# Concurrency in Java


**Dozent:** Prof. Dr. Michael Eichberg
**Kontakt:** michael.eichberg@dhbw.de

**Version:** 1.0.3.1

---

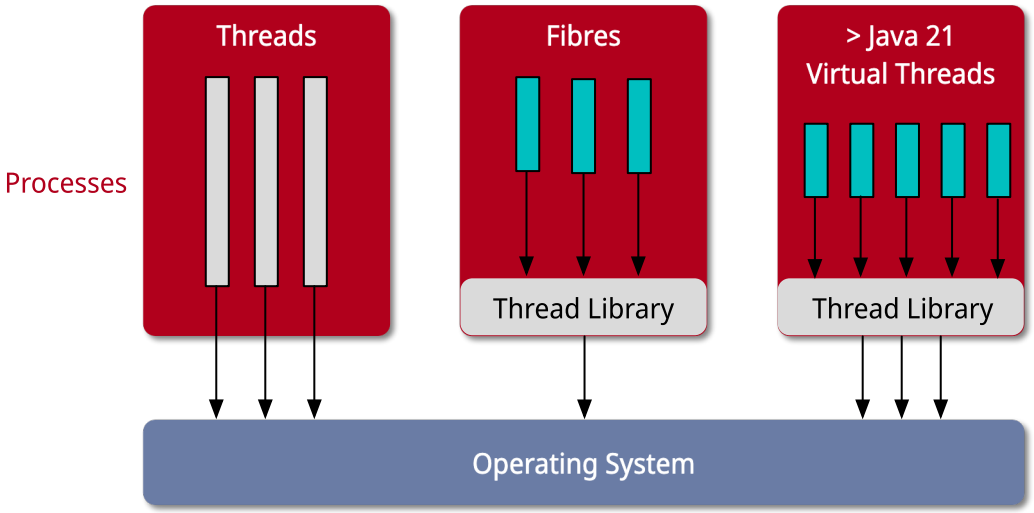
**Slides/Scripts:** https://delors.github.io/ds-concurrency-in-java/folien.en.md.html
https://delors.github.io/ds-concurrency-in-java/folien.en.md.html.pdf

**Reporting errors:** https://github.com/Delors/delors.github.io/issues

**AI Usage:** AI assistants (in particular Claude, but also ChatGPT, Ollama with Gemma/LLama/Qwen, OpenCode with Kimi, etc.) were used to assist in the creation of the slides. This was done, for example, to efficiently generate graphics (i.e. SVG files), or to generate summary tables. Furthermore, AI was used for general quality assurance. Content that may have been suggested by an AI was explicitly validated and adapted whenever it was incorporated.

A good understanding of concurrent programming is essential for the development of distributed applications, as servers always process several requests simultaneously.

# Processes vs. threads




- Processes are isolated from each other and can only communicate with each other via explicit mechanisms; processes do not share the same address space.
- All threads of a process share the same address space. *Native threads* are threads supported by the operating system that are managed directly by the operating system. Standard Java threads are *native threads*.
- Fibres* (also *coroutines*) always use cooperative multitasking. This means that a fibre explicitly passes control to another fibre. (Formerly also referred to as *green threads*.) These are invisible to the operating system.
- As of Java 21, Java not only supports classic (native) threads but also virtual threads (which are "somewhere" between green threads and native threads. The latter in particular allow very natural programming of middleware that takes care of parallelization/concurrency.

# Uncontrolled Parallelization

```
1  int counter = 0; // or long, or float, or ...
2
3  void main() throws InterruptedException {
4      Runnable increment = () -> {
5          for (int i = 0; i < 100_000; i++)
6              counter++;
7      };
8
9      var t1 = new Thread(increment);
10     var t2 = new Thread(increment);
11     t1.start();      t2.start();
12     t1.join();       t2.join();
13
14     IO.println("Counter: " + counter);
15 }
```

**? Question**

What value will the counter have when both threads have ended?

