Threads & Tasks: Executor Framework

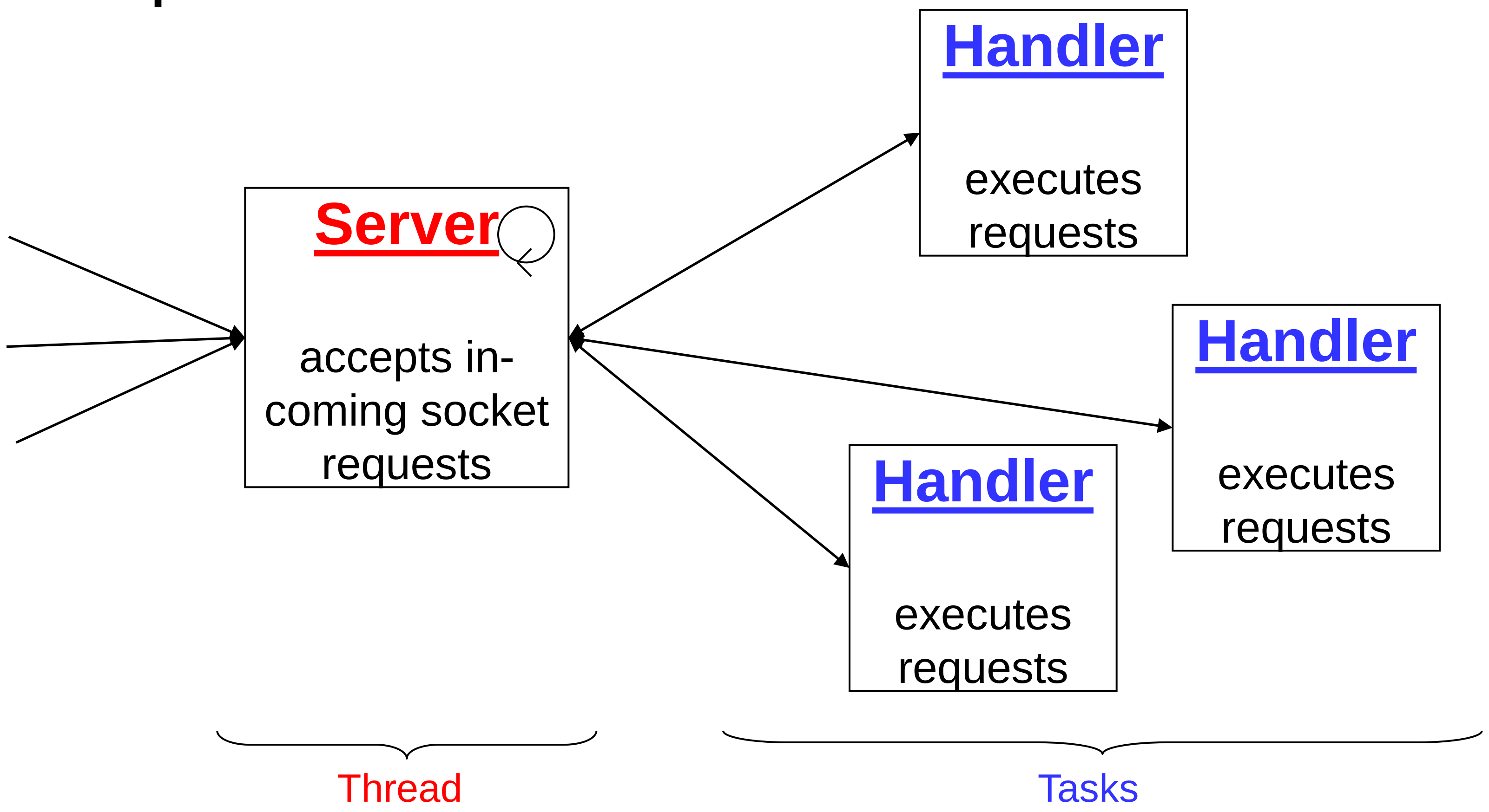
- **Introduction & Motivation**
  - WebServer
- **Executor Framework**
- **Callable and Future**

