Article

Adopting inheritance in SwiftData

Add flexibility to your models using class inheritance.

Overview

SwiftData supports class inheritance, an object-oriented programming feature that enables you to expand the kinds of use cases your models can support by creating new subclasses that add properties and capabilities to an existing base class.

Like other Swift subclasses, SwiftData models can inherit the properties and capabilities of a parent or superclass. In SwiftData this allows you to add new properties and behaviors that extend the capabilities of your models by creating a hierarchical relationship between them that you can operating on using query, predicate, and history operations. This enables you to build in more flexibility into your app as your models become more specialized and able to serve more diverse use cases.

An example of such an app might one that tracks trips: both personal trips, such as family vacations, and business trips. At a high level, a trip might be captured in a very concise model, like the one shown here.

```
@Model LivingAccommodation { ... }

@Model class Trip {
    @Attribute(.preserveValueOnDeletion)
    var name: String
    var destination: String

    @Attribute(.preserveValueOnDeletion)
    var startDate: Date

    @Attribute(.preserveValueOnDeletion)
```

```
    var endDate: Date

    @Relationship(deleteRule: .cascade, inverse: \BucketListItem.trip)
    var bucketList: [BucketListItem] = [BucketListItem]()

    @Relationship(deleteRule: .cascade, inverse: \LivingAccommodation.trip)
    var livingAccommodation: LivingAccommodation?

    var isBusinessTrip: Boolean = false
}
```

A name, a destination, start and end dates, living accommodations, and `true` or `false` values that indicate a business or personal trip: this is functional, but as the capabilities for a trip tracking app grow, it might expand to track more kinds of information, and this is where inheritance can provide more flexibility.

Design for specialization

As described above, inheritance allows you to develop models that share fundamental properties but that diverge as use cases become more specialized. For example, a more expansive view of business and personal trips may include the addition of new elements depending upon the specific use case. The following table shows a few more elements that these trips this might record.

Personal Trip	Business Trip
attractionsToVisit	expenseCode
familyMembers	businessMeals
frequentFlyerCodes	milageRate
venuePasses	milesDriven
	conferenceSessions

Subclasses that build upon the base `Trip` model make use of its properties and any common behavior, but keep these new behaviors isolated. For example, a `PersonalTrip` doesn't have to have a state or behavior for calculating the cumulative miles traveled in a reimbursement calculation: that's more relevant to a `BusinessTrip`.

Given the outline of personal and business trip specialization above, you'd create model subclasses for SwiftData subclasses as you would in Swift, but with the addition of the `@Model` macro, to indicate the new class is a model to SwiftData. A refactoring of our trip classes into a parent (base) class and two subclasses could resemble these classes:

```
import SwiftData

@Model BucketListActivity { ... }

@Model class Trip {
    @Attribute(.preserveValueOnDeletion)
    var name: String
    var destination: String

    @Attribute(.preserveValueOnDeletion)
    var startDate: Date

    @Attribute(.preserveValueOnDeletion)
    var endDate: Date

    @Relationship(deleteRule: .cascade, inverse: \BucketListItem.trip)
    var bucketList: [BucketListItem] = [BucketListItem]()

    @Relationship(deleteRule: .cascade, inverse: \LivingAccommodation.trip)
    var livingAccommodation: LivingAccommodation?
}
```

Here, an expanded `Trip` base class no longer uses the Boolean value that previously indicated the kind of trip. The following `BusinessTrip` and `PersonalTrip` subclasses describe additional properties and behaviors that could define these specialized trip types and create a hierarchical relationship between the parent and the subclasses.

```
@available(iOS 26, *)
@Model class BusinessTrip: Trip {
    var purpose: String
    var itinerary: MeetingItinerary
    var expenseCode: String
    var perDiemRate: Double
    var mileageRate: Double

    @Relationship(deleteRule: .cascade, inverse: \DailyMilageRecord.trip)
```

```
    var milesDriven: [DailyMilageRecord]

    @Relationship(deleteRule: .cascade, inverse: \BusinessMeal.trip)
    var businessMeals: [BusinessMeal]

    @Relationship(deleteRule: .cascade, inverse: \ConferenceSession.trip)
    var sessionsAttended: [ConferenceSession]
}
```

The `PersonalTrip` subclass may have a very different set of properties, its design and use case shares very little with a business trip, beside the name, place, optional transportation, and duration, as shown here.

```
@Model Attraction { ... }
@Model FamilyMember { ... }

@available(iOS 26, *)
@Model class PersonalTrip: Trip {
    enum Reason: String, CaseIterable, Codable, Identifiable {
        case family
        case reunion
        case wellness
        case unknown

        var id: Self { self }
    }

    var reason: Reason
    @Relationship(deleteRule: .cascade, inverse: \BucketListActivities.trip)
    var bucketList: [BucketListActivity]
    var attractionsToVisit: [Attraction]
    var familyMembers: [FamilyMember]
}
```

Determine whether inheritance is right for your use case

Inheritance is useful when defining class hierarchies where you need to specialize a model representation that derives from a common base. For example `Trip`, `PersonalTrip`, and

`BusinessTrip` form a natural hierarchy: a business trip and a personal trip extend the basic concept of a "trip" and create, in the language of Object Oriented design, an "IS-A" relationship. A `BusinessTrip` IS-A `Trip`. and a `PersonalTrip` also IS-A `Trip`.

Avoid using inheritance in scenarios where the specialized subclass would center on common properties, such as a trip's name, or starting or ending dates; subclassing at this level of granularity, the class hierarchy would contain many subdomains that only share a single property. In these cases if common properties need some kind of specialized behavior, protocol conformance is a better tool.