# Fundamental Problems in Concurrent Programming

Concurrent access to shared mutable state introduces three classes of problems:

**Race Condition:** *Read / modify / write* of a variable is non-atomic.

**Visibility:** Reading of *stale values across threads*.

**Ordering:** *Instructions do not execute in source order*.


Summary

All three problems share the same root cause:

**Unsynchronised access to shared mutable state**.


---

A **Race Condition** can happen, because a compound operation such as `counter++` consists of three separate steps (read, increment, write). When the scheduler preempts a thread between any two of these steps, another thread can operate on a stale value — an increment is silently lost.

Reading of *stale values across threads* is likely because modern CPUs (since the 1990s) cache values in registers or L1/L2 caches. A write by thread A may never reach main memory before thread B reads the same variable — thread B therefore operates on an arbitrarily old value.

Both the compiler (in case of Java the Just-In-Time Compiler not the frontend compiler) and the CPU may reorder instructions for optimisation. What *looks sequential in source code may execute in a completely different order at runtime*. This can make partially constructed objects visible to other threads.

Hint

A **monitor** eliminates all three — mutual exclusion prevents races, and the *happens-before* guarantee provided by Java's memory model ensures both visibility and ordering, as we will discuss next.

Remark

In Java "happens-before" does not mean "immediately visible across all cores at the instant of the write". It means: **if you observe the write, you observe everything that preceded it.**

## ✎ Review Question

Would it make a difference if we only loop at most 60 times and the type of the variable is the primitive type `byte`?

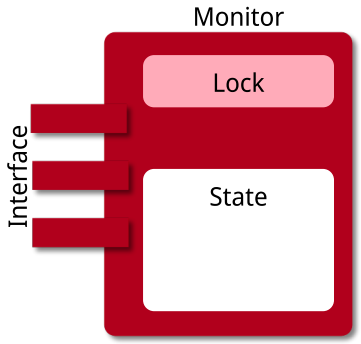
```
byte counter = 0;

void main() throws InterruptedException {
    Runnable increment = () -> {
        for (int i = 0; i < 60; i++)
            counter++;
    };

    long runs = 0;
    do {    runs++;
            counter = 0;
            var t1 = new Thread(increment);     var t2 = new Thread(increment);
            t1.start();                         t2.start();
            t1.join();                          t2.join();
    } while (counter == 120);
    IO.println("Needed runs: " + runs);
}
```

# Communication and synchronization with the help of *monitors*

A *monitor* is an object in which the methods are executed in *mutual exclusion*.




## Condition synchronization

- expresses a condition on the order in which operations are executed.
- For example, data can only be removed from a buffer once data has been entered into the buffer.
- Java only supports one (anonymous) condition variable per monitor, with the classic methods `wait` and `notify` or `notifyAll`.

---

**⚠ Warning**

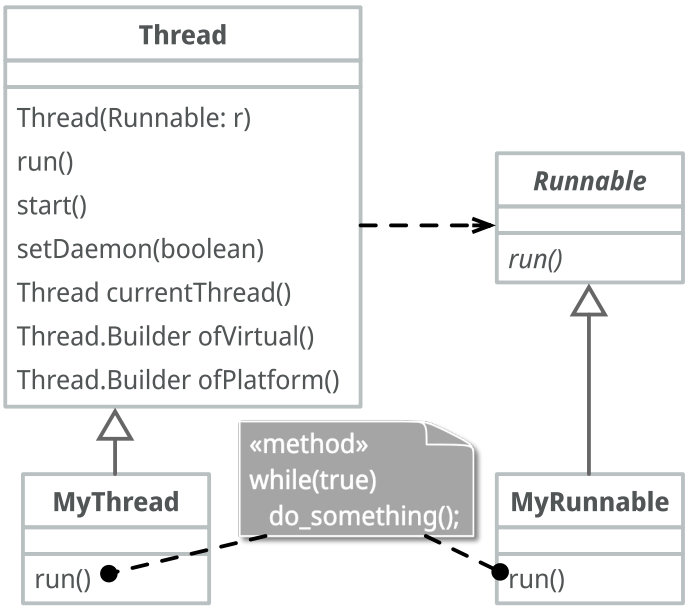
In Java, mutual exclusion only takes place between methods that have been explicitly declared as `synchronized`.

*Monitors* are just one model (alternatives: *Semaphores*, *Message Passing*) that enables the communication and synchronization of threads. It is the standard model in Java and is directly supported by the Java Virtual Machine (JVM).

