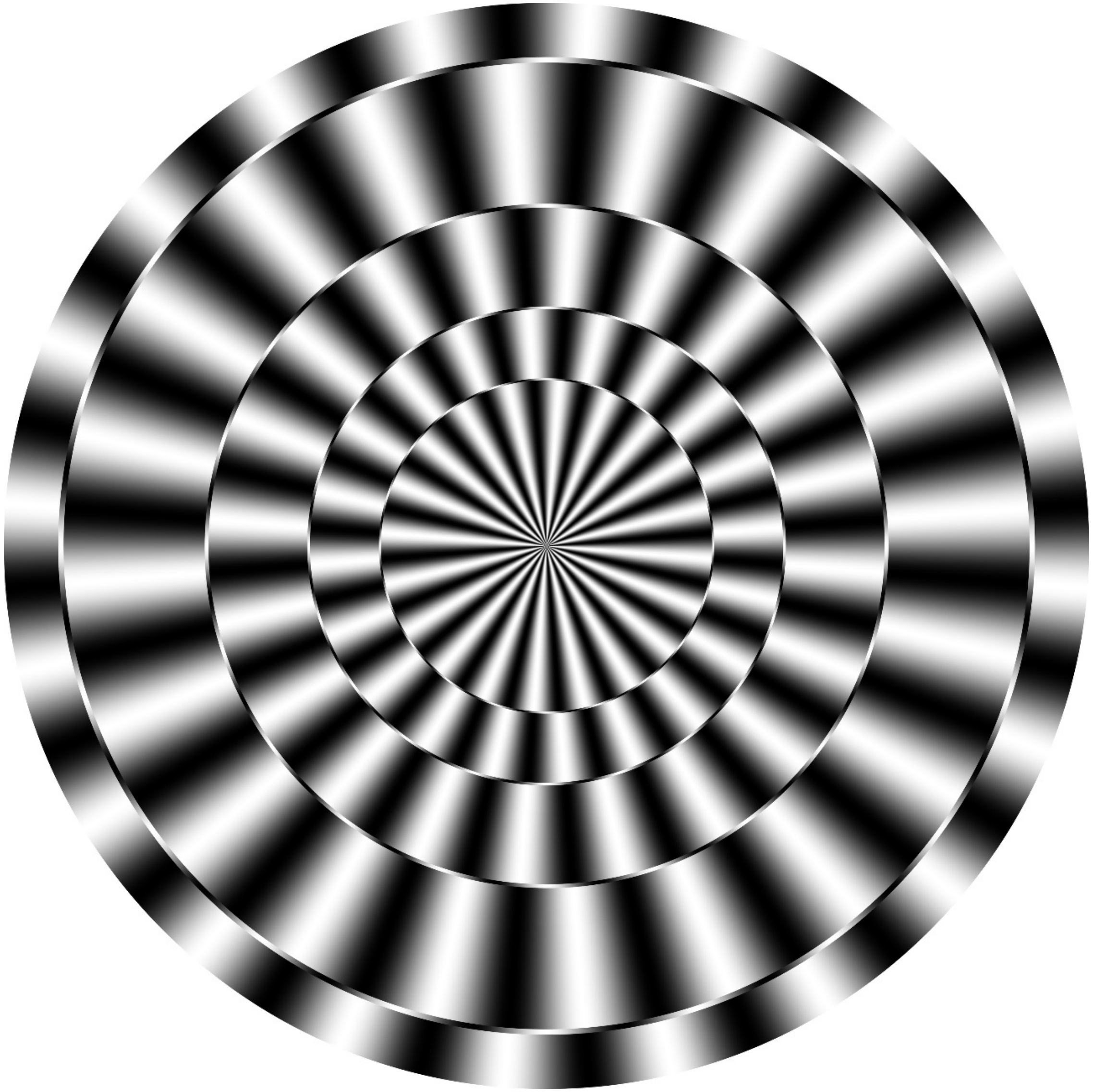
**Todorovic 1997**



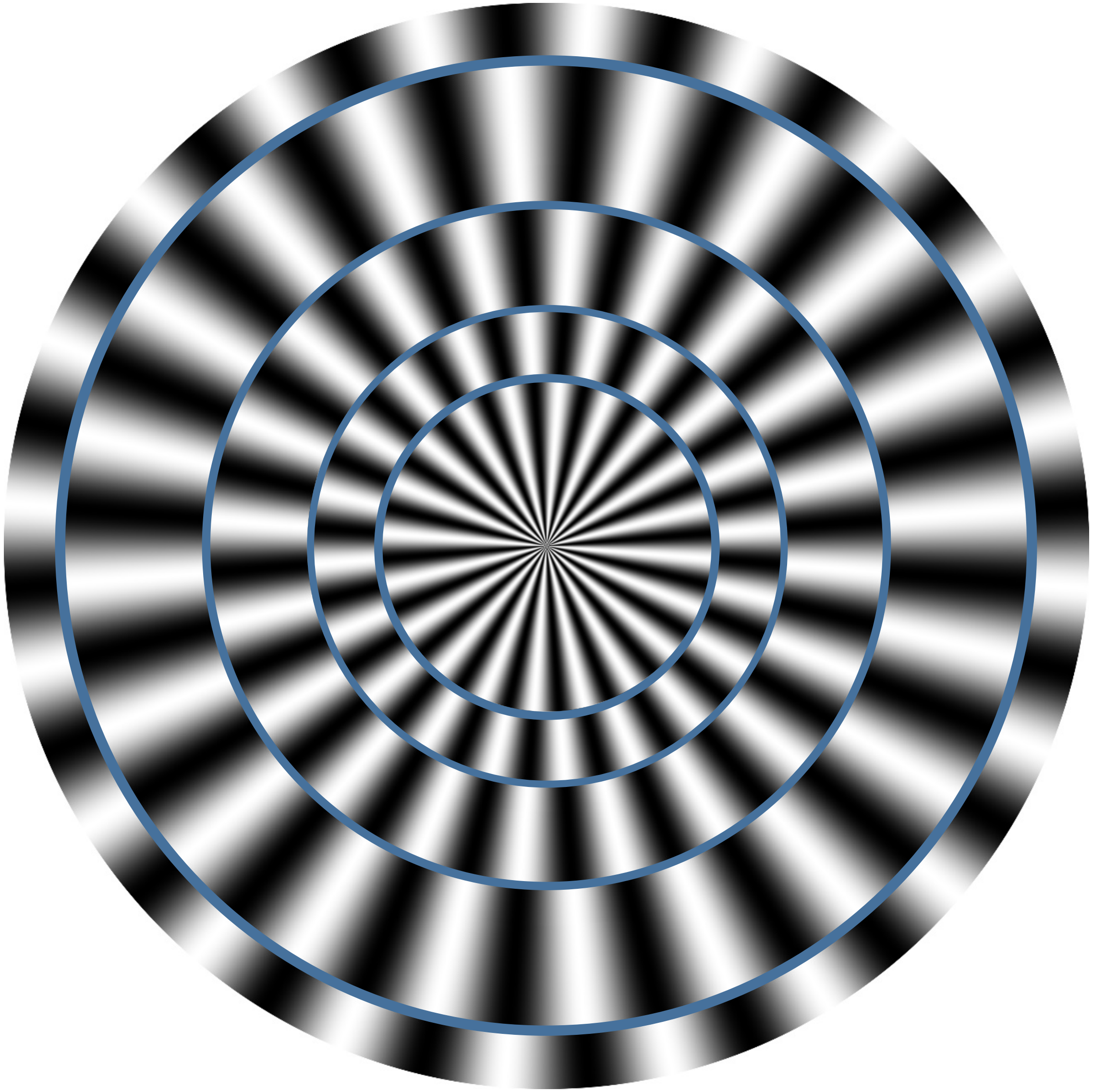
Todorovic 1997



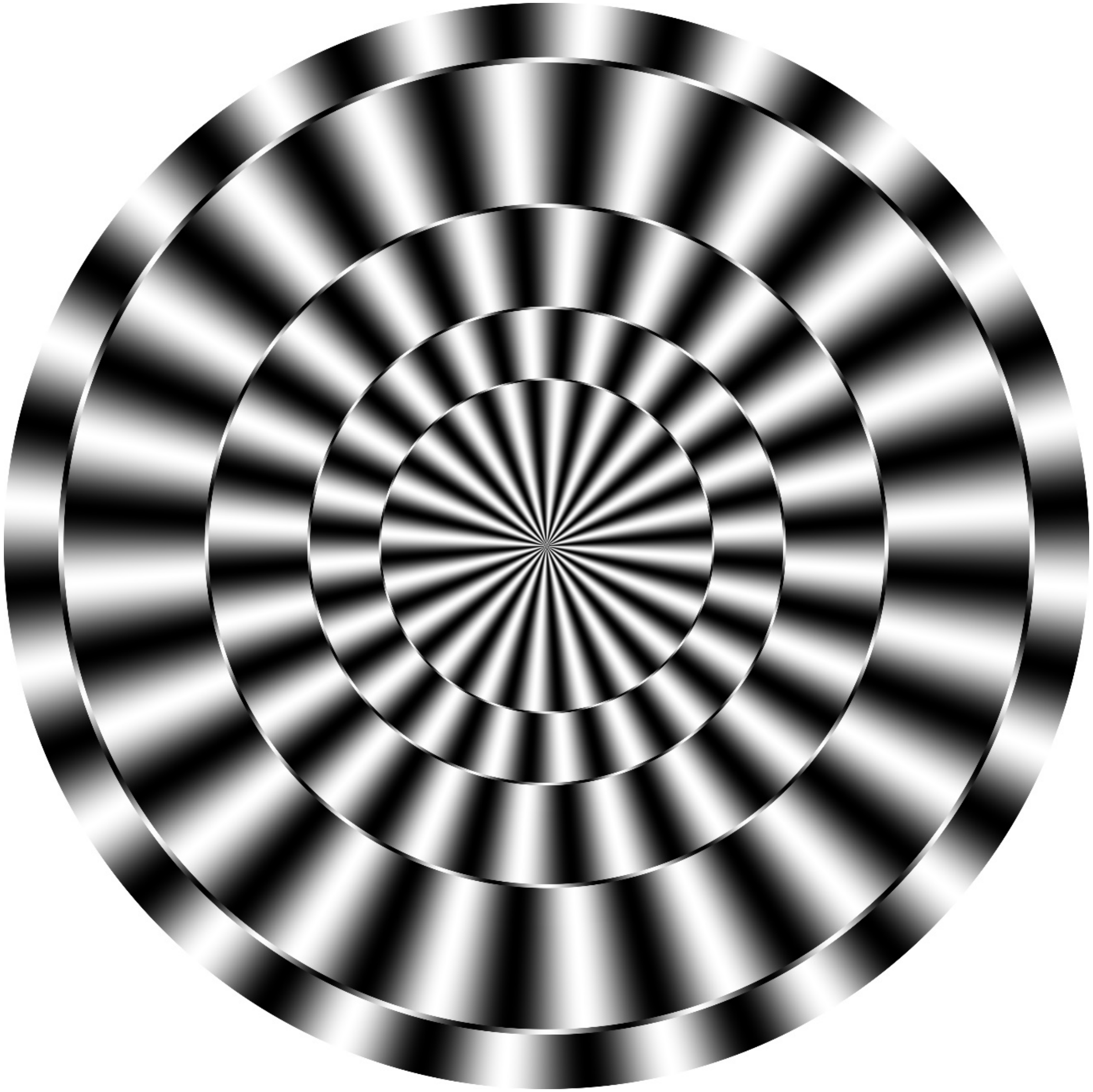
