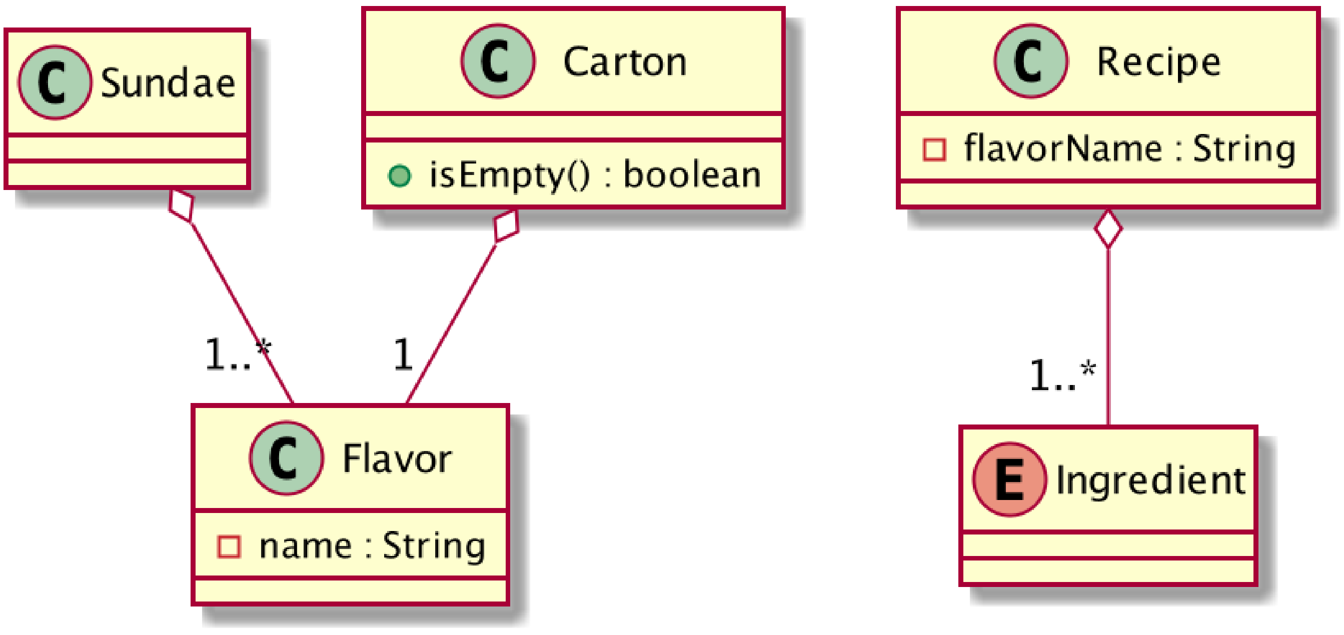
Lambda Expressions & Functional Interfaces IceCreamParlorService - Lambda-flavored

Introduction

We're returning to the `IceCreamParlorService` from a prior lesson. It's evolved a bit since then, but some of the key elements are still the same.

Key models:

- `Carton`: A carton of ice cream of a particular flavor that is available in the parlor. If the carton `isEmpty()`, however, there is none of that flavor left!
- `Flavor`: A particular flavor of ice cream. Each flavor has a unique combination of `Ingredient`s, which are specified by a `Recipe`.
- `Ingredient`: An ingredient needed to make ice cream. Some ingredients are shared across flavors, some are unique to a flavor.
- `Recipe`: For a given flavor, the recipe indicates the ingredients to add to the mixture (as well as the order in which they should be added, in the form of a `Queue`).
- `Sundae`: What our customers order! They contain a list of scoops of `Flavor` objects representing each scoop in the sundae.