# Communication between threads with the help of monitors

- By reading and writing data encapsulated in shared objects that are protected by monitors.
- Each object is implicitly derived from the class `java.lang.Object`, which defines a mutual exclusion lock.
- Methods in a class can be marked as `synchronized`. The method is only executed when the lock is present. It waits until then. This process happens automatically.
- The lock can also be acquired via a `synchronized` statement that names the object.
- A thread can wait for a single (anonymous) condition variable and notify it.

# Concurrency in Java




---

- Threads are provided in Java via the predefined class *java.lang.*Thread.
- Alternatively, the interface:

  ```
  public interface Runnable { void run(); }
  ```

  can be implemented and an instance can then be passed to a Thread-Objekt.
- Threads only start their execution when the `start` method is called in the thread class. The `thread.start` method calls the `run` method. Calling the `run` method directly does not lead to parallel execution.
- The current thread can be determined using the static method Thread.*currentThread*().
- A thread is terminated when the execution of its run method ends either normally or as the result of an unhandled exception.
- Java distinguishes between *user* threads and *daemon* threads.

  *Daemon threads* are threads that provide general services and are normally never terminated.

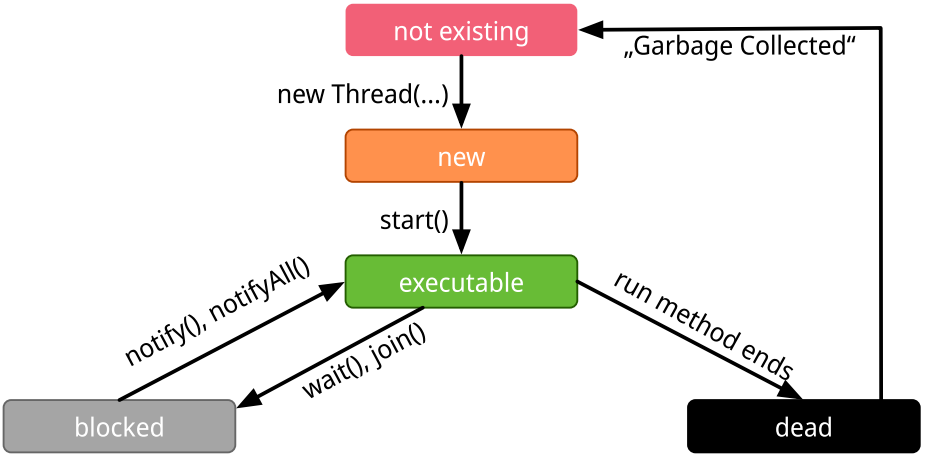
  When all user threads are terminated, the daemon threads are terminated by the JVM and the main programme is terminated.

  The method `setDaemon` must be called before the thread is started.
- The Thread.*stop*() method was removed in Java 26, after being deprecated in Java 1.2 (December 1998).

## Inter-thread communication and coordination

- A thread can wait (with or without a timeout) for another thread (the target) to finish, by calling the `join` method of the target thread.
- A thread can use the `isAlive` method to determine whether the target thread has ended.

# Java Thread States



## `synchronized`-Methods and `synchronized`-Blocks

- A mutual exclusion lock is assigned to each object. The lock cannot be accessed explicitly by the application. This happens implicitly if:
  - a method uses the method modifier `synchronized`
  - block synchronization with the keyword `synchronized` is used
- If a method is marked as `synchronized`, the method can only be accessed if the system has received the lock.
- Therefore, `synchronized` methods have mutually exclusive access to the data encapsulated by the object, if this data is only accessed in other synchronized contexts.
- Non-`synchronized` methods do not require a lock and can therefore be called *at any time*.

# Example: synchronized method

```
public class SynchronizedCounter {

  private int count = 0;

  public synchronized void increment() {
    count++;
  }

  public synchronized int getCount() {
    return count;
  }
}
```

```
public class SharedLong {

  private long theData; // reading (or writing) a single
                        // non-volatile long is not atomic

  public SharedLong(long initialValue) {
    theData = initialValue;
  }

  public synchronized long read() { return theData; }

  public synchronized void write(long newValue) { theData = newValue; }

  public synchronized void incrementBy(long by) {
    theData = theData + by;
  }
}

SharedLong myData = new SharedLong(42);
```

```
public class SynchronizedCounter {

  private int count = 0;

  public void increment() {
    synchronized(this) {
      count++;
    }
  }

  public int getCount() {
    synchronized(this) {
      return count;
    }
  }
}
```

---

**⚠ Warning**

When `synchronized` is used in all its generality, it can undermine one of the advantages of classic monitors: The encapsulation of synchronization constraints associated with an object in a single place in the program!

