Karlsruhe Institute of Technology

Programmierparadigmen

Prof. Dr. Ralf Reussner

Topic 5.5

Parallel Programming with Java – Advanced Concepts

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu

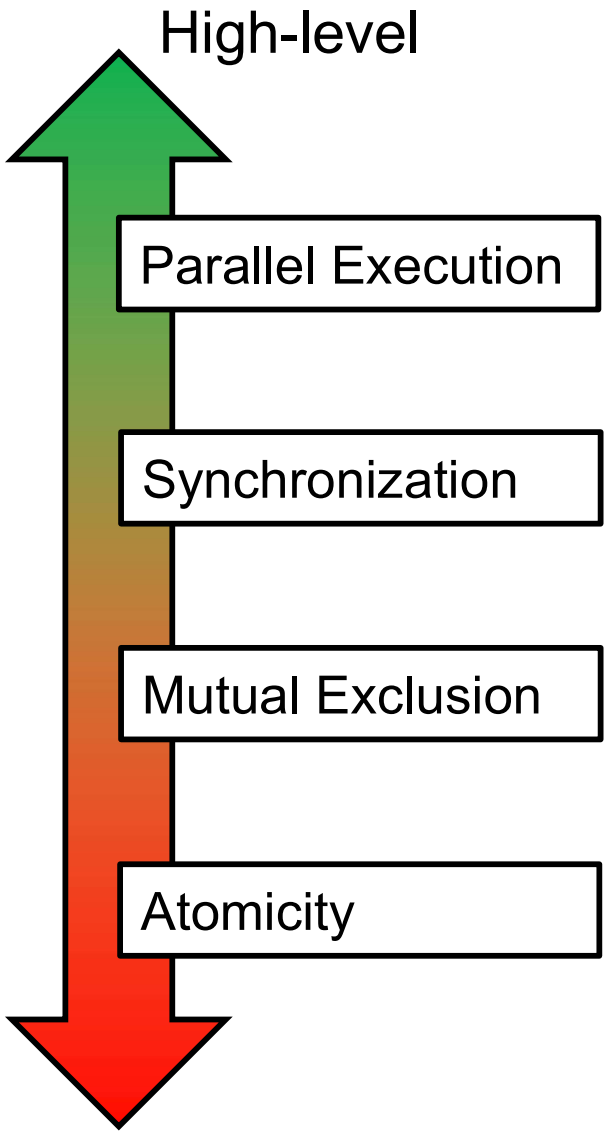
Overview on Today's Lecture

- Content
  - Advanced parallel programming constructs in Java:
    - Atomic types and locks
    - `Executors`, `ThreadPools`, `Callables` and `Futures`
    - Fork/Join and parallel streams
  - Now in Appendix – Actor Model:
    - Motivation for the actor model as a parallel computation concept
    - History and basics of the actor model
    - Actor model implementations with focus on Akka
- Learning goals: participants –
  - know about advanced parallelization principles and constructs in Java
  - can discuss which principles are useful to solve a parallelization problem
  - can apply the introduced constructs to simple problems
  - understand components, functioning and benefits of the actor model
  - are able to write simple actor-based applications with Akka

Advanced Java Threading Concepts

- Threads are rather low-level constructs, making certain tasks difficult to implement, e.g.
  - Managing (certain pools of) threads is effortful
  - Returning results is only possible on detours
- Since Java 1.5, further constructs for parallel execution were added to the `java.util.concurrent` package
- In the following, we will discuss:
  - Atomic types and non-blocking algorithms
  - Complex locks and semaphores for better synchronization
  - `Executors` and `ThreadPools` to reduce thread management efforts
  - `Callables` and `Futures` to easily return results from threads
  - The Fork-Join principle to split a problem into smaller ones
  - Parallel streams for parallel data processing

Overview: Parallel Constructs in Java



Concurrent Streams

Concurrent Classes

Actors

Fork/Join

Futures

Join

Barriers

Monitors (synchronized)

Locks

Signals (wait & notify)

Semaphores

Atomic Data Types

Atomic Operations

Atomicity

- Atomic statements …
  - are executed complete or not at all
  - produce no visible side effects until their completion
  - are the smallest unit of code that cannot be interleaved
    → no race conditions due to interleaving
  - can still produce synchronization errors due to memory consistency errors (missing happens-before relationship)
- By atomically accessing shared variables, synchronization can sometimes be avoided
  - Much more performant
  - More difficult, error-prone and challenging to maintain
  - Should only be applied if highly optimized code is necessary

Atomic Statements in Java

- Reads and writes of reference variables
- Reads and writes of most primitive variables
  - Atomic for 32bit values
  - Potentially not atomic for `long` and `double`, depending on the system
- Reads and writes of *all* variables declared `volatile`
  - `volatile` introduces a happens-before relationship between a write of that variable and all subsequent reads
  - In contrast to using `synchronized`, `volatile` does not perform a mutual exclusive lock

- *Not* atomic are multiple, combined accesses, e.g.
  - `x=y+1`
  - Increment/decrement operations (like shown previously)

Atomic Types

- Java provides atomic types that support atomic operations on single variables in the `java.util.concurrent.atomic` package
- Example: `AtomicInteger`
  - `int get()`: just returns the variable
  - `int incrementAndGet()`: atomically increments and returns the variable
  - `int decrementAndGet()`: atomically decrements and returns the variable
  - `boolean compareAndSet(int oldValue, int newValue)`: checks for the old value and replaces it with the new one
    - If successful, **`true`** is returned
    - If the `AtomicInteger` does not contain the `oldValue`, **`false`** is returned and nothing is done
- Allows to prevent thread interferences without the need to use synchronization
  - Less performance impact
  - Less liveness impact

Non-blocking Algorithms

- Non-blocking algorithms allow for parallel execution without using potentially blocking synchronization constructs, e.g. using …
  - Atomic types
  - `volatile` keyword

```
public class AtomicCounter {
    private AtomicLong count = new AtomicLong(0);

    public void increment() {
        boolean updated = false;
        while(!updated){
            long prevCount = this.count.get();
            updated = this.count.compareAndSet(prevCount, prevCount + 1);
        }
    }

    public long count() {
        return this.count.get();
    }
}
```

