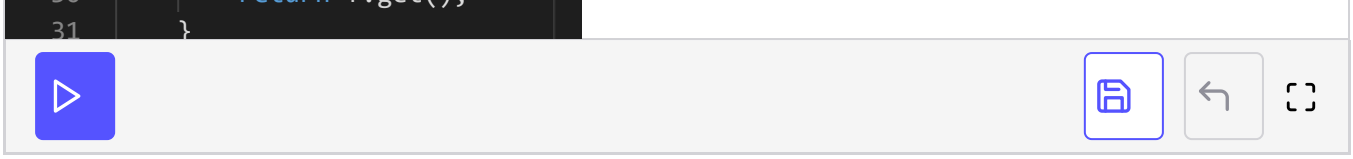
Future Interface

This lesson discusses the Future interface.

Future Interface

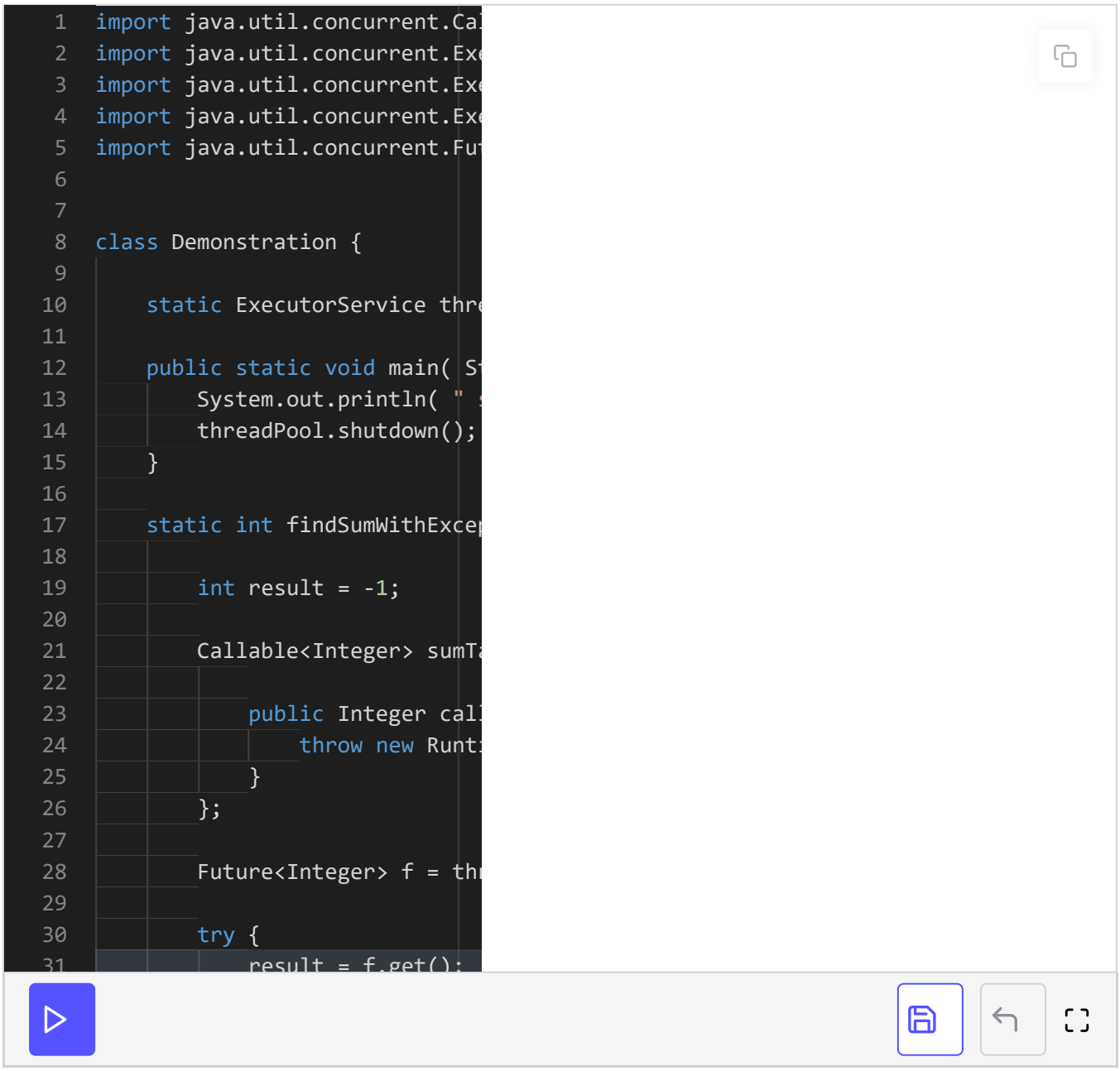
The `Future` interface is used to represent the result of an asynchronous computation. The interface also provides methods to check the status of a submitted task and also allows the task to be cancelled if possible. Without further ado, let's dive into an example and see how callable and future objects work in tandem. We'll continue with our sumTask example from the previous lesson.

```
1  import java.util.concurrent.Ca
2  import java.util.concurrent.Ex
3  import java.util.concurrent.Ex
4  import java.util.concurrent.Ex
5  import java.util.concurrent.Fu
6
7  class Demonstration {
8
9      // Create and initialize a
10     static ExecutorService thr
11
12     public static void main( S
13         System.out.println( "s
14         threadPool.shutdown();
15     }
16
17     static int findSum(final i
18
19         Callable<Integer> sumT
20
21             public Integer cal
22                 int sum = 0;
23                 for (int i = 1
24                     sum += i;
25                 return sum;
26             }
27         };
28
29         Future<Integer> f = th
30         return f.get();
```