This is because it is not possible to understand the synchronization associated with a particular object just by looking at the object itself. Other objects can use a `synchronized` block in relation to the object.

## Complex return values

```
public class SharedCoordinate {

  private int x, y;

  public SharedCoordinate(int initX, int initY) {
    this.x = initX; this.y = initY;
  }

  public synchronized void write(int newX, int newY) {
    this.x = newX; this.y = newY;
  }

  /*⚠*/ public /* synchronized irrelevant */ int readX() { return x; } /*⚠*/
  /*⚠*/ public /* synchronized irrelevant */ int readY() { return y; } /*⚠*/

  public synchronized SharedCoordinate read() {
    return new SharedCoordinate(x, y);
} }
```

---

The two methods: `readX` and `readY` are not synchronized, as reading `int` values is atomic. However, they do allow an inconsistent state to be read! It is conceivable that the corresponding thread is interrupted directly after a `readX` and another thread changes the values of x and y. If the original thread is then continued and calls `readY`, it receives the new value of y and thus has a pair of x, y that never existed in this form.

A consistent state can only be determined by the method `read`, which reads the values of x and y in one step and returns them as a pair.

If it can be ensured that a reading thread names the instance in a `synchronized` block, then the reading of a consistent state can also be ensured for several consecutive method calls.

```
SharedCoordinate point = new SharedCoordinate(0,0);
synchronized (point1) {
  var x = point1.readX();
  var y = point1.readY();
}
// do something with x and y
```

However, this "solution" is very dangerous, as the probability of programming errors is *very high* and this can lead to either *race conditions* (here) or *deadlocks* (in general).

# Conditional synchronization

For the purpose of conditional synchronisation, the methods `wait`, `notify` and `notifyAll` can be used in Java. These methods allow you to wait for certain conditions and notify other threads when the condition has changed.

- These methods can only be used within methods that hold the object lock; otherwise a `IllegalMonitorStateException` is thrown.
- The `wait` method always blocks the calling thread and releases the lock associated with the object.
- The `notify` method wakes up *a* waiting thread. Which thread is woken up is not specified.

  `notify` does not release the lock; therefore, the awakened thread must wait until it can receive the lock before it can continue.
- Use `notifyAll` to wake up all waiting threads.

  If the threads are waiting due to different conditions, `notifyAll` must always be used.
- If no thread is waiting, then `notify` and `notifyAll` have no effect.

!! Important

When a thread is woken up, it cannot assume that its condition has been fulfilled!

The condition must always be checked in a loop and the thread may have to be put back into the wait state.

# Example: Synchronisation with *condition variables*

If a thread is waiting for a condition, no other thread can wait for the other condition.

With the primitives presented so far, direct modelling of this scenario is not possible. Instead, all threads must always be woken up to ensure that the intended thread is also woken up. This is why it is also necessary to check the condition in a loop.

A *BoundedBuffer* traditionally uses two condition variables: *BufferNotFull* und *BufferNotEmpty*.

```
1  public class BoundedBuffer {
2    private final int buffer[];
3    private int first;
4    private int last;
5    private int numberInBuffer = 0;
6    private final int size;
7
8    public BoundedBuffer(int length) {
9      size = length;
10     buffer = new int[size];
11     last = 0;
12     first = 0;
13   };
14   public synchronized void put(int item) throws InterruptedException {
15     while (numberInBuffer == size)
16       wait();
17     last = (last + 1) % size;
18     numberInBuffer++;
19     buffer[last] = item;
20     notifyAll();
21   };
22   public synchronized int get() throws InterruptedException {
23     while (numberInBuffer == 0)
24       wait();
25     first = (first + 1) % size;
26     numberInBuffer--;
27     notifyAll();
28     return buffer[first];
29   }
30 }
```

