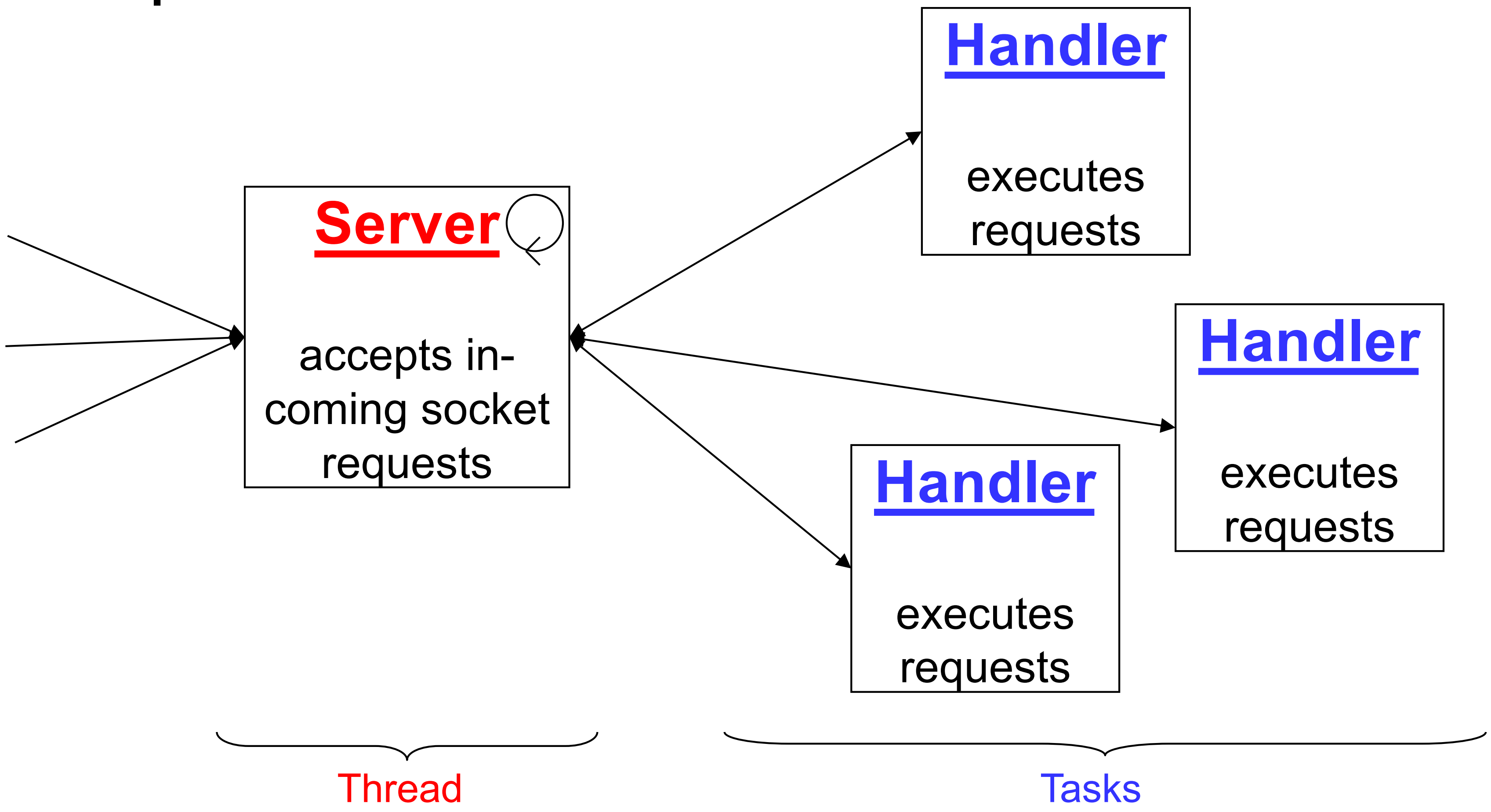
# Threads & Tasks: Executor Framework

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

# 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) {
         Socket s = serverSocket.accept();
         Thread t = new Thread(() -> 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(() -> {
            while (true) {
               try { handleRequest(serverSocket.accept()); }
               catch (IOException e) { /* ... */ }
            }
         });
         t.start();
      }
   }
}
```

# Introduction: Implementation of a WebServer3

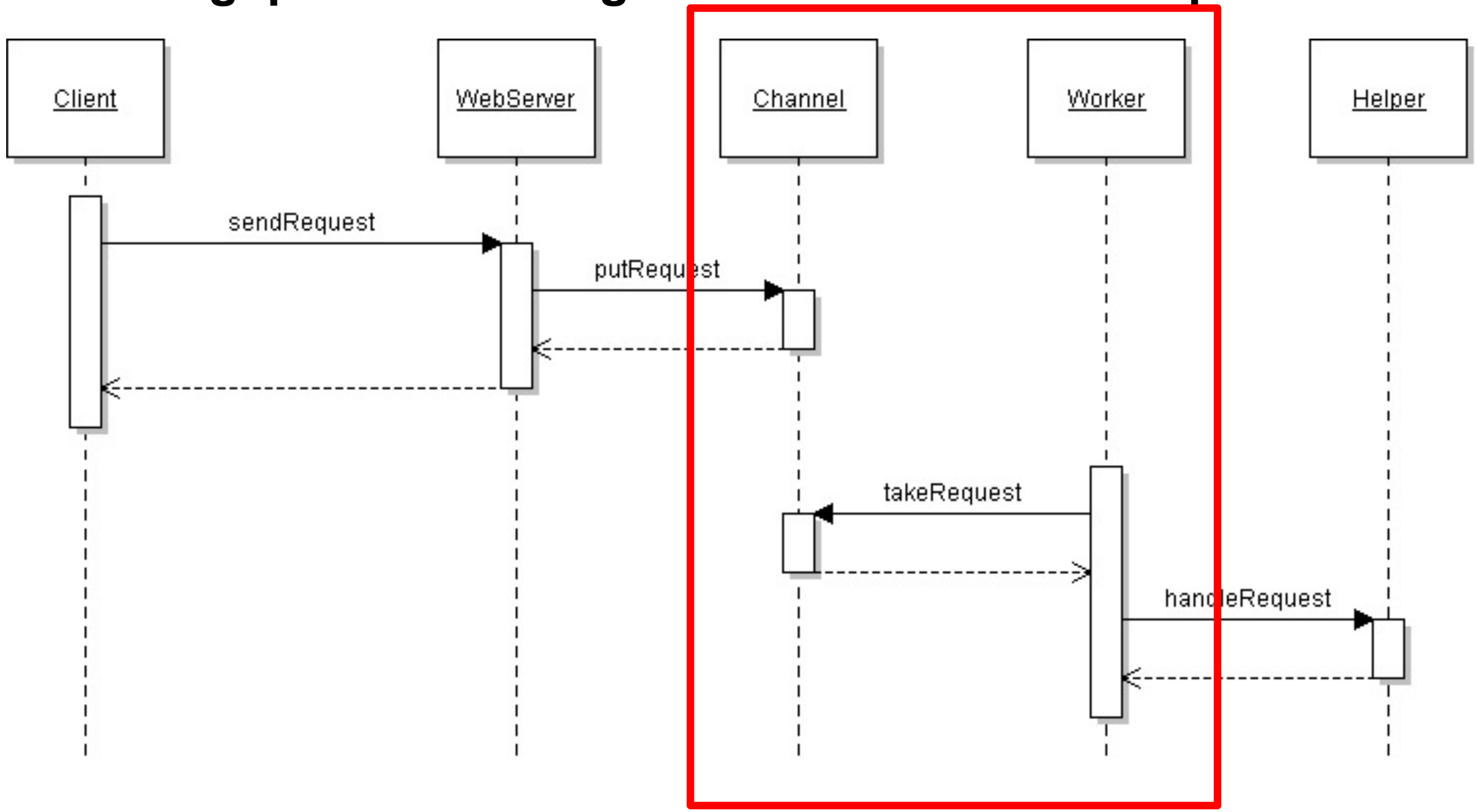
