Concurrent programming

Advanced thread synchronization and coordination

In previous section of this unit we have learnt some basic techniques of coordinating and synchronizing threads, such as joining threads, or synchronizing methods or blocks of code. In this document we are going to see some advanced strategies that were added to Java API from version 5 and later.

1. Using thread executors

When we are dealing with multiple threads in our application, we may face two problems:

- The performance of our application is not as good as it is expected to be, since there are too many threads running at the same time.
- Our code gets a little bit confusing, because we have to create and start every thread that we need.

These problems can be partially avoided by using **thread executors**. These structures allow us to create a pool of threads, and let a special object handle the threads in our place. For instance, if we define a thread like this:

```java
public class MyThread implements Runnable
{
    ...

    @Override
    public void run()
    {
        ...
    }
}
```

Then we can create a thread executor that handles objects of type `MyThread` (and any other `Runnable` object), this way:

```
ThreadPoolExecutor executor =
    (ThreadPoolExecutor)Executors.newCachedThreadPool();

MyThread t1 = new MyThread();
MyThread t2 = new MyThread();
executor.execute(t1);
executor.execute(t2);

executor.shutdown();
```

We can typically define a loop to create threads using a lambda expression, and add them to the executor, this way:

```
ThreadPoolExecutor executor = ...

for (int i = 0; i < N; i++)
{
    executor.execute(() -> {
        // Thread code
    });
}

executor.shutdown();
```

In both cases, we use the `ThreadPoolExecutor` class (from `java.util.concurrent` package) to handle the pool of threads. There are many ways of getting an object of this type, but the one used in the code above is quite simple. Then, we can create as many threads as we need, and call the `execute` method from the thread executor. From then on, the thread executor is in charge of calling the `start` method of each thread. When all the threads have been added to the pool, we must call the `shutdown` method of the executor to let the program finish when all the threads finish their task.

1.1. Advantages of using executors

What advantages do executors offer if we compare them with traditional thread management?

- The thread executor can reuse a thread (or a position in the pool) to put a new thread if any previous thread has finished its task, so we can save some memory space.
- We can also tell the executor how many threads we want to have running at the same time, by using this instruction instead of the first one used before:

```
ThreadPoolExecutor executor =
    (ThreadPoolExecutor)Executors.newFixedThreadPool(10);
```

If we limit the size of the pool, every thread that exceeds this size will be waiting for a free slot before starting its task. This can be particularly useful if we adjust the total size to the total number of cores of our processor, this way:

```
ThreadPoolExecutor executor =(ThreadPoolExecutor)
    Executors.newFixedThreadPool(
        Runtime.getRuntime().availableProcessors()
    );
```

We can also use this method from *Executors* class to fit the pool size to the number of available processors.

```
ThreadPoolExecutor executor =
    (ThreadPoolExecutor) Executors.newWorkStealingPool();
```

Besides, we have some useful methods in *ThreadPoolExecutor* class, such as `getPoolSize` (it returns how many threads are currently added to the pool), `getActiveCount` (it returns how many threads in the pool are still alive) or `shutdownNow` (it forces all the threads in the pool to finish immediately).

2. Using *Callables* and *CompletableFutures*

In this section we are going to see some alternatives to just launch *Runnable* objects. Depending on whether we want to get a result back after each thread execution or not, we may need some other ways of launching threads. This is when callables and completable futures come into scene.

2.1. Using Callable

In addition to `Runnable`, executors support another kind of task named `Callable`. Callables are **functional interfaces** just like runnables but instead of being void they **return** a value. The `Callable` interface defines the type of data returned using generics.

In this example, we are going to create a `Callable` thread using an `ExecutorService` (`ThreadPoolExecutor` implements `ExecutorService`, so it’s the same). When submitting a *Callable* to an Executor, it will return a `Future` object. It's just an interface that has the necessary methods to get the result returned by the *Callable*.

