Executing tasks in Threads

- Executing sequentially
- Disadvantages of unbound thread creation

Executor framework

- Execution policies
- Thread pools
- The Executors Factory
- The factory methods

Exploitable parallelism

- The Callable Interface
- Completion service
- Delay and periodic tasks

Task Execution Summary

- Completion Service Examples

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Part II
Structuring Concurrent Applications

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Tasks are natural

Tasks a building blocks

Most concurrent applications are organized around the execution of *tasks*: abstract, discrete units of work. Dividing the work of an application into tasks simplifies program organization, facilities error recovery by providing natural transaction boundaries and promotes concurrency by providing a natural structure for parallellizing work.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Designing your program organization

- Identify sensible task boundaries
- Ideally, tasks are *independent* activities: work that does not depend on other tasks.
- Independence facilitates concurrency
- To be able to do load balancing, the tasks should make out only small fractions of the application's processing capacity.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Server applications and degradation

- Server applications should show *good throughput* and *good responsiveness* under normal load.
- The application should exhibit *graceful degradation* under heavy (over) load.
- Choosing good task boundaries and a sensible task execution policy will help achieve these goals
- Most server applications have a natural choice of task boundaries

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

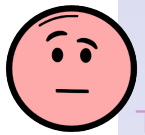
Completion Service Examples

Singlethreadedness, too little...

- Network and disk operations block. This makes the cpu free for other tasks. But here there is no opportunity to exploit that.

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

    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```



Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

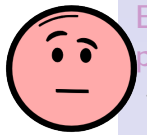
Completion Service Examples

Unbound MultiThreadedness

On the other hand, a thread per task might not be a good idea.

```
public class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }

    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```



Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Consequences of multi threading

In the second solution a thread is started per request.
This has Three consequences:

- Task processing is offloaded from the main (network daemon) thread, enabling the main loop to accept incoming connections more quickly
- Task can be processed in parallel, so more requests can be serviced simultaneously.
- Task handling code must be thread-safe, because it may be invoked concurrently.
- In the example the number of Threads generated is unbound, risking resource depletion, which wins it a

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Disadvantages of unbound thread creation

For production purposes (large webservers for instance) the task per thread approach has some drawbacks.

Thread life cycle overhead Thread creation and tear down is not for free. It involves the JVM and underlying OS. For lots of lightweight threads this is not very efficient.

Resource consumption Active Threads consume extra memory, for instance to provide for a thread stack.

Thread/CPU ratio If there are less CPU's than threads, some threads sit idle. If you have many of these, these idle threads use up lots of memory.

Stability There is a limit on how many threads you can have concurrently. If you hit this limit your program will most likely become unstable.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Unbounded thread creation example

```
package boom ;
public class ThreadBomb implements Runnable {
    private static int b=0;
    /**
     * @param args
     */
    public static void main ( String [] args ) {
        try {
            while ( true ){
                new Thread (new ThreadBomb ()). start ();
                b++;
            }
        } finally {
            System .err . println (b);
            System . exit (0);
        }
    }
    ThreadBomb (){ }
    public void run (){
        while ( true ) {
            Thread . yield ();
        }
    }
}
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Executor framework

