Parkeringsapp - Uppgift 1: Grundläggande Datamodeller och CLI

Introduktion

I denna första uppgift ska ni börja utveckla grunden för en parkeringsapplikation. Ni ska skapa de grundläggande datamodellerna och ett enkelt CLI (program som körs från terminalen) för att hantera dessa modeller.

Mål

- Skapa grundläggande datamodeller för parkeringsapplikationen
- Implementera ett enkelt CLI för att hantera dessa modeller
- Börja tänka på hur olika delar av applikationen interagerar med varandra

Resurser

- Börja här!
- Fippla lite här!
- Kom igång här!
- Börja koda ditt CLI med dart:io!

Uppmaning

- Gör inte allt på en och samma gång! Välj en sak i taget och försök ta kortaste möjliga steget till att lösa det problemet. Kursen förväntar sig mycket självstudier av er så gör en plan och försök följa den. Ni förväntas läsa en hel del dokumentation för att styra era beslut. Ta ett beslut, försök, testa, misslyckas, utvärdera, gör om, lyckas, gå vidare!

Krav

1. Datamodeller

Skapa följande klasser i Dart:

1. `Person`
    - Attribut: namn, personnummer
2. `Vehicle`
    - Attribut: registreringsnummer, typ (bil, motorcykel, etc.), ägare (en `Person`)
3. `ParkingSpace`
    - Attribut: id, adress, pris per timme
4. `Parking`
    - Attribut: fordon, parkeringsplats, starttid, sluttid (kan förslagsvis vara null om pågående)

2. Datahanteringsklasser

För att hantera samlingar av våra huvudklasser, skapa följande klasser:

1. `PersonRepository`
    - Ansvarar för att lagra och hantera en samling av `Person`-objekt
2. `VehicleRepository`
    - Ansvarar för att lagra och hantera en samling av `Vehicle`-objekt
3. `ParkingSpaceRepository`
    - Ansvarar för att lagra och hantera en samling av `ParkingSpace`-objekt
4. `ParkingRepository`
    - Ansvarar för att lagra och hantera en samling av `Parking`-objekt

Varje repository-klass bör innehålla metoder för att lägga till, ta bort, uppdatera och hämta objekt. Exempel på metodnamn kan vara:

- `add(item)`
- `getAll()`
- `getById(id)`
- `update(item)`
- `delete(id)`

3. CLI (Command Line Interface)

Skapa ett enkelt CLI som låter användaren:

1. Välja vilken datamodell de vill arbeta med (Person, Fordon, Parkeringsplats, Parkering)
2. För varje datamodell, erbjud följande operationer:
    - Skapa ny (Create)
    - Visa alla (Read)
    - Uppdatera befintlig (Update)
    - Ta bort (Delete)

Exempel på CLI-flöde

```
Välkommen till Parkeringsappen!
Vad vill du hantera?
1. Personer
2. Fordon
3. Parkeringsplatser
4. Parkeringar
5. Avsluta

Välj ett alternativ (1-5): 1

Du har valt att hantera Personer. Vad vill du göra?
1. Skapa ny person
2. Visa alla personer
3. Uppdatera person
4. Ta bort person
5. Gå tillbaka till huvudmenyn

Välj ett alternativ (1-5):
```

Tips

- Börja med att skapa enkla versioner av klasserna och lägg till mer funktionalitet efter hand.
- Använd listor eller maps för att lagra data i minnet (ingen permanent lagring krävs än).
- Tänk på hur de olika klasserna relaterar till varandra. Till exempel, ett `Vehicle` har en ägare som är en `Person`.
- Använd Darts inbyggda `stdin` och `stdout` för att hantera input och output i CLI:n. Börja här kanske?
- Använd listor eller maps i dina repository-klasser för att lagra data i minnet.
- Tänk på att uppdatera både huvudobjektet (t.ex. en `Person`) och motsvarande repository (t.ex. `PersonRepository`) när du gör ändringar. Undersök: är objekten call by reference eller call by value? Ändras svaret på detta när objekten är lagrade i en databas? Vill du ha eller vill du inte ha mutability? Än finns det inga rätt och fel så låt din egen programmerarstil lysa i denna första uppgift så får vi mer utrymme för diskussion senare!

Potentiellt användbara koncept att läsa upp på

- Class Definitions
- Getters
- Abstract Classes
- Interfaces
- Inheritance
- Enumerated Types (enum)
- Generics
- Nullable Types
- Named Constructors
- Mixins
- Method Overriding
- Future
- Required Named Parameters
- Optional Parameters
- Default Parameters
- Numbers (int, double)
- Strings (String)
- Booleans (bool)
- Records (Tuples)
- Lists (List)
- Maps (Map)
- Null (Null)
- Lambda Functions

Utmaningar (förslag för ökad kvalité för att uppnå VG)

- Implementera enkel **felhantering**, t.ex. för ogiltiga inputs.
- Lägg till en funktion för att **beräkna kostnaden för en parkering** baserat på tid och pris per timme.
- Implementera en enkel **sökfunktion** för att hitta specifika objekt baserat på vissa kriterier (t.ex. hitta alla fordon för en specifik ägare).
- **Designmönster**: Implementera Singleton-mönstret för dina repository-klasser för att säkerställa att endast en instans av varje repository existerar i programmet.

Utmaningar för framtida övningar (förslag för ökad kvalité för att uppnå VG)

- **Serialisering och Deserialisering**: Implementera `toJson()` och `fromJson()` metoder för varje klass. Detta kommer att vara användbart när ni senare ska arbeta med HTTP-servrar och databaser.
- **Datavalidering**: Skapa en metod `isValid()` för varje klass som kontrollerar om alla nödvändiga fält är korrekt ifyllda. Till exempel, att ett personnummer har rätt format eller att ett registreringsnummer är giltigt.
- **Databasförberedelse**: Lägg till ett unikt `id`-fält till varje klass. Detta kommer att vara användbart när ni senare ska lagra data i en databas.
- **Asynkron programmering**: Modifiera några av era metoder att returnera sitt svar i en `Future`. Detta kommer att förbereda er för asynkron programmering som är vanlig i HTTP-anrop och databasoperationer.

Dessa utmaningar kommer att ge er en försmak av koncept som ni kommer att möta i kommande sprintar och förbereda er kodbas för framtida utökningar. Lycka till!

Gör minst 4/8 av dessa förslag för att få VG på uppgiften.

Lycka till med er första uppgift! Detta kommer att ge er en bra grund för de kommande sprintarna i projektet.