Optimistic locking:
Create a copy of the data and check if the value is still up-to-date when setting the changed value via atomic `compareAndSet` operation, otherwise nothing is done and **`false`** is returned

Adapted from: http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html

Executor

- Executors abstract from thread creation
  - Simple implementations only start a thread
  - Other implementations, for example, reuse already created threads
- Java defines three executor interfaces in `java.util.concurrent`
- The most generic interface is `Executor`
  - A simple interface supporting the execution of tasks
  - Provides an `execute` method that accepts a `Runnable`

```
void execute(Runnable runnable)
```

ExecutorService

- The most important interface is `ExecutorService`:
  - A subinterface of `Executor`
  - Provides further lifecycle management logic
- The class `Executors` (not to confuse with the `Executor` interface) provides convenient factory methods for creating an `ExecutorService`
  - `newSingleThreadExecutor()` creates an `Executor` using a single thread
  - `newFixedThreadPool(int)` creates a thread pool with reused threads of fixed size
  - `newCachedThreadPool()` creates a thread pool with reused threads of dynamic size

ExecutorService

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

try {
        executor.shutdown();
        executor.awaitTermination(5, TimeUnit.SECONDS);
} catch(InterruptedException ex) {
} finally {
        if (!executor.isTerminated()) {
                executor.shutdownNow();
        }
}
```

Shutdown and wait for termination

If shutdown was not successful, enforce shutdown

- An `ExecutorService` provides lifecycle management
  - Listens for new tasks until they are stopped
  - The above example will not terminate without `shutdown()`

Use of Executor vs. Thread

```
Runnable task = new Runnable() {
        public void run() {
                System.out.println("Executed by: " +
                        Thread.currentThread().getName());
        }
};
Thread t = new Thread(task, "my_thread");
t.start();

Executor e = Executors.newSingleThreadExecutor();
e.execute(task);
```

- Executor separates task from execution
- Executor provides the execute() method and can execute any number of Runnable tasks
- Thread receives the task as constructor argument and can only execute one Runnable task

Adapted from: https://javarevisited.blogspot.com/2016/12/difference-between-thread-and-executor.html

Returning Results from Threads

- A thread cannot directly return a result
  - The `run` method has the result type `void`
  - The `run` method is executed asynchronously, so it is unclear when it finishes and delivers the result
- To wait for a thread to finish its execution, the method `Thread.join()` can be called on a thread
- Returning a result can be achieved in the following way:
  - Define an instance variable in the custom thread class
  - Add a getter for that variable
  - Start the thread with `start()`
  - Wait for the thread to finish with `join()`
  - Get the result using the getter

```
class MyThread extends Thread {
        …
        public String getResult()
…
}
…
MyThread t = new MyThread();
t.start();
t.join();
String result = t.getResult();
```

Callables

- The `Callable` interface allows to return results
  - Is similar to the `Runnable` interface
  - Instead of run() it defines the method `call()`, which has a return type other than `void`

```
public class MyCallable implements Callable<String> {
        int id;

        public MyCallable(int id) {
                this.id = id;
        }

        public String call() {
                return "Run " + id;
        }
}
```

The type parameter defines the return type of the `call()` method

Futures: Representation of Results

- A `Future<V>` represents the (future) result of an asynchronous computation (i.e. `Callable`)
- The computation can either be (not yet) finished or cancelled
- Results can only be acquired when the computation is finished
- `ExecutorService` provides an additional `submit()` method, which expects a `Runnable`, but also a `Callable` in an overloaded version

```
<T> Future<T> submit(Callable<T> callable)
```

- `submit()` returns a `Future`, which represents the (future) result of the provided `Callable`

```
ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 10; i++) {
        final int currentValue = i;
        Callable<Integer> myCallable = () -> {return currentValue;};
        Future<Integer> myFuture = executorService.submit(myCallable);
}
```

Futures: Retrieving Results

- The result of the `Callable` can be retrieved from the `Future` using its `get()` method

```
ExecutorService executorService = Executors.newCachedThreadPool();
List<Future<Integer>> futures = new ArrayList<Future<Integer>>();

for (int i = 0; i < 10; i++) {
        final int currentValue = i;
        Callable<Integer> myCallable = () -> {return currentValue;};
        futures.add(executorService.submit(myCallable));
}

for (Future<Integer> future : futures) {
        try {
                Integer result = future.get();
                System.out.println(result);
        } catch (ExecutionException ex) {}
}

executorService.shutdown();
```

Blocks until the Thread finished and the Future contains the value

Futures: Waiting for Results

- The `get()` method of `Future` blocks until the result is available
- A `Future` provides further methods for waiting for the completion of a submitted `Callable`:
  - `isDone()`: Returns whether the task finished
  - `get(int timeout, TimeUnit unit)`: Allows to wait for a specified amount of time

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) { }
});

future.get(1, TimeUnit.SECONDS);
```

- The above example will result in a `TimeoutException`

Adapted from: http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

ExecutorService and Futures: Example

```
public double parallelMean(final List<Double> allValues, int numThreads) {
        List<Double> resultList = new ArrayList<>();
        int blockSize = allValues.size() / numThreads();

        ExecutorService pool = Executors.newFixedThreadPool(numThreads);
        List<Future<Double>> futureList = new ArrayList<>();
        for (int i = 0; i < allValues.size(); i += blockSize) {
                List<Double> subList = allValues.subList(i, i + blockSize);
                Future<Double> future = pool.submit(() -> {
                        double sum = 0;
                        for (double value : subList) {
                                sum += value;
                        }
                        return sum / subList.size();
                });
                futureList.add(future);
        }
        for (int i = 0; i < futureList.size(); i++) {
                try {
                        resultList.add(futureList.get(i).get());
                } catch (Exception e) { /* ... */ }
        }
        /* ... calculate average sequentially based on subList-averages ...
}
```

Submit task to be executed in parallel

Split lists in non-overlapping sublists that threads can work on

Collect results

Streams

- Streams were introduced in Java 8
- Streams provide additional and often used operations on data collections, such as
  - `filter`,
  - `map`, `reduce`,
  - `collect`,
  - `findAny`, `findFirst`,
  - `min`, `max`,
  - etc.
- Streams do not require additional memory
  - Results of stream operations are calculated on demand
  - Stream operations are executed lazy
- Any Java collection can be treated as a stream by calling the `stream()` method