However, whenever we call `get()` on the Future object, the current thread is blocked until the *Callable* thread returns something. So it is a good idea to call `isDone()` method before, just to check if the *Callable* has finished its task. Besides, we do not use `execute()` method to add Futures to the executor: we use `submit` instead, in order to get the result later.

```
public static void main(String[] args)
{
    Callable<Integer> callInt = () -> {
        try
        {
            TimeUnit.SECONDS.sleep(3);
            return 20;
        } catch (InterruptedException e) {
            throw new IllegalStateException("task interrupted", e);
        }
    };

    ExecutorService executor = Executors.newFixedThreadPool(1);

    // Calling submit executes the thread and returns a Future
    Future<Integer> future = executor.submit(callInt);

    executor.shutdown();

    System.out.println("future done? " + future.isDone());
    Integer result;
    try
    {
        result = future.get(); // It BLOCKS main thread until it returns!
        System.out.println("future done? " + future.isDone());
        System.out.println("Result: " + result); // Prints 20
    } catch (InterruptedException ex) {
    } catch (ExecutionException ex) { }
}
```

Passing a timeout

When calling `get()` on the Future object to retrieve the result, we can pass a timeout, so when that time passes, if the thread hasn't finished, it will throw a `TimeoutException`. It's also a good idea to cancel the task when that happens:

```
try
{
    result = future.get(1, TimeUnit.SECONDS); // Blocks 1 second maximum
    System.out.println("Result: " + result);
} catch (InterruptedException ex) {
} catch (ExecutionException ex) {
} catch (TimeoutException ex) { // When the timeout expires...
    System.err.println("The thread took more than 1 second to complete!");
    executor.shutdownNow(); // Cancel immediately all pending tasks
}
```

Launching several *Callable* tasks at the same time

We can launch more than one *Callable* thread at the same time using an executor. If we pass a list of *Callables* using `invokeAll`, it will return a list of *Futures*. Iterating over the *Future* list and calling `get()` should give us the results. We'll get all results when the last thread finishes.

```java
public static Callable<Integer> getSumCallable(int num1, int num2,
    int secondsSleep)
{
    return () -> {
        try
        {
            TimeUnit.SECONDS.sleep(secondsSleep);
            return num1 + num2;
        } catch (InterruptedException e) {
            throw new IllegalStateException("task interrupted", e);
        }
    };
}

public static void main(String[] args)
{
    List<Callable<Integer>> callables = Arrays.asList(
        getSumCallable(3, 6, 2),
        getSumCallable(5, 8, 3),
        getSumCallable(12, 3, 1)
    );

    ExecutorService executor = Executors.newWorkStealingPool();
    List<Future<Integer>> futures;
    try
    {
        futures = executor.invokeAll(callables);
        executor.shutdown();
        futures.forEach(future -> {
            try
            {
                System.out.println(future.get());
            } catch (InterruptedException | ExecutionException e) {
                throw new IllegalStateException(e);
            }
        });
    } catch (InterruptedException ex) {}
}
```

In this case, we have created a static method that returns the *Callable* object. We call this method many times to add many callables to our list. Then we invoke all of them from the executor. You can download here the source code of this example.

If we don't want to wait until all tasks finish, and instead, want to get only the result of the task that finishes in first place, we can use `invokeAny`. This will return a single *Future* object that should get the result of the first thread that finishes without error. When a task finishes first and returns a value, the rest of tasks are cancelled.

```
ExecutorService executor = Executors.newWorkStealingPool();
try
{
    // Blocks and returns first result
    int firstResult = executor.invokeAny(callables);
    executor.shutdown();
    System.out.println(firstResult); // 15 -> 12 + 3 finishes in 1 second
} catch (InterruptedException ex) {
} catch (ExecutionException ex) { }
```

2.2. Scheduled executors

If we wanted to run a task periodically, instead of doing it manually, we could use a `ScheduledExecutorService`. First of all, we'll see an example of a task that doesn't run periodically, but instead has a delay and waits for a number of seconds before starting. This kind of executor service returns an `ScheduledFuture` object.