Executor Interface

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

- Task are logical units of work. And threads provide a mechanism to run them asynchronously.
- Thread pools are the means to keep the amount of threads manageable.
- The abstraction for task execution in Java is *not* Thread but Executor.
- Although the interface is simple, it is the basis of a flexible framework for asynchronous execution.
- Executor decouples task-*submission* from task-*execution*.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Executor web server

```
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
            = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

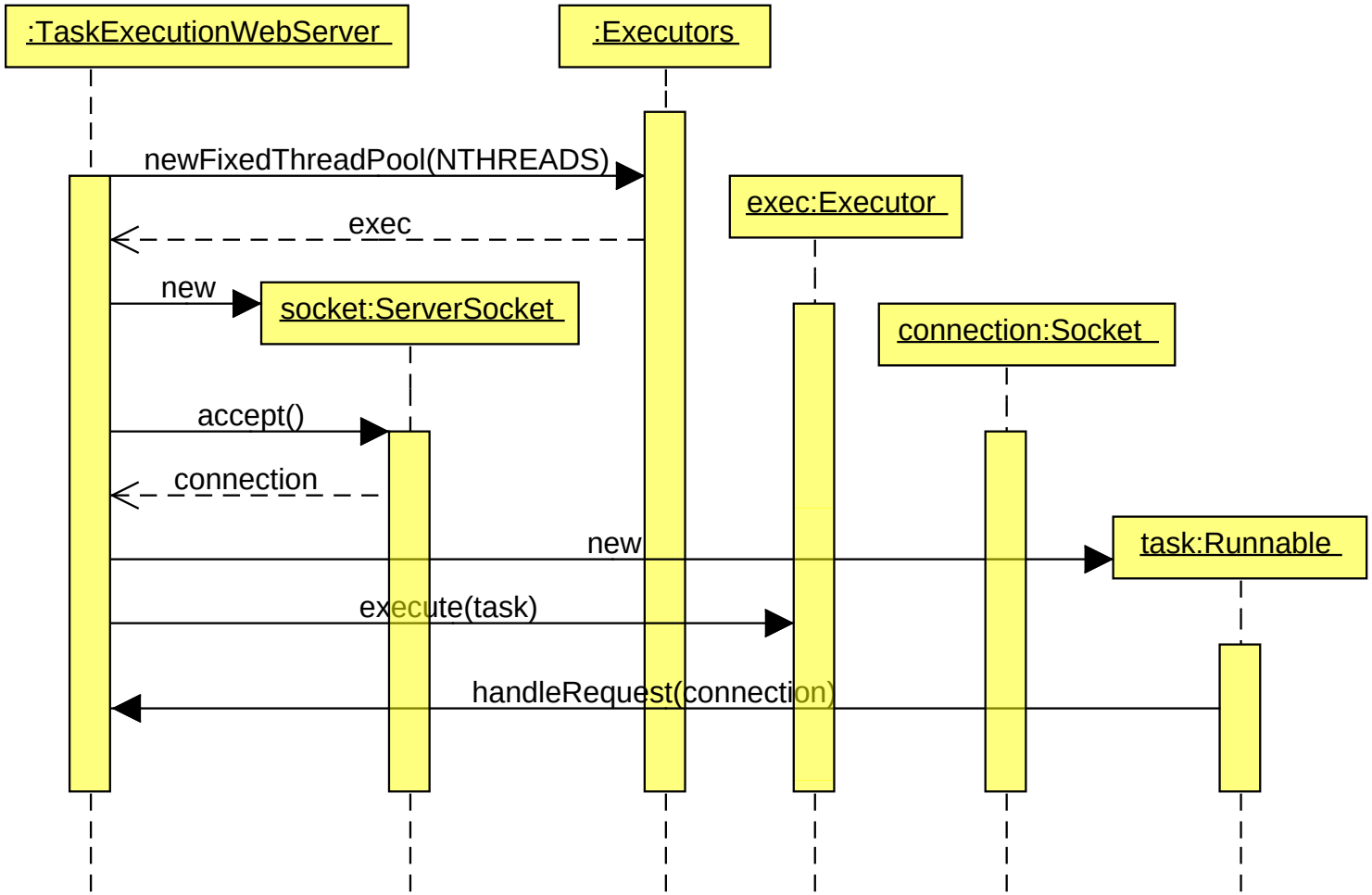
Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

TaskExecutionWebServer sequencediagram



Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Flexibility gained: Execution policies

The flexibility you gain with Executor, is that you can still simulate the previous sub-optimal models, but since you separated the submission from actual execution, various execution policies can be used and tuned.

- In what thread will the task be executed?
- In what order should the tasks be executed (Queue, Stack, priority)?
- What about the amount of parallelism?
- How many task may be queued?
- If there is a *cancellation* victim, how should you select this victim?
- What (if any) actions are there to take before or after task execution?

Whenever you see `new Thread(runnable).start();` you might want to consider replacing this use with an Executor.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

The simple life of a Thread in a pool

Untitled Sheets

HOM

Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation

Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods

Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks

Task Execution Summary
Completion Service Examples

Thread pools are fabricated

The class library provides a number of factory methods for various ThreadPools.

newFixedThreadPool Fixed size pool, that is the maximum number of threads in the pool is fixed.

newCachedThreadPool Boundless, but the pool shrinks and grows when demand dictates so.

newSingleThreadedExecutor Helps to enforce a task order, dependent on the task queue used.

newScheduledThreadPool Supports delayed and periodic execution like a timer but with more flexibility.

These executors and its factory methods in Executors give opportunities for tuning, management, monitoring, logging, error reporting etc, which are much more difficult to achieve without the framework.

Untitled Sheets

HOM

Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation

Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods

Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks

Task Execution Summary
Completion Service Examples

The Executors class

