1. REFACTORING

The aim of this exercise is to work on the concept of refactoring and its application.

Below is the **StaffCost** class, which contains the **WorkersCost** method.
This method has the function of calculating the final cost of each worker based on the values of the parameters that define them. Specifically, if the type of worker is director or deputy director, the worker's final cost will be the corresponding to his payroll. But if he is neither a director nor deputy director, the extra hours he has worked for a value of 20 must be added to his payroll.

Given the following code with the implementation of this class previously explained, it is necessary to analyze it and tell if it can be refactored. If refactoring can be done, it is requested to refactor the code and show the resulting code.

```
public class StaffCost {
	static float WorkersCost(Worker workers[]) {
		float finalCost = 0;
		Worker worker;
		for (int i = 0; i < workers.length; i++) {
			worker= workers[i];
			if (worker.getWorkerType() == Worker.DIRECTOR ||
worker.getWorkerType() == Worker.DEPDIRECTOR) {
				finalCost += worker.getPayroll();
			} else {
				finalCost += worker.getPayroll() +
(worker.getExtraHours() * 20);
			}
		}
		return finalCost;
	}
}
```

2. REFACTORING VALIDATION

The aim of this exercise is to take a step further in the application of refactoring and its relationship with unitary testing.

Answer the following question: How can you validate that the refactored code is still correct?

Use examples to demonstrate your answer.