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
try
{
    // Usage: schedule(Callable/Runnable, delay, Time unit)
    ScheduledFuture<Integer> schedFuture = executor.schedule(
        getSumCallable(3, 6, 2), 3, TimeUnit.SECONDS);

    executor.shutdown();
    TimeUnit.MILLISECONDS.sleep(1500); // Sleeps for about 1.5 seconds
    long remainingDelay = schedFuture.getDelay(TimeUnit.MILLISECONDS);
    System.out.printf("Remaining Delay: %dms\n", remainingDelay);
    // 1498ms
    int result = schedFuture.get();
    // blocks 3.5 sec. (1.5 delay + 2 task)
    System.out.println("Result: " + result);
} catch (InterruptedException ex) {
} catch (ExecutionException ex) { }
```

To run a scheduled task (task that runs every X time), we should call one of these two methods: `scheduledAtFixedRate` or `scheduleWithFixedDelay`.

`scheduledAtFixedRate` always launches a new thread every X time and doesn't care about the task running time. For example, if we schedule to run a new task every 3 seconds but the task needs 5 seconds, the executor will try to run a new task when 3 seconds have passed since it launched the previous one (the second task will be running at the same time as the first one, and so on)

Using `scheduleWithFixedDelay` can be usually a better idea, because the delay for the next task will begin when the current task finishes (not when it starts). It's important **not to** call `executor.shutdown()` until we want to cancel the scheduled task.

```
public static void main(String[] args)
{
    Runnable task = () -> {
        System.out.println("Time now: " + LocalTime.now().toString());
    };

    ScheduledExecutorService executor =
        Executors.newScheduledThreadPool(1);
    // Delay (1 second), runs every 3 seconds
    executor.scheduleWithFixedDelay(task, 1, 3, TimeUnit.SECONDS);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
    String command;
    try
    {
        do
        {
            // When user presses "q" and "enter", program will end
            command = in.readLine();
        } while (!command.equals("q"));
        executor.shutdown(); // Cancel the scheduled task
    } catch (IOException ex) {}
}
```

Exercise 1:

Create a project named **CallableWordCounting**. Launch 3 Callable threads at the same time using executor's method `invokeAll`.

Each thread will read a different text file (create 3 text files with a lot of text inside, or use these ones) and search how many times a text appears in that file. At the end return the number of times the text has appeared in the file (*Integer*).

Hint: Create a class that implements `Callable<Integer>` and pass to the constructor the file name and the text to search, or create a static method that receives these 2 parameters and returns a *Callable* lambda.

The main thread will get the results and add them, printing the total number of times the word or text has appeared in all files (notice that you don't need any synchronized section or variable for this exercise)

2.3. Using *CompletableFuture*

CompletableFuture is a class that implements `Future` and `CompletionStage` interfaces. `CompletionStage` represents a Promise. This is a great advantage over the previous methodology, because it doesn't block the current thread while waiting for the task to finish. Instead, we'll provide a callback (a function to be executed after the task finishes and returns its result) to it. This kind of elements support both runnables and callables.

This example uses a runnable (this is, a `CompletableFuture` that does not return anything) to print a message in the screen after 3 seconds. We launch it with `runAsync` method.

```java
public static void main(String[] args)
{
    // runAsync receives a runnable that doesn't return anything
    CompletableFuture<Void> compRunnable =
            CompletableFuture.runAsync(() -> {
        try
        {
            TimeUnit.SECONDS.sleep(3);
            System.out.println("Task completed");
        } catch (InterruptedException ex) {}
    });

    // thenRun runs another task (in main thread)
    // when the current task finishes
    compRunnable.thenRun(() ->
        System.out.println("CompletableFuture finish"));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println("Press enter to exit (let the task finish first)");
    try
    {
        in.read();
    } catch (IOException ex) { }
}
```

If we use `Callable` instead of `Runnable`, this is, if we want to get a value back from the thread(s) involved, then we should use `supplyAsync` method instead of `runAsync`. We can use this to process the result returned by the async task (this processing is also asynchronous). In this example, the callable returns a random integer between 0 and 100, and this data is stored in a *CompletableFuture* object.

