Parallel Programming Practice

Threads and Tasks

Susanne Cech Previtali
Thomas Gross

Last update: 2009-10-29, 09:12

Thread objects

`java.lang.Thread`

- Each thread is associated with an instance of the class `Thread`

Two strategies for using `Thread` objects

- To *directly control* thread creation and management
  - Instantiate `Thread` each time for an asynchronous task
- Abstract *thread management* from the rest of the application
  - Pass the tasks to an `Executor`

Today

Low-level: basic building blocks

- Thread API
- Wait and notify mechanism

High-level: concurrency API

- Executor framework

Thread API

How to create a thread

1. Declare a class that implements the **Runnable** interface

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
    }
}
```

preferable way!

- Separates **Runnable** task from the **Thread** object that executes the task
- Applicable to high-level thread management APIs (**Executor**)

How to create a thread

2. Declare a class to be a subclass of `Thread`

```java
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread t = new HelloThread();
        t.start();
    }
}
```

java.lang.Thread: Properties

Property	Getter	Setter	Description
`long id`	✓		Identifier
`int priority`	✓	✓	Priority
`String name`	✓	✓	Name
`boolean isDaemon`	✓	✓	User or daemon thread

java.lang.Thread: Queries

Instance methods	Description
`boolean isAlive()`	Is the current thread alive?
`boolean isInterrupted()`	Has the current thread been interrupted?

Class methods	Description
`Thread currentThread()`	Reference to the currently executing thread
`boolean interrupted()`	Has the current thread been interrupted?

java.lang.Thread: Commands

Instance methods	Description
`void run()`	Default: returns ⇒ override
`void start()`	Start a Thread instance and execute its `run()` method
`void interrupt()`	Interrupt the current thread
`void join([long])`	Block until the other thread exits [for at most the given milliseconds]

Class methods	Description
`void sleep(long)`	Stop temporarily (for the given milliseconds) the execution of the current thread

JMM: Happens-before rules for threads

Thread start rule

- `T1.start()` *happens-before* every action in `T1`

Thread termination rule

- Any action in `T1` *happens-before* any action in `T2` that detects that `T1` has terminated
- Detection in `T2`: `T1.join()` returns or `T1.isAlive() == false`

Interruption rule

- In `T1`: `T2.interrupt()` *happens-before* interrupt detection (by any thread including `T2`)
- Detection: throw `InterruptException`, invoke `T2.isInterrupted()`, `Thread.interrupted()`

Thread control example: Main

```
public class SimpleThreads {
  public static void main(String args[]) throws InterruptedException {
    long patience = 1000 * 60 * 60; // 1 hour delay
    long startTime = System.currentTimeMillis();

    Thread t = new Thread(new MessageLoop()).start();
    while (t.isAlive()) {
       t.join(1000); // wait for t to finish (max. 1 second)
       if (((System.currentTimeMillis() - startTime) > patience) &&
              t.isAlive()) {
           t.interrupt(); // tired of waiting -> interrupt t
           t.join();  // wait indefinitely for t to finish
       }
    }
  }
}
```

See example at http://java.sun.com/docs/books/tutorial/essential/concurrency/simple.html

Thread control example: MessageLoop

```
public class MessageLoop implements Runnable {
    public void run() {
        String importantInfo[] = { "A", "B", "C", "D" };
        try {
            for (int i = 0; i < importantInfo.length; i++) {
                Thread.sleep(4000); // pause for 4 seconds
                printMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            printMessage("I wasn't done!");
        }
    }
}
```

Wait and notify

Wait sets and notification

Each `Object` has an associated *lock* and *wait set*

Wait set

- Set of threads
- Holds threads blocked by `Object.wait()` until notifications/wait done
- Used by `wait()`, `notify()`, `notifyAll()` and thread scheduling

Wait sets interact with locks

- `t.wait()`, `t.notify()`, `t.notifyAll()` must be called only when synchronization lock is hold on `t`
    - Otherwise `IllegalMonitorStateException` is thrown

`Object.wait()` and `Object.wait(long)`

If current thread T has been interrupted by another thread

- return

else T is blocked

- T is placed in wait set of `obj`
- T releases any locks for `obj` (keeps other locks)
    - Lock status is restored upon later resumption

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    // Perform action appropriate to condition
}
```

