Java training

Threads, synchronous and asynchronous processing

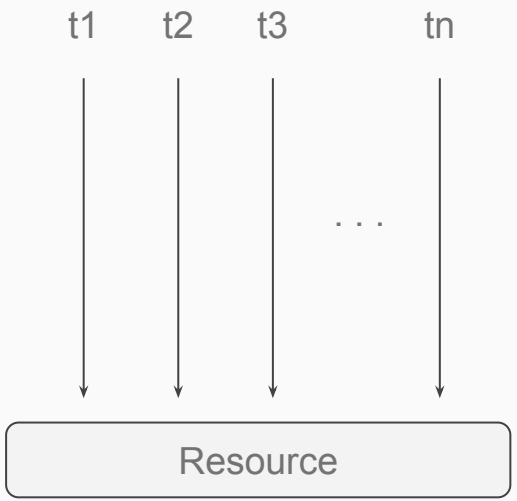
Session overview

- **Threads:**
  - Concurrency modes overview
  - Concurrent processing
  - Async / parallel processing
- **Hands-on**

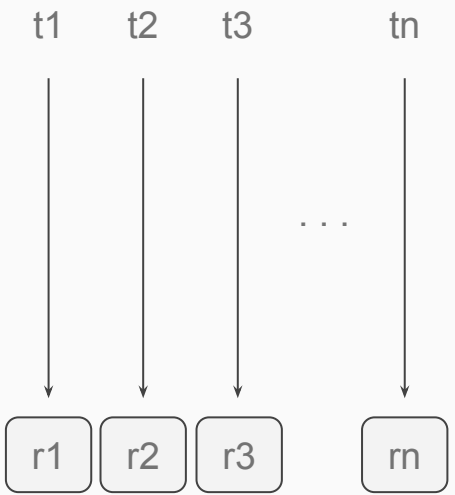
Multi-threaded programming categories

- **Synchronous (concurrent, blocking)** programming - multiple threads accessing shared resources
  - Especially projects started in the past (> 3-5 years), when processing resources were more expensive
- **Asynchronous (parallel, non-blocking)** programming - parallelizing programming computations

Synchronous vs asynchronous processing



Synchronous



Asynchronous

Concurrency / multithreading benefits

- **Thread** - an execution ‘line’ of a program
- **Multi-threaded application** - application which uses multiple threads, running in parallel
  - Speeding up the processing + making it more efficient
- Cross-cutting aspect in programming → added transversally besides the business logic flow
  - One of the most complex aspects
- Multiple processor cores → scaled / parallel program execution
  - → Improve the performance and responsiveness of the application

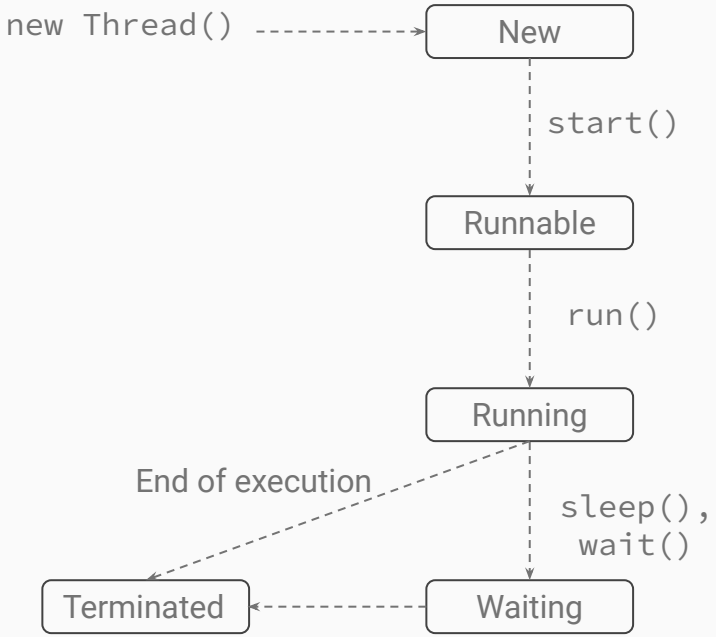
Threading - main classes and methods

- `java.lang.Thread` - main thread handling class
  - `join()`
  - `run()`
  - `start()`
  - `sleep()`
  - `yield()`
- `java.lang.Object` - threads inner-working / communication
  - `wait()`
  - `notify()`
- `java.lang.Runnable` - main interface for the thread creation
  - `run()`