- **Disadvantages**
  - Maybe incorrect, ServerSocket.accept() is *not* documented as threadsafe
  - No flexibility, i.e. exactly 10 threads
    - No creation of new threads
    - No deletion of unused threads
  - All threads are pre-created, no lazy allocation
  - No life-cycle management, i.e. "pool"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

# Threads & Tasks: Executor Framework

- Introduction & Motivation
- Executor Framework
- Callable and Future
- Fork-Join Motivation
- Fork-Join Framework

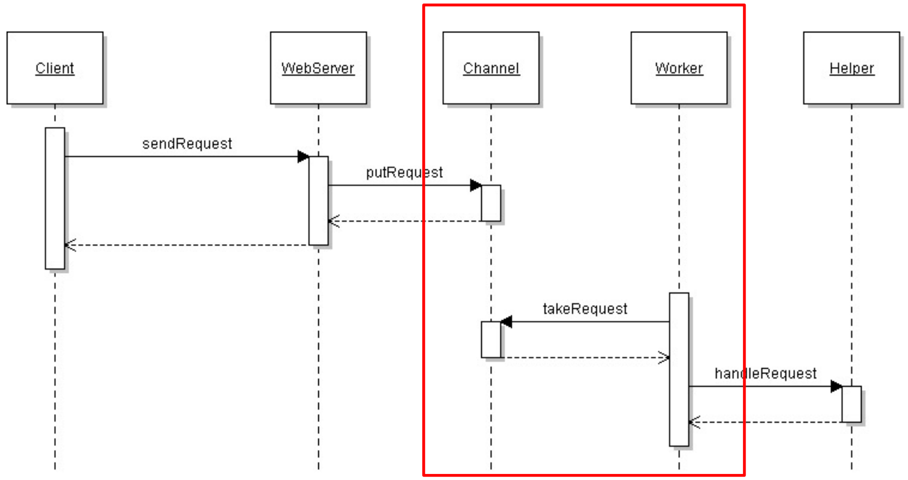
# Executor Framework

- **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 task);
}
```