Object.wait(long) waits for a maximum time given

`Object.notify()` and `Object.notifyAll()`

A thread T is arbitrarily chosen from wait set of `obj`

- No guarantees which thread

T re-obtains lock on `obj`

- T blocks until `notify()` releases the lock
- T may block if some other thread obtains lock first

T resumes after `wait()`

- `wait()` returns

`notifyAll()`

- Similar as `notify()` but for all threads in wait set of `obj`

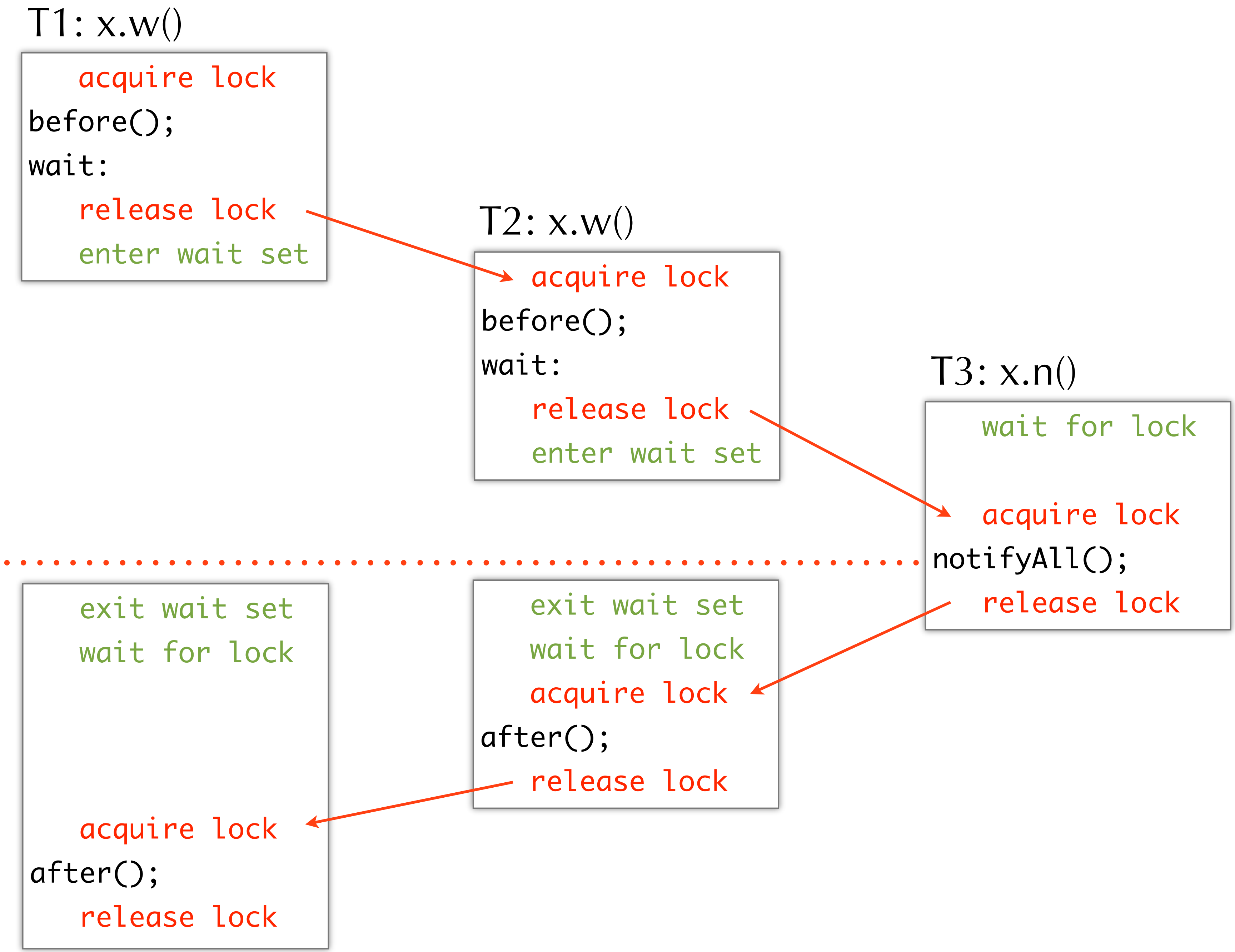
Example with useless class

To illustrate the underlying mechanisms

✗

```
class X {
    synchronized void w() throws InterruptedException {
        before(); wait(); after();
    }
    synchronized void n() {
        notifyAll();
    }
    void before() {}
    void after() {}
}
```