Thread class methods

Method	Method Type	Short Description
`Thread currentThread()`	Static method	Returns reference to the current thread.
`String getName()`	Instance method	Returns the name of the current thread.
`int getPriority()`	Instance method	Returns the priority value of the current thread.
`void join()`, `void join(long)`, `void join(long, int)`	Overloaded instance methods	The current thread invoking join on another thread waits until the other thread completes. You can optionally give the timeout in milliseconds (given in `long`) or timeout in milliseconds as well as nanoseconds (given in `long` and `int`).
`void run()`	Instance method	Once you start a thread (using the `start()` method), the `run()` method will be called when the thread is ready to execute.
`void setName(String)`	Instance method	Changes the name of the thread to the given name in the argument.
`void setPriority(int)`	Instance method	Sets the priority of the thread to the given argument value.
`void sleep(long)` `void sleep(long, int)`	Overloaded static methods	Makes the current thread sleep for given milliseconds (given in `long`) or for given milliseconds and nanoseconds (given in `long` and `int`).
`void start()`	Instance method	Starts the thread; JVM calls the `run()` method of the thread.
`String toString()`	Instance method	Returns the string representation of the thread; the string has the thread's name, priority, and its group.

Thread life cycle



State	Details
New	Beginning state - remains in this state until the program starts the thread; also referred to as a **born thread**
Runnable	After the thread is started, it becomes runnable → executing its task
Waiting	A thread may transition to the 'waiting' state if it waits for another thread to perform a task. It can resume to the 'running' state only when another thread signals it
Timed waiting	Waiting for a specified interval of time
Terminated	After it has completed its task

Creating threads

- Two main ways
  - Implementing the `Runnable` interface → the recommended way
    - Implement an (anonymous) instance of the **Runnable** interface
    - Wrap the Runnable object in a **Thread** object
  - Extending the `Thread` class
    - Create a class which extends the Thread class
    - Override the .run() method (the default does nothing)
- Call the `.start()` method on the created thread
  - The `.run()` method will be automatically invoked by the JVM

Hands-on - creating and running a thread

Threads synchronization

- **Race condition** - concurrent threads accessing the same resources - objects, variables, methods
  - May produce unforeseen / unpredictable / unwanted effects
  - Examples: accessing the same file, accessing the same network resource
- **Synchronizing** the action of multiple threads → ensuring only one thread can access the resource at a given time
- Implemented using **monitor locks** (**monitors**):
  - Each object is associated with a monitor, which a thread can lock or unlock
  - Only one thread may hold a lock on a monitor

synchronized

- **Synchronized** blocks - accessing shared resources

```
synchronized (lock) {
    // accessing shared resources
}
```

- **Synchronized** methods

```
public synchronized void processProduct(Product product) { ... }
```

Thread synchronization - continued

- When a thread executes a `synchronized` method for an object, all other threads that invoke that method for the same object *will block* (suspend their execution), until the first thread has finished with the object (+ released the lock)

- **Synchronized methods** - enable a simple means for preventing thread interference - if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods

synchronized (this)

- When synchronized access to some class fields is needed - ‘this' can be used as a lock / monitor

```
public void setNewPrice(int price) {
    synchronized (lock) {
        // update the price AND any related properties - balance, profit, ...
    }
}
```

Concurrent collections

- `ArrayBlockingQueue`
- `LinkedBlockingQueue`
- `ConcurrentHashMap`
- `ConcurrentNavigableMap`
- `ConcurentLinkedQueue + ConcurrentLinkedDeque`
- `Vector`