```
public static void main(String[] args)
{
    CompletableFuture<Integer> compRunnable =
                CompletableFuture.supplyAsync(
    () -> {
        try
        {
            TimeUnit.SECONDS.sleep(3);
            // Return random 0 - 100
            return (new Random()).nextInt(100);
        } catch (InterruptedException ex) {
            return -1;
        }
    });

    // thenAccept receives the result of the previous task to process
    compRunnable.thenAccept((num) ->
        System.out.println("Number generated: " + num));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println(
            "Press enter to exit (let the task finish first)");
    try
    {
        in.read();
    } catch (IOException ex) { }
}
```

`CompletableFuture` can be seen as an asynchronous version of Stream in Java. It allows us to apply/chain multiple filters to the result obtained in first place. To execute intermediate tasks use `thenAccept` (doesn't return anything) or `thenApply` (returns another result).

This example uses a *CompletableFuture* to get a string formatted with a person name and an age (separated by a semicolon). Once the data is obtained, it launches a second asynchronous task to split the string and return a *Person* object with these attributes. Finally, it launches a third asynchronous task to print the person on the screen.

```
public static void main(String[] args)
{
    CompletableFuture<String> compRunnable =
                CompletableFuture.supplyAsync(
    () -> {
        try
        {
            TimeUnit.SECONDS.sleep(3);
            return "Peter;28";
        } catch (InterruptedException ex) {
            return "Error;0";
        }
    });

    // thenApply gets the previous result and returns another (Person)
    CompletableFuture<Person> comPerson = compRunnable.thenApply((str) ->
    {
        String[] parts = str.split(";");
        return new Person(parts[0], Integer.parseInt(parts[1]));
    });

    // thenRun runs the final task with the last processed result
    comPerson.thenAccept((person) -> System.out.println(person));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println(
        "Press enter to exit (let the task finish first)");
    try
    {
        in.read();
    } catch (IOException ex) { }
}
```

What happens if the original task throws an exception and we want to recover from it? (return valid data that can be processed). We can use `exceptionally` method. This method will act like a *catch* statement for exceptions that are thrown in the previous task (it must return the same type of data as the previous task). In this example, we'll see how we can chain all these tasks without using intermediate variables:

```
public static void main(String[] args)
{
    CompletableFuture.supplyAsync(() -> {
        try
        {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException ex) {}
        return "Peter;28";
    }).exceptionally((error) -> {
        // Only if previous step throws an error
        System.err.println("Error: " + error.getMessage());
        return "Error;0";
    }).thenApply((str) -> {
        // Process the string and return a Person
        String[] parts = str.split(";");
        return new Person(parts[0], Integer.parseInt(parts[1]));
    }).thenAccept((person) -> System.out.println(person));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println(
        "Press enter to exit (let the task finish first)");
    try
    {
        in.read();
    } catch (IOException ex) { }
}
```

Finally (although this CompletableFuture API has many more possibilities), we'll see how to execute a task when a number (greater than 1) of CompletableFutures end their tasks, using `CompletableFuture.allOf`. In this example, we'll create tasks that try to ping some servers and at the end, we'll show the results and end the program.