Streams: Example

```
public class Person {
  private final boolean isStudent;
  private final int age;

  public boolean isStudent() {
    return isStudent;
  }

  public int getAge() {
    return age;
  }

  public Person(boolean isStudent,
                int age) {
    this.isStudent = isStudent;
    this.age = age;
        }
}
```

Calculation of average age of students in a given `Person` collection `personsInAuditorium`:

```
double average =
personsInAuditorium
  .stream()
  .filter(Person::isStudent)
  .mapToInt(Person::getAge)
  .average()
  .getAsDouble();
```

Maps to an `IntStream` on which `average()` can be called

Remark: Using `map` with a function that returns a stream of integers instead of `mapToInt` returns an ordinary `Stream` that does not provide the `average` function.

Streams: Evaluation Example

```
List<Person> personsInAuditorium =          Arrays.asList(
        new Person(true, 18),
        new Person(false, 21),
        new Person(false, 19),
        new Person(true, 28));

double average = personsInAuditorium
        .stream()          {(true, 18), (false, 21), (false, 19), (true, 28)}
        .filter(Person::isStudent)        {(true, 18), (true, 28)}
        .mapToInt(Person::getAge)         {18, 28}
        .average()                        23
        .getAsDouble();                   23d
```

- The results of stream operations are not physically present but are calculated on demand

Streams: The `collect` Operation

- A `collect` operation performs a reduction given three arguments:
  - Supplier: Delivers a new result container
  - Accumulator: A function for incorporating a new element in the result
  - Combiner: Combines two values and must be compatible with the result

```
R collect(Supplier<R> supplier,
BiConsumer<R, ? super T> accumulator,
BiConsumer<R, R> combiner);
```

R: result type
T: collection type

- Its behavior in pseudocode is as follows:

```
R result = supplier.get();
for (T element : this stream)
    accumulator.accept(result, element);
return result;
```

- Note that the combiner is not used: it is only used for parallel streams to combine results calculated in parallel

Why „? super T“ ?

```
R collect(Supplier<R> supplier,
BiConsumer<R, ? super T> accumulator,
BiConsumer<R, R> combiner);
```

- Question: Is `List<Circle> I = new List<Circle>()` a subclass of `List<Shape>` if `Circle` is a subclass of `Shape`?
- Answer: It depends:
  - Is it written into I (I consumes) it is not, but Is it read from I (I produces) it is
- Rationale:
  - Liskov Substition Principle:
    an instance of a subclass s can always be used typesafely instead of on instance c of a superclass.
  - Contravariance in method signatures is type-safe
    - generalization for the parameters (where is written to, from the perspective if the caller), specialization for the result type (where is read from, from the perspective if the caller).
    - From the perspective of the method implementer: the method consumes parameters and produces result.
- Covariantly typed languages: PECS = Producer „`extends`“, consumer „`super`“
- Note: here this mnemonic is to be seen from the perspective of the list. When it is written into the list, the list „consumes“, when it is read from it, it „produces“.
- See also https://stackoverflow.com/questions/2723397/what-is-pecs-producer-extends-consumer-super

Streams: `collect` Operation Example

```
R collect(Supplier<R> supplier,
BiConsumer<R, ? super T> accumulator,
BiConsumer<R, R> combiner);
```

R: result type
T: collection type

- A collector for summing up the ages of all persons in a collection can be basically defined as follows:

```
personsInAuditorium.stream().collect(
  () -> 0,
  (currentSum, person) -> { currentSum += person.getAge(); },
  (leftSum, rightSum) -> { leftSum += rightSum; });
```

Combined value has to be assigned to the first input reference

- Nevertheless, an `Integer` value is immutable, so the operation will always return `0` → ensure that the result container is mutable!

Streams: `collect` Operation Example

- The previous example can, for example, be solved with a mutable `Integer` object:

```
personsInAuditorium.stream().collect(
  () -> new MutableInt(),
  (currentSum, person) -> { currentSum.add(person.getAge()); },
  (leftSum, rightSum) -> { leftSum.add(rightSum.intValue()); })
        .intValue();
```

- Alternatively, the `Collector` interface can be implemented and passed to the `collect` method

```
public class MutableInt {
  private int value;

  public void add(int value) {
    this.value += value;
  }

  public int intValue() {
    return value;
  }
}
```

Streams: Predefined Collectors

- For many purposes, Java provides predefined collectors in the `java.util.stream.Collectors` class, e.g. …
  - for mapping elements
  - for calculating the average, the sum, the maximum of elements
  - for grouping elements, e.g. by age in our example:

```
Map<Integer, List<Person>> groupedByAge =
  personsInAuditorium
  .stream()
  .collect(Collectors.groupingBy(Person::getAge));
```

Parallel Streams

- On parallel streams, retrieved by `parallelStream()`, certain operations are automatically executed in parallel
- The previous example can also be calculated in parallel:

```
double average = personsInAuditorium
        .parallelStream()
        .filter(p -> p.isStudent())
        .mapToInt(Person::getAge)
        .average()
        .getAsDouble();
```

- Parallel execution of some operations depends on certain properties
  - `collect()` is only executed in parallel if the collection is unordered and if the collector is concurrent. Therefore, specific predefined collectors exist:

```
Map<Integer, List<Person>> groupedByAge =
  personsInAuditorium
  .parallelStream()
  .collect(Collectors.groupingByConcurrent(Person::getAge));
```

Conclusion

- Advanced low-level thread programming
  - Atomic types: less synchronization overhead and non-blocking algorithms
  - `Locks`, `Semaphores` and `Barriers` for more sophisticated locking
- Advanced high level synchronization and parallel execution
  - `ThreadPools` and `Executors` for simplified thread management and reuse
  - `Callables` and `Futures` for returning results from threads
  - Fork-Join for dividing a problem into independent smaller ones
  - Thread-safe collection classes
  - Parallel streams for automatic parallelization of data collection operations