\+ others → `java.util.concurrent` package

Atomic primitive wrappers

- Synchronized primitive wrapper classes
  - `AtomicInteger`
  - `AtomicLong`
  - `AtomicBoolean`
- Example:

```
AtomicInteger anImportantValue = new AtomicInteger(0);
anImportantValue.incrementAndGet();
anImportantValue.set(20);
anImportantValue.getAndAdd(2);
```

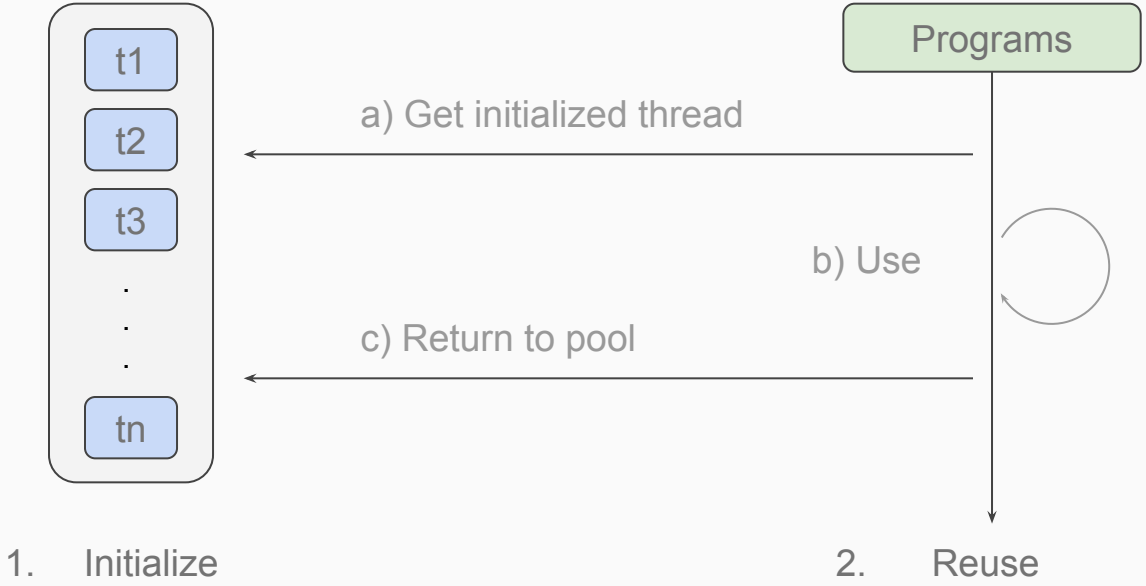
Common locking related threading problems

- **Deadlock** - when two threads are each trying to acquire a lock on a resource locked by the other thread → they wait indefinitely for the other thread to terminate
  - Advice - avoid using multiple locks
  - Non-deterministic - they do not occur every time
- **Livelock** - two threads which are continuously responding to the actions of the other thread, hence not progressing (reciprocally) in their tasks
- **Starvation** - lower priority threads unable to gain access to a shared resource, due to other (more) ‘greedy’ threads holding locks on them

Parallel / asynchronous processing

- **Thread pool** - container of reusable threads
  - Threads are initialized once, reused afterwards
    - Thread initialization - expensive operation (resource wise)
  - Submitted tasks are backed by a queue
  - Very configurable; use-case specific
- ***ExecutorService, ExecutorCompletionService*** - thread pool management
  - Multiple static methods for various thread pools creation
  - Used to perform **async** and / or **fork / join** processings
- ***Future* interface** - represents a future terminated operation
  - The finished async processings are `Future` wrapped objects
  - Further extended and improved in Java 8 → `CompletableFuture`