Threads & Tasks

- **Motivations for using threads**
  - Actor-based
    - Goal: Create an autonomous, active object which can react on events
    - Example: swing event handler thread
  - Task-based
    - Task = Logical unit of work
    - Goal: Asynchronous execution of a task
    - Example: mandelbrot slice task
- **Task based concurrency**
  - Simplifies program organization
  - Facilitates error recovery by providing natural transaction boundaries
  - Promotes concurrency by providing a natural structure for parallelizing work

Threads & Tasks

- **Example: Webserver**



Introduction: Implementation of a WebServer1

```
public class WebServer1 {
   public static void main(String[] args) throws IOException {
      ServerSocket serverSocket = new ServerSocket(80);
      while(true) {
         Socket s = serverSocket.accept();
         handleRequest(s);
      }
   }
}
```

- **Single-Threaded Server:**
  Tasks are executed within the server thread
  – Poor performance, can only handle one request at a time
  – While server is processing, new requests must wait
  – Possible if request processing is very fast

Introduction: Implementation of a WebServer2

```
public class WebServer2 {
   public static void main(String[] args) throws IOException {
      ServerSocket serverSocket = new ServerSocket(80);
      while(true) {
         final Socket s = serverSocket.accept();
         Thread t = new Thread(){
            public void run(){ handleRequest(s); }
         };
         t.start();
      }
   }
}
```

- **Explicitly creating threads for tasks**
  Each task is executed in its own thread
  - handleRequest must be thread-safe
  - Excessive thread creation: Scheduling overhead / memory consumption

Introduction: Implementation of a WebServer3

```
public class WebServer3 {
   public static void main(String[] args) throws IOException {
      final ServerSocket serverSocket = new ServerSocket(80);
      for (int i = 0; i < 10; i++) {
         Thread t = new Thread() {
            public void run() {
               while (true) {
                  try { handleRequest(serverSocket.accept()); }
                  catch (IOException e) { /* ... */ }
               }
            }
         };
         t.start();
      }
   }
}
```