Bonus: The Dos and Don’ts (1)

- Parallelization is only profitable if the problem exceeds a certain size
  - If partitioned tasks are too small, the overhead for managing threads and synchronization consumes the performance improvement
- Come to a compromise between understandability and performance
  - If possible, use high-level language constructs. They take care of correct synchronization on their own
  - Fine-grained/highly optimized synchronization makes the code complex, error-prone and hard to understand → only optimize where necessary
  - On the other hand: oversized critical sections are not good as well
- Do not trust in a certain timing
  - Use joins, signals or barriers instead
  - In the worst case, a device goes to standby during execution. Then every sleep statement is too short

Bonus: The Dos and Don’ts (2)

- Do not use threads as monitors
  - Confusing: `thread.wait()` does not pause the referenced thread, but the current one
  - Additional notify signals because of the thread’s lifecycle
- Be aware of potential problems using threads:
  - Race conditions through interleaving
  - Memory inconsistencies
  - Deadlocks, livelocks and starvation
  - Bottlenecks through coarse-grained synchronization

Literature and References

[Hewitt1973] Carl Hewitt, Peter Bishop, and Richard Steiger. “A universal modular ACTOR formalism for artificial intelligence” in Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI'73). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235-245, 1973

[Agha1986] Gul Agha. “Actors: a Model of Concurrent Computation in Distributed Systems”. MIT Press, Cambridge, MA, USA, 1986

[Akka] http://doc.akka.io/docs/akka/current/java.html

APPENDIX

Locks

- An option to realize critical sections are locks
- Java provides different implementations of a `Lock` interface [1]
  - `ReentrantLock`
  - `ReentrantReadWriteLock`
- A constructor parameter allows to specify if a lock shall be fair
  - "Unfair" locks are more performant
- Reentrant: Locked section can be entered multiple times by the same thread
  - E.g. via recursion

```
public class Fibonacci {
  private int n1 = 0, n2 = 0, n3 = 0;
  Lock lock = new ReentrantLock(false);
  public void fibonacci(int count) {
    lock.lock();
    n3 = n1 + n2; n1 = n2; n2 = n3;
    fibonacci(count - 1);
    lock.unlock();
  }
}
```

[1] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ Lock.html

Locks: Pitfalls

```
public void doSomething() {
        lock.lock();
        // do something
        lock.unlock();
}
```

What if this section includes a return or throws an exception?

```
public void doSomething() {
  lock.lock();
  try {
    // do something
  } finally {
    lock.unlock();
  }
}
```

Locks: Attempted Acquisition

- A simple kind of lock in Java we have already seen are monitors
  - A `synchronized` block can be seen as a pair of lock and unlock operations on the monitor object
  - Drawback is that there is no possibility to back out of an attempt to acquire a lock
- Locks provide a `tryLock()` method, which acquires the lock if possible and returns whether it was successful
  - Useful to acquire several locks without blocking
  - Allows to avoid deadlocks by not fulfilling *hold and wait*
  - Be sure to correctly unlock the acquired locks (and none else)

Locks: Deadlock-free Locking Principle

```
public class DeadlockFreeLock {
        Lock lock = new ReentrantLock();

        private boolean getLocks(DeadlockFreeLock other) {
                boolean myLock = false;
                boolean theirLock = false;
                try {
                        myLock = lock.tryLock();
                        theirLock = other.lock.tryLock();
                } finally {
                        if (!(myLock && theirLock)) {
                                if (myLock) {
                                        lock.unlock();
                                }
                                if (theirLock) {
                                        other.lock.unlock();
                                }
                        }
                }
                return myLock && theirLock;
        }
```

If tryLock() returns true, acquisition of the lock was successful

Locks: Deadlock-free Locking Principle

```
public class DeadlockFreeLock {
        Lock lock = new ReentrantLock();

        private boolean getLocks(DeadlockFreeLock other) {
                boolean myLock = false;
                boolean theirLock = false;
                try {
                        myLock = lock.tryLock();
                        theirLock = other.lock.tryLock();
                } finally {
                        if (!(myLock && theirLock)) {
                                if
                                }
                                if
                                }
                        }
                }
                return myLock && th
        }
```

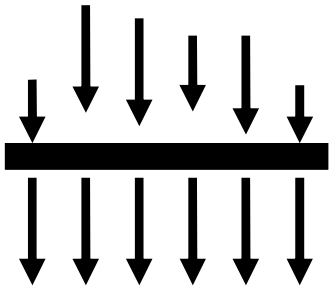
If tryLock() returns true, acquisition of the lock was successful

```
        …
private void use(DeadlockFreeLock
other) {
  if (getLocks(other)) {
    // Do something
    lock.unlock();
    other.lock.unlock();
    }
  }
}
```

Barriers

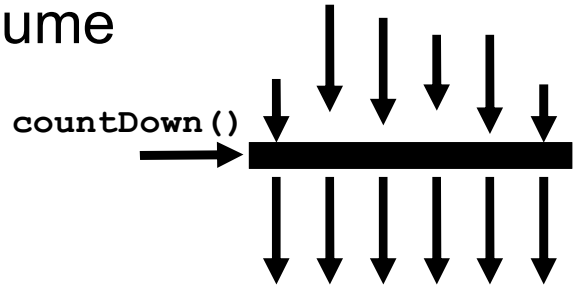
`CyclicBarrier(int n)`

- `await()` blocks the calling thread
- If `await()` was called *n* times, all threads resume
- Barrier can be reused afterwards



`CountDownLatch(int n)`

- `await()` blocks the calling thread
- If `countdown()` was called *n* times, all threads resume
- Latch cannot be restarted afterwards
- Further calls to `await()` return immediately



CyclicBarrier Example

```
public class BarrierDemo {
        static CyclicBarrier barrier;

        public static void main(String[] args) {
             barrier = new CyclicBarrier(2);
             new Thread(BarrierDemo::runSingleThread).start();
             new Thread(BarrierDemo::runSingleThread).start();
        }

        public static void runSingleThread() {
             try {
                        // Do first task
                        System.out.println("Reached first barrier");
                        barrier.await();
                        // Do second task
                        System.out.println("Reached second barrier");
                        barrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {}
        }
}
```