Error situation that could occur when using `notify` instead of `notifyAll`:

```
1 BoundedBuffer bb = new BoundedBuffer(1);
2 Thread g1,g2 = new Thread(() => { bb.get(); });
3 Thread p1,p2 = new Thread(() => { bb.put(new Object()); });
4 g1.start(); g2.start(); p1.start(); p2.start();
```

| | Operation | Change of State of the Buffer | Waiting for the lock | Waiting for the condition |
|---|---|---|---|---|
| 1 | **g1:bb.get()** g2:bb.get(), p1:bb.put(), p2:bb.put() | empty | {g2,p1,p2} | {g1} |
| 2 | **g2:bb.get()** | empty | {p1,p2} | {g1,g2} |
| 3 | **p1:bb.put()** | empty → not empty | {p2,g1} | {g2} |

| | Operation | Change of State of the Buffer | Waiting for the lock | Waiting for the condition |
|---|---|---|---|---|
| 4 | **p2:bb.put()** | not empty | {g1} | {g2,p2} |
| 5 | **g1:bb.get()** | not empty → empty | {g2} | {p2} |
| 6 | **g2:bb.get()** | empty | ∅ | {g2,p2} |

---

In step 5, the VM woke up the `g2` thread - instead of the `p2` thread - due to the call of `notify` by `g1`. The awakened thread `g2` checks the condition (step 6) and realises that the buffer is empty. It goes back to the wait state. Now both a thread that wants to write a value and a thread that wants to read a value are waiting.

# Java's `volatile` Keyword — Lightweight Visibility Guarantee

- Declaring a field `volatile` guarantees *visibility*: every write is immediately flushed to main memory; every read fetches the current value from main memory.
- It also guarantees *ordering*: a write *happens-before* every subsequent read of the same variable.

> ▲ Attention!
>
> Usage of `volatile` does **not** guarantee **atomicity**: compound operations such as `counter++` (read / modify / write) are still unsafe.

---

Behind the scenes, `volatile` inserts *memory barriers* that prevent both the Just-In-Time-Compiler (JIT Compiler) and the CPU from reordering instructions across the barrier.

Hence, `volatile` eliminates only two of the three fundamental problems: visibility and ordering. To eliminate all three a *monitor* (`synchronized`) or an atomic type (e.g. `AtomicInteger`) is required.

## Rule of thumb

`volatile` is sufficient when *exactly one thread writes* and one or more threads only read — with no read/modify/write involved.

## Using `volatile` — the Stop-Flag Pattern

### *Insufficiently* Synchronized Example

```
1  boolean running = true;              // shared variable
2
3  void main() throws InterruptedException {
4      var worker = new Thread(() -> {
5          while (running) {            // read
6          }
7      });
8      worker.start();
9      Thread.sleep(1000);
10     running = false;                  // write
11 }
```

### Correctly Synchronized Example

```
1  volatile boolean running = true;     // evey read/write hits main memory
2
3  void main() throws InterruptedException {
4      var worker = new Thread(() -> {
5          while (running) {            // read from main memory
6          }
7      });
8      worker.start();
9      Thread.sleep(1000);
10     running = false;                 // write to main memory
11 }
```

> ▤ Summary
>
> `volatile` is correct here because only the *main thread writes* and only the *worker thread reads* — no compound operation is involved.

Without `volatile`, the JIT compiler is free to hoist the `running` read out of the loop — it has no evidence that another thread will ever change it. The worker may therefore spin forever even after the flag is set to `false`.

Note that `volatile` does *not* guarantee *when* the worker reacts to the flag change — it only guarantees that the write will eventually (and typically immediately) be seen.

## ✎ Review Question

0.1. Why is `volatile int counter` still broken for the counting example, even though `volatile boolean running` is correct for the stop-flag?

# 1. Advanced synchronisation mechanisms, primitives and concepts.

# Java API for concurrent programming

**java.util.concurrent:**

Provides various classes to support common concurrent programming paradigms, e.g. support for *BoundedBuffers* or thread pools.

**java.util.concurrent.atomic:**

Provides support for *lock-free*, thread-safe programming on simple variables - such as atomic integers.

**java.util.concurrent.locks:**

Provides various lock algorithms that complement the Java language mechanisms, e.g. read-write locks and conditional variables. This enables, for example: "Hand-over-Hand" or "Chain Locking".