Introduction: Implementation of a WebServer3

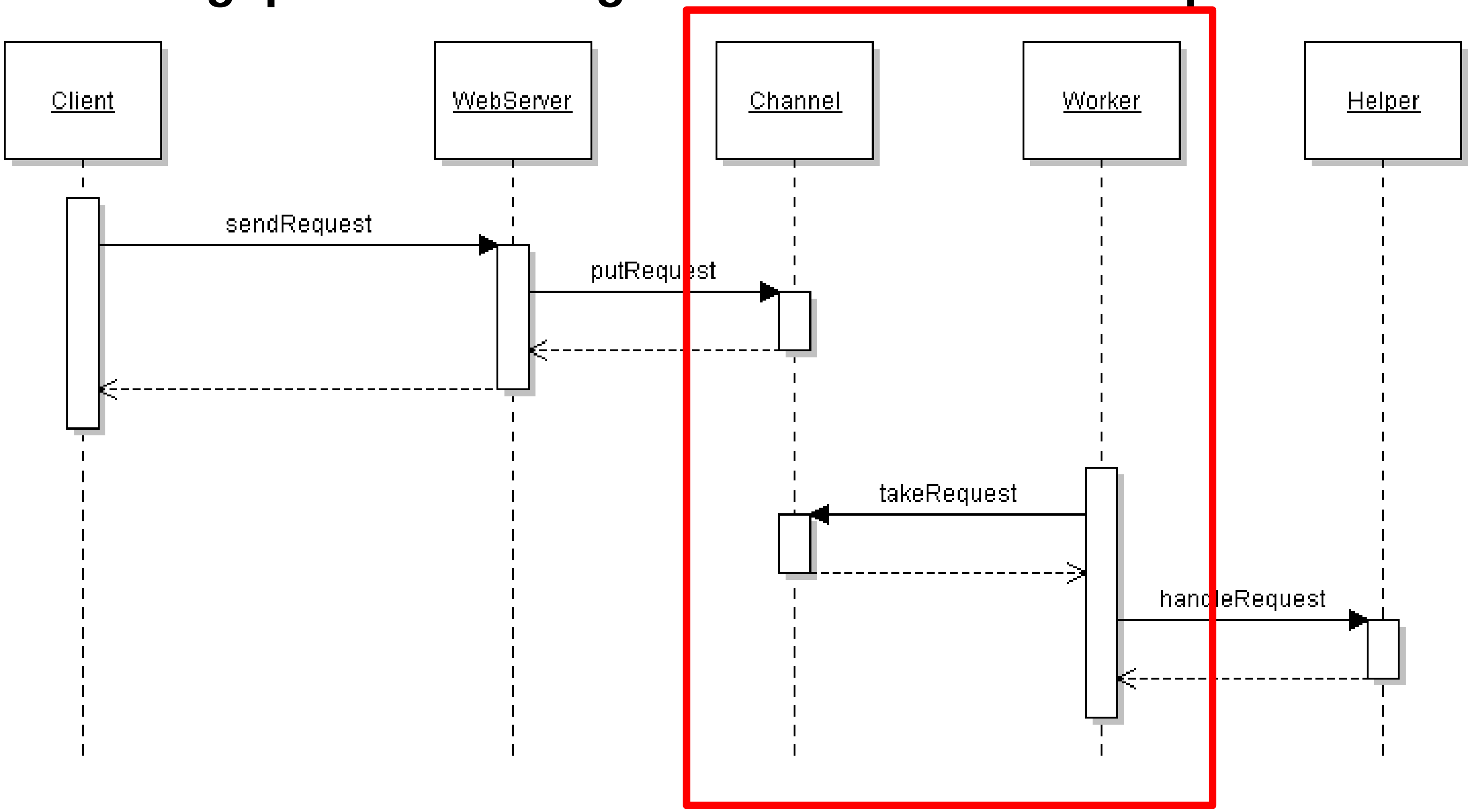
- **Disadvantages**
  - Maybe incorrect, ServerSocket.accept() is *not* documented as threadsafe
  - No flexibility, i.e. exactly 10 threads
    - No creation of new threads
    - No deletion of unused threads
  - All threads are pre-created, no lazy allocation
  - No life-cycle management, i.e. "pool" cannot be stopped
  - No error handling, if an exception is not catched, thread terminates silently
  - No flexibility in the order of pending requests
    - They are stored in the Server-Socket queue
  - No full control over size of queue for pending requests
    - backlog parameter only specifies *maximum* length of the queue

Threads & Tasks: Executor Framework

- **Introduction & Motivation**
- **Executor Framework**
- **Callable and Future**

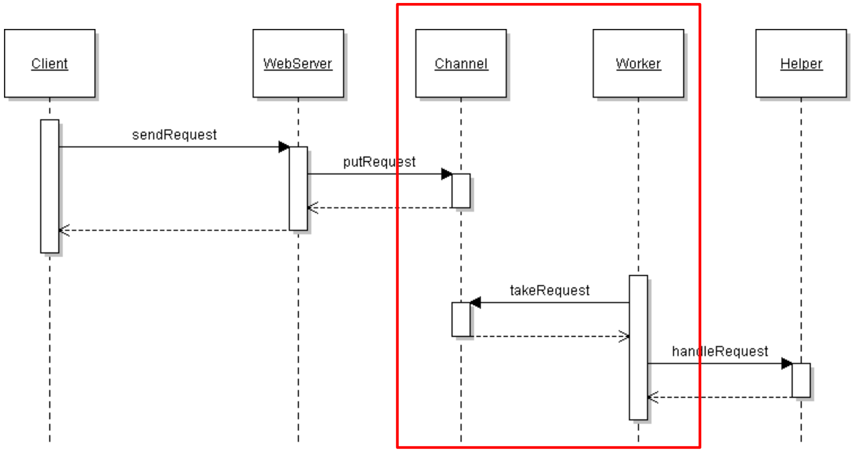
Executor Framework

- **Fills the gap between single-threaded and thread-per-task**



Executor Framework

- **Participants**
  - Worker: One thread is used to execute many unrelated tasks
    - These threads are called worker threads / background threads
    - May be organized in a thread pool (if more than one thread is used)
    - May provide a flexible thread management
  - Channel: A buffer which holds pending requests
    - May be bounded
    - Implements the producer-consumer pattern