Avoid using inheritance if your querying model would depend on fetching all of the model data all the time and then filtering the results, — this is known as a *deep query*. It's possible the specialization (here the difference between personal and business trips) is something that a Boolean type could represent as it did in the initial `Trip` model where a Boolean value differentiated the type of trip. Another method to keep models "flatter," reduce the number of properties, and avoid inheritance is to add an enumeration type that has a value which captures the type of trip and it's value, here the personal trip's reason or the business trip's per diem value instead of an `isBusinessTrip` Boolean property, as shown here:

```
enum Category: Codable {
    case personal(Reason)
    case business(perdiem: Double)
}
```

As your model's and use cases expand, adding more properties may become impractical or difficult to maintain.

Conversely, if your app's query strategy *only* focuses on the specialization — specific properties that define a business trip or a personal trip and never on the properties of the base trip model — known as a *shallow query* — then using two models might be a better approach, even though each individual model might contain some apparent duplication.

Lastly, If the query strategy for your app involves allowing a person to perform elements of both deep and shallow searches, in order find and return information from both a base and the subclasses, then inheritance may be a good fit for your app's model and use case.

Fetch and Query Data

> **Note**
>
> These are abbreviated examples that show only the code relevant to fetches, queries, and predicates.

Combining inheritance and customized predicates it's possible to create any number of customized searching and filtering mechanisms that can select from any of the properties available in your parent or subclasses.

For example, to search for text in both `BusinessTrip` and `PersonalTrip` types, perform a deep search on properties using the base class `Trip`, as this the outline demonstrates:

```
struct TripListView: View {
    @Environment(\.modelContext) private var modelContext
    // All trips, in ascending order, by start date.
    @Query(sort: \Trip.startDate, order: .forward)
    var trips: [Trip]

    init(searchText: String) {
        let searchPredicate = #Predicate<Trip> {
            searchText.isEmpty ? true : $0.name.localizedStandardContains(searchTex
        }
        // Filter the trips array using predicate, which searches the trip name and
        _trips = Query(filter: searchPredicate, sort: \.startDate, order: .forward)
    }

    var body: some View {
        /* View body contents */
    }
}
```

Selecting trips based on type, or all trips is a similar predicate: this example uses a simple enumeration and a switch to indicate which kind of trips to select.

```
enum TripKind: String, CaseIterable {
    case all = "All"
    case personal = "Personal"
    case business = "Business"
}


struct TripListView: View {
        @Environment(\.modelContext) private var modelContext
        /// All trips, in ascending order, by start date.
        @Query(sort: \Trip.startDate, order: .forward)
        var trips: [Trip]
```

```
        init(tripKind: Binding<ContentView.TripKind>) {
            // Create a predicate, selected by the provided enumeration
            // case, that examines the object's class to determine if it's
            // a `BusinessTrip`, `PersonalTrip`, or `Trip`.
            let classPredicate: Predicate<Trip>? = {
                switch tripKind.wrappedValue {
                // Returns a `nil` predicate representing all trips.
                case .all:
                    return nil
                // Returns a predicate that matches on `PersonalTrip` objects.
                case .personal:
                    return #Predicate { $0 is PersonalTrip }
                // Returns a predicate that matches on `BusinessTrip` objects.
                case .business:
                    return #Predicate { $0 is BusinessTrip }
                }
            }()


           // Filter the trips array using predicate, which matches on trips of a sp
           _trips = Query(filter: classPredicate, sort: \.startDate, order: .forward
        }

        var body: some View {
            // View body contents that displays the matching trips.
        }
 }
```

The following example demonstrates how to combine both of these predicates to search for text in the trip's' `name` and `destination` properties, as well as by trips or by type of trip, if specified:

```
enum TripKind: String, CaseIterable {
    case all = "All"
    case personal = "Personal"
    case business = "Business"
}


struct TripListView: View {
    @Environment(\.modelContext) private var modelContext
    /// All trips, ordered by start date.
```

```
    @Query(sort: \Trip.startDate, order: .forward)
    var trips: [Trip]

    init(searchString: String, tripKind: Binding<ContentView.TripKind>) {
        // Create a predicate that examines the object's class to determine
        // if it's a `BusinessType`, `Personal`, or `Trip` ("all trips").
        let classPredicate: Predicate<Trip>? = {
            switch tripKind.wrappedValue {
            // Returns a `nil` predicate representing all trips.
            case .all:
                return nil
            // Returns a predicate that matches on `PersonalTrip` objects.
            case .personal:
                return #Predicate { $0 is PersonalTrip }
            // Returns a predicate that matches on `BusinessTrip` objects.
            case .business:
                return #Predicate { $0 is BusinessTrip }
            }
        }()

        // If there's search text, create a predicate than can search the trip's nam
        let searchPredicate = #Predicate<Trip> {
            searchText.isEmpty ? true : $0.name.localizedStandardContains(searchTex
        }

        let fullPredicate: Predicate<Trip>
        if let classPredicate {
            fullPredicate = #Predicate { classPredicate.evaluate($0) && searchPredic
        } else {
            fullPredicate = searchPredicate
        }
        // Filter trips on other `searchText` or trip type, or both.
        _trips = Query(filter: fullPredicate, sort: \.startDate, order: .forward)
    }

    var body: some View {
        // View body contents that displays the matching trips.
    }
}
```

See Also

Essentials

- Preserving your app's model data across launches

  Describe your model classes to SwiftData using the framework's macros, and store instances of those models so they exist beyond the app's runtime.

- Adding and editing persistent data in your app

  Create a data entry form for collecting and changing data managed by SwiftData.

- Adopting SwiftData for a Core Data app

  Persist data in your app intuitively with the Swift native persistence framework.

- SwiftData updates

  Learn about important changes to SwiftData.