# Example: Synchronization with *ReentrantLock*s.

A *BoundedBuffer*, for example, traditionally has two condition variables: *BufferNotFull* and *BufferNotEmpty*.

```
1  public class BoundedBuffer<T> {
2
3    private final T buffer[];
4    private int first;
5    private int last;
6    private int numberInBuffer;
7    private final int size;
8
9    private final Lock lock = new ReentrantLock();
10   private final Condition notFull = lock.newCondition();
11   private final Condition notEmpty = lock.newCondition();
12   public BoundedBuffer(int length) { /* Normal constructor. */
13     size = length;
14     buffer = (T[]) new Object[size];
15     last = 0;
16     first = 0;
17     numberInBuffer = 0;
18   }
19   public void put(T item) throws InterruptedException {
20     lock.lock();
21     try {
22
23       while (numberInBuffer == size) { notFull.await(); }
24       last = (last + 1) % size;
25       numberInBuffer++;
26       buffer[last] = item;
27       notEmpty.signal();
28
29     } finally {
30       lock.unlock();
31     }
32   }
33   public T get() ... {
34     lock.lock();
35     try {
36
37       while (numberInBuffer == 0) { notEmpty.await(); }
38       first = (first + 1) % size;
39       numberInBuffer--;
40       notFull.signal();
41       return buffer[first];
42
43     } finally {
44       lock.unlock();
45     }
46   }
47 }
```

## ✎ Review Question

1.1. Would replacing `volatile boolean running` with `AtomicBoolean running` in the Stop-Flag Pattern example be correct? Would it be necessary?

# Thread Priorities

- Although priorities can be assigned to the Java threads (`setPriority`), they only serve the underlying scheduler as a guideline for resource allocation.
- A thread can explicitly give up the processor resources by calling the `yield` method.
- `yield` places the thread at the end of the queue for its priority level.
- However, Java's scheduling and priority models are weak:
  - There is no guarantee that the thread with the highest priority that can run will always be executed.
  - Threads with the same priority may or may not be divided into time slices.
  - When using native threads, different Java priorities can be mapped to the same operating system priority.

## Best Practices

!! Important

- `synchronized` code should be kept as short as possible.
- Nested monitor calls should be avoided as the outer lock is not released when the inner monitor is waiting. This can easily lead to a deadlock occurring.

⚠ Warning

If two (or more) threads access the same resources in a different order, a deadlock can occur.

!! Important

**Resources must always be locked in the same order** to avoid deadlocks.

# 2. Thread Safety

# Thread Safety - Prerequisites

For a class to be thread-safe, it must behave correctly in a single-threaded environment.

I.e. if a class is implemented correctly, then no sequence of operations (reading or writing public fields and calling public methods) on objects of this class should be able to

- set the object to an invalid state,
- observe the object in an invalid state, or
- violate one of the invariants, preconditions or postconditions of the class.

The class must also behave correctly when accessed by multiple threads.

- Independent of *scheduling* or the interleaving of the execution of these threads by the runtime environment,
- Without additional synchronisation on the part of the calling code.

**⚑ Assessment**

As a result, operations on a thread-safe object appear to all threads as if the operations were performed in a fixed, globally consistent order.

# Thread Safety Level

**Immutable:** The objects are constant and cannot be changed.

**Thread-safe:** The objects can be changed, but support concurrent access as the methods are synchronized accordingly.

**Conditionally thread-safe:**
All objects where each individual operation is thread-safe, but certain sequences of operations may require external synchronization.

**Thread-compatible:**
All objects that have no synchronization at all. However, the caller can take over the synchronization externally if necessary.

**Thread-hostile:** Objects that are not thread-safe and cannot be made thread-safe as they manipulate global status, for example.

# ✎ Exercise

## 2.1. Delayed Execution

Implement a class (`DelayingExecutor`) that accepts tasks (instances of *`java.lang.Runable`*) and executes them after a certain time. The class must not block or be locked during this time.

Consider using virtual threads. A virtual thread can be created using the method: `Thread.ofVirtual()`. A `Runnable` object can then be passed to the `start` method.