```
public class Executors extends Object
```

It provides Factory and utility methods for:

- `Executor`,
- `ExecutorService`,
- `ScheduledExecutorService`,
- `ThreadFactory`, and
- Callable

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Factory methods in Executors.

Executors supports the following methods: Methods that create and return

- an **ExecutorService** set up with commonly useful configuration settings.
- a **ScheduledExecutorService** set up with commonly useful configuration settings.
- a **"wrapped" ExecutorService**, that disables reconfiguration by making implementation-specific methods inaccessible.
- a **ThreadFactory** that sets newly created threads to a known state.
- a **Callable** out of other closure-like forms, so they can be used in execution methods requiring Callable.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

The factory methods I

```
public static ExecutorService
newFixedThreadPool(int nThreads)
```

- Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue.
- If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.
- Parameters: nThreads - the number of threads in the pool

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

The factory methods II

`public static ExecutorService`

`newCachedThreadPool()`

- Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
- These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks.
- Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads not being used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources.
- Note that pools with similar properties but different details (for example, timeout parameters) may be created using ThreadPoolExecutor constructors.

Untitled Sheets

HOM

Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation

Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods

Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks

Task Execution Summary
Completion Service Examples

The factory methods III

```
public static ScheduledExecutorService
newSingleThreadScheduledExecutor()
```

- Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.
- Note however that if this single thread terminates prior to shutdown, a new one will take its place if needed to execute subsequent tasks.
- Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent newScheduledThreadPool(1) the returned executor is guaranteed not to be reconfigurable to use additional threads.

Untitled Sheets
HOM
Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation
Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods
Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks
Task Execution Summary
Completion Service Examples

The factory methods IV

```
public static ScheduledExecutorService
newScheduledThreadPool(int corePoolSize)
```

- Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
- Parameters: corePoolSize - the number of threads to keep in the pool, even if they are idle.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Exploiting Parallelism

With an Executor you write a Runnable, thereby defining a so called *task boundary*. For the web server example this coincides with the actual task boundary: that's okay. In other applications this might be less obvious.

Example in this section: the page-rendering part of a browser application. An HTML-page is rendered to an image buffer. The page consists of text and images.

First approach: sequentially processing. See the code on the next slide.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Listing 6.10: Rendering page elements sequentially

```
public abstract class SingleThreadRenderer {
    void renderPage(CharSequence source) {
        renderText(source);
        List<ImageData> imageData = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : scanForImageInfo(source))
            imageData.add(imageInfo.downloadImage());
        for (ImageData data : imageData)
            renderImage(data);
    }

    interface ImageData {
    }

    interface ImageInfo {
        ImageData downloadImage();
    }

    abstract void renderText(CharSequence s);
    abstract List<ImageInfo> scanForImageInfo(CharSequence s);
    abstract void renderImage(ImageData i);
}
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Sequential processing

- Simple implementation
- Disadvantage: page is rendered gradually as in *slow*.
- Better but still annoying: first text elements and then the pictures. User can start reading!
- Latter approach is in listing 6.10.

Fetching the images takes time, just like expensive calculations or database retrieval. Instead of a `Runnable` task a better abstraction is the `Callable` interface.

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

Untitled Sheets
HOM
Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation
Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods
Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks
Task Execution Summary
Completion Service Examples

From the java api: Callable

```
public interface Callable<V>
```

A task that returns a result and may throw an exception. Implementers define a single method with no arguments called `call` with return type V.