Todorovic 1997



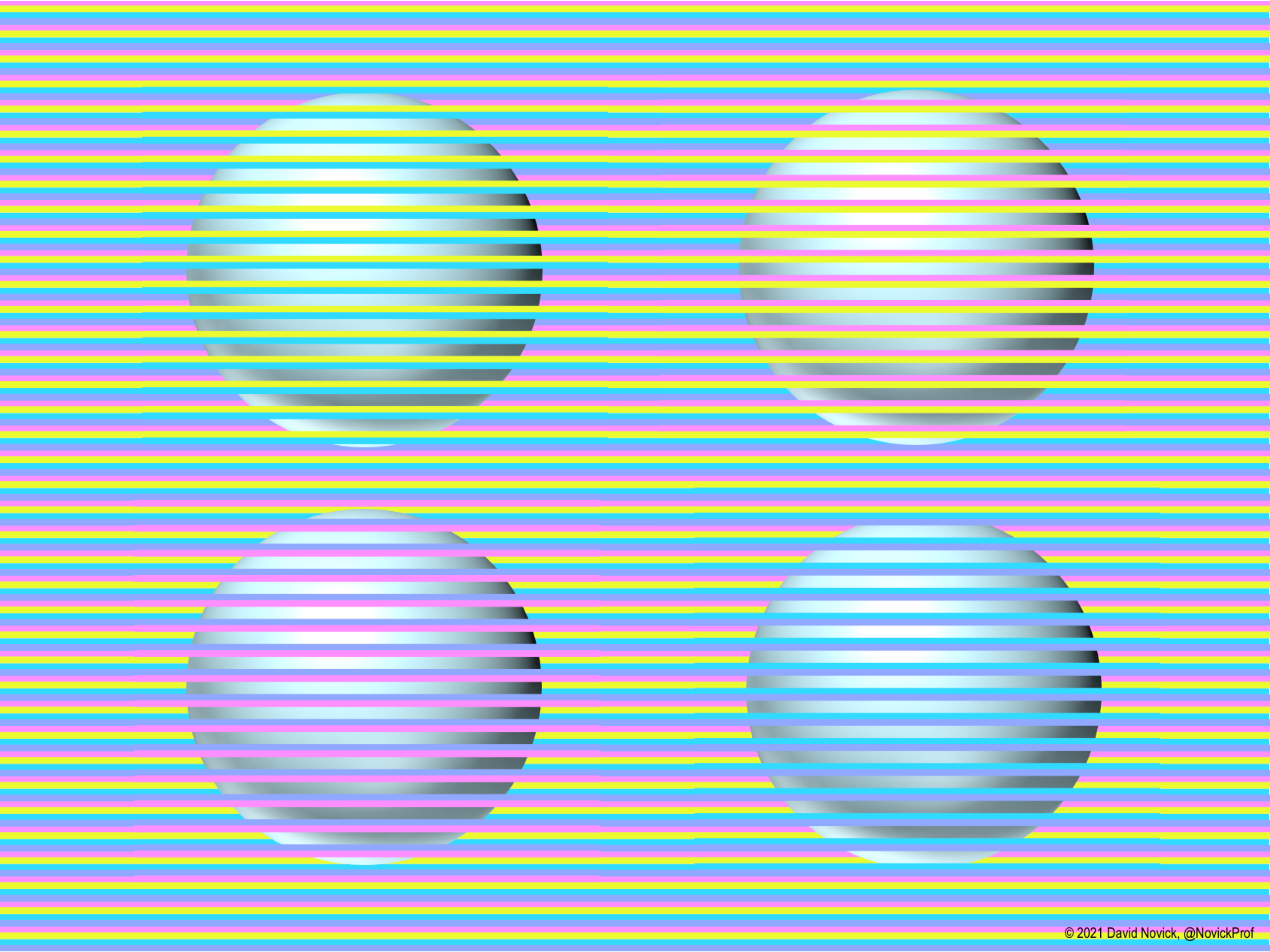
Kitaoka 2020



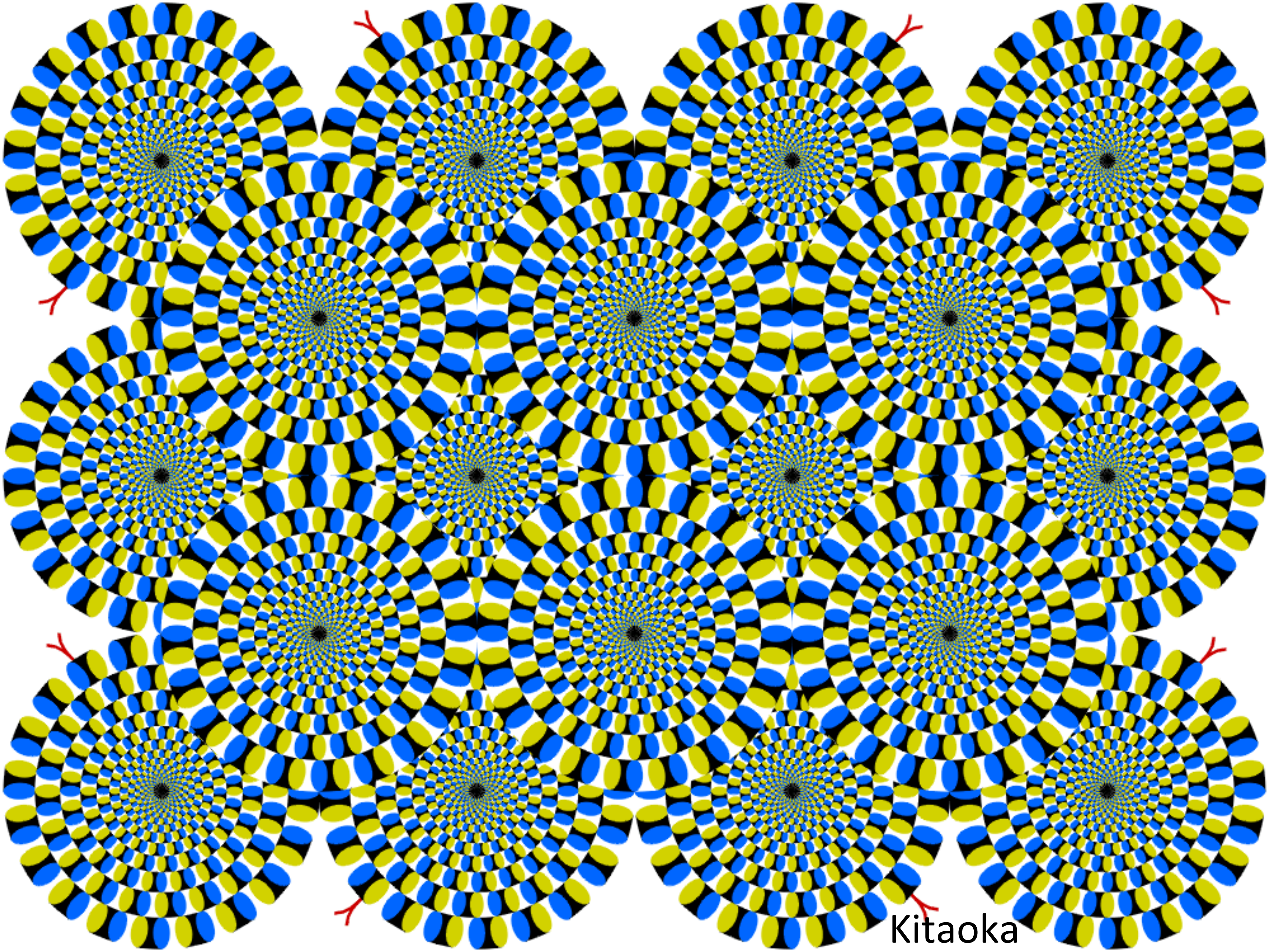
Kitaoka 2020



Kitaoka 2020



© 2021 David Novick, @NovickProf



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) `[-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 `int`s

# 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…

```java
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)

- `int`s aren't integers
  - Think about overflow
- The comparison technique `(i1, i2) -> i1 - i2` requires |i1 - i2| <= `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;
        }
    }
}
```