Executor Framework

- **Executor = Channel + Worker**

```
public interface Executor {
    void execute(Runnable command);
}
```

  - Decouples task submission from task execution
  - Executes the given command at some time in the future
  - The command may be executed
    - in a new thread / in a pooled thread / in the calling thread

- **Runnable = Task**

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

  - Limitation:
    - Method *run* cannot return a result (results are placed in shared fields)
    - Method *run* cannot declare a checked exception

Introduction: Implementation of a WebServer4

```
public class WebServer4 {
   public static void main(String[] args) throws IOException {
      Executor exec = new ThreadPoolExecutor(10);
      ServerSocket serverSocket = new ServerSocket(80);
      while(true){
         final Socket s = serverSocket.accept();
         Runnable task = new Runnable(){
            public void run(){ handleRequest(s); }
         };
         exec.execute(task);
      }
   }
}
```

Fachhochschule Nordwestschweiz

Executor: Implementation

```
class ThreadPoolExecutor implements Executor {
   private final BlockingQueue<Runnable> queue
                     = new LinkedBlockingQueue<Runnable>();

   public void execute(Runnable r) { queue.offer(r); }

   public ThreadPoolExecutor(int nrThreads) {
      for (int i = 0; i < nrThreads; i++) { activate(); }
   }

   private void activate() {
      new Thread(new Runnable() {
         public void run() {
            try {
               while (true) { queue.take().run(); }
            } catch (InterruptedException e) { /* die */ }
         }
      }).start();
   }
}
```

Executor: Simple implementations

- **DirectExecutor: Synchronous execution (in calling thread)**

```
class DirectExecutor implements Executor {
   public void execute(Runnable r) { r.run(); }
}
```

  – With this executor Webserver4 = Webserver1

- **Thread per task executor**

```
class ThreadPerTaskExecutor implements Executor {
   public void execute(Runnable r) {
      new Thread(r).start();
   }
}
```

  – With this executor Webserver4 = Webserver2

Executor: Advanced Implementations

- **Execution Policies**
  - Execution order of submitted tasks (FIFO, LIFO, Priority Queue)
    - More control as with Java's setPriority
  - Number of threads which execute concurrently
  - Maximal size of queue with pending tasks
  - Actions taken before / after task execution (startup/cleanup)

- **Factory: java.util.concurrent.Executors**
  - Provides several implementations for thread pools which implement different policies
  - All factory methods return an executor which implements the ExecutorService interface

Executor: Advanced Implementations

- **Executors.newFixedThreadPool**
  - Threads are created up to a fixed number
  - Threads which die due to an unexpected exception are replaced
- **Executors.newCachedThreadPool**
  - Creates new threads as needed, reusing previously constructed threads if they are available
- **Executors.newSingleThreadExecutor**
  - Uses single worker thread
  - Worker thread is replaced if an unexpected exception occurs
- **Executors.newScheduledThreadPool**
  - Creates a ScheduledExecutorService which supports
    - Periodic tasks (scheduleAtFixedRate / scheduleWithFixedDelay)
    - Delayed tasks (schedule)

ThreadFactory

- **Some factory methods on j.u.c.Executors take a ThreadFactory**

```
public interface ThreadFactory {
  Thread newThread(Runnable r);
}
```

- **Enables applications to use**
  - special Thread subclasses
  - priorities
  - custom named threads
  - daemon flag

Executors: Design Considerations

- **Identity**
  - Different tasks are executed with the same thread
    => thread-specific contextual control techniques are difficult
    - Use of ThreadLocals
    - Use of security contexts
- **Queuing**
  - Runnable tasks that are waiting in queues do not run
    => if a currently running task blocks waiting for a condition
    produced by a task still waiting in a queue => system may freeze up
    - Use enough worker threads
    - Restrict dependencies
    - Create custom queues that understand the dependencies
      => Fork-Join Framework

ExecutorService: Executor Life-Cycle

- **ExecutorService: provides life-cycle management methods**

```
interface ExecutorService extends Executor {
   void shutdown();
   List<Runnable> shutdownNow();
   boolean isShutdown();
   boolean isTerminated();
   boolean awaitTermination(long timeout, TimeUnit unit)
                               throws InterruptedException;
}
```

