Hellas Direct Insurance Coding Exercise

Implementation Notes + Thinking Rationale

First thing to do is gather the requirements in a bulleted list.
This helps me stay focused on the deliverables and write my tests.

For now, I only have Functional Requirements:

- Base annual premium with one bedroom = £110
- no bedrooms = 1 bedroom => will be insured
- more than 4 bedrooms => will not be insured
- thatched roof => will not be insured
- Third party service integrations
    - subsidence area => will not be insured
    - high risk of flooding => will not be insured
    - high risk of flooding => will be insured with 15% extra charge
    - any other risk => will be insured (no effect)

Software Design:

- Since we are dealing with currency, I chose the `BigDecimal` class to manipulate and perform operations for better accuracy.
- The base premium is by default £110. This could change in the future, so I put this logic in a separate class (`BaseAnnualPremiumChargeService`). The class accepts house information as parameter and returns the base premium value.
- The base premium is modified by a percentage based on several factors. I created a registry class (`ExtraChargesCalculatorRegistry`) and separate each factor in its own logic. The registry goes through each one of them and calculates the total percentage to apply on the base premium. Each factor implements an interface (`PremiumExtraChargeCalculator`).
- The `HouseInsuranceServiceImpl` class returns the premium amount using the two above classes, `BaseAnnualPremiumChargeService` and `ExtraChargesCalculatorRegistry`. It returns a `ServiceResponse` object with code, message (in case of error) and data (the calculated premium).
- I changed the parameters of `HouseInsuranceService.getCalculatedPremium` to a single class object (`HouseDetails`), because the number might grow in the future.
- Each class that implements `PremiumExtraChargeCalculator` returns a percentage value (in decimal) that would be applied to the base premium. In case of error or logic that decides against the insurance of a house, an exception is thrown. There are different types of exception, depending on the error and logic.
- I wrote tests to test my implementation (`Junit`). I created mock classes to test different scenarios and the third-party services (see `Mocks.java`)

Other Notes:

- You can run the tests to test the implementation.
- The IDE I used is Intellij Jetbrains. I used the IDE's feature to load libraries (`Junit`) and their default Code Style scheme for Java.

- There is some inconsistency between the two third party services. I tried to mitigate their implementation differences by slightly changing the logic in their corresponding `ExtraChargeCalculator` classes, i.e. the `FloodAreaChecker` throws exception if postcode is not found, but the `SubsidenceAreaChecker` returns null. I made sure to throw an exception in both cases.
- Potential improvements:
    - The service is limited to responding on the first reason it fails to insure a house. It could be altered to allow listing all the reasons.
    - What happens if the third-party services are not responsive? What do we do in that case? Should we implement a mechanism to retry later? Ideally, a faulty third-party service should not translate to a non-insurable house.
    - We could implement some caching mechanisms for the third-party services. Most probably inputs/outputs are not regularly changed.