  - Decouples task submission from task execution
  - Executes the given task at some time in the future
  - The task 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 MyThreadPoolExecutor(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);
      }
   }
}
```

- **Using lambda notation**

```
         exec.execute(() -> handleRequest(s));
```

# Executor: Implementation

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

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

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

   private void activate() {
      new Thread(() -> {
         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)
  - Number of threads which execute concurrently
  - Maximal size of queue with pending tasks
  - Actions taken before / after task execution
    - E.g. extension hooks in class ThreadPoolExecutor
      - `void beforeExecute(Thread t, Runnable r) { }`
      - `void afterExecute(Runnable r, Throwable t) { }`
- **Factory: java.util.concurrent.Executors**
  - Factory methods for preconfigured ThreadPoolExecutor instances
  - E.g. creating an executor using 10 worker threads:

```
Executor es = Executors.newFixedThreadPool(10);
```

# 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
  - custom named threads
  - daemon flag

# Executors: Design Considerations

- **Identity**
  - Different tasks are executed with the same thread
    - If task uses ThreadLocals it needs to clean up afterwards
  - Logical flows can be executed on multiple threads
    - Information in a ThreadLocal is not available inside the whole flow

> ➢ There is no more a one-to-one relation between a logical flow and a technical Thread

- **Granularity**
  - Promotes concurrency by providing a natural structure for parallelizing work

# 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
- Fork-Join Motivation
- Fork-Join Framework

# 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 extends Executor {
   // ... Lifecycle methods
   <T> Future<T> submit(Callable<T> task);

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

- `take` retrieves and removes the next completed task, potentially waiting

- **ExecutorCompletionService**
  - Uses a separate Executor to schedule the tasks

# 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()

# Threads & Tasks: Executor Framework

- Introduction & Motivation
- Executor Framework
- Callable and Future
- Fork-Join Motivation
- Fork-Join Framework

Fachhochschule Nordwestschweiz

# MergeSort using Executor

```
public class MS1 implements Runnable {
  public final int[] is, tmp; private final int l, r;
  private final ExecutorService ex;

  public MS1(int[] is, int[] tmp, int l, int r, ExecutorService ex) {
    this.is = is; this.tmp = tmp; this.l = l; this.r = r; this.ex = ex;
  }

  public void run() {
    if(r – l <= 1) return;
    else {
      int mid = (l+r) / 2;
      MS1 left = new MS1(is, tmp, l, mid, ex);
      MS1 right = new MS1(is, tmp, mid, r, ex);

      Future<?> lf = ex.submit(left);
      Future<?> rf = ex.submit(right);
     try {lf.get(); rf.get(); } catch (InterruptedException | ExecutionException e) {}
      merge(is, tmp, l, mid, r);
    }
  }
  private void merge(int[ ] is, int[ ] tmp, int l, int m, int r) {...}
}
```

# MergeSort using Executor