The `Callable` interface is similar to `Runnable`, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

The `Executors` class contains utility methods to convert from other common forms to Callable classes.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

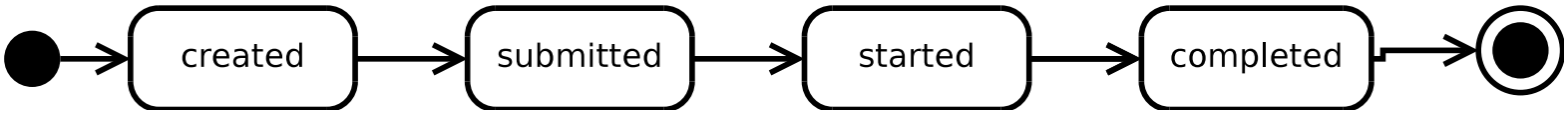
Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

States of a task



A task can be in one of the four states: *created*, *submitted*, *started* and *completed*.

A user might want to cancel a task! This is now possible with the Executor framework. When submitted but not yet started a task can be canceled. The Future interface represents the life cycle of a task.

Untitled Sheets

HOM

Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation

Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods

Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks

Task Execution Summary
Completion Service Examples

From the java api: Future

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


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

From the java api: ExecutorService

```
public interface ExecutorService extends Executor
```

Some methods:

- `<T> Future<T> submit(Callable<T> task)`
- boolean isTerminated()
- void shutdown()

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

An ExecutorService can be shut down, which will cause it to stop accepting new tasks. After being shut down, the executor will eventually terminate, at which point no tasks are actively executing, no tasks are awaiting execution, and no new tasks can be submitted.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

From the java api: ExecutorService (2)

Method submit extends base method Executor.execute(java.lang.Runnable) by creating and returning a Future that can be used to cancel execution and/or wait for completion. Methods `invokeAny()` and `invokeAll()` perform the most commonly useful forms of bulk execution, executing a collection of tasks and then waiting for at least one, or all, to complete. (Class ExecutorCompletionService can be used to write customized variants of these methods.)

The Executors class provides factory methods for the executor services provided in this package.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Divide the work in runnable tasks

- To be able to use executors successfully, your application should have some natural exploitable parallelism.
- Sometimes these tasks come natural, like in a web server request-response cycle.
- Sometimes you need result bearers; that is where Callable and Future come in.
- Callable has a better mechanism to deal with exceptions.
- Runnable and Callable describe abstract computational tasks which are usually finite.
- Cancellation is also provided, avoiding the problems with `Thread.stop()` etc.

Untitled Sheets

HOM

Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation

Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods

Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks

Task Execution Summary
Completion Service Examples

Parallelism payoff

Best performance payoff with homogeneous tasks

The real performance payoff of dividing a program's workload into tasks comes when there are a large number of independent, homogeneous tasks that can be processed concurrently.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Executor meets BlockingQueue

Why poll and wait when someone else can wait for you? CompletionService is the marriage of an Executor with a BlockingQueue:

```
public interface CompletionService<V> {
    Future<V> submit(Callable<V> task);
    Future<V> submit(Runnable task, V result);
    Future<V> take() throws InterruptedException;
    Future<V> poll();
    Future<V> poll(long timeout, TimeUnit unit)
        throws InterruptedException;
}
```

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

ExecutorCompletionService I. FutureTask queues itself.

Extracted form the implementation src.zip, all comments removed

```
package java.util.concurrent;
public class ExecutorCompletionService<V>
    implements CompletionService<V> {
    private final Executor executor;
    private final AbstractExecutorService aes;
    private final BlockingQueue<Future<V>> completionQueue;

    /**
     * FutureTask extension to enqueue upon completion
     */
    private class QueueingFuture extends FutureTask<Void> {
        QueueingFuture(RunnableFuture<V> task) {
            super(task, null);
            this.task = task;
        }
        protected void done() { completionQueue.add(task); }
        private final Future<V> task;
    }
```


Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation
Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods
Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks
Task Execution Summary
Completion Service Examples

ExecutorCompletionService II

```
    private RunnableFuture<V> newTaskFor(Callable<V> task) {
        if (aes == null)
            return new FutureTask<V>(task);
        else
            return aes.newTaskFor(task);
    }

    private RunnableFuture<V> newTaskFor(Runnable task, V result) {
        if (aes == null)
            return new FutureTask<V>(task, result);
        else
            return aes.newTaskFor(task, result);
    }
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

ExecutorCompletionService, constructors

