Turbo Charge CPU Utilization in Fork/Join Using the ManagedBlocker
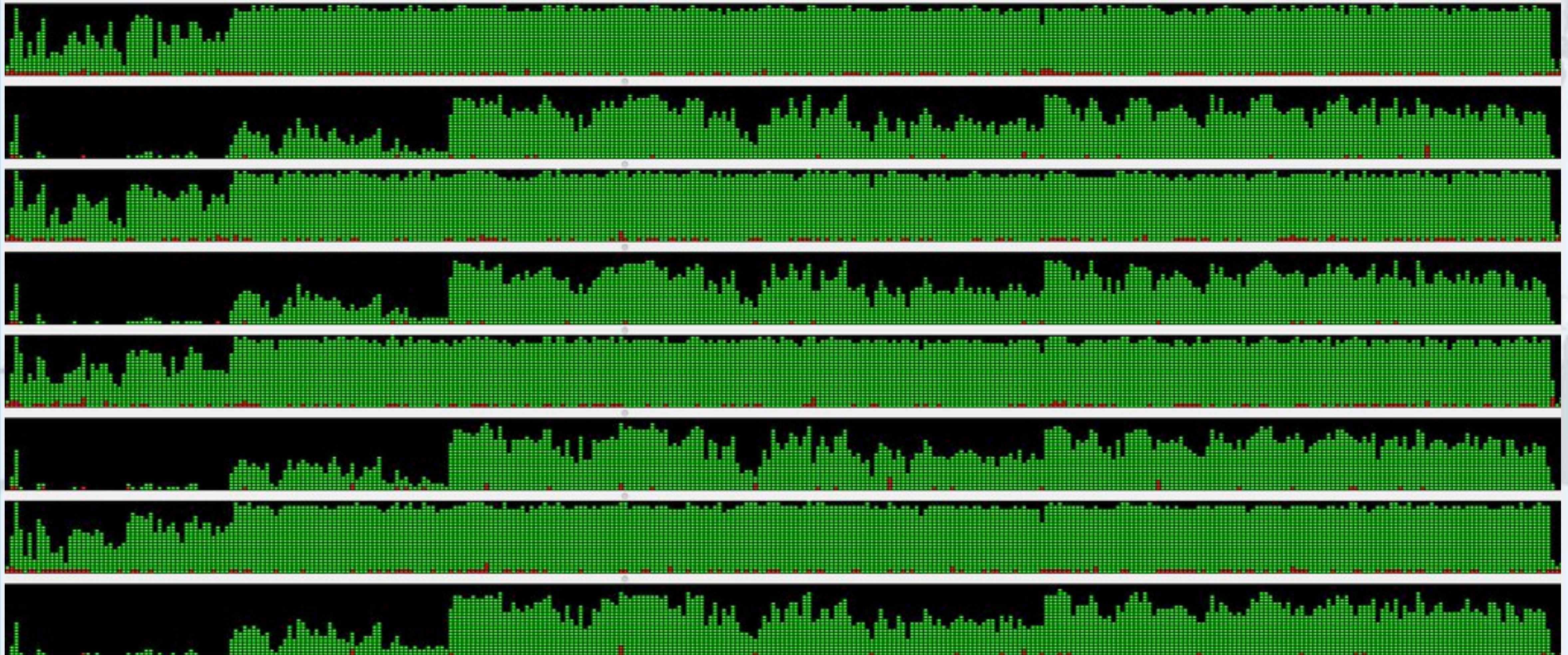
Dr Heinz M. Kabutz

