Class

NSManagedObject

The base class that all Core Data model objects inherit from.

iOS 3.0+ | iPadOS 3.0+ | Mac Catalyst 13.1+ | macOS 10.4+ | tvOS | visionOS 1.0+ | watchOS 2.0+

```
nonisolated
class NSManagedObject
```

Mentioned in

- Syncing a Core Data Store with CloudKit
- Generating code
- Modeling data
- Using Core Data in the background

Overview

A managed object has an associated entity description (`NSEntityDescription`) that provides metadata about the object, including the name of the entity that the object represents and the names of its attributes and relationships. A managed object also has an associated managed object context that tracks changes to the object graph.

You can't use instances of direct subclasses of `NSObject`, or any other class that doesn't inherit from `NSManagedObject`, with a managed object context. You may create custom subclasses of `NSManagedObject`, although this isn't always necessary. If you don't need custom logic, you can create a complete object graph with `NSManagedObject` instances.

If you instantiate a managed object directly, you must call the designated initializer `init(entity:insertInto:)`.

Data Storage

In some respects, an `NSManagedObject` acts like a dictionary—it's a generic container object that provides efficient storage for the properties defined by its associated `NSEntityDescription` instance. `NSManagedObject` supports a range of common types for attribute values, including string, date, and number (see `NSAttributeDescription` for full details). Therefore, typically you don't need to define instance variables in subclasses. Sometimes, however, you want to use types that aren't supported directly, such as colors and C structures. For example, in a graphics application you might want to define a Rectangle entity that has color and bounds attributes that are an instance of `NSColor` and an `NSRect` struct, respectively. For some types you can use a transformable attribute, for others this may require you to create a subclass of `NSManagedObject`.

> Note
>
> The default value for `automaticallyNotifiesObservers(forKey:)` is `false` for managed properties of a `NSManagedObject`, and `true` for unmanaged properties.

Faulting

Managed objects typically represent data held in a persistent store. In some situations a managed object may be a *fault*—an object whose property values haven't yet been loaded from the external data store. When you access persistent property values, the fault "fires" and the data is retrieved from the store automatically. This can be a comparatively expensive process (potentially requiring a round trip to the persistent store), and you may wish to avoid unnecessarily firing a fault. See Faulting and Uniquing for more details on faults.

You can safely invoke the following methods and properties on a fault without causing it to fire: `isEqual(_:)`, `hash`, `superclass`, `class`, `self()`, `isProxy()`, `isKind(of:)`, `isMember(of:)`, `conforms(to:)`, `responds(to:)`, `description`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, `faultingState`, and `isFault`. Because `isEqual` and `hash` don't cause a fault to fire, managed objects can typically be placed in collections without firing a fault. Note, however, that invoking key-value coding methods on the collection object might in turn result in an invocation of `valueForKey:` on a managed object, which would fire the fault.

Although the `description` property doesn't cause a fault to fire, if you implement a custom `description` that accesses the object's persistent properties, this does cause a fault to fire. You are strongly discouraged from overriding `description` in this way.

Subclassing Notes

In combination with the entity description in the managed object model, `NSManagedObject` provides a rich set of default behaviors including support for arbitrary properties and value validation. If you decide to subclass `NSManagedObject` to implement custom features, make sure you don't disrupt Core Data's behavior.

Methods and Properties You Must Not Override

`NSManagedObject` itself customizes many features of `NSObject` so that managed objects can be properly integrated into the Core Data infrastructure. Core Data relies on the `NSManagedObject` implementation of the following methods and properties, which you therefore absolutely must not override: `primitiveValue(forKey:)`, `setPrimitiveValue(_:forKey:)`, `isEqual(_:)`, `hash`, `superclass`, `class`, `self()`, `isProxy()`, `isKind(of:)`, `isMember(of:)`, `conforms(to:)`, `responds(to:)`, `managedObjectContext`, `entity`, `objectID`, `isInserted`, `isUpdated`, `isDeleted`, and `isFault`, `alloc`, `allocWithZone:`, `new`, `instancesRespond(to:)`, `instanceMethod(for:)`, `method(for:)`, `methodSignatureForSelector:`, `instanceMethodSignatureForSelector:`, or `isSubclass(of:)`.

Methods and Properties You Shouldn't Override

As with any class, you are strongly discouraged from overriding the key-value observing methods such as `willChangeValue(forKey:)` and `didChangeValue(forKey:withSetMutation:using:)`. Avoid overriding `description`—if this method fires a fault during a debugging operation, the results may be unpredictable. Also avoid overriding `init(entity:insertInto:)`, or `dealloc`. Changing values in the `init(entity:insertInto:)` method won't be noticed by the context, and if you aren't careful, those changes may not be saved. Perform most initialization customization in one of the `awake…` methods. If you do override `init(entity:insertInto:)`, make sure you adhere to the requirements set out in the method description. See `init(entity:insertInto:)`.