```
int[] data = ...
int[] tmp = new int[data.length];
ExecutorService es = Executors.newFixedThreadPool(3);

MS1 ms = new MS1(data, tmp, 0, data.length-1, es);
Future<?> f = es.submit(ms);
f.get();
// When this line is reached, ms.is is sorted!
```

# MergeSort using hard-working Executor

```
public class MS2 implements Runnable {
  ...
  public void run() {
    if(r - l <= 1) return;
    else {
      int mid = (l+r) / 2;
      MS2 left = new MS2(is, tmp, l, mid, hex);
      MS2 right = new MS2(is, tmp, mid, r, hex);
      Future<?> lf = ex.submit(left);
      Future<?> rf = ex.submit(right);

      // don't hang around, work!
      while (!(lf.isDone() && rf.isDone())) {
         ex.helpOtherWorkers();
      }
     try {lf.get(); rf.get(); } catch (InterruptedException | ExecutionException e) {}
      merge(is, tmp, l, mid, r);
    }
  }
}
```

# MergeSort using hard-working Executor

```
public class MS2 implements Runnable {
  ...
  public void run() {
    if(r - l <= 1000) Arrays.sort(is); return;
    else {
      int mid = (l+r) / 2;
      MS2 left = new MS2(is, tmp, l, mid, hex);
      MS2 right = new MS2(is, tmp, mid, r, hex);
      Future<?> lf = ex.submit(left);
      Future<?> rf = ex.submit(right);

      // don't hang around, work!
      while (!(lf.isDone() && rf.isDone())) {
         ex.helpOtherWorkers();
      }
     try {lf.get(); rf.get(); } catch (InterruptedException | ExecutionException e) {}
      merge(is, tmp, l, mid, r);
    }
  }
}
```

# ForkJoin decomposition

- **Parallel version of divide-and-conquer algorithms**

```
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

# Examples of divide-and-conquer algorithms

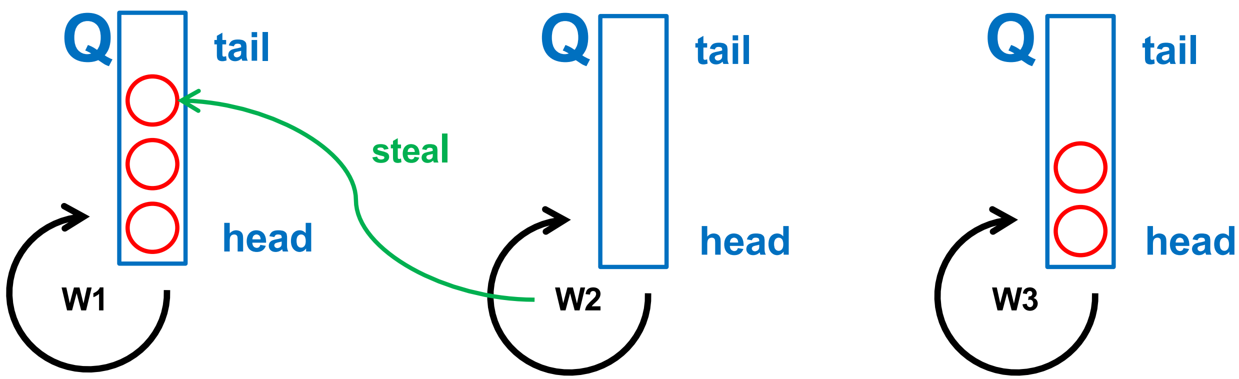
- **Sort**
  - Mergesort
  - Quicksort
- **Search in unsorted data**
  - Arrays
  - Trees
- **Numerics**
  - Matrix multiplications
- **Geometrics**
  - Convex hull

# Threads & Tasks: Executor Framework

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

# ForkJoin Framework: Work stealing

- Create a limited number of worker threads
- Each worker thread maintains a private double-ended work queue
- When forking, worker pushes new task at the head of its deque
- When waiting or idle, worker pops a task off the head of its deque and executes it instead of sleeping
- If a worker's deque is empty, it steals a task off the tail of another randomly chosen worker



# Work stealing: Efficiency

- **Reduced contention compared to shared work queue**
  - No contention for head since it is only accessed by the owner
  - No contention between head and tail access (good deque algorithms)
  - Almost never contention for tail because stealing is infrequent
    - Workers access their deque in LIFO (Stack) order
    - Size of tasks gets smaller as problem is divided
    - Tasks are stolen from the tail and are generally a big chunk!
- **Good load balancing**
  - If work is unequally distributed, it is rebalanced via stealing
  - Without central coordination
  - With little scheduling overhead
  - With minimal synchronization costs