CountDownLatch Example

```
public class BarrierDemo {
        static CountDownLatch latch;

        public static void main(String[] args) {
                latch = new CountDownLatch(2);
                new Thread(BarrierDemo::runSingleThread).start();
                new Thread(BarrierDemo::runSingleThread).start();
                try {
                        latch.await();
                } catch (InterruptedException e) {}
                // Do tasks that require completion of the threads
        }

        public static void runSingleThread() {
                try {
                        // Do some task
                } catch (InterruptedException e) {}
                latch.countDown();
        }
}
```

Latch cannot be reused after await() has returned

Fork-Join (1)

- Fork-Join is a pattern for effectively computing divide-and-conquer algorithms in parallel
  - Problems are solved by splitting them into subtasks, solving them in parallel and finally composing the results
  - General algorithm in pseudocode:

```
Result solve(Problem problem) {
  if (problem is small enough) {
      directly solve problem
    } else {
      split problem into independent parts
      fork new subtasks to solve each part
      join all subtasks
      compose results from subresults
    }
}
```

Fork-Join (2)

- Java provides a `ForkJoinPool` on which `ForkJoinTasks` can be executed
- Implementations of `ForkJoinTask` must override the `compute()` method
  - `RecursiveAction` (no result) and `RecursiveTask` (returns result) are concretizations of such tasks
  - They can be executed by a `ForkJoinPool` calling its `invoke()` method

- If `MyTask` is a `ForkJoinTask`, it can be invoked as follows:

```
…
ForkJoinPool fjPool = new ForkJoinPool();
MyTask myTask = new MyTask(…);
fjPool.invoke(myTask);
…
```

Fork-Join (3)

- A task returning a value has to override `RecursiveTask`:

```
public class MyTask extends RecursiveTask<Integer> {
  private int[] array;
  private static final int THRESHOLD = 20;

  public MyTask(int[] arr) {this.arr = arr;}

  @Override
  public Integer compute() {
    if (arr.length > THRESHOLD) {
      return ForkJoinTask.invokeAll(createSubtasks())
                  .stream().mapToInt(ForkJoinTask::join).sum();
    } else {
      return processing(arr);
    }
  }

  private Collection<CustomRecursiveTask> createSubtasks() {
    //divide array into smaller parts, e.g., two halves
  }

  private Integer processing(int[] arr) {
    //do actually interesting computation, e.g., calculate average of array
} }
```

Split task if it is bigger than the threshold

Join subtasks

Adapted from: https://www.baeldung.com/java-fork-join

Actor Model

- Parallel computation with threads/shared memory –
  - requires locks to avoid race conditions
  - can easily lead to deadlocks when using locks wrong
  - uses blocking method calls

- The actor model is a conceptual, computational model for concurrent computation without locks and with asynchronous calls
  - Originally proposed by Hewitt, Bishop and Steiger in 1973 [Hewitt1973]
  - Refined by Gul Agha [Agha1986]
- The model is based on actors and messages
  - Actors are computation units
  - Actors communicate via messages → message passing