Attention! Broken program: liveness failure ⇒ *missed signal*



T1: x.w()

acquire lock
before();
wait:
release lock
enter wait set

T2: x.w()

acquire lock
before();
wait:
release lock
enter wait set

T3: x.n()

wait for lock

acquire lock
notifyAll();
release lock

exit wait set
wait for lock

acquire lock
after();
release lock

exit wait set
wait for lock
acquire lock
after();
release lock

Remarks

Place checks for *condition variables* in while loops

- Thread only knows that is has been waken up, must re-check

Methods with *guarded waits* are not completely atomic

- On `wait()` lock is released ⇒ other thread can be scheduled
- Objects must be in consistent state before calling `wait()`

Typical usage

```
public class PatientWaiter {
    @GuardedBy(“this”) private volatile boolean flag = false;
    public synchronized void waitTillChange() {
        while (!flag) {
            try {
                this.wait();
            } catch (InterruptedException e) {}
        }
        // whatever needs to be done after condition is true
    }
    public synchronized void change() {
        flag = true;
        this.notifyAll();
    }
```

slipped condition

if two synchronized blocks

Executor framework

Threaded web server

```
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) { ... }
}
```

See example at http://www.javaconcurrencyinpractice.com/listings/ThreadPerTaskWebServer.java

Problems of the threaded solution

Discussion

- Up to a certain point: more threads improve throughput
- Beyond that: slow down, crash

Poor resource management

- Thread lifecycle overhead
  - Thread creation and teardown
- Resource consumption
  - More runnable threads than processors ⇒ may hurt performance
  - Memory, garbage collection
- Stability
  - Number of threads limited ⇒ OutOfMemoryError

Tasks versus threads

Task

- Logical unit of work

Thread

- Mechanism by which tasks can run asynchronously

Web server example

- Each task is executed in its thread
- Poor resource management

Need: High-level abstraction for task execution

Low-level constructs

- wait()/notify()

High-level concurrency API

- Prefer executors and tasks to threads
- Prefer concurrency utilities to wait()/notify()

Producer-consumer design pattern

Producer

- Places work items on a “to do” list

Consumer

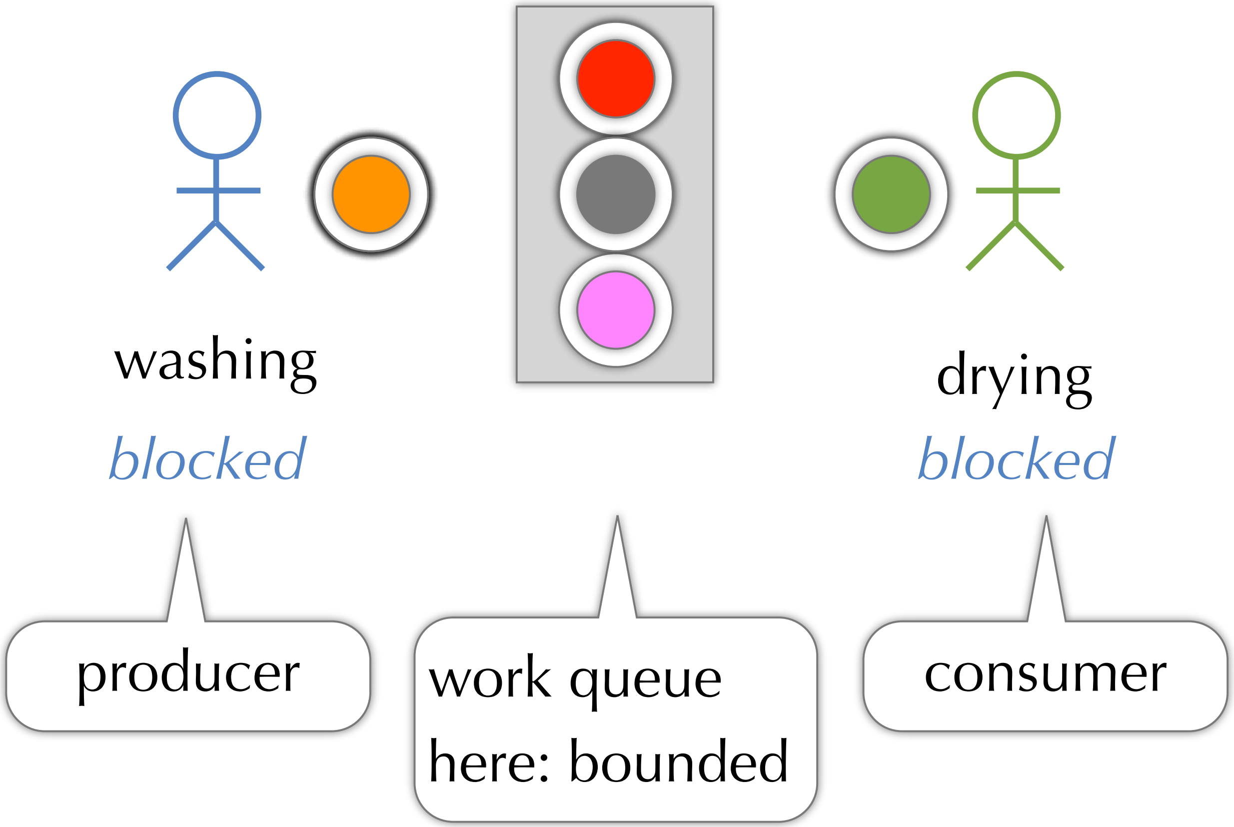
- Takes work items from the “to do” list for processing