# Task granularity and structure

- **Small Tasks (low threshold)**
  - Maximizing parallelism
  - More fine-grained tasks keep more CPUs busy
  - Improves load balancing and locality
  - Decreases time that CPUs must wait for one another
  - Fast if data fits into cache

- **Large Tasks (high threshold)**
  - Minimizing overhead
  - Task creation and management versus sequential execution
  - Memory consumption, garbage collection

**Fork-join task framework is designed to minimize per task overhead for compute-intensive tasks.**

# MergeSort using ForkJoin

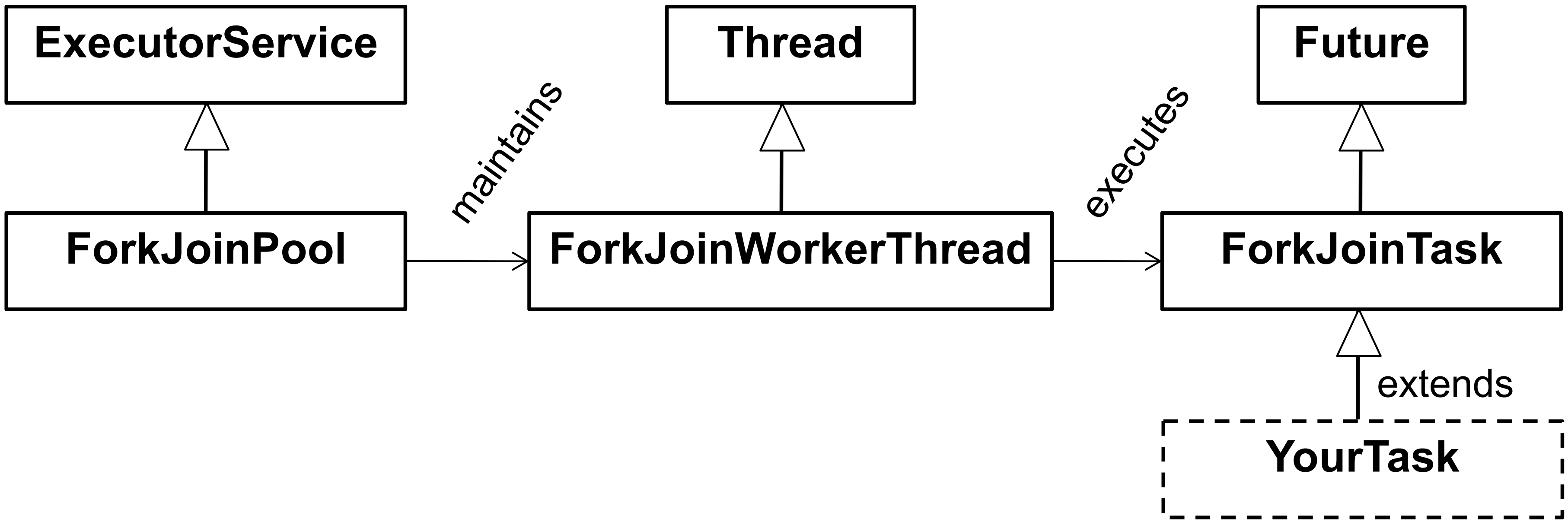
```
public class FJMS extends RecursiveAction {
  public final int[] is, tmp; private final int l, r;

  public FJMS(int[] is, int[] tmp, int l, int r) {
    this.is = is; this.tmp = tmp; this.l = l; this.r = r;
  }

  protected void compute() {
    if (r - l<= 100000) Arrays.sort(elems, l, r);
    else {
      int mid = (l + r) / 2;
      FJMS left = new FJMS (is, tmp, l, mid);
      FJMS right = new FJMS (is, tmp, mid, r);
      left.fork();
      right.invoke();
      left.join();
      merge(is, tmp, l, mid, r);
    }
  }

  private void merge(int[ ] es, int[ ] tmp, int l, int m, int r) {...}
}
```