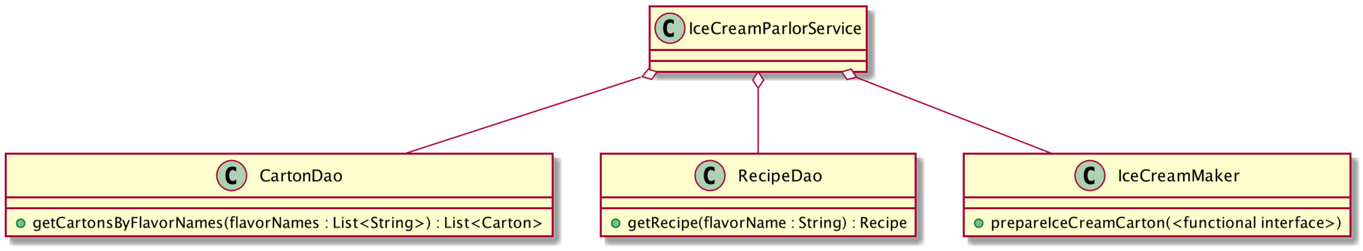
Ice Cream Parlor Service Models diagram PlantUML source

Customers can order a sundae by calling `getSundae()` with a list of flavors they want in the sundae.

There is also a method on our service, `prepareFlavors()` for preparing more ice cream by passing in a list of flavors to create. These are then looked up in the recipe store, which gives us the ingredients to mix into the ice cream.

The `IceCreamParlorService` depends on:

- two DAOs, `CartonDao` and `RecipeDao`, for accessing `Carton`s and `Recipe`s, respectively.
- `IceCreamMaker`, which accepts ingredients and then does the mixing and freezing.



IceCreamParlorService
CartonDao
getCartonsByFlavorNames(flavorNames : List<String>) : List<Carton>
RecipeDao
getRecipe(flavorName : String) : Recipe
IceCreamMaker
prepareIceCreamCarton(<functional interface>)

Ice Cream Parlor Service class diagram PlantUML source

There are several places in the code that need functional interfaces to be created and passed into methods. Your job is to provide those functional interface implementations, sometimes using lambda expressions, sometimes with method references.

Phase 0: It builds

1. Build your project. *(If necessary, null out instantiations to your Dagger component to complete the initial build.)*
2. Run the `Phase0Test` and verify that it passes

GOAL: Make sure that your project builds and the starter test passes.

Phase 0 is complete when:

- The `Phase0Test` runs and passes

Phase 1: No empty scoops

You're going to make sure we don't deliver any "empty scoops" of ice cream in our customers' sundaes by updating the `IceCreamParlorService.getSundae()` method.

You'll have a list of cartons of ice cream, and need to remove any cartons that are empty (`isEmpty()` returns true). The functional interface that you will be implementing with a lambda expression (and later a method reference) accepts a `Carton` and returns `true` if the carton is empty, `false` if not.

GOAL: use a method reference to remove empty cartons so we only add non-empty flavors to the sundae.

Phase 1 is complete when:

- `Phase1Test` is passing

Phase 2: Build that sundae!

Now you get to build the sundae of your customers' dreams by updating the `IceCreamParlorService.buildSundae()` method. You'll create a lambda expression that implements a functional interface. The functional interface will accept each carton of ice cream and adds a scoop to the `Sundae`.

The functional interface you will be implementing with a lambda expression accepts a `Carton` and doesn't need to return anything.

GOAL: Use a lambda expression implementing a functional interface that adds a scoop from each carton to the customer's sundae.

Phase 2 is complete when:

- `Phase1Test` and `Phase2Test` are passing

Phase 3: Get the recipe

You're going to update `IceCreamParlor`'s `prepareFlavors()` method to convert a list of `Recipe`s into a list of `Queue<Ingredient>`.

You'll create a lambda expression to do this. It will accept a `Recipe` and need to return a `Queue<Ingredient>`.

We request that you use `RecipeConverter`'s static method to do the conversion.

GOAL: use a method reference to convert recipes to their `Queue<Ingredient>` components

Phase 3 is complete when:

- `Phase1Test`, `Phase2Test`, and `Phase3Test` are passing.

Phase 4: Make that ice cream

You're going to update `IceCreamService`'s `makeIceCreamCartons()` method to pass a functional interface to an `IceCreamMaker` method it calls. The functional interface will supply a new ingredient each time it's called by `IceCreamMaker`'s method.

The functional interface you will implement with a lambda expression returns a new `Ingredient` each time it is called (with no arguments).

GOAL: Implement logic that accepts a lambda expression and call that method with a lambda expression, so that the proper ingredients are mixed in for each flavor.

Phase 4 is complete when:

- All tests are passing