Last Updated 2017-02-07
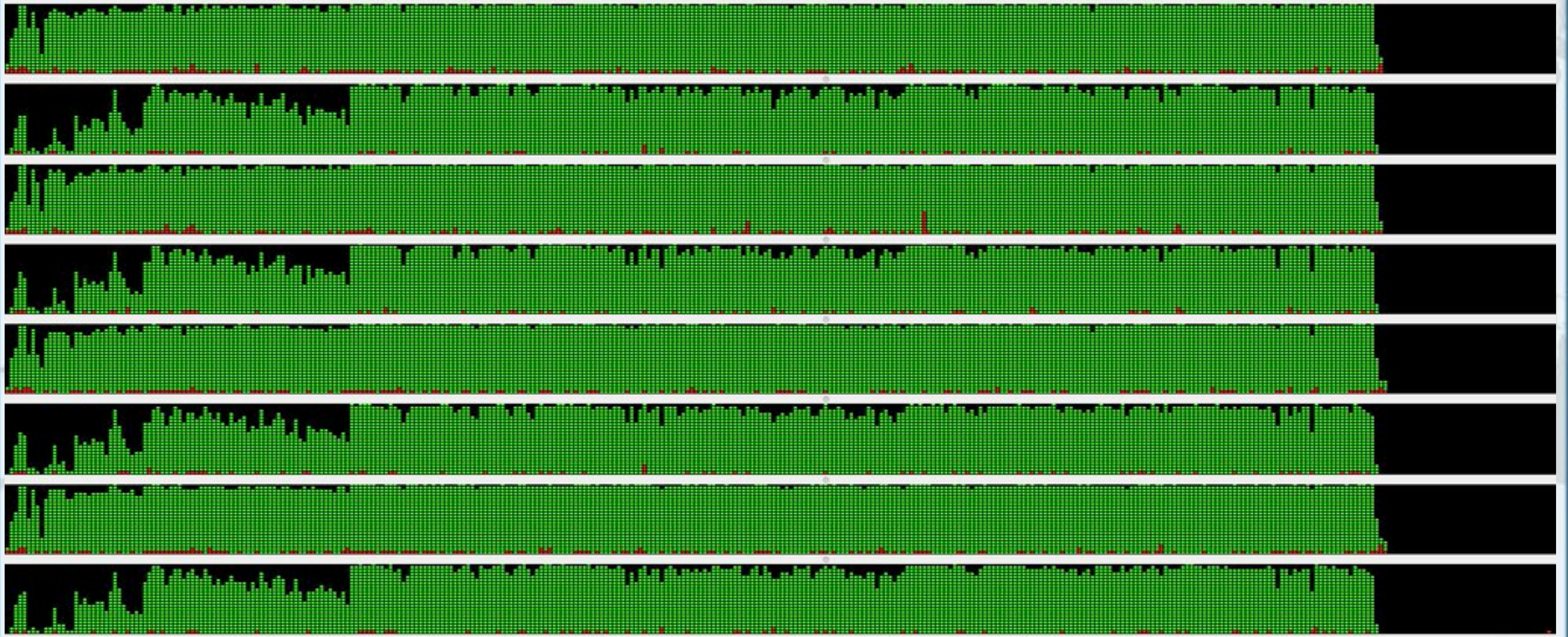
Javaspecialists.eu
java training

