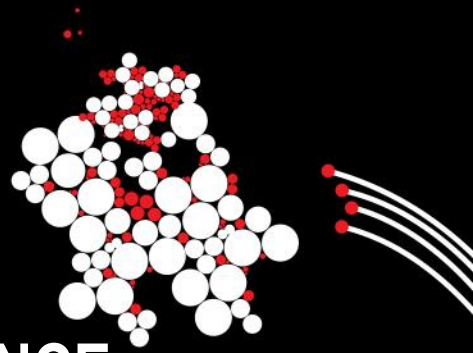
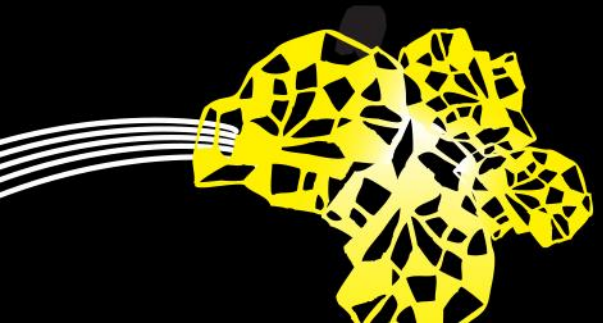
UNIVERSITY OF TWENTE.



# OPTIMIZING JAVA CODE FOR PERFORMANCE

MODULE 2 SOFTWARE SYSTEMS

31 JANUARY 2022



# WHAT DO YOU WANT TO LEARN?

Please go to Wooclap and answer the question:

- What do you want to learn today?

# TOPICS

1. What do we mean with performance
2. When to improve performance
3. How to measure performance
4. Examples of performance improvements (in general and in Java)
5. Performance and multithreading

# WHAT IS PERFORMANCE

What do we/you mean with performance?

- Think what you mean with performance
- Discuss with your neighbour(s)
- Share with everyone

# WHAT IS PERFORMANCE

Do you remember a concrete example of difference in performance in M1 or M2?

# WHAT IS PERFORMANCE

- Time
- Responsiveness
- Memory

# TRADE-OFFS

Do you know examples of:

- Time vs responsiveness
- Time vs memory
- Trading time for time

# THE COST OF PERFORMANCE

Consider different properties of software…

**Question**:

- Give an example of something that is threatened when performance is a priority
- Give an example solution to mitigate this problem

- Think about the answer (30 sec)
- Discuss with your neighbour
- Share with everyone

# DON'T OPTIMIZE EARLY

**Rule 1: Don't optimize early**

First consider functionality;
Then ask yourself: is there really a problem?

| | |
|---|---|
| ▪ Maintainability and readability | first isolate / simplify to minimize impact |
| ▪ Correctness of code | first write correct code and unit tests |
| ▪ Optimizing the wrong thing | use a profiler to find "hot spots" |
| ▪ Other code gets slower | write automated tests for performance |

# THE NUMBERS TELL THE TALE

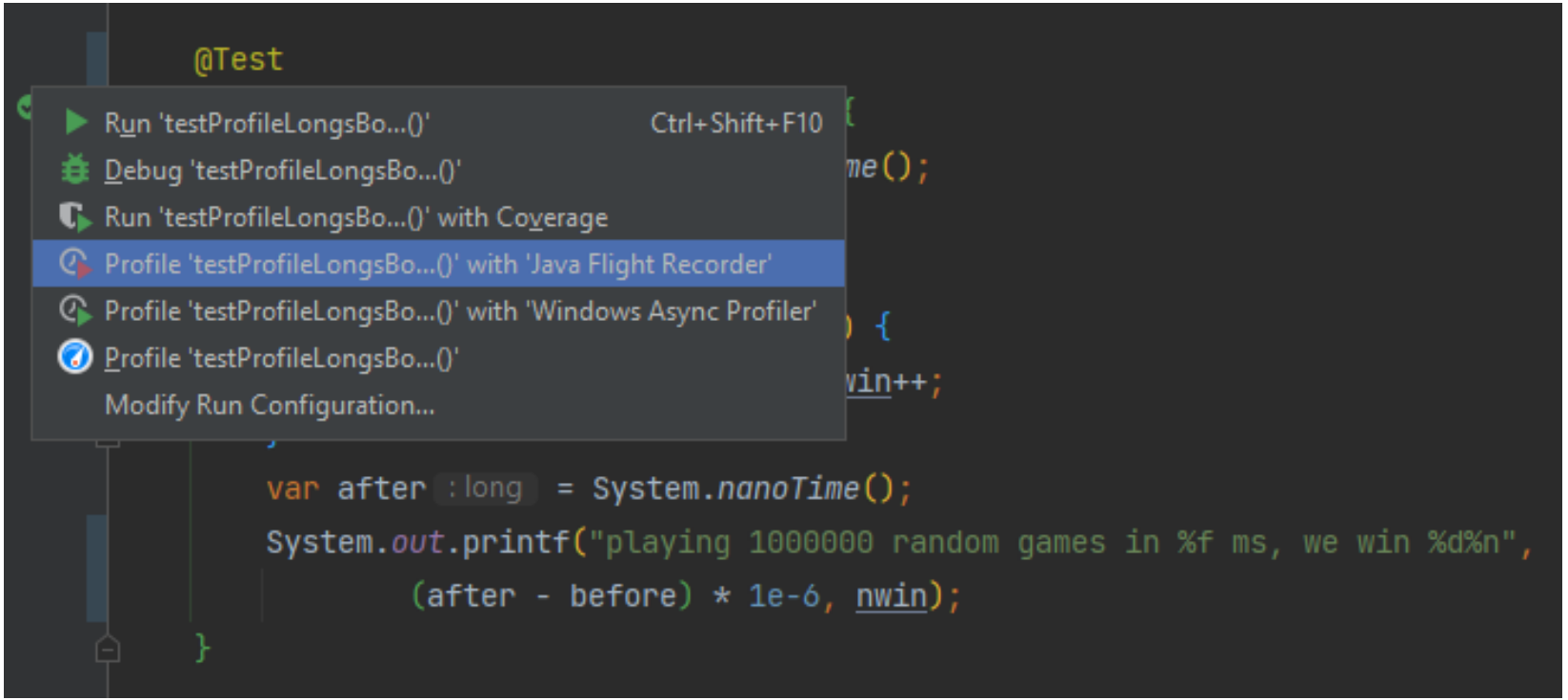
**Rule 2: The numbers tell the tale (meten is weten)**