Don't override `dealloc` because `didTurnIntoFault()` is usually a better time to clear values —a managed object may not be reclaimed for some time after it has been turned into a fault. Core Data doesn't guarantee that `dealloc` will be called in all scenarios (such as when the application quits). Therefore, don't include required side effects (like saving or changes to the file system, user preferences, and so on) in these methods.

In summary, for `init(entity:insertInto:)` and `dealloc`, Core Data reserves exclusive control over the life cycle of the managed object (that is, raw memory management). This is so that the framework can provide features such as uniquing and by consequence, relationship maintenance, as well as much better performance than would be possible otherwise.

Additional Override Considerations

The following methods are intended to be fine grained and aren't suitable for large-scale operations. Don't fetch or save in these methods. In particular, they shouldn't have side effects on the managed object context.

- `init(entity:insertInto:)`
- `didTurnIntoFault()`
- `willTurnIntoFault()`
- `dealloc`

In addition, if you plan to override `awakeFromInsert`, `awakeFromFetch`, and validation methods, first invoke `super.method()`, the superclass's implementation. Don't modify relationships in `awakeFromFetch()`—see the method description for details.

Custom Accessor Methods

Typically, you don't need to write custom accessor methods for properties that are defined in the entity of a managed object's corresponding managed object model. If you need to do so, follow the implementation patterns described in Managed Object Accessor Methods in Core Data Programming Guide.

Core Data automatically generates accessor methods (and primitive accessor methods) for you. For attributes and to-one relationships, Core Data generates the standard get and set accessor methods; for to-many relationships, Core Data generates the indexed accessor methods as described in Achieving Basic Key-Value Coding Compliance in Key-Value Coding Programming Guide. You do however need to declare the accessor methods or use Objective-C properties to suppress compiler warnings. For a full discussion, see Managed Object Accessor Methods in Core Data Programming Guide.

Custom Instance Variables

By default, `NSManagedObject` stores its properties in an internal structure as objects, and in general Core Data is more efficient working with storage under its own control rather than by using custom instance variables.

`NSManagedObject` provides support for a range of common types for attribute values, including string, date, and number (see `NSAttributeDescription` for full details). If you want to use types that aren't supported directly, like colors and C structures, you can either use transformable attributes or create a subclass of `NSManagedObject`.

Sometimes it's convenient to represent variables as scalars—in drawing applications, for example, where variables represent dimensions and x and y coordinates and are frequently used in calculations. To represent attributes as scalars, you declare instance variables as you do in any other class. You also need to implement suitable accessor methods as described in Managed Object Accessor Methods.

If you define custom instance variables for example to store derived attributes or other transient properties, clean up these variables in `didTurnIntoFault()` rather than `dealloc`.

Validation Methods

`NSManagedObject` provides consistent hooks for validating property and inter-property values. You typically shouldn't override `validateValue(_:forKey:)`. Instead implement methods of the form `validate<Key>:error:`, as defined by the NSKeyValueCoding protocol. If you want to validate inter-property values, you can override `validateForUpdate()` and/or related validation methods.

Don't call `validateValue:forKey:error:` within custom property validation methods—if you do, you create an infinite loop when `validateValue:forKey:error:` is invoked at runtime. If you do implement custom validation methods, don't call them directly. Instead, call `validateValue:forKey:error:` with the appropriate key. This ensures that any constraints defined in the managed object model are applied.

If you implement custom inter-property validation methods like `validateForUpdate()`, call the superclass's implementation first. This ensures that individual property validation methods are also invoked. If there are multiple validation failures in one operation, collect them in an array and add the array—using the key `NSDetailedErrorsKey`—to the userInfo dictionary in the `NSError` object you return. For an example, see Managed Object Validation.

Topics

Creating a Managed Object

`init(entity: NSEntityDescription, insertInto: NSManagedObjectContext?)`

Initializes a managed object from an entity description and inserts it into the specified managed object context.

`convenience init(context: NSManagedObjectContext)`

Initializes a managed object subclass and inserts it into the specified managed object context.

Getting a Managed Object's Identity

`var entity: NSEntityDescription`

The entity description of the managed object.

`var objectID: NSManagedObjectID`

The object ID of the managed object.

`class func entity() -> NSEntityDescription`

Returns the entity description that is associated with this subclass.

Getting State Information

`var managedObjectContext: NSManagedObjectContext?`

The managed object context with which the managed object is registered.

`var hasChanges: Bool`

A Boolean value that indicates whether the managed object has been inserted, has been deleted, or has unsaved changes.

`var isInserted: Bool`

A Boolean value that indicates whether the managed object has been inserted in a managed object context.

`var isUpdated: Bool`

A Boolean value that indicates whether the managed object has unsaved changes.

`var isDeleted: Bool`

A Boolean value that indicates whether the managed object will be deleted during the next save.

