Concurrency: Futures

Tutorial 15 (2 June 2021)

Daniël Kuijsters

WE START AT 8:30

Generics

```
import java.lang.reflect.Array;

public class Box<E> {
    private final E[] elements;

    @SuppressWarnings("unchecked")
    public Box (Class<E> componentType, int size) {
        elements = (E[]) Array.newInstance(componentType, size);
    }
}

…

@SuppressWarnings("unchecked")
Box<Integer> box = new Box(Integer.class, 10);
```

Generics (2)

```
import java.util.ArrayList;
import java.util.List;

public class Box<E> {
    private final List<E> elements;

    public Box (int size) {
        elements = new ArrayList<>(size);
    }
}
```

Lock Objects

```
private final ReentrantLock lock = new ReentrantLock();
...

lock.lock()
try {
        // Critical section
} finally {
        lock.unlock();
}
```

Lock Objects (2)

```
private final ReentrantLock lock = new ReentrantLock();

public void claim() {
        lock.lock();
}

public void free() {
        lock.unlock();
}
```

Possibly in a different class:

```
claim();
try {
        // Critical section
} finally {
        free();
}
```

Condition Objects

```
private final ReentrantLock lock = new ReentrantLock();
private final Condition condition = lock.newCondition();
...

lock.lock()
try {
        while (// Some predicate) {
                condition.await();
        }
        ...
} finally {
        lock.unlock();
}
```

Condition Objects (2)

```
private final ReentrantLock lock = new ReentrantLock();
private final Condition condition = lock.newCondition();
...

lock.lock()
try {
        // Statements that might lead to some predicate being false
        condition.signalAll();
} finally {
        lock.unlock();
}
```

Result-bearing Tasks

- We created Tasks by making classes **active**, i.e., by making them implement the **Runnable** interface.
- In other words, these classes implemented a **run** method.
- However, run takes no argument and it returns nothing, i.e., its return type is **void**.
- In order to obtain a result, we had to store it into an instance variable of the Runnable.
- There exists a result-bearing sibling: **Callable<T>**.

Callable<T>

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

Futures

```
void                        execute(Runnable task);
Future<?>                   submit(Runnable task);
<T> Future<T>               submit(Runnable task, T result);
<T> Future<T>               submit(Callable<T> task);
<T> List<Future<T>>         invokeAll(Collection<? extends Callable<T>> tasks)
```

Futures (2)

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

Demo: Max Finder

Interruption

- In order to cancel a single task, we call **cancel(true)** on the corresponding **Future** object.
- Alternatively, if we want to cancel all tasks and close the thread pool, we call **shutdownNow()** on the **ExecutorService** object.
- Both methods may (and most likely will) interrupt the threads that are executing the tasks, i.e., call their **interrupt()** methods.

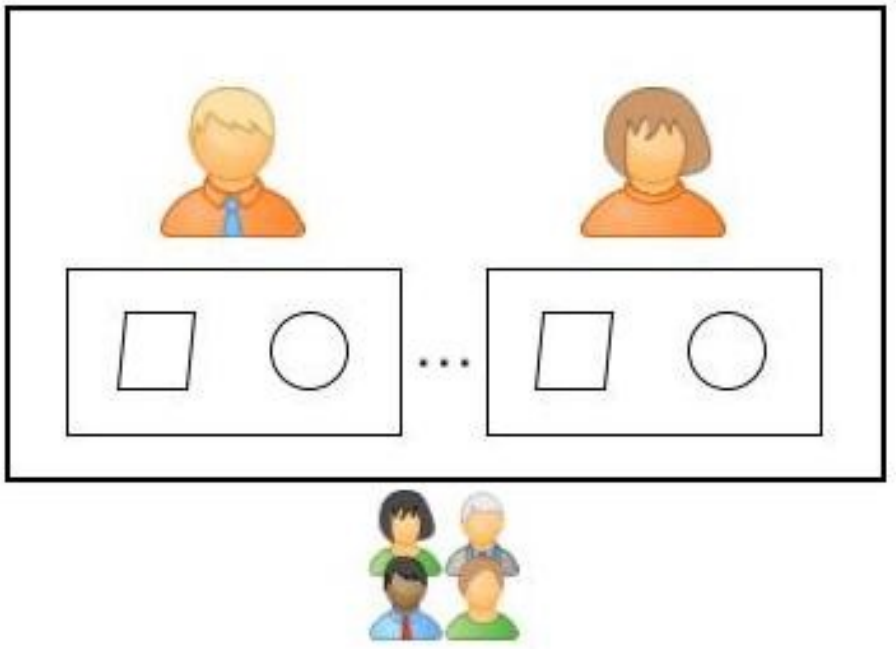
Interruption (2)

```
public void run() {
        try {
                while (!Thread.currentThread().isInterrupted()) {
                        // Do work (might contain blocking calls,
                        // e.g., sleep(), await(), or not)
                }
                // Interrupt detected through status flag
                // In case of no blocking calls
        } catch (InterruptedException e) {
                // Interrupt detected through exception
                // Thrown by blocking call
                // Allow thread to exit
        }
}
```

Interruption (3)

```
try {
        // Work that involves a blocking call
        // Examples include sleep() and await()
} catch (InterruptedException ie) {
        // Catching the exception clears interrupt flag
        // Either throw the exception up (and include in method signature):
        //     throw ie;
        // Or set the flag again:
        //     Thread.currentThread().interrupt();
}
```

Demo: Interrupts



Demo: Coffee Shop