Regular

ManagedBlocker
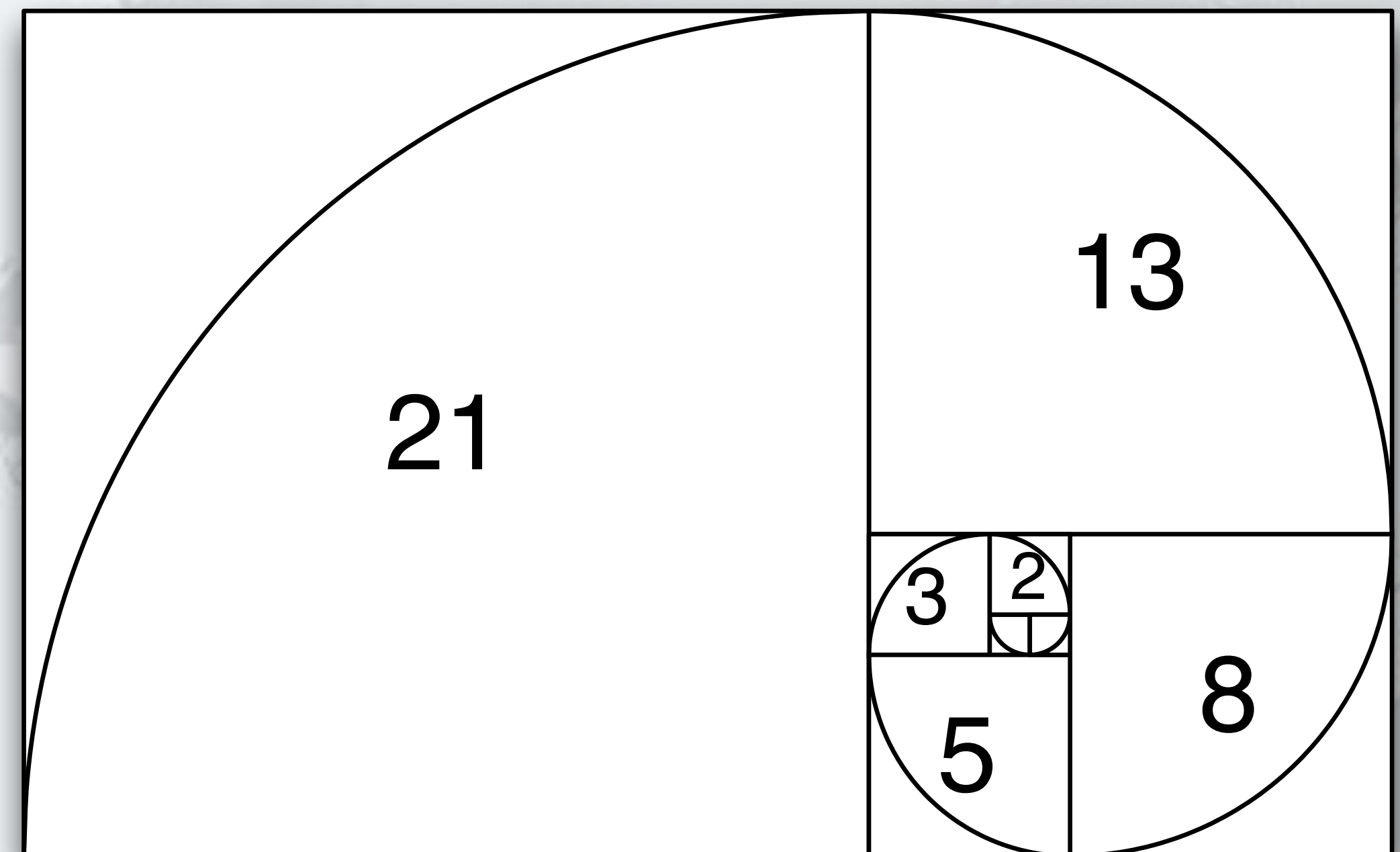
Speeding Up Fibonacci

- **By Leonardo of Pisa**
  - **F0 = 0**
  - **F1 = 1**
  - **Fn = Fn-1 + Fn-2**

- **Thus the next number is equal