# Executing FJ Tasks: ForkJoinPool

```
public class ForkJoinRunner {
  public static void main(String[] args) {
    int[] data = ...
    int[] tmp = new int[data.length];
    ForkJoinPool fjPool = new ForkJoinPool();
    FJMS ms = new FJMS(data,tmp,0,data.length);
    fjPool.invoke(ms);
    // When this line is reached, ms.is is sorted!
  }
}
```

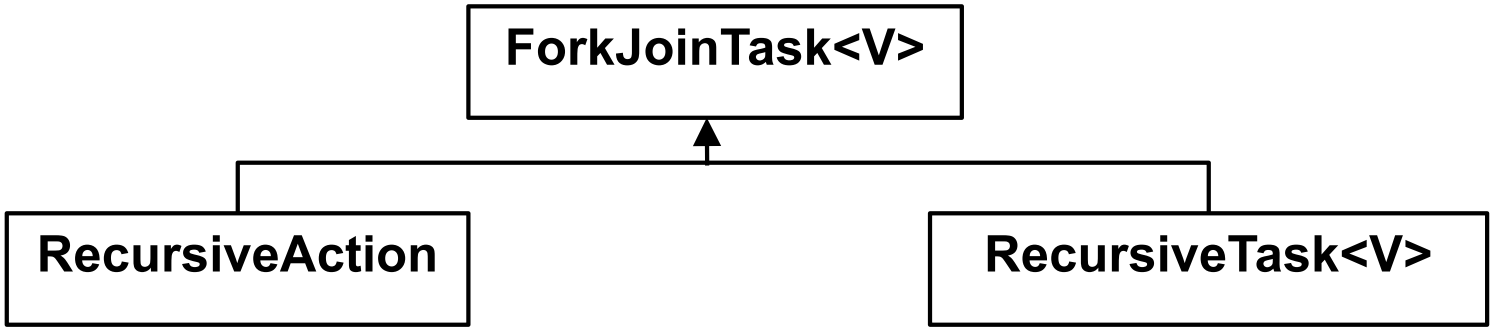
- Default constructor of `ForkJoinPool` sets parallelism to `Runtime.getRuntime().availableProcessors()`
- Normally a single `ForkJoinPool` is used for all parallel task execution in a program or subsystem: **`ForkJoinPool.commonPool()`**
- Because it uses daemon threads, there is no need for explicit shutdown

# ForkJoin Framework: Overview




- A ForkJoinPool maintains typically #CPU ForkJoinWorkerThreads
- Each ForkJoinWorkerThread executes several ForkJoinTasks
- Your tasks subclass ForkJoinTask

# RecursiveAction vs. RecursiveTask




```
protected abstract void compute();
```

- Resultless
- Typically reuses argument field for result

```
protected abstract V compute();
```

- Result-bearing

# ForkJoinPool: API

```
public class ForkJoinPool extends AbstractExecutorService {
   // ... ExecutorService methods

   // Constructors
   public ForkJoinPool(int parallelism) { ... }

   // Task processing
   public <T> T invoke(ForkJoinTask<T> task) { ... } //block.
   public void execute(ForkJoinTask<?> task) { ... } //nonblock.
   public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) { ... }

   // Monitoring operations (used for tuning)
   public long getStealCount() { ... }
   public long getQueuedTaskCount() { ... }
   public int getQueuedSubmissionCount() { ... }

   ...
}
```

# ForkJoinTask: API

```
public abstract class ForkJoinTask<V> implements Future<V>, Serializable
{
   // ... Future methods

   // Subtask control
   public final ForkJoinTask<V> fork() { ... } // put into dequeue
   public final V join() { ... } // work on other tasks
   public final V invoke() { ... } // invoke with current FJWT

   // Bulk invocations
   public static void invokeAll(ForkJoinTask<?> t1,
                                ForkJoinTask<?> t2) { ... }
   public static void invokeAll(ForkJoinTask<?>... tasks) { ... }
   public static <T extends ForkJoinTask<?>> Collection<T>
                           invokeAll(Collection<T> tasks) { ... }

   // adapters
   public static ForkJoinTask<?> adapt(Runnable runnable) { ... }
   ...
}
```

# Summary: ForkJoin Framework

- **Framework for executing computational tasks**
  - Offers a portable way to express many parallel algorithms
  - Optimized for fine grained tasks
  - Based on work stealing
  - Ready for the massive multi core future
- **Usage**
  - Implement a subclass of RecursiveAction or RecursiveTask
  - Execute it on a ForkJoinPool