Callables and synchronisation

Key terms

Callables

> The callable interface is a generic interface containing a single method, call(). The call() method accepts zero or more arguments, and returns an object. The callable interface is used for having threads return values unlike the Runnable interface.

Future

> A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready.

Syncronisation

> Synchronization is the capability to control the access of multiple threads to any shared resource.

Callables

Runnables do not return a result. If we want to execute a task that returns a result, we can use the `Callable` interface. The `Callable` interface is a functional interface that has only one method:

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

The `call` method returns a result of type `V`. The `call` method can throw an exception. The `Callable` interface is used to execute tasks that return a result. For instance we can use the `Callable` interface to execute a task that returns the sum of two numbers:

```
Callable<Integer> sumTask = () -> 2 + 3;
```

In order to execute a task that returns a result, we can use the `submit` method of the `ExecutorService` interface. The `submit` method takes a `Callable` object as a parameter. The `submit` method returns a `Future` object. The `Future` interface has a method called `get` that returns the result of the task. The `get` method is a blocking method. It waits until the task is completed and then returns the result of the task.

```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<Integer> future = executorService.submit(() -> 2 + 3);
Integer result = future.get();
```

Futures can be used to cancel tasks. The `Future` interface has a method called `cancel` that can be used to cancel a task. The `cancel` method takes a boolean parameter. If the boolean parameter is `true`, the task is cancelled even if the task is already running. If the boolean parameter is `false`, the task is cancelled only if the task is not running.

```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<Integer> future = executorService.submit(() -> 2 + 3);
future.cancel(false);
```

Adder and Subtractor

- Create a count class that has a count variable.
- Create two different classes `Adder` and `Subtractor`.
- Accept a count object in the constructor of both the classes.
- In `Adder`, iterate from 1 to 100 and increment the count variable by 1 on each iteration.
- In `Subtractor`, iterate from 1 to 100 and decrement the count variable by 1 on each iteration.
- Print the final value of the count variable.
- What would the ideal value of the count variable be?
- What is the actual value of the count variable?
- Try to add some delay in the `Adder` and `Subtractor` classes using inspiration from the code below. What is the value of the count variable now?

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Adder

```
public class Adder implements Runnable {
    private Count count;

    public Adder(Count count) {
        this.count = count;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            count.increment();
        }
    }
}
```

Subtractor

```
public class Subtractor implements Runnable {
    private Count count;

    public Subtractor(Count count) {
        this.count = count;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            count.decrement();
        }
    }
}
```

Runner

```
public class Runner {
    public static void main(String[] args) {
        Count count = new Count();
        Adder adder = new Adder(count);
```

```
        Subtractor subtractor = new Subtractor(count);

        Thread adderThread = new Thread(adder);
        Thread subtractorThread = new Thread(subtractor);

        adderThread.start();
        subtractorThread.start();

        adderThread.join();
        subtractorThread.join();

        System.out.println(count.getCount());
    }
}
```

Synchronisation

Running the above code should ideally result in the count variable being 0. However, if you increase the number of iterations in the `Adder` and `Subtractor` classes, you will see that the count variable is not 0. This is because the `Adder` and `Subtractor` classes are running in parallel and are not synchronised. This means that the `Adder` and `Subtractor` classes are not waiting for each other to finish before they start their own iterations. This results in the count variable being incremented and decremented at the same time, resulting in the count variable not being 0.

The major issue is that multiple threads are accessing a shared resource at the same time.

Characteristics of synchronisation problems

- `Critical section` - A section of code that is accessed by multiple threads. When multiple threads access the same critical section, the result is a synchronisation problem that might yield wrong or inconsistent results.

```
    public void run() {
        for (int i = 0; i < 100; i++) {
            count.increment(); // Critical section
        }
    }
```

- `Race Conditions` - When more than one thread tries to enter the critical section at the same time.
- `Preemption` - When a thread is interrupted by another thread. It could be possible that the interrupted thread is in the middle of a critical section. This could result in the interrupted thread not being able to finish the critical section and yield inconsistent results.

Properties of a good solution

- `Mutual Exclusion` - Only one thread can access the critical section at a time.
- `Progress` - If a thread wants to enter the critical section, it will eventually be able to do so.

- `Bounded Waiting` - If a thread wants to enter the critical section, it will eventually be able to do so, but only after a finite number of other threads have entered the critical section.
- `No busy Waiting` - If a thread wants to enter the critical section, it will not be able to do so until the critical section is free. It has to keep checking if the critical section is free. This is called busy waiting.
- `Notification` - If a thread is waiting to enter the critical section, it should be notified when the critical section is free.

Solutions to synchronisation problems

Mutex Locks

Mutex locks are a way to solve the synchronisation problem. Mutex locks are a way to ensure that only one thread can access a critical section at a time. Mutex locks are also known as `mutual exclusion locks`.

A thread can only access the critical section if it has the lock. If a thread does not have the lock, it cannot access the critical section. If a thread has the lock, it can access the critical section. If a thread has the lock, it can release the lock and allow another thread to access the critical section.

Think of a room with a lock. Only one person can enter the room at a time. If a person has the key, they can enter the room. If a person does not have the key, they cannot enter the room. If a person has the key, they can leave the room and give the key to another person. This is the same as a mutex lock.

Properties of a mutex lock

- `Lock` - A thread can only access the critical section if it has the lock.
- Only one thread can have the lock at a time.
- Other threads cannot access the critical section if a thread has the lock and thus have to wait.
- Lock will automatically be released when the thread exits the critical section.

Syncronised keyword

The `synchronized` keyword is a way to solve the synchronisation problem. The `synchronized` keyword is a way to ensure that only one thread can access a critical section at a time.

A synchronized method or block can only be accessed by one thread at a time. If a thread is accessing a synchronized method or block, other threads cannot access the synchronized method or block. If a thread is accessing a synchronized method or block, other threads have to wait until the thread exits the synchronized method or block.

Following is an example of a synchronized method:

```
public class Count {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized void decrement() {
        count--;
```

```
    }

    public int getCount() {
        return count;
    }
}
```

In the above example, the `increment()` and `decrement()` methods are synchronized. This means that only one thread can access the `increment()` and `decrement()` methods at a time. If a thread is accessing the `increment()` method, other threads cannot access the `increment()` method. If a thread is accessing the `decrement()` method, other threads cannot access the `decrement()` method. If a thread is accessing the `increment()` method, other threads have to wait until the thread exits the `increment()` method. If a thread is accessing the `decrement()` method, other threads have to wait until the thread exits the `decrement()` method.

Similarly, the `synchronized` keyword can be used to synchronize a block of code. Following is an example of a synchronized block:

```
public class Count {
    private int count = 0;

    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public void decrement() {
        synchronized (this) {
            count--;
        }
    }

    public int getCount() {
        return count;
    }
}
```

If you declare a method as `synchronized`, only one thread will be able to access any `synchronized` method in the class. This is because the `synchronized` keyword is associated with the object.

Reading List

- Web Browser architecture