Discussion

- Separates identification of work to be done from execution of that work
- Removes code dependencies between producer and consumer classes
- Simplifies workload management

Producer-consumer example

Dish washing and drying



washing

blocked

drying

blocked

producer

work queue
here: bounded

consumer

Executor framework

Based on producer-consumer pattern

Producers

- Submit tasks

Consumers

- Threads that execute tasks

Web server using Executor

```
public class TaskExecutionWebServer {
    private static final Executor exec = ...;  // see later

    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);
        }
    }
}
```

Different Executor implementations

Behavior like `ThreadPerTaskWebServer`

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

Behavior like a single threaded web server

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

java.util.concurrent.Executor

```
public interface Executor {
    // Execute the given command at some time in the future
    void execute(Runnable command);
}
```

Executor implementations

Tasks may execute in

- a newly created thread
- an existing task-execution thread
- or the thread calling `execute()`

Tasks may execute sequentially or concurrently

Execution policies

Executor decouples submission from execution

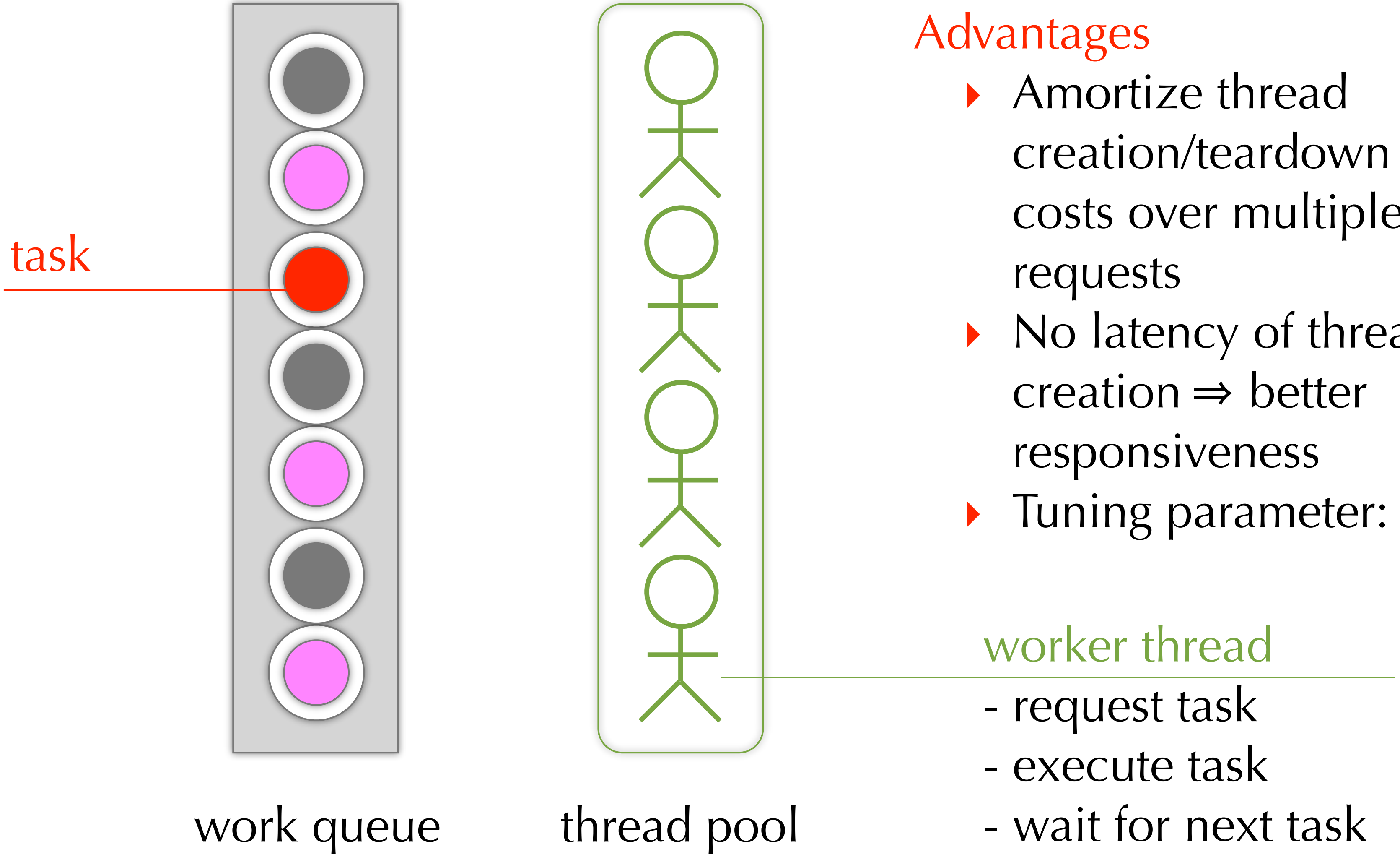
Resource management tool

- What resources are available?
- Which QOS requirements?

Policies decide

- In what threads will tasks execute
- In what order? -- FIFO, LIFO, priority queue?
- How many concurrent tasks?
- How many tasks may be queued pending execution?
- If system overloaded: choose victim task? notify application?
- Actions before/after executing a task?

Thread pool



- request task
- execute task
- wait for next task

Advantages

- Amortize thread creation/teardown costs over multiple requests
- No latency of thread creation ⇒ better responsiveness
- Tuning parameter: size

Factory methods to create thread pools

```
public class Executors {
    // maintain n threads, unbounded queue
    public static ExecutorService newFixedThreadPool(int n)
    // create threads as needed (reused), unbounded queue
    public static ExecutorService newCachedThreadPool()
    // create one thread, unbounded queue
    public static ExecutorService newSingleThreadExecutor()
    // delayed and periodic task execution
    public static ExecutorService newScheduledThreadPool(int size)

    // ... more methods... consider also overloaded variants
}
```

Web server using thread pool

factory method for creating a thread pool

```
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);
        }
    }
}
```