```
public ExecutorCompletionService(Executor executor) {
    if (executor == null)
        throw new NullPointerException();
    this.executor = executor;
    this.aes = (executor instanceof AbstractExecutorService) ?
        (AbstractExecutorService) executor : null;
    this.completionQueue = new LinkedBlockingQueue<Future<V>>();
}

public ExecutorCompletionService(Executor executor,
                                 BlockingQueue<Future<V>> completionQueue) {
    if (executor == null || completionQueue == null)
        throw new NullPointerException();
    this.executor = executor;
    this.aes = (executor instanceof AbstractExecutorService) ?
        (AbstractExecutorService) executor : null;
    this.completionQueue = completionQueue;
}
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

ExecutorCompletionService submitters

```
public Future<V> submit(Callable<V> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task);
    executor.execute(new QueueingFuture(f));
    return f;
}

public Future<V> submit(Runnable task, V result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task, result);
    executor.execute(new QueueingFuture(f));
    return f;
}
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

ExecutorCompletionService result getters

```
    public Future<V> take() throws InterruptedException {
        return completionQueue.take();
    }

    public Future<V> poll() {
        return completionQueue.poll();
    }

    public Future<V> poll(long timeout, TimeUnit unit)
            throws InterruptedException {
        return completionQueue.poll(timeout, unit);
    }

}
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Time limits

Better then speed limits...

Sometimes you cannot afford to wait forever.

- The difficulty is to wait just long enough, no longer.
- A timed version of Future.get provides that. *I will wait to become a millionaire but no longer then 2 months.*
- A timed Future.get will throw an *I'm Bored* exception (actually: TimeoutException).
- By catching this exception you can cancel those tasks, so that they no longer take up resources. (as in: take them of the road.)

Untitled Sheets
HOM
Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation
Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods
Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks
Task Execution Summary
Completion Service Examples

Timer is not that good.

The Timer class has its problems:

- A Timer creates one Thread to execute its tasks.
- Trigger misses during long tasks if period is short compared to task execution time.
- Timer behave poorly on Exceptions being thrown. An uncaught exception kills this Thread. This Thread is not recreated.
- See example on next sheet.

Untitled Sheets

HOM

Executing tasks in Threads
Executing sequentially
Disadvantages of unbound thread creation

Executor framework
Execution policies
Thread pools
The Executors Factory
The factory methods

Exploitable parallelism
The Callable Interface
Completion service
Delay and periodic tasks

Task Execution Summary
Completion Service Examples

Out of Time

```
public class OutOfTime {
    public static void main(String[] args) throws Exception {
        Timer timer = new Timer();
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(1);
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(5);
    }

    static class ThrowTask extends TimerTask {
        public void run() {
            throw new RuntimeException();
        }
    }
}
```


Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Summary

- Structuring applications with tasks can both simplify development and facilitate concurrency.
- The Executor framework permits decoupling of task submission and execution. with a range of execution policies
- Consider and executor if you do `new Thread(runnable).start()` a lot.
- Identify sensible task boundaries. Sometimes they come natural, sometimes you must do some analysis to find more homogeneous tasks to provide optimum parallelism.

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Completion service example 1

Your want ALL results, optimising concurrency

```
void solve(Executor e,
           Collection<Callable<Result>> solvers)
    throws InterruptedException, ExecutionException {

    CompletionService<Result> ecs
     = new ExecutorCompletionService<Result>(e);

    for (Callable<Result> s : solvers) {
        ecs.submit(s);
    }

    int n = solvers.size();

    for (int i = 0; i < n; ++i) {
        Result r = ecs.take().get();
        if (r != null)
            use(r);
    }
}
```

from api javadoc

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples

Completion service example 2

Your want ANY results, quickest wins

```
        try {
            for (Callable<Result> s : solvers)
                futures.add(ecs.submit(s));
            for (int i = 0; i < n; ++i) {
                try {
                    Result r = ecs.take().get();
                    if (r != null) {
                        result = r;
                        break;
                    }
                } catch (ExecutionException ignore) {}
            }
        }
        finally {
            for (Future<Result> f : futures) {
                f.cancel(true);
            }
        }
```

from api javadoc

Untitled Sheets

HOM

Executing tasks in Threads

Executing sequentially

Disadvantages of unbound thread creation

Executor framework

Execution policies

Thread pools

The Executors Factory

The factory methods

Exploitable parallelism

The Callable Interface

Completion service

Delay and periodic tasks

Task Execution Summary

Completion Service Examples