to the sum of the two previous numbers**
  - **e.g. 0, 1, 1, 2, 3, 5, 8, 13, 21, …**

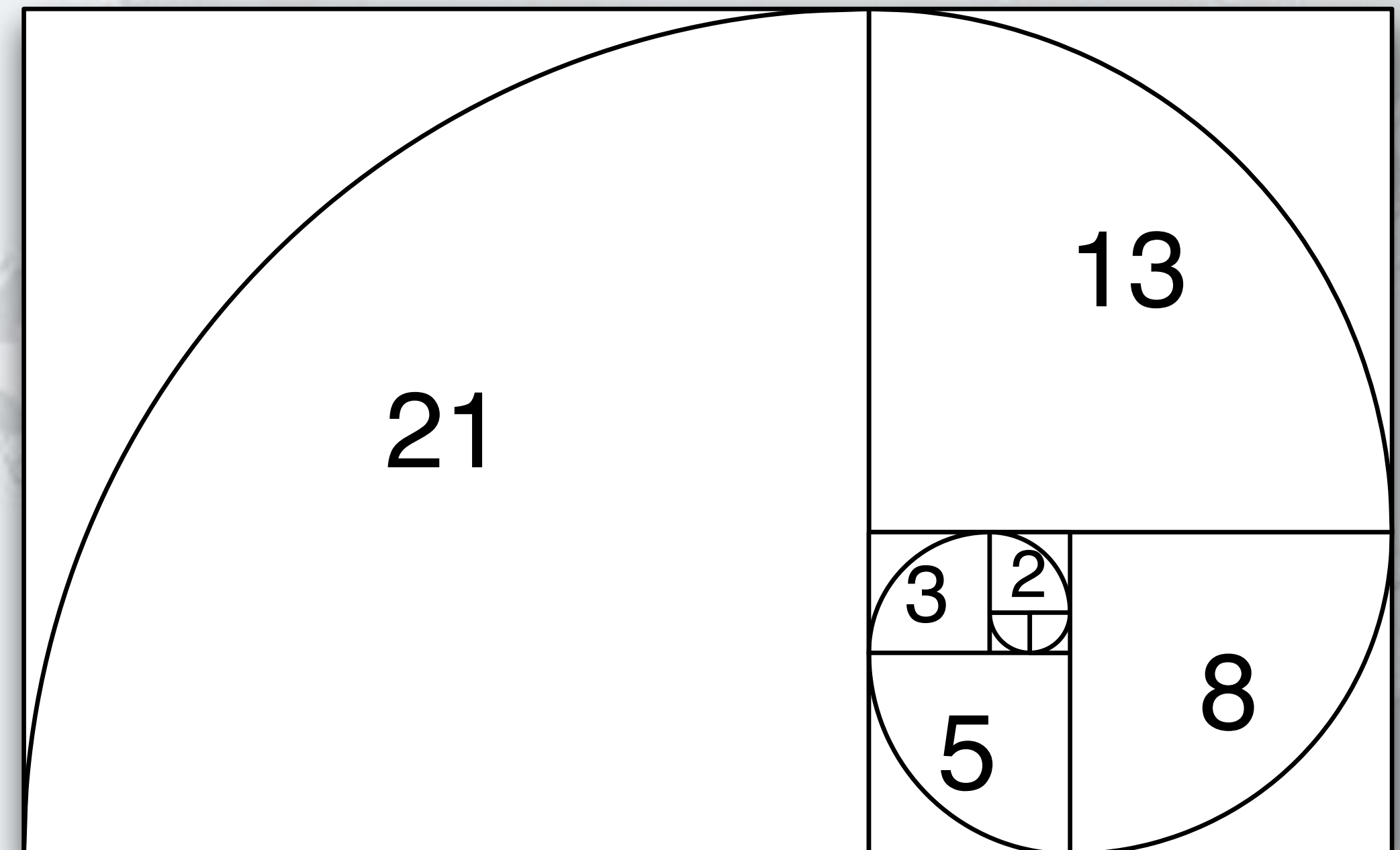- **The numbers get large quickly, like Australian rabbit population did**