Using Future and Callable Together

Thread pools implementing the `ExecutorService` return a future for their task submission methods. In the above code on **line 29** we get back a future when submitting our task. We retrieve the result of the task by invoking the `get` method on the future. The get method will return the result or throw an instance of `ExecutionException`. Let's see an example of that now.



Callable throwing Exception

On **line 31** of the above code, we make a `get` method call. The method throws an execution exception, which we catch. The reason for the exception can be determined by using the `getCause` method of the execution exception. If you run the above snippet, you'll see it prints the runtime exception that we throw on **line 24**.

The `get` method is a blocking call. It'll block till the task completes. We can also write a polling version, where we poll periodically to check if the task is complete or not. Future also allows us to cancel tasks. If a task has been submitted but not yet executed, then it'll be cancelled. However, if a task is currently running, then it may or may not be cancellable. We'll discuss cancelling tasks in detail in future lessons.

Below is an example where we create two tasks. We poll to check if the task has completed. Also, we cancel the second submitted task.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    static ExecutorService threadPool = Executors.newSingleThreadExecutor();

    public static void main( String args[] ) throws Exception {
        System.out.println(pollingStatusAndCancelTask(10));
        threadPool.shutdown();
    }

    static int pollingStatusAndCancelTask(final int n) throws Exception {

        int result = -1;

        Callable<Integer> sumTask1 = new Callable<Integer>() {

            public Integer call() throws Exception {

                // wait for 10 milliseconds
                Thread.sleep(10);

                int sum = 0;
                for (int i = 1; i <= n; i++)
                    sum += i;
```

```
            return sum;
        }
    };

    Callable<Void> randomTask = new Callable<Void>() {

        public Void call() throws Exception {

            // go to sleep for an hours
            Thread.sleep(3600 * 1000);
            return null;
        }
    };

    Future<Integer> f1 = threadPool.submit(sumTask1);
    Future<Void> f2 = threadPool.submit(randomTask);

    // Poll for completion of first task
    try {

        // Before we poll for completion of second task,
        // cancel the second one
        f2.cancel(true);

        // Polling the future to check the status of the
        // first submitted task
        while (!f1.isDone()) {
            System.out.println("Waiting for first task to complete.");
        }
        result = f1.get();
    } catch (ExecutionException ee) {
        System.out.println("Something went wrong.");
    }

    System.out.println("\nIs second task cancelled : " + f2.isCancelled());

    return result;
  }
}
```

Note the following about the above code

- On **lines 44 and 45** we submit two tasks for execution.
- **line 45** the second task submitted doesn't return any value so the future is parametrized with `Void`.
- On **line 52**, we cancel the second task. Since our thread pool consists of a single thread and the first task sleeps for a bit before it starts executing, we can assume that the second task will not have started

executing, we can assume that the second task will not have started executing and can be cancelled. This is verified by checking for and printing the value of the `isCancelled` method later in the program.

- On **lines 56 - 58**, we repeatedly poll for the status of the first task.

The final output of the program shows messages from polling and the status of the second task cancellation request.

FutureTask

Java also provides an implementation of the future interface called the `FutureTask`. It can wrap a callable or runnable object and in turn be submitted to an executor. Though, the class may not be very useful if you don't intend to create customized tasks but we mention it for the sake of completeness.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;

class Demonstration {

    @SuppressWarnings("unchecked")
    public static void main( String args[] ) throws Exception{

        FutureTask<Integer> futureTask = new FutureTask(new Callable() {

            public Object call() throws Exception {
                try{
                    Thread.sleep(1);
                }
                catch(InterruptedException ie){
                    // swallow exception
                }
                return 5;
            }
        });

        ExecutorService threadPool = Executors.newSingleThreadExecutor();
        Future duplicateFuture = threadPool.submit(futureTask);
```

```
        // Awful idea to busy wait
        while (!futureTask.isDone()) {
            System.out.println("Waiting");
        }

        if(duplicateFuture.isDone() != futureTask.isDone()){
            System.out.println("This should never happen.");
        }

        System.out.println((int)futureTask.get());

        threadPool.shutdown();
    }
}
```