- Write unit tests so code is still correct
- Measure performance with unit tests
  - Use `System.nanoTime()` to measure elapsed time
  - Use the git commit history to track improvements
- Use a profiler to find “hot spots”
  - Java Flight Recorder, Async Profiler (see IntelliJ)

# THE NUMBERS TELL THE TALE

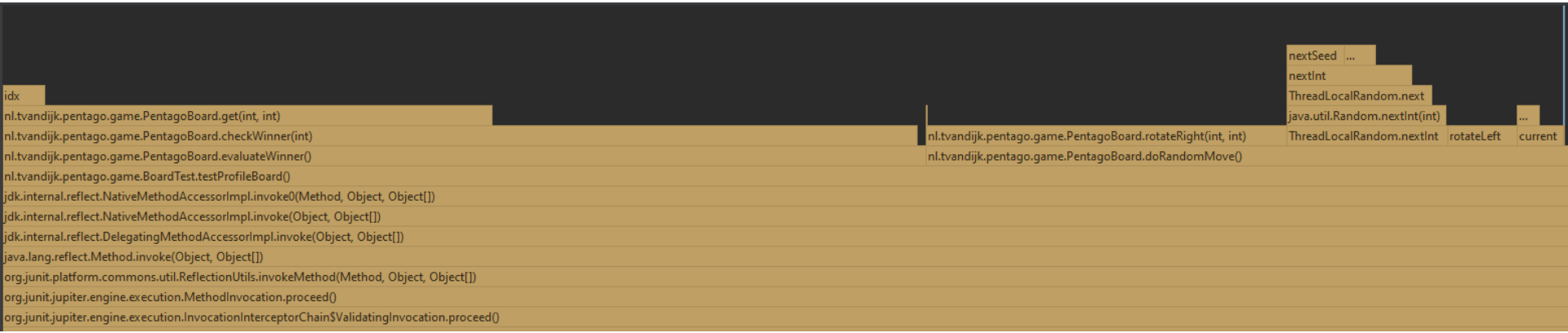
```java
@Test
void testProfileLongsBoardPlayout() {
    var before :long = System.nanoTime();
    long nwin = 0;
    var b = new PentagoLongsBoard();
    for (int i = 0; i < 1000000; i++) {
        if (b.doRandomPlay() == 1) nwin++;
    }
    var after :long = System.nanoTime();
    System.out.printf("playing 1000000 random games in %f ms, we win %d%n",
            (after - before) * 1e-6, nwin);
}
```