Javaspecialists.eu

Naive Implementation

- **Taking our recursive definition**

  - $F_0 = 0$, $F_1 = 1$

  - $F_n = F_{n-1} + F_{n-2}$

- **Converting this into Java:**

```java
public long f(int n) {
    if (n <= 1) return n;
    return f(n–1) + f(n–2);
}
```

- **But this has exponential time complexity, so gets terribly slow**

javaspecialists.eu

2nd Attempt at Coding Fibonacci

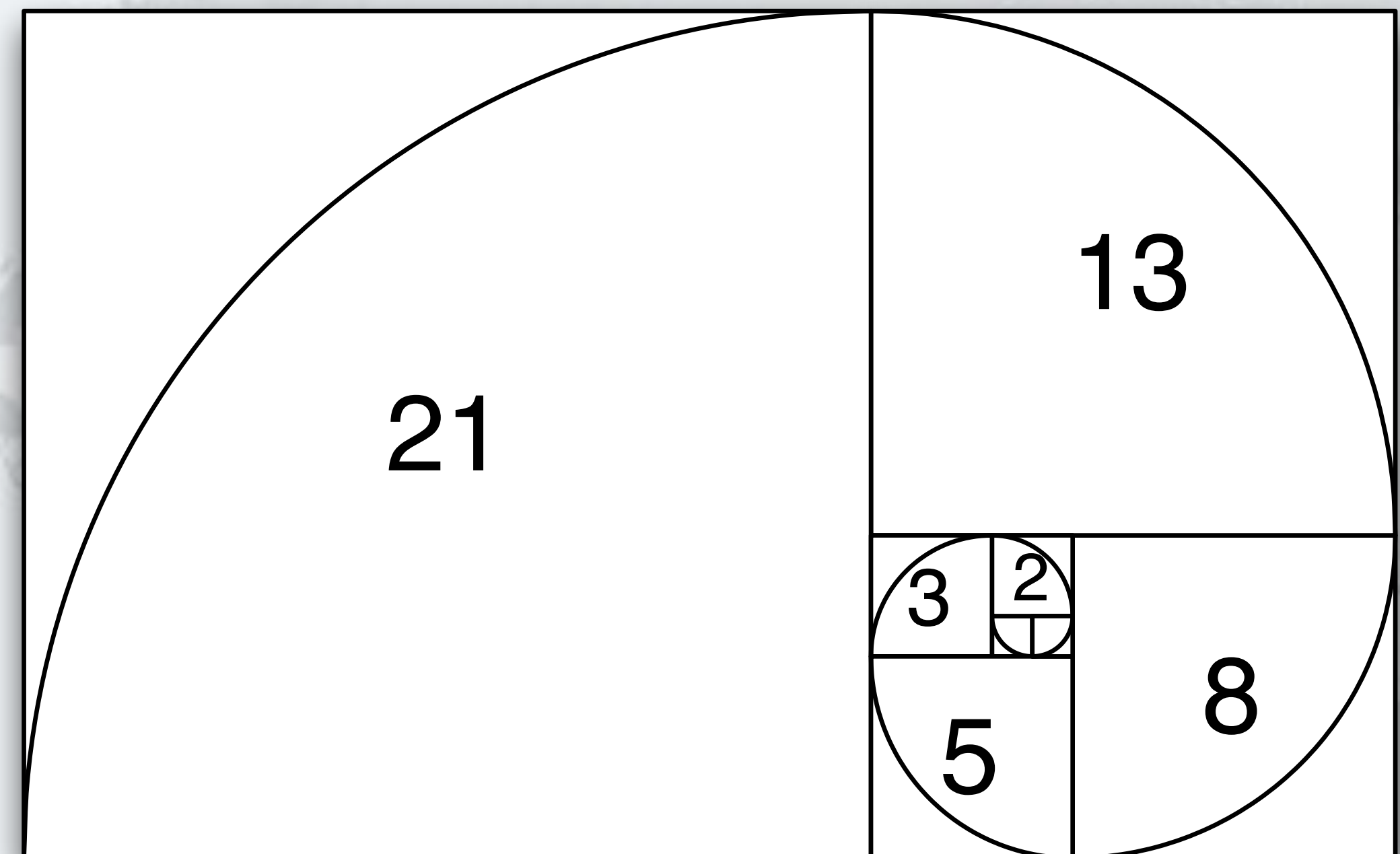- ## Iterative algorithm

```java
public static long f(int n) {
    long n0 = 0, n1 = 1;
    for (int i = 0; i < n; i++) {
        long temp = n1;
        n1 = n1 + n0;
        n0 = temp;
    }
    return n0;
}
```