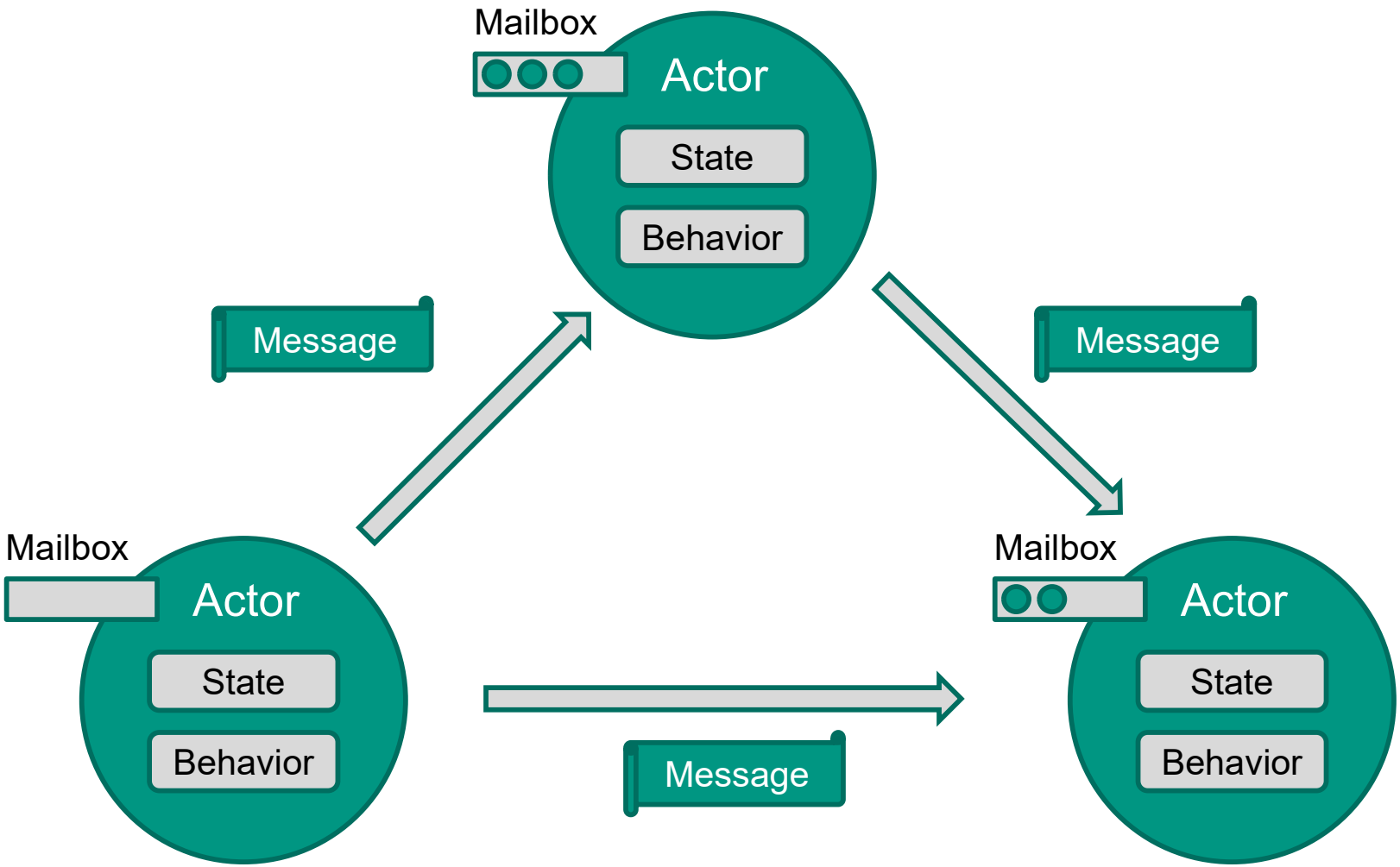
Actors

- Basic philosophy: *Everything is an actor*
- After receiving a message, an actor can –
  - send (a finite numbers of) messages to other actors
  - instantiate (a finite numbers of) new actors
  - designate the behavior when receiving the next message
    - which means that the actor can make computations and modify its local state, which influences what it does when receiving the next message
- Actors cannot access and modify the local state of other actors
- Actors keep the mutable state internal and communicate only via messages

Messages

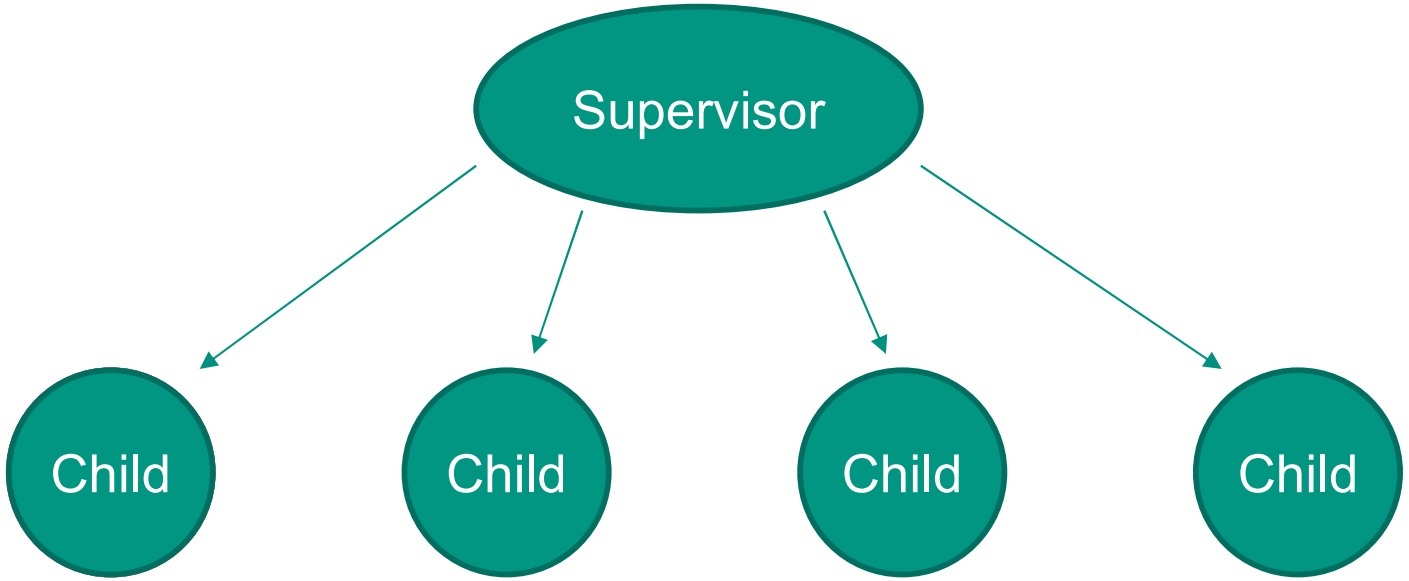
- A Message is delivered using an address, sometimes called “mailing address”
  - An actor can only communicate with actors whose addresses it has
  - An actor knows an address either through a received message or because it created the other actor itself
- Messages are sent asynchronously and stored in a mailbox of the receiving actor
- Messages are processed sequentially
  - An actor can only process one received message at a time
  - An actor processes messages in the order in which they arrived
  - Processing *n* message concurrently requires *n* actors

Actors and Messages



Actors: Fault Tolerance

- Classic, imperative programs have to be implemented defensively to handle problems that may happen
  - It is impossible to think about all potential problems
- In the actor model, an actor that creates a new (child) actor becomes its supervisor
  - It delegates work to the child actor
  - It can restart the child actor if it crashes



Distribution

- Actors allow for different distribution strategies
  - All actors of an application can be executed on one machine
  - The actors can be arbitrarily distributed on different machines
- Actors only have an internal state, a mailbox and respond to messages. They can do this independent from the machine they are running on
- This allows to improve the scalability of an application as further actors can be easily distributed on further computation nodes

Relation to Object-Oriented Programming

- The behavior of OO applications is implemented in methods of objects and their calls
  - An object has an internal state
  - An object receives messages (method calls)
  - An object performs operations based on the message (the method that was called)
- The behavior of an actor-based application is implemented in the actor behavior depending on messages
- While objects in OO programs operate on shared memory, actors are completely isolated
  - Actors can be seen as a most stringent form of OO programming
  - Actors can be easily distributed

Relation to MPI

- Both the actor model concept as well as MPI use message passing
- MPI –
  - is for parallel, distributed computation
  - is for data parallelism (SIMD)
  - has a fixed number of processes after the program starts
  - uses message passing only if necessary
  - is node-intransparent (the node number determines what to do)
- Actor Model –
  - is for abstraction (and for parallel computation)
  - allows dynamic amounts of actors (on-demand creation/deletion)
  - uses message passing whenever possible
  - is node-transparent (an actor can be distributed on any node)

Actor Model Implementations

- The actor model is just a conceptual computation model
- Several actor model implementations exist for different programming languages, some are:
  - Erlang
    - Pure functional
    - Designed to be distributed and fault-tolerant
    - Implements the actor model: actors are first-level entities
  - Akka (for Java/Scala)
    - JVM library with implementations of concurrent programming models, especially the actor model
    - Implemented in Scala
  - Akka.NET
    - A porting of the Akka library for .NET
- For more details on Akka Actors, their implementation and API, take a look into the appendix