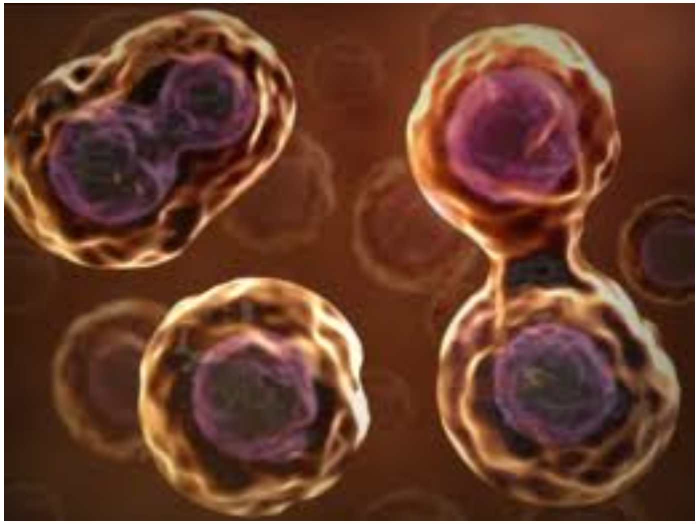
# The moral

- 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);
    }
}
```

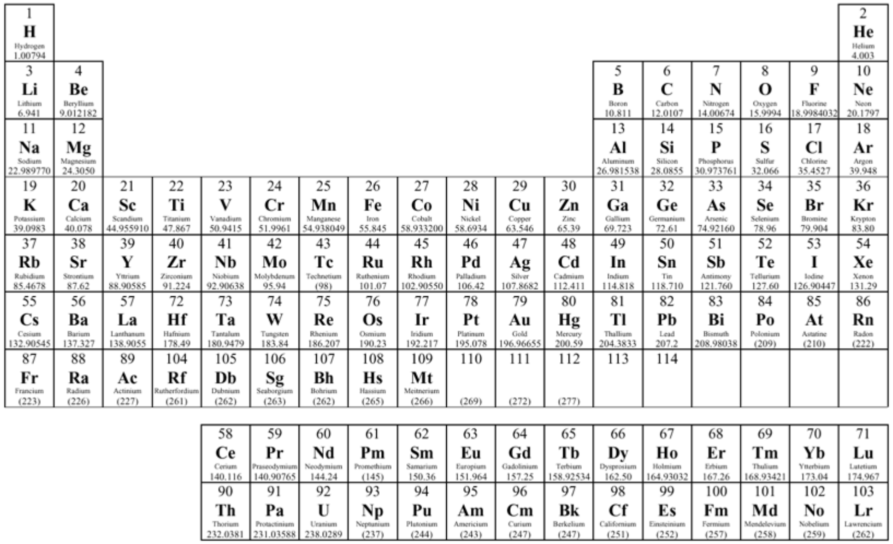
Prints 1000

# 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 + 5432l);
        System.out.println(01234 + 43210);
    }
}
```