- ## Linear time complexity

  – **f(1_000_000_000) in 1.7 seconds**

    • **However, long overflows so the result is incorrect**

    • **We can use BigInteger, but its add() is also linear, so time is quadratic**

Javaspecialists.eu

3rd Attempt Dijkstra's Sum of Squares

- **Dijkstra's clever formula**

  – $F_{2n-1} = F_{n-1}^2 + F_n^2$

  – $F_{2n} = (2 \times F_{n-1} + F_n) \times F_n$
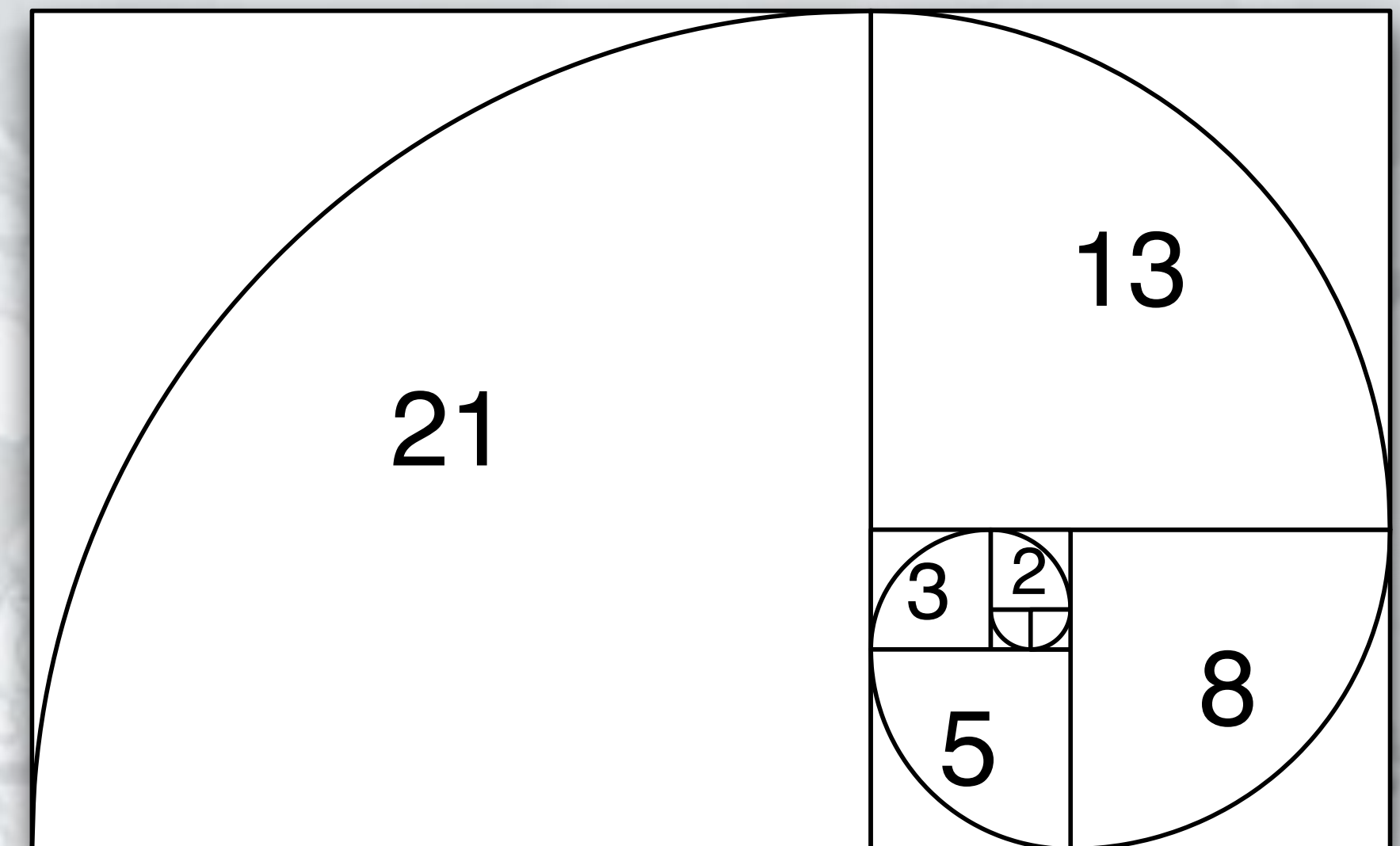
- **Logarithmic time complexity**

  – **Multiply in Java BigInteger**

    - **Karatsuba complexity is O($n^{1.585}$)**

    - **3-way Toom Cook complexity is O($n^{1.465}$)**

    - **Prior to Java 8, multiply() had complexity O(n^2)**

    - **BigInteger.multiply() single-threaded in Java - we'll fix that later**

javaspecialists.eu

javaspecialists.eu

Demo 1: Dijkstra's Sum of Squares

- **We implement this algorithm using BigInteger**

  – $F_{2n-1} = F_{n-1}^2 + F_n^2$

  – $F_{2n} = (2 \times F_{n-1} + F_n) \times F_n$

Demo 2: Parallelize Our Algorithm

- **We can parallelize by using common Fork/Join Pool**

  - **Next we fork() the 1ˢᵗ task, do the 2ⁿᵈ and then join 1ˢᵗ**

```java
ForkJoinTask<BigInteger> f0_task = new RecursiveTask<BigInteger>() {
    protected BigInteger compute() {
        return f(half - 1);
    }
}.fork();
BigInteger f1 = f(half);
BigInteger f0 = f0_task.join();
```

Demo 3: Parallelize BigInteger

- **Let's hack fork/join into:**
  - **multiplyToomCook3()**
  - **squareToomCook3()**

- **These probably won't reach the threshold**
  - **multiplyKaratsuba()**
  - **squareKaratsuba()**

Demo 4: Cache Results

- **Dijkstra's Sum of Squares needs to work out some values several times. Cache results to avoid this.**
  - **Careful to avoid a memory leak**
    - **No static maps**

Demo 5: Reserved Caching Scheme

● **Instead of calculating same value twice:**

– **Use putIfAbsent() to insert special placeholder**

– **If result is null, we are first and start work**

– **If result is the placeholder, we wait**

Demo 6: ManagedBlocker

- **ForkJoinPool is configured with *desired parallelism***
  - **Number of active threads**
  - **ForkJoinPool mostly used with CPU intensive tasks**
- **If one of the FJ Threads has to block, a new thread can be started to take its place**
  - **This is done with the ManagedBlocker**
- **We use ManagedBlocker to keep parallelism high**

Questions?

Dr Heinz M. Kabutz

heinz@javaspecialists.eu

@heinzkabutz

Javaspecialists.eu
java training