```java
public class ThreadsExamples
{
    public static Deque<String> ipMessages =
        new ConcurrentLinkedDeque<>();

    public static CompletableFuture<Void> pingIp(String address)
    {
        return CompletableFuture.supplyAsync(() -> {
            try
            {
                InetAddress inet = InetAddress.getByName(address);
                if(inet.isReachable(4000)) // 4 seconds
                {
                    return true;
                }
            } catch (UnknownHostException ex) {
            } catch (IOException ex) { }
            return false;
        }).thenAccept(result -> {
            ipMessages.add(address + (result?" ping OK":" unreachable"));
        });
    }

    public static void main(String[] args)
    {
        CompletableFuture<Void> allTasks = CompletableFuture.allOf(
            pingIp("google.es"),
            pingIp("iessanvicente.com"),
            pingIp("apache.org"),
            pingIp("facebook.com")
        );
        allTasks.thenRun(() -> {
            System.out.println("All tasks finished");
            System.out.println(ipMessages);
        });

        while(!allTasks.isDone())
        {
            try
            {
                TimeUnit.MILLISECONDS.sleep(500);
            } catch (InterruptedException ex) {}
        }
    }
}
```

If instead of `allOf`, we use `CompletableFuture.anyOf`, it would execute `thenRun` when the first task ends, instead of waiting for all of them. You can download here the source code from previous examples.

Exercise 2:

Create a project called **FastestWordCounting**. This exercise will be similar to previous *Exercise 1*, but with some differences.

- Create a `CompletableFuture<Integer>` object instead of a `Callable<Integer>` object for reading each file.
- This time, use the `CompletableFuture.anyOf` method. This method will return also a `CompletableFuture<Integer>`, which will receive the value of the thread that finishes first. When this first thread finishes (`thenRun`), print a message like this:

```
The first thread has finished and found the text “cat” 24 times.
```

3. Synchronizing with *Lock*

Since Java 5 there is another way of synchronizing threads (besides using `synchronized` keyword), when trying to get to critical sections. It consists in implementing an interface called `Lock` (from package `java.util.concurrent.locks`). This interface provides methods to lock and unlock a given resource, so that the operations in the in between are guaranteed to be run not concurrently.

```
Lock myLock;
...
public void myMethod()
{
    myLock.lock();
    ... // Do some not concurrent tasks
    myLock.unlock();
}
```

We should define a class that implements this interface. Fortunately, Java provides such a class: `ReentrantLock` (from the same package), so we can use this class directly to create our lock:

```
Lock myLock = new ReentrantLock();
```

Exercise 3:

Create a project call **BankAccountLock**, that is a copy of the project created in previous documents (*BankAccountSynchronized*). Replace the old *synchronized* methods with the *Lock* mechanisms that we have just seen, and check that everything goes OK.

3.1. Read / Write locks

Besides, there is an improvement brought by this `Lock` interface: the possibility of having read and write operations working separately, so that there can be multiple read operations running at the same time on a given file or resource, but only one write operation (when a thread is writing, no one else can be reading or writing). We can achieve this with the `ReadWriteLock` interface and its implementation in `ReentrantReadWriteLock` class. This class has two locks, one for reading operations and one for writing operations, so that we can use any of them depending on the operation we actually want to do.

```
ReadWriteLock lock = new ReentrantReadWriteLock();
...
public void readOperation()
{
    lock.readLock().lock();
    ... // This area can be achieved by multiple reading threads
    lock.readLock().unlock();
}

public void writeOperation()
{
    lock.writeLock().lock();
    ... // When a writing thread gets here, no one else can have the lock
    lock.writeLock().unlock();
}
```

As you can see in the code below, the read lock allows to lock an object for reading, so that any other reading operation can also get to the critical section. However, when a write lock wants to be set, no other lock can be currently applied. In other words, we can have multiple readers running the critical section at the same time, but whenever a writer is running the critical section, no other thread can be running it.

Let's see how it works with the following example: we are going to create a class that stores an integer value:

```java
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class MyData
{
    int value;
    ReentrantReadWriteLock lock;

    public MyData(int value)
    {
        this.value = value;
        lock = new ReentrantReadWriteLock();
    }

    public int getValue()
    {
        lock.readLock().lock();
        try { Thread.sleep(2000); } catch (Exception e) {}
        System.out.println("Thread #" + Thread.currentThread().getId() +
            " reads value " + value);
        int v = value;
        lock.readLock().unlock();
        return v;
    }

    public void setValue(int value)
    {
        lock.writeLock().lock();
        try { Thread.sleep(2000); } catch (Exception e) {}
        System.out.println("Thread #" + Thread.currentThread().getId() +
            " sets value to " + (this.value + value));
        this.value += value;
        lock.writeLock().unlock();
    }
}
```

Now, we define a thread class that tries to read it, and a thread class that tries to change its value:

```java
public class ReadingThread extends Thread
{
    MyData sharedData;

    public ReadingThread(MyData sharedData)
    {
        this.sharedData = sharedData;
    }

    @Override
    public void run()
    {
        int value = sharedData.getValue();
    }
}

public class WritingThread extends Thread
{
    MyData sharedData;

    public WritingThread(MyData sharedData)
    {
        this.sharedData = sharedData;
    }

    @Override
    public void run()
    {
        sharedData.setValue(10);
    }
}
```

If we run a main program like this one:

```
MyData mds = new MyData(10);
ReadingThread[] threadsR = new ReadingThread[5];
WritingThread[] threadsW = new WritingThread[2];

for (int i = 0; i < threadsW.length; i++)
{
    threadsW[i] = new WritingThread(mds);
}

for (int i = 0; i < threadsR.length; i++)
{
    threadsR[i] = new ReadingThread(mds);
}

for (int i = 0; i < threadsW.length; i++)
{
    threadsW[i].start();
}
for (int i = 0; i < threadsR.length; i++)
{
    threadsR[i].start();
}
```

we will notice that all the reading threads print their results at the same time, whereas the writing threads print their messages separately, after 2 seconds. The output may be something like this (it may differ depending on the order in which threads are actually started):

```
Thread #9 sets value to 20
Thread #13 reads value 20
Thread #11 reads value 20
Thread #14 reads value 20
Thread #12 reads value 20
Thread #15 reads value 20
Thread #10 sets value to 30
```

Exercise 4:

Create a project called **ReadersWritersLock** and copy the previous example on it. Make changes to the code so that there are 10 reading threads (instead of 5), and each thread (reader or writer) will sleep a random number of seconds (between 1 and 10), and then it will do its job. This way, there should be some reading operations at the beginning, some in the middle of the two writings, and some at the end. Your output should look like this one:

```
Thread #13 reads value 10
Thread #11 reads value 10
Thread #9 sets value to 20
Thread #14 reads value 20
Thread #12 reads value 20
Thread #15 reads value 20
Thread #10 sets value to 30
Thread #16 reads value 30
Thread #18 reads value 30
...
```

4. The Fork/Join framework

The thread executor that we have just seen was added in Java 5, and it allows us to separate the thread creation and its execution. Since Java 7, we can go a step further thanks to the *Fork/Join* framework.

With this framework, we can divide complex or big problems into smaller ones. So, this framework is based on two operations: **fork** (divide a task into smaller tasks) and **join** (a task waits for its subtasks to finish). However, tasks involved in *Fork/Join* framework have no other synchronization mechanism.

Fork/Join framework relies on two classes: `ForkJoinPool` (it will manage the tasks and will offer information about their execution) and `ForkJoinTask` (the base class of every task added to the `ForkJoinPool`). This class has two implemented subclasses: `RecursiveAction` (for tasks that will not return any result) and `RecursiveTask` (for tasks that will return a result). All these classes belong to `java.util.concurrent` package.

4.1. Example: tasks that do not return any result

Let's see how this framework can be used with the following example: we are going to create a list of video games, with their titles and prices. Then, we are going to look for a given title in the list, so that, if the list size is smaller than 5 video games, only one task will be needed, but if not, a task will be created to search a subset of up to 5 video games from the list.

Our VideoGame class would be like this one:

```
public class VideoGame
{
    String title;
    float price;

    public VideoGame(String title, float price)
    {
        this.title = title;
        this.price = price;
    }

    public String getTitle()
    {
        return title;
    }

    public float getPrice()
    {
        return price;
    }
}
```

Our thread or task to search in the list would be like this one:

```
public class GameSearch extends RecursiveAction
{
    /* How many video games will each task be in charge of? */
    public static final int MAX_GAMES = 5;
    /* List of video games */
    ArrayList<VideoGame> list;

    /* First index of the list to search */
    int first;

    /* Last index of the list to search */
    int last;

    /* Text to be searched in the list */
    String text;

    public GameSearch(ArrayList<VideoGame> list, String text, int first,
        int last)
    {
        this.list = list;
        this.text = text;
        this.first = first;
        this.last = last;
    }

    @Override
    protected void compute()
    {
        if (last - first <= MAX_GAMES)
            search();
        else
        {
            int middle = (last - first)/2;
            System.out.println("Creating 2 subtasks...");
            GameSearch s1 = new GameSearch(list, text, first, middle+1);
            GameSearch s2 = new GameSearch(list, text, middle+1, last);
            invokeAll(s1, s2);
        }
    }

    public void search()
    {
        for (int i = first; i < last; i++)
        {
            try { TimeUnit.SECONDS.sleep(1); } catch (Exception e) {}
            if (list.get(i).getTitle().contains(text))
                System.out.println("Found at position " + i + ": " +
                    list.get(i).getTitle());
        }
        }
    }
```

Notice that, when we extend `RecursiveAction` class, we need to define a `compute` method. This would be the equivalent to the *run* method in common threads. Inside this method, we check the size of the game list. If it is smaller than 5, we just call the search method to solve the problem. Otherwise, we divide the list in two parts and create two subtasks; each one will be in charge of searching in one half of the list.

We can also create a list of tasks, and call the `invokeAll` method passing that list as a parameter:

```
ArrayList<GameSearch> subtasks = new ArrayList<>();
...
subtasks.add(new GameSearch(...));
subtasks.add(new GameSearch(...));
subtasks.add(new GameSearch(...));

invokeAll(subtasks);
```

From our main program, we create the video game list, create a `GameSearch` task to look for the word "Assassin's", and launch it in the *Fork/Join* pool, as we did before with thread executors:

```
public static void main(String[] args)
{
    ArrayList<VideoGame> list = new ArrayList<VideoGame>();
    list.add(new VideoGame("Assassin's Creed", 19.95f));
    list.add(new VideoGame("The last of us", 49.90f));
    list.add(new VideoGame("Fifa 14", 39.95f));
    list.add(new VideoGame("Far Cry 2", 14.95f));
    list.add(new VideoGame("Watchdogs", 59.95f));
    list.add(new VideoGame("Assassin's Creed II", 24.90f));
    list.add(new VideoGame("Far Cry 3", 39.50f));
    list.add(new VideoGame("Borderlands", 19.90f));

    GameSearch v = new GameSearch(list, "Assassin's", 0, list.size());
    ForkJoinPool pool = new ForkJoinPool();
    pool.execute(v);
    do
    {
        try { Thread.sleep(100); } catch (Exception e) {}
    } while (!v.isDone());
    pool.shutdown();
}
```

Main program has to wait until task finishes (using its `isDone` method), before shutting down.

4.2. Example: tasks that return a result

How could we adapt the previous example so that tasks do not print anything to the output, and return a set or list of results found? We have to use a subclass of `RecursiveTask` instead of a subclass of `RecursiveAction`. When we extend `RecursiveTask`, we have to take into account that it is a parameterized class, this is, we need to provide the type of result that will be returned. So our `GameSearch` class from previous example would look like this one now:

```java
public class GameSearch extends RecursiveTask<ArrayList<String>>
{
    /* How many video games will each task be in charge of? */
    public static final int MAX_GAMES = 5;
    /* List of video games */
    ArrayList<VideoGame> list;
    /* First index of the list to search */
    int first;
    /* Last index of the list to search */
    int last;
    /* Text to be searched in the list */
    String text;

    public GameSearch(ArrayList<VideoGame> list, String text, int first,
    int last)
    {
        this.list = list;
        this.text = text;
        this.first = first;
        this.last = last;
    }

    @Override
    protected ArrayList<String> compute()
    {
        ArrayList<String> results = new ArrayList<String>();
        if (last - first <= MAX_GAMES)
            results = search();
        else
        {
            int middle = (first + last)/2;
            System.out.println("Creating 2 subtasks...");
            GameSearch s1 = new GameSearch(list, text, first, middle+1);
            GameSearch s2 = new GameSearch(list, text, middle+1, last);
            invokeAll(s1, s2);
            try
            {
                results = s1.get();
                ArrayList<String> aux = s2.get();
                results.addAll(aux);
            } catch (Exception e) {}
        }
        return results;
    }

    public ArrayList<String> search()
    {
        ArrayList<String> results = new ArrayList<String>();
        for (int i = first; i < last; i++)
        {
            try { TimeUnit.SECONDS.sleep(1); } catch (Exception e) {}
            if (list.get(i).getTitle().contains(text))
                results.add("Found at " + i + ": " +
                    list.get(i).getTitle());
        }
        return results;
    }
}
```

We are going to return an `ArrayList` of *Strings* as a result, each one containing each occurrence of the searched text. In `search` method, we just create the list of games matching the text and return it. In `compute` method, we call the `search` method directly if there are less than 5 games to search, or we split the work in two tasks, and join their results in the `try..catch` block (calling to `get` method may throw exceptions).

Our main program will get the results after the main task has finished, and it will print them to the standard output:

```
public static void main(String[] args)
{
    ...
    /* Main method is the same until we call the shutdown method, then we
    need to add some lines to get and print the results */
    try
    {
        ArrayList<String> results = v.get();
        for (int i = 0; i < results.size(); i++)
            System.out.println(results.get(i));
    } catch (Exception e) {
        System.out.println("Exception occurred: " + e.getMessage());
    }
    pool.shutdown();
}
```

> **Exercise 5:**
>
> Create a project called **ForkJoinFile**. Create a text file in it, with some lines (at least 50, you can copy them from any source). Use the *Fork/Join* framework to create tasks that will check the contents of the text file (up to 10 lines for each task). The tasks must replace every occurrence of the word "java" with "Java" (of course, try to add some occurrences of the word "java" in the text file). At the end, main program will get the results of all the subtasks (i.e., the pieces of text with the replacements done), will join them and will rewrite the text file with the updated text.

4.3. Launching asynchronous subtasks

In the examples shown above, when we call the `invokeAll` method, the task that calls it waits until the subtasks invoked finish their job. This is, we are using a synchronous way of calling tasks and subtasks.

We can also call the subtasks in an asynchronous way (this is, the main task continues its job while it launches the subtasks), by using the `fork` and `join` methods, instead of `invokeAll` and other synchronous methods. So previous code of `compute` method could be changed into an asynchronous one this way:

```
@Override
protected void compute()
{
    if (last - first <= MAX_GAMES)
        search();
    else
    {
        int middle = (first + last)/2;
        System.out.println("Creating 2 subtasks...");
        GameSearch s1 = new GameSearch(list, text, first, middle+1);
        GameSearch s2 = new GameSearch(list, text, middle+1, last);
        s1.fork();
        s2.fork();
        // At this point, this task continues running its code
        ...
        // Wait for the 1st subtask to finish
        s1.join();
        ...
        // Wait for the 2nd subtask to finish
        s2.join();
    }
}
```