Akka Actors

- Akka provides an implementation of the actor model for JVM-based languages [Akka]
  - Akka is implemented in Scala, which is why many required classes are defined in the Scala namespace
  - Akka can also be used in Java
- All classes on the following slides are defined in one of the following packages:
  - `akka.actor`
  - `akka.util`
  - `scala.concurrent`

Defining an Akka Actor

- An Akka actor has to extend the class `AbstractActor` and must at least implement the method:

```
Receive createReceive();
```

  - Is called if a message is sent to the actor
  - Has to define how a `Receive` object is created after receiving a message
- The method receiveBuilder() returns a pre-implemented builder for `Receive` objects
  - Provides several methods to match a received message
  - `match(Class<P> type, UnitApply<P> apply)` allows to handle messages of the specified `type` by the defined `apply` method (a functional interface with a method expecting an object of type `P`)
  - `match(Class<P> type, TypedPredicate<P> predicate, UnitApply<P> apply)` works like above but also checks the predicate
  - `matchAny(UnitApply<Object> apply)` handles messages of any type by the specified `apply` method

A HelloWorld Actor

- An actor that prints “Hello World” in response to a message with the value "`printHello`" can be implemented as follows:

```
public class HelloWorldActor extends AbstractActor {
        @Override
        public Receive createReceive() {
                return receiveBuilder()
                        .match(String.class,
                                message ->
message.equals("printHello"),
                                message ->
System.out.println("Hello World!"))
                        .matchAny(message ->
unhandled(message))
                        .build();
        }
}
```

This generic match can also be omitted, as it is automatically executed

Further Methods of an Actor

- Actors can override further methods, which are executed if the actor state changes, especially –
  - `preStart()`: Executed before an actor is started
  - `postStop()`: Executed after an actor is stopped
  - `preRestart()` / `postRestart()`: Executed before/after an actor is restarted
- Actors provide three important methods:
  - `getSelf()` delivers a reference (`ActorRef`) to the actor
  - `getContext()` delivers an `ActorContext`, which is especially an `ActorRefFactory` (for creating new actors)
  - `getSender()` delivers a reference to the actor sending the currently processed message

Creating an Actor

- Actors are created using so called `Props`
  - Configuration class for specifying options for an actor creation
  - A `Props` for creating an actor of the given type can be instantiated with:

```
Props Props.create(Class<?> actorType, Object… parameters)
```

  - Props can also configure constructor parameters etc.
- An actor can be created using an `ActorRefFactory`'s method:

```
ActorRef actorOf(Props props)
```

  - Such a factory is implemented –
    - by the `ActorSystem`, which has to be instantiated once per application
    - by the actor context
- `Props`-based instead of ordinary constructor- or factory-based Actor instantiation is used due to different reasons:
  - Ensures that Actors only exists within an `ActorSystem`
  - Returns an `ActorRef` rather than an `Actor`, making it impossible to manually call Actor methods not using messages → less error-prone

Creating a `HelloWorldActor`

- A `HelloWorldActor` can be created with the following code:

```
ActorSystem actorSystem = ActorSystem.create("MySystem");
ActorRef helloWorldActor =
    actorSystem.actorOf(Props.create(HelloWorldActor.class));
```

Is an `ActorRefFactory`

- The first actor has to be created on the actor system
- Further actors can be created from within other actors using their context

Sending Messages

- Messages in Akka are instances of any `Object`
- Messages should be immutable
  - The Akka framework does/can not enforce that
  - Messages have to be immutable by convention
- There are two essential methods for sending messages:
  - `tell` sends a message asynchronously an returns immediately
  - `ask` sends a message asynchronously and returns a `Future` for the possible reply
- `tell`: fire-forget
  - Simplest form of messages
  - Method is defined in the `ActorRef` class and has to be called on the message target actor:

```
void tell(Object message, ActorRef sender)
```

Calling the HelloWorldActor

```
public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("MySystem");
        ActorRef helloWorldActor =

        actorSystem.actorOf(Props.create(HelloWorldActor.class));
        helloWorldActor.tell("printHello", ActorRef.noSender());
        actorSystem.terminate();
}
```

Shuts down the system, otherwise the application runs endlessly

Can be used if no sender can/shall be specified

- Why does this program potentially not print “Hello World” as expected?
  - Using tell has a fire-and-forget semantics, which means that the caller does not block but immediately proceed execution

→ The system is shutdown before the Actor handled the received message

Sending Messages with ask

- `ask`: send-and-receive-future
  - Sends a message to an actor and returns a `Future`, which contains the result of the target actor when it has finished
  - Is not defined in the `ActorRef` class but as a static pattern:

```
Future<?> Patterns.ask( ActorRef target, Object msg,
                        Timeout timeout)
```

  - The asked actor can deliver the result by sending a message with the result back to the sender
  - The sender can wait for the result by calling:

```
T Await.result(Future<T> future, Duration duration)
```

  - Nevertheless, programs should use await as rarely as possible
    - Only if blocking is really necessary
    - Use asynchronous communication otherwise

Stopping Actors

- Actors can be stopped using the following method of an `ActorRefFactory`:

```
void stop(ActorRef actorToStop)
```

  - The context of an actor is used for stopping the actor itself or child actors
  - The actor system is used for stopping top level actors
  - Stopping is performed asynchronously, so the actor may still run when the method already returned
- Actors can also be stopped sending a poison pill:

```
PoisonPill.getInstance()
```

  - The poison pill can be sent via ordinary send operations
  - It is processed like an ordinary message, thus all messages received before will be handled before stopping the actor

Running Actor-based Applications

- An actor-based application runs in an `ActorSystem`, which can be instantiated in two ways:
- Manually (like on a previous slide):

```
ActorSystem actorSystem = ActorSystem.create("MySystem");
// … create actors and send messages to them
```

- Automatically, calling

```
akka.Main.main(String[] supervisorActorName)
```

with the name of the root supervisor actor as the only argument

  - Can, e.g., be called from an ordinary main method
  - The root actor usually has to override the `preStart()` method to initialize the communication

