# ThreadPool

## Executor

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

## ExecutorService

```
public interface ExecutorService extends Executor {
	public void shutdown();
	// gestione con future
	public Future<?> submit(Runnable task); // solo eccezioni
	public <T> Future<T> submit(Callable<T> task); // eccezioni e valori
	// gestione con callback
	public void submit(Runnable task, Callback<?> callback); // solo eccezioni
	public <T> void submit(Callable<T> task, Callback<T> callback); // eccezioni e valori
}
```

## Executors

```
public class Executors {
	public static ExecutorService newThreadPool() {
		return new SimpleThreadPool();
	}
	public static ExecutorService newThreadPool(int numOfThreads) {
		return new SimpleThreadPool(numOfThreads);
	}
}
```

## Callable

```
public interface Callable<T> {
	public T call();
}
```

## Callback

```
public interface Callback<T> {
	public void onSuccess(T arg);
	public default void onFailure(Throwable throwable) {
		throwable.printStackTrace();
	}
}
```

## Future

```
public interface Future<T> {
	public Object get() throws InterruptedException, Throwable;
	public boolean isDone();
}
```

## SimpleFuture

```
public class SimpleFuture<T> implements Future<T> {
	Lock lock;
	Condition condition;
	T value;
	boolean done;
	Throwable throwable;
	public SimpleFuture() {
		this.lock = new ReentrantLock();
		this.condition = lock.newCondition();
		this.value = null;
		this.done = false;
		this.throwable = null;
	}


	public void setValue(T value) {
		lock.lock();
```

```
		try {
			if(done)
				throw new IllegalMonitorStateException("done == true");
			this.value = value;
			done = true;
			condition.signalAll();
		} finally {
			lock.unlock();
		}
	}
	public void setThrowable(Throwable throwable) {
		if(throwable == null)
			throw new IllegalMonitorStateException("throwable == null");
		lock.lock();
		try {
			if(done)
				throw new IllegalMonitorStateException("done == true");
			this.throwable = throwable;
			done = true;
			condition.signalAll();
		} finally {
			lock.unlock();
		}

	}
	@Override
	public Object get() throws InterruptedException, Throwable {
		lock.lock();
		try {
			while(!done)
				condition.await();
			if(throwable != null)
				throw throwable;
			return value;
		} finally {
			lock.unlock();
		}
	}
	@Override
	public boolean isDone() {
		lock.lock();
		try {
			return done;
		} finally {
			lock.unlock();
		}
	}
}
```

**SimpleThreadPool**

```
public class SimpleThreadPool implements ExecutorService {
	boolean shutdown;
	BlockingQueue<Runnable> tasks;
	Thread[] workers;
	final static int maxTasks = 20;
	final static int maxWorkers = 20;

	public SimpleThreadPool() {
		this(maxWorkers);
	}
	public SimpleThreadPool(int workersNum) {
		if (workersNum <= 0 || workersNum > maxWorkers) {
```

```
                        throw new IllegalArgumentException("\"workerNum <= 0 || workerNum >
maxWorkersNum");
                }
                this.shutdown = false;
                this.tasks = new ArrayBlockingQueue<Runnable>(maxTasks);
                this.workers = new Worker[workersNum];
                for(int i = 0; i < workersNum; i++) {
                        Worker worker = new Worker();
                        workers[i] = worker;
                        worker.start();
                }
        }
        @Override
        public void execute(Runnable command) {
                if(command == null)
                        throw new IllegalArgumentException("command == null");
                if(shutdown)
                        throw new IllegalMonitorStateException("shutdown == true");
                try {
                        tasks.put(command);
                } catch(InterruptedException e) {
                        System.out.println(e.getClass());
                }
        }
        @Override
        public void shutdown() {
                synchronized (tasks) {
                        for(int i = 0; i < workers.length; i++) {
                                workers[i].interrupt();
                        }
                }
        }
        @Override
        public Future<Void> submit(Runnable task) {
                if(task == null)
                        throw new IllegalArgumentException("task == null");
                SimpleFuture<Void> future = new SimpleFuture<Void>();
                Runnable runnable = ()-> {
                        try {
                                task.run();
                                future.setValue(null); // runnable riuscita
                        } catch(Throwable e) {
                                future.setThrowable(e); // runnable lancia eccezione
                        }
                };
                this.execute(runnable);
                return future;
        }
        @Override
        public <T> Future<T> submit(Callable<T> task) {
                if(task == null)
                        throw new IllegalArgumentException("task == null");
                SimpleFuture<T> future = new SimpleFuture<T>();
                Runnable runnable = ()-> {
                        try {
                                future.setValue(task.call());
                        } catch(Throwable e) {
                                future.setThrowable(e);
                        }
                };
                this.execute(runnable);
```

```
		return future;
	}
	@Override
	public void submit(Runnable task, Callback<?> callback) {
		if(task == null)
			throw new IllegalArgumentException("task == null");
		if(callback == null)
			throw new IllegalArgumentException("callback == null");
		Runnable runnable = ()->{
			try {
				task.run();
				callback.onSuccess(null); // runnable riuscita
			} catch (Throwable e) {
				callback.onFailure(e); // runnable lancia eccezione
			}
		};
		this.execute(runnable);
	}
	@Override
	public <T> void submit(Callable<T> task, Callback<T> callback) {
		if(task == null)
			throw new IllegalArgumentException("task == null");
		if(callback == null)
			throw new IllegalArgumentException("callback == null");

		Runnable runnable = ()-> {
			try {
				callback.onSuccess(task.call()); // runnable riuscita
			} catch (Throwable e) {
				callback.onFailure(e); // runnable lancia eccezione
			}
		};
		this.execute(runnable);
	}
	class Worker extends Thread {
		@Override
		public void run() {
			Runnable runnable;
			while(true) {
				synchronized (tasks) {
					if(shutdown && tasks.isEmpty())
						return;
				}
				try {
					runnable = tasks.take();
					runnable.run();
				} catch(InterruptedException e) {
					System.out.println("Worker interrupted");
				} catch(Throwable e) {
					e.printStackTrace();
					System.out.println(e.getClass());
				}
			}
		}
	}
}
```