Executor lifecycle

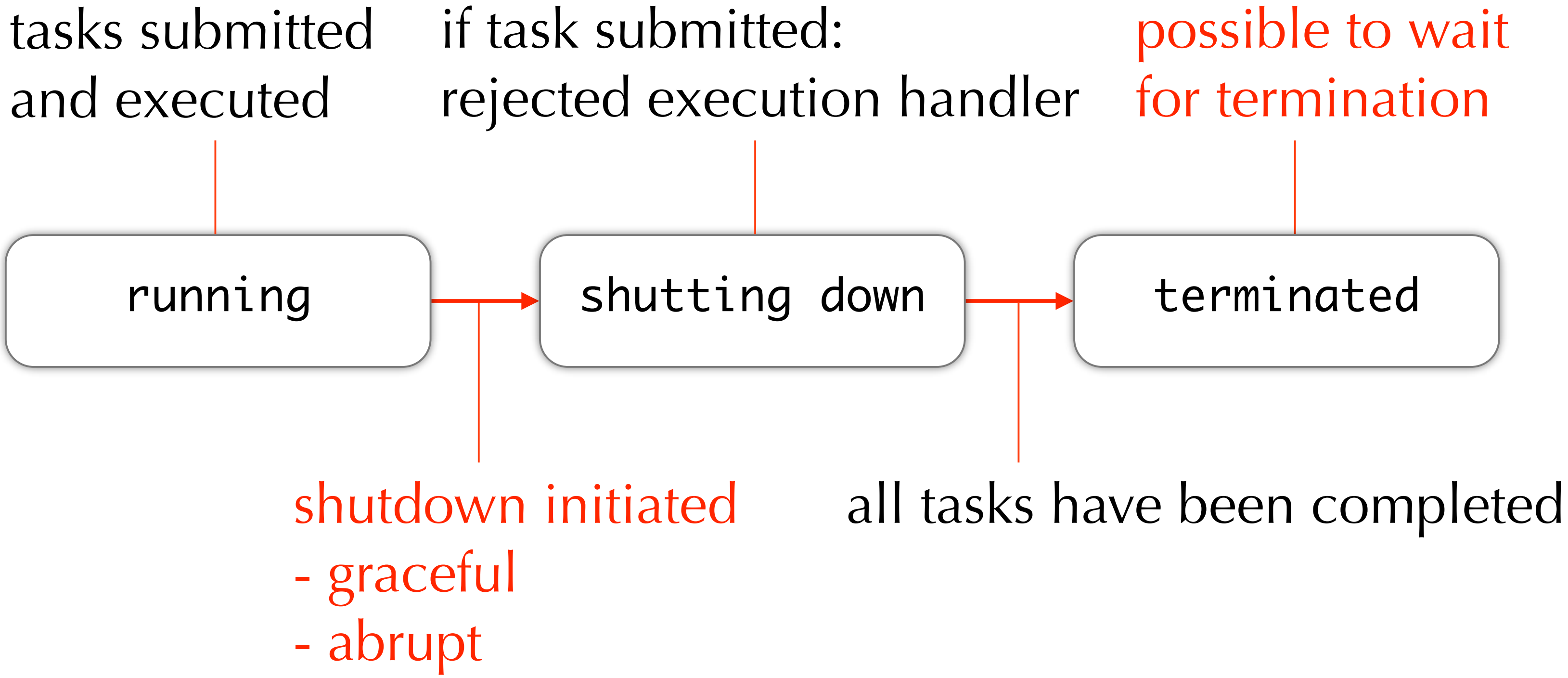
Executor processes task asynchronously

- State of tasks may not be obvious

Executor provides service to applications: must be able to

- Shutdown
- Report status of tasks
- Also: executor implementation must shut down
  - JVM can exit only after all threads have terminated