`var isFault: Bool`

A Boolean value that indicates whether the managed object is a fault.

`var faultingState: Int`

The faulting state of the managed object.

`func hasFault(forRelationshipNamed: String) -> Bool`

Returns a Boolean value that indicates whether the relationship for a given key is a fault.

`var hasPersistentChangedValues: Bool`

A Boolean value that indicates whether the managed object has persistent changes.

Managing Change Events

`class var contextShouldIgnoreUnmodeledPropertyChanges: Bool`

A Boolean value that indicates whether to mark instances of the class as having changes when an unmodeled property changes.

`func awakeFromFetch()`

Provides an opportunity to add code into the life cycle of the managed object when fufilling it from a fault.

`func awakeFromInsert()`

Provides an opportunity to add code into the life cycle of the managed object when initially creating it.

`func awake(fromSnapshotEvents: NSSnapshotEventType)`

Provides an opportunity to add code into the life cycle of the managed object when fulfilling it from a snapshot.

`func changedValues() -> [String : Any]`

Returns a dictionary containing the keys and new values of persistent properties with changes since the last fetching or saving of the managed object.

`func changedValuesForCurrentEvent() -> [String : Any]`

Returns a dictionary containing the keys and new values of persistent properties with changes since the last fetching or saving of the managed object.

`func committedValues(forKeys: [String]?) -> [String : Any]`

Returns a dictionary of the most recent fetched or saved values of the managed object for the properties of the specified keys.

`func prepareForDeletion()`

Provides an opportunity to add code into the life cycle of the managed object before deleting it.

`func willSave()`

Provides an opportunity to add code into the life cycle of the managed object before saving it.

`func didSave()`

Provides an opportunity to add code into the life cycle of the managed object after the managed object's context completes a save operation.

`func willTurnIntoFault()`

Provides an opportunity to add code into the life cycle of the managed object before converting it to a fault.

`func didTurnIntoFault()`

Provides an opportunity to add code into the life cycle of the managed object after converting it to a fault.

```
class func fetchRequest() -> NSFetchRequest<any NSFetchRequestResult>
```

Returns an initialized fetch request with the entity this subclass represents.

Supporting Key-Value Coding

```
func value(forKey: String) -> Any?
```

Returns the value for the property specified by `key`.

```
func setValue(Any?, forKey: String)
```

Sets the specified property of the managed object to the specified value.

```
func primitiveValue(forKey: String) -> Any?
```

Returns the value for the specified property from the managed object’s private internal storage .

```
func setPrimitiveValue(Any?, forKey: String)
```

Sets the value of a given property in the managed object’s private internal storage.

```
func objectIDs(forRelationshipNamed: String) -> [NSManagedObjectID]
```

Returns the object IDs for all of the managed objects that are in the named relationship.

Managing Data Validation

```
func validateValue(AutoreleasingUnsafeMutablePointer<AnyObject?>, for
Key: String) throws
```

Validates a property value for a given key.

```
func validateForDelete() throws
```

Determines whether the managed object can be deleted in its current state.

```
func validateForInsert() throws
```

Determines whether the managed object can be inserted in its current state.

```
func validateForUpdate() throws
```

Determines whether the managed object’s current state is valid.

Validation error codes

Error codes relating to the validation of managed objects.

`let NSValidationKeyErrorKey: String`

The error key for the attribute that failed to validate.

`let NSValidationObjectErrorKey: String`

The error key for the object that failed to validate.

`let NSValidationPredicateErrorKey: String`

The error key for the predicate that failed to validate.

`let NSValidationValueErrorKey: String`

The error key for the value that failed to validate.

Supporting Key-Value Observing

`func didAccessValue(forKey: String?)`

Provides support for key-value observing access notification.

`func observationInfo() -> UnsafeMutableRawPointer?`

Returns the observation info of the managed object.

`func setObservationInfo(UnsafeMutableRawPointer?)`

Sets the observation info of the managed object.

`func willAccessValue(forKey: String?)`

Provides support for key-value observing access notification.

`func didChangeValue(forKey: String)`

Provides an opportunity to respond when a value of a given property has changed.

`func didChangeValue(forKey: String, withSetMutation: NSKeyValueSetMutationKind, using: Set<AnyHashable>)`

Provides an opportunity to respond when a change was made to a specified to-many relationship.

`func willChangeValue(forKey: String)`

Provides an opportunity to respond when a value of a given property is about to change.

`func willChangeValue(forKey: String, withSetMutation: NSKeyValueSetMutationKind, using: Set<AnyHashable>)`

Provides an opportunity to respond when a change is about to be made to a specified to-many relationship.

Reinitializing Values

`struct NSSnapshotEventType`

Constants that specify the reason the managed object may need to reinitialize its values.

Subscripts

`subscript(String) -> Any?`

Relationships

Inherits From

`NSObject`

Conforms To

`CVarArg`
`