Delay the execution (`Thread.sleep()`) by an average of 100ms with a standard deviation of 20ms. (Use `Random.nextGaussian(mean,stddev)`)

Start 100 000 virtual threads. How long does the execution take? How long does the execution take with 100 000 platform (*native*) threads?

It is recommended to use the template.

---

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class DelayingExecutor {

  private final Random random = new Random();

  private Thread runDelayed(int id, Runnable task) {
    // TODO
  }

  public static void main(String[] args) throws Exception {
    var start = System.nanoTime();
    DelayingExecutor executor = new DelayingExecutor();
    List<Thread> threads = new ArrayList<>();
    for (int i = 0; i < 100000; i++) {
      final var no = i;
      var thread = executor.runDelayed(
          i,
          () -> System.out.println("i'm no.: " + no));
      threads.add(thread);
    }
    System.out.println("finished starting all threads");
    for (Thread thread : threads) {
      thread.join();
    }
    var runtime = (System.nanoTime() - start)/1_000_000;
    System.out.println(
      "all threads finished after: " + runtime + "ms"
    );
  }
}
```

# ✎ Exercise

## 2.2. Thread-safe programming

Implement a class `ThreadsafeArray` to store non-`null` objects (*`java.lang.Object`*) at selected indices - comparable to a normal array. Compared to a normal array, however, a thread which wants to read a value should be blocked if the cell is occupied. The class should provide the following methods:

*`get`*`(int index)`: Returns the value at the position `index`. The calling thread may be blocked until a value has been saved at the `index` position. (The `get` method does not remove the value from the array).

*`set`*`(int index, Object value)`: Stores the value `value` at the position `index`. If a value has already been saved at the position `index`, the calling thread is blocked until the value at the position `index` has been deleted.

*`delete`*`(int index)`: Deletes the value at position `index` if a value exists. Otherwise, the thread is blocked until there is a value that can be deleted.

a. Implement the `ThreadsafeArray` class using only the standard primitives: `synchronized`, `wait`, `notify` and `notifyAll`. Use the template.

b. Can you use both `notify` and `notifyAll`?

c. Implement the `ThreadsafeArray` class using `ReentrantLock`s and `Condition`s. Use the template.

d. What are the advantages of using `ReentrantLock`s?

---

You can also consider the class `ThreadsafeArray` as an array of *BoundedBuffers* with the size 1.

```
1  public class ThreadsafeArray {
2
3    private final Object[] array;
4
5    public ThreadsafeArray(int size) {
6      this.array = new Object[size];
7    }
8
9    // complete method signatures and implementations
10   Object get(int index)
11   void set(int index, Object value)
12   void remove(int index)
13
14   public static void main(String[] args) throws Exception {
15     final var ARRAY_SIZE = 2;
16     final var SLEEP_TIME = 1; // ms
17     var array = new ThreadsafeArray(ARRAY_SIZE);
18     for (int i = 0; i < ARRAY_SIZE; i++) {
19       final var threadId = i;
20
21       final var readerThreadName = "Reader";
22       var t2 = new Thread(() -> {
23         while (true) {
24           int j = (int) (Math.random() * ARRAY_SIZE);
25           try {
```

```
            out.println(readerThreadName + "[" + j + "]" );
            var o = array.get(j);
            out.println(readerThreadName +
                "[" + j + "] ⇒ #" + o.hashCode());
            Thread.sleep(SLEEP_TIME);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }, readerThreadName);
      t2.start();

      // One Thread for each slot that will eventually
      // write some content
      final var writerThreadName = "Writer[" + threadId + "]";
      var t1 = new Thread(() -> {
        while (true) {
          try {
            var o = new Object();
            out.println(writerThreadName + " = #" + o.hashCode());
            array.set(threadId, o);
            out.println(writerThreadName + " done");
            Thread.sleep(SLEEP_TIME);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }, writerThreadName);
      t1.start();

      // One Thread for each slot that will eventually
      // delete the content
      final var deleterThreadName = "Delete[" + threadId + "]";
      var t3 = new Thread(() -> {
        while (true) {
          try {
            out.println(deleterThreadName);
            array.delete(threadId);
            Thread.sleep(SLEEP_TIME);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }, deleterThreadName);
      t3.start();
    }
  }
}
```