# THE NUMBERS TELL THE TALE



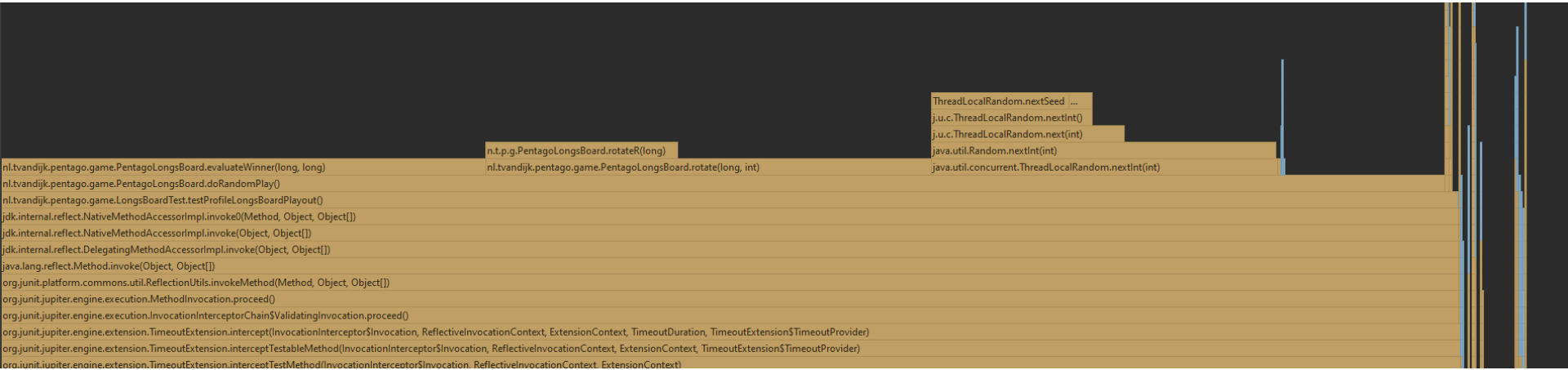
# THE NUMBERS TELL THE TALE

- 49% `evaluateWinner` of which 53% in `get`
- 48% in `doRandomMove`
  - 48% `rotateRight/Left`
  - 17% `Random.nextInt`()



# THE NUMBERS TELL THE TALE

- 33% `evaluateWinner`
- 31% `rotate`
- 24% `Random.nextInt()`
- ~12% rest of `doRandomPlay()`



# WHAT IS EXPENSIVE

On Wooclap, please provide examples of things that are **expensive / bad for performance**

(either time or memory)

# WHAT IS EXPENSIVE

- I/O (network, files)
- Waiting for user input
- Memory reading and especially writing
  - Creating objects
  - Following references
- Branching, method calls, recursion
- Many threads using the same lock

# STARTING PRINCIPLES

**Rule 3: Keep it simple**

- First understand the code / problem
- Simplify, refactor, clean up
  - Isolate the problem
  - Minimize side effects
  - Remove wasteful code

# MEMORY EFFICIENCY

**Rule 4: Datastructures before algorithms**

- Memory/information efficiency ➔ speed
- `ArrayList` vs `LinkedList`
- Use `Maps` (examples?)
- Be careful with `contains` (list vs set)
- Use external libraries with optimized datastructures

# MEMORY EFFICIENCY

**Rule 4: Datastructures before algorithms**

- Memory/information efficiency ➔ speed
- Only store what you need
  - Avoid creating a lot of objects (why? how can this happen?)
  - Avoid creating copies (how could this happen?)
  - Avoid indirections (what is this? how to prevent it?)
  - Prefer primitive types over objects (why?)
  - Bit-level efficiency
  - Store related information nearby (memory locality)

# I/O

Anyone: How to be efficient with I/O? What have you learned in M2?

- Avoid accessing disk/network
- Use buffers (BufferedReader, BufferedWriter)
- Use threads (this is why servers use threads)

# IMPROVING AT CODE LEVEL

- Method calls are expensive (but inlining, JIT)
- Avoid recursion, use loops (example?)
- Use a cache to avoid recomputing (example?)
- Switches are sometimes faster than repeated if-else statements
- Be careful with loops
  - Recomputing `size()`
  - Recomputing something every iteration
- Avoid branches if possible

# JAVA SPECIFIC

- Java 8 streams are convenient but also slower (any guess why?)
- For-each loops are slightly slower than numeric for loops (except `LinkedList`, why?)
- `ArrayList` vs `LinkedList`
  - Iterating
  - Modifications
  - Contains is always slow
- Overhead of exceptions vs return values (any guess why?)
- Use a `StringBuilder` if you lose a lot of time concatenating `Strings`
- Avoid `Vector` and `Stack`
- `Scanner` is convenient but slow
- Avoid `Random`

# MULTITHREADING

Wooclap: What is the major source of bad performance when multithreading?

- Minimize interaction between threads
- Avoid locks
  - Don't make everything `synchronized`
  - Use short-lived, local locks
- Use atomic variables: `AtomicInteger`, etc.
- Specialized datastructures for "scalable" performance:
  - `ConcurrentHashMap`
  - `BlockingQueue`
- Not discussed: parallel computation

# THE NUMBERS TELL THE TALE

Playing 1,000,000 random games of Pentago:

- Before: 5059 ms
- After: 1611 ms

Applied techniques:

- Primitive types (no objects)
- Bitboard (two bits per marble) with as few bitwise operations as possible
- Minimize branches
- Avoid object creation/copying
- Faster random number generator (`ThreadLocalRandom` or `SplittableRandom`)
- Precomputation/caching for efficient win check

# WRAPPING UP

- Which three types of performance were discussed?
- What are the four rules of optimizing for performance?
- How to make multithreaded code fast?

- Did you learn what you wanted to learn?