  – Supports shutdown methods
    - `shutdown`: Graceful shutdown: finish pending tasks, do not accept new ones
    - `shutdownNow:` Abrupt shutdown: running tasks are interrupted, returns list of tasks that were not started
  – awaitTermination: awaits until executor service is terminated

running → shutting down → terminated

Executors and JMM

- **Memory consistency effects**
  - Actions in a thread prior to submitting a Runnable object to an **Executor** happen-before its execution begins (possibly in another thread)

  - Actions in a task which is executed by a **SingleThreadExecutor** happen-before actions executed in subsequent tasks (even if the subsequent task is executed by another thread due to an exception)

Threads & Tasks: Executor Framework

- **Introduction & Motivation**
- **Executor Framework**
- **Callable and Future**

Result-bearing tasks: Callable & Future

- **Callable: Task with a result / exception**

```
interface Callable<V> {
   V call() throws Exception;
}
```

- **Future: represents a future result of a task**

```
interface Future<V> {
   boolean cancel(boolean mayInterruptIfRunning);
   boolean isCancelled();
   boolean isDone();
   V get() throws InterruptedException, ExecutionException,
                  CancellationException;
   V get(long timeout, TimeUnit unit)
         throws InterruptedException, ExecutionException,
                  CancellationException, TimeoutException;
}
```

Callable & Future

- **Submitting tasks**

```
interface ExecutorService {
   // ... Lifecycle methods
   Future<?> submit(Runnable task);
   <T> Future<T> submit(Callable<T> task);
   <T> Future<T> submit(Runnable task, T result);

   <T> List<Future<T>> invokeAll(
           Collection<? extends Callable<T>> tasks)
           throws InterruptedException;

   <T> T invokeAny(
           Collection<? extends Callable<T>> tasks)
           throws InterruptedException, ExecutionException;
}
```

Callable & Future

- **States of a Task**

created → submitted → started → completed

- **get()**
  - If completed: returns immediately
(returns result or throws ExecutionException)
  - If not completed: method call blocks
    - If terminates regularly => result
    - If terminates with an exception => ExecutionException
    - If cancelled => CancellationException
    - If thread calling get was interrupted => InterruptedException

CompletionService

- **CompletionService = Executor + BlockingQueue**
  - Decouples production of new tasks from the consumption of the results of completed tasks
    - Producers submit tasks for execution
    - Consumers take completed tasks and process their results

```
interface CompletionService {
   Future<V> poll(); // returns available future or null
   Future<V> poll(long timeout, TimeUnit unit) throws IE;
   Future<V> take() throws IE; // waits for a future
   Future<V> submit(Callable<V> task);
   Future<V> submit(Runnable task, V result);
}
```

      - `take` retrieves and removes the next completed task, potentially waiting
- **ExecutorCompletionService**
  - Uses a separate Executor to schedule the tasks

CompletionService: Sample

- **Solver**
  - Method solve takes a list of solvers (of type Callable<Result>)
  - Result of first solver is returned

```
void solve(Executor e, Collection<Callable<Result>> solvers)
                                      throws InterruptedException {
  CompletionService<Result> cs = new ExecutorCompletionService<>(e);
  int n = solvers.size();
  List<Future<Result>> futures = new ArrayList<Future<Result>>(n);
  Result result = null;
  try {
    for (Callable<Result> s : solvers) futures.add(cs.submit(s));
    ...
```

CompletionService: Sample

```
    for (int i = 0; i < n; ++i) {
      try {
        Result r = cs.take().get();
        if (r != null) { // solver may return null
          result = r;
          break;
        }
      } catch (ExecutionException ignore) {}
    }
  }
  finally {
    // cancel all pending solvers
    for (Future<Result> f : futures) f.cancel(true);
  }

  if (result != null)  use(result);
}
```

Callable & Future and JMM

- **Memory consistency effects**
  - Actions in a thread prior to the submission of a Runnable or Callable task to an ExecutorService happen-before any actions taken by that task
  - Any actions taken by a Runnable or Callable task executed by an ExecutorService happen-before the result is retrieved via Future.get()