The Periodic Table of the Elements

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1<br>H<br>Hydrogen<br>1.00794 | | | | | | | | | | | | | | | | | 2<br>He<br>Helium<br>4.003 |
| 3<br>Li<br>Lithium<br>6.941 | 4<br>Be<br>Beryllium<br>9.012182 | | | | | | | | | | | 5<br>B<br>Boron<br>10.811 | 6<br>C<br>Carbon<br>12.0107 | 7<br>N<br>Nitrogen<br>14.00674 | 8<br>O<br>Oxygen<br>15.9994 | 9<br>F<br>Fluorine<br>18.9984032 | 10<br>Ne<br>Neon<br>20.1797 |
| 11<br>Na<br>Sodium<br>22.989770 | 12<br>Mg<br>Magnesium<br>24.3050 | | | | | | | | | | | 13<br>Al<br>Aluminum<br>26.981538 | 14<br>Si<br>Silicon<br>28.0855 | 15<br>P<br>Phosphorus<br>30.973761 | 16<br>S<br>Sulfur<br>32.066 | 17<br>Cl<br>Chlorine<br>35.4527 | 18<br>Ar<br>Argon<br>39.948 |
| 19<br>K<br>Potassium<br>39.0983 | 20<br>Ca<br>Calcium<br>40.078 | 21<br>Sc<br>Scandium<br>44.955910 | 22<br>Ti<br>Titanium<br>47.867 | 23<br>V<br>Vanadium<br>50.9415 | 24<br>Cr<br>Chromium<br>51.9961 | 25<br>Mn<br>Manganese<br>54.938049 | 26<br>Fe<br>Iron<br>55.845 | 27<br>Co<br>Cobalt<br>58.933200 | 28<br>Ni<br>Nickel<br>58.6934 | 29<br>Cu<br>Copper<br>63.546 | 30<br>Zn<br>Zinc<br>65.39 | 31<br>Ga<br>Gallium<br>69.723 | 32<br>Ge<br>Germanium<br>72.61 | 33<br>As<br>Arsenic<br>74.92160 | 34<br>Se<br>Selenium<br>78.96 | 35<br>Br<br>Bromine<br>79.904 | 36<br>Kr<br>Krypton<br>83.80 |
| 37<br>Rb<br>Rubidium<br>85.4678 | 38<br>Sr<br>Strontium<br>87.62 | 39<br>Y<br>Yttrium<br>88.90585 | 40<br>Zr<br>Zirconium<br>91.224 | 41<br>Nb<br>Niobium<br>92.90638 | 42<br>Mo<br>Molybdenum<br>95.94 | 43<br>Tc<br>Technetium<br>(98) | 44<br>Ru<br>Ruthenium<br>101.07 | 45<br>Rh<br>Rhodium<br>102.90550 | 46<br>Pd<br>Palladium<br>106.42 | 47<br>Ag<br>Silver<br>107.8682 | 48<br>Cd<br>Cadmium<br>112.411 | 49<br>In<br>Indium<br>114.818 | 50<br>Sn<br>Tin<br>118.710 | 51<br>Sb<br>Antimony<br>121.760 | 52<br>Te<br>Tellurium<br>127.60 | 53<br>I<br>Iodine<br>126.90447 | 54<br>Xe<br>Xenon<br>131.29 |
| 55<br>Cs<br>Cesium<br>132.90545 | 56<br>Ba<br>Barium<br>137.327 | 57<br>La<br>Lanthanum<br>138.9055 | 72<br>Hf<br>Hafnium<br>178.49 | 73<br>Ta<br>Tantalum<br>180.9479 | 74<br>W<br>Tungsten<br>183.84 | 75<br>Re<br>Rhenium<br>186.207 | 76<br>Os<br>Osmium<br>190.23 | 77<br>Ir<br>Iridium<br>192.217 | 78<br>Pt<br>Platinum<br>195.078 | 79<br>Au<br>Gold<br>196.96655 | 80<br>Hg<br>Mercury<br>200.59 | 81<br>Tl<br>Thallium<br>204.3833 | 82<br>Pb<br>Lead<br>207.2 | 83<br>Bi<br>Bismuth<br>208.98038 | 84<br>Po<br>Polonium<br>(209) | 85<br>At<br>Astatine<br>(210) | 86<br>Rn<br>Radon<br>(222) |
| 87<br>Fr<br>Francium<br>(223) | 88<br>Ra<br>Radium<br>(226) | 89<br>Ac<br>Actinium<br>(227) | 104<br>Rf<br>Rutherfordium<br>(261) | 105<br>Db<br>Dubnium<br>(262) | 106<br>Sg<br>Seaborgium<br>(263) | 107<br>Bh<br>Bohrium<br>(262) | 108<br>Hs<br>Hassium<br>(265) | 109<br>Mt<br>Meitnerium<br>(266) | 110<br>(269) | 111<br>(272) | 112<br>(277) | 113 | 114 | | | | |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 58<br>Ce<br>Cerium<br>140.116 | 59<br>Pr<br>Praseodymium<br>140.90765 | 60<br>Nd<br>Neodymium<br>144.24 | 61<br>Pm<br>Promethium<br>(145) | 62<br>Sm<br>Samarium<br>150.36 | 63<br>Eu<br>Europium<br>151.964 | 64<br>Gd<br>Gadolinium<br>157.25 | 65<br>Tb<br>Terbium<br>158.92534 | 66<br>Dy<br>Dysprosium<br>162.50 | 67<br>Ho<br>Holmium<br>164.93032 | 68<br>Er<br>Erbium<br>167.26 | 69<br>Tm<br>Thulium<br>168.93421 | 70<br>Yb<br>Ytterbium<br>173.04 | 71<br>Lu<br>Lutetium<br>174.967 |
| 90<br>Th<br>Thorium<br>232.0381 | 91<br>Pa<br>Protactinium<br>231.03588 | 92<br>U<br>Uranium<br>238.0289 | 93<br>Np<br>Neptunium<br>(237) | 94<br>Pu<br>Plutonium<br>(244) | 95<br>Am<br>Americium<br>(243) | 96<br>Cm<br>Curium<br>(247) | 97<br>Bk<br>Berkelium<br>(247) | 98<br>Cf<br>Californium<br>(251) | 99<br>Es<br>Einsteinium<br>(252) | 100<br>Fm<br>Fermium<br>(257) | 101<br>Md<br>Mendelevium<br>(258) | 102<br>No<br>Nobelium<br>(259) | 103<br>Lr<br>Lawrencium<br>(262) |

# What does it print?

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