States of the `ExecutorService`



tasks submitted
and executed

if task submitted:
rejected execution handler

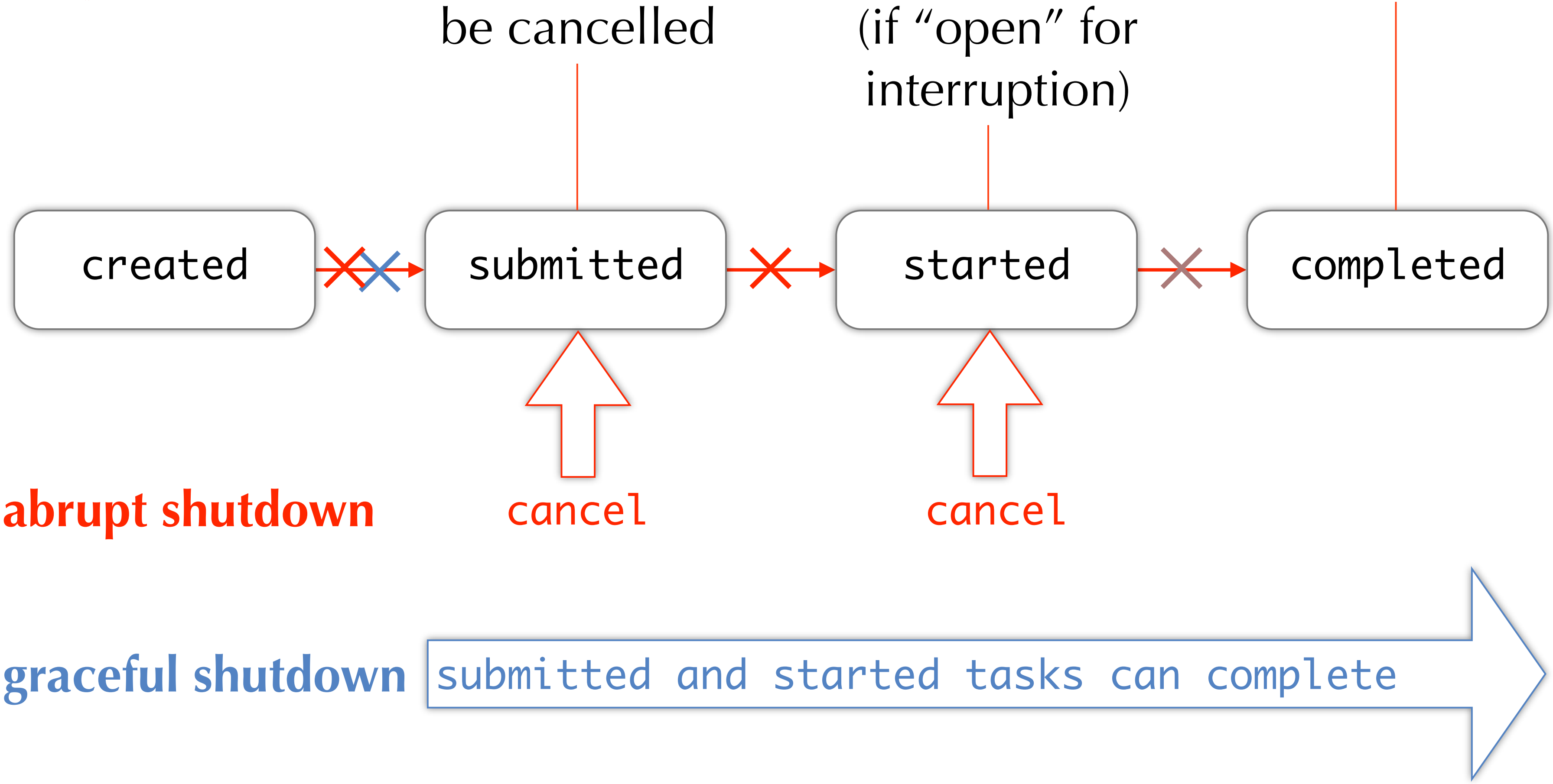
possible to wait
for termination

`running` → `shutting down` → `terminated`

shutdown initiated
- graceful
- abrupt

all tasks have been completed

Shutdown



Lifecycle of a task

can always be cancelled

may be cancelled (if “open” for interruption)

cancel: no effect

`created` → `submitted` → `started` → `completed`

abrupt shutdown

`cancel` `cancel`

graceful shutdown

`submitted and started tasks can complete`

java.util.concurrent.ExecutorService

```
public interface ExecutorService extends Executor {
    // graceful shutdown
    void shutdown();
    // abrupt shutdown
    // -> return list of tasks awaiting execution
    List<Runnable> shutdownNow();

    // query about state change
    boolean isShutdown();
    // ... more methods... discussed later
}
```

java.util.concurrent.ExecutorService

```
public interface ExecutorService extends Executor {
    // block until one event happens
    // (1) all tasks have completed
    // (2) the timeout occurs
    // (3) the current thread is interrupted
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedExecution;

    // Have all tasks been completed? following shut-down
    boolean isTerminated();

    // ... more methods... discussed later
}
```

```java
public class LifecycleWebServer {
    private final ExecutorService exec = Executors.newCachedThreadPool();
    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() {
                        handleRequest(conn);
                    }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }
    public void stop() { exec.shutdown(); }
}
```

See complete code at http://www.javaconcurrencyinpractice.com/listings/LifecycleWebServer.java

Executor revisited

```
public interface Executor {
    // Execute the given command at some time in the future
    void execute(Runnable command);
}
```