Appendix: Counting Semaphore

- Concept originates from Edsger Dijkstra
- Makes critical sections with capacity *n* possible
  - The critical section can be entered *n* times
  - One access to the critical section is called a *permit*
- The constructor expects the capacity and the fairness

```
Semaphore(int capacity, boolean fair)
```

- Important methods:
  - `acquire()`: takes a permit an reduces the number of available permits; blocks if all permits are acquired
  - `release()`: releases a permit and increases the number of available permits; potentially releases a blocking acquirer
  - `tryAcquire()`: like `acquire()` but does not block
- All calls are also possible with an arbitrary number of permits
  - E.g. `void acquire(int amount)`

Appendix: `ScheduledExecutorService`

- Another kind of `Executor` is the `ScheduledExecutorService`
  - A subinterface of `ExecutorService`
  - Supports scheduled execution of tasks
    - Future:

```
schedule(Runnable task, long delay, TimeUnit timeunit)
```

    - Periodic:

```
scheduleAtFixedRate(Runnable, long initialDelay,
        long period, TimeUnit timeunit)
```

Appendix: ScheduledExecutorService

- This example schedules a task after a delay of three seconds:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
Runnable task = () -> System.out.println("Scheduling: " +
                                         System.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3,
                                              TimeUnit.SECONDS);
TimeUnit.MILLISECONDS.sleep(1337);
long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.println("Remaining Delay: “ + remainingDelay);
```

Adapted from: http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

Appendix: CompletableFutures (1)

- Drawback of Futures: The caller can query a result, but not register a callback
- Java 5 provided the ExecutorCompletionService for that purpose
- In Java 8, the CompletableFuture was introduced, which provides the supplyAsync method, to which the asynchronous task can be passed:

```
CompletableFuture<Integer> futureCount = CompletableFuture.supplyAsync(
        () -> {
                try {
                        // simulate long running task
                        Thread.sleep(5000);
                } catch (InterruptedException ex) { }
                return 20;
        }
);
int count = futureCount.get();
```

Adapted from: http://www.vogella.com/tutorials/JavaConcurrency/article.html#completablefuture

Appendix: CompletableFutures (2)

- On a CompletableFuture, the thenApply method can be called to define behavior that is executed after the task has finished
- Given our previous futureCount, we can, for example, call:

```
CompletableFuture<String> modified =
        futureCount.thenApply((Integer count) -> {
                int transformedValue = count * 10;
                return transformedValue;
        })
        .thenApply(transformed -> "Finally create a string: " +
transformed);

System.out.println(modified.get());
```

  - After execution, the value is multiplied by ten and put into a string
  - The result of the new CompletableFuture is printed: "Finally created a string: 200"
  - Chaining two thenApply calls is not necessary in this example, but can be useful when passing a CompletableFuture between different contexts and calling thenApply with different context-dependent information

Appendix: Thread-safe Classes

- The Java API provides several thread-safe classes, which can be safely used by multiple threads concurrently
- `BlockingQueue` (interface)
  - Queue with operations that block if queue is full/empty when putting/retrieving an element
  - Methods:
    - `put(…)` blocks if full
    - `take()` blocks if empty
  - Concrete implementations (extract):
    - `ArrayBlockingQueue`: limited capacity
    - `LinkedBlockingQueue`: optionally limited capacity
    - `PriorityBlockingQueue`: sorted
- `ConcurrentHashMap`
- Concurrently accessing non-thread-safe classes should be avoided (memory inconsistencies)

Appendix: An Improved HelloWorld Actor

- The `match` methods have functional interfaces as parameters, which allows to define the logic for handling a certain message in a dedicated method and pass a reference to that method:

```
public class HelloWorldActor extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(String.class,
                message -> message.equals("printHello"),
                    this::handleStringMessage)
            .matchAny(message -> unhandled(message))
            .build();
    }

    private void handleStringMessage(String message) {
        System.out.println("Hello World!");
    }
}
```

Appendix: Calculator Example for ask (1)

- The following actor expects an integer array as message and returns the sum of the array elements via message:

```
public class AdditionActor extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(Integer[].class, this::performAddition)
            .build();
    }

    private void performAddition(Integer[] array) {
        int sum = 0;
        for (int value : array) {
            sum += value;
        }
        getSender().tell(sum, getSelf());
    }
}
```

Appendix: Calculator Example for ask (2)

- The result of the addition can be retrieved using the ask operation:

```
public class AdditionUser {
    public static void main(String[] args) throws Exception {
        ActorSystem actorSystem = ActorSystem.create("MySystem");
        ActorRef additionActor =
            actorSystem.actorOf(Props.create(AdditionActor.class));

        Timeout timeout = new Timeout(1, TimeUnit.SECONDS);
        Integer[] sourceValues = new Integer[] {1, 2, 3, 4};
        Future<Object> future =
            Patterns.ask(additionActor, sourceValues, timeout);

        int result = (Integer) Await.result(future, timeout.duration());
        System.out.println(result);

        actorSystem.terminate();
    }
}
```

Appendix: Supervision

- An actor gets supervisor of another actor if it creates it
- If a child actor fails (i.e. throws an exception), it suspends and sends a message to the supervisor
  - So called “let it crash” semantics
- The supervisor has four options to handle the situation
  - Resume the child with the existing state
  - Restart the child with a clear state
  - Stop the child permanently
  - Fail itself
- Fail Strategies
  - One-For-One-Strategy: Fail handling is applied only to the failing child
    - Recommended strategy
  - All-For-One-Strategy: Fail handling is applied to all children
    - E.g. all childs are restarted if one of them fails
    - Only reasonable if children have tight dependencies

Appendix: Additional Capabilities

- Forwarding messages, preserving the original sender, by calling on the target `ActorRef`:

```
void forward(Object message, ActorContext context)
```

- Defining timeouts for message receives
- Hot-swapping actor behavior: e.g. for implementing finite state machines
- Message stashing: Messages can be temporarily stashed away
- Typed actors: Typed interfaces instead of untyped messaged

- More about Akka actors can be found in its documentation: [Akka]