Thread pool visualization



Parallel (asynchronous) processing

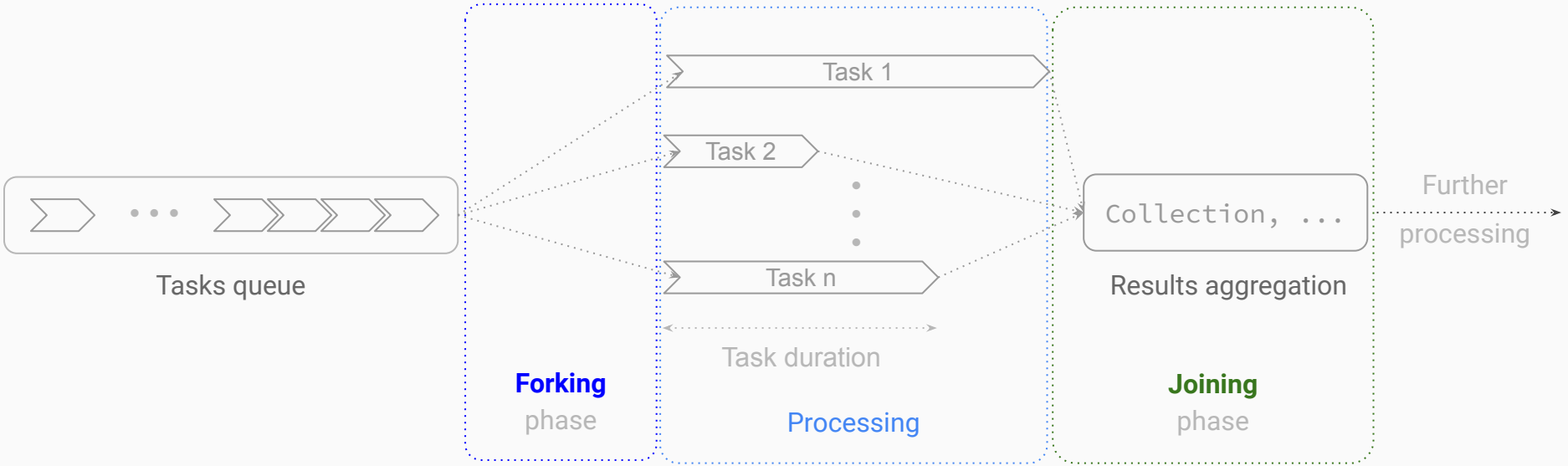
Components:

1. An `ExecutorService` + an `ExecutorCompletionService`
   a. The ExecutorService → the used thread pool
   b. The ExecutorCompletionService → handles the async task processing
2. A class which implements Callable<returned-type>

Steps:

1. Submit processing tasks to the `ExecutorCompletionService`
2. Poll the `ExecutorCompletionService` for finished tasks
3. Use the results of the finished processing tasks

Fork / join processing



Asynchronous processing - main classes

- **`ExecutorService`** - abstracts a thread pool
- **`ExecutorCompletionService`** - uses an `ExecutorService` for asynchronous task processing, by:
  - Submitting it processing tasks
  - Polling it for processing results
- **`Callable<Result>`** - the async tasks must implement the `Callable` interface → the `Executor*` classes use them for processing
- **`Future<Result>`** - the (future terminated) result of an async processing

Hands-on

- Trying an asynchronous processing service
- Discuss various use-cases in projects

Fork/Join framework

- Modelling ‘divide and conquer’ tasks → recursively dividing tasks into smaller sub-tasks, joining them after they’re finished
  - Dividing the tasks → fork
  - Re-joining them → join
- ‘Work stealing algorithm’ - free threads ‘steal’ work from the tasks queue
- Main components:
  - `ForkJoinPool` → thread-pool used to run fork/join tasks
  - `ForkJoinTask` → tasks executed by the ForkJoinPool
  - `RecursiveTask` → a task which has a return value; can be further forked
  - `RecursiveAction` → a task which doesn’t have a return value

Programming trends

- **Reactive** programming - lock-free programming
  - Java 9+ offers built-in support for reactive programming
- **Stateless** programming - no state maintained in the code (service or model)
- `CompletableFuture` - new async programming model, introduced in Java 8

Further reading

- Java concurrency essentials
- Java concurrency in practice
- Java concurrent animated
- Java threads and concurrency utilities

Q&A session

1. You ask, I answer
2. I ask, you answer