`Runnable` as basic task representation

- Cannot return a value
- Cannot throw checked exceptions

Other task abstractions necessary

- `Callable`: task
- `Future`: result

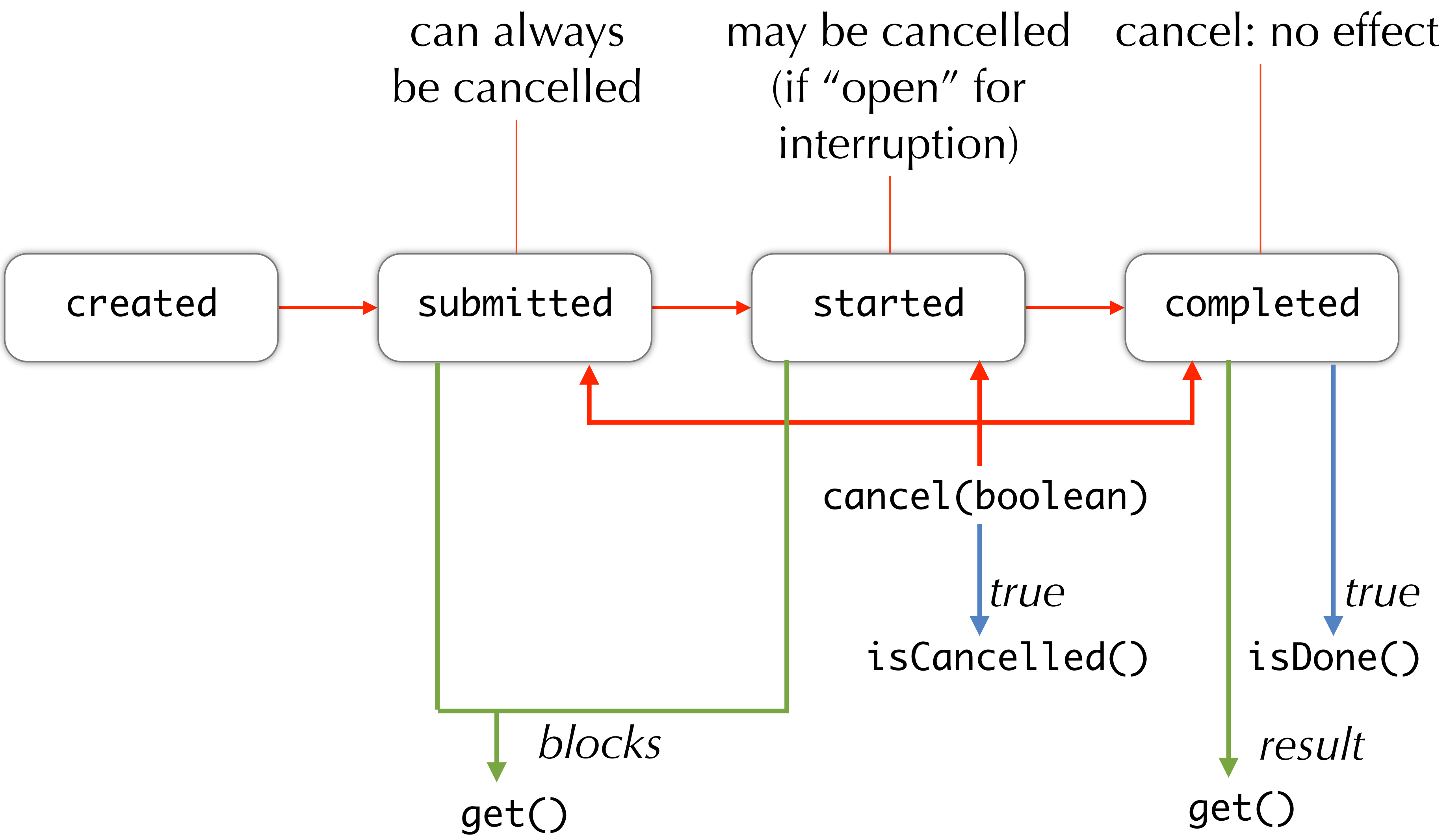
Task abstraction `Callable`

```
public interface Callable<V> {
    // Task that returns a result and may throw an exception
    V call() throws Exception;
}
```

See `Executors` for utility factory methods

- Example: wrap a `Runnable` in a `Callable`

Lifecycle abstraction with Future



can always be cancelled

may be cancelled (if “open” for interruption)

cancel: no effect

`created` → `submitted` → `started` → `completed`

`cancel(boolean)`

true `isCancelled()`

true `isDone()`

blocks `get()`

result `get()`

Future

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

Create a future

- Interface `ExecutorService: Future<V> submit([Callable|Runnable])`
- Class `FutureTask<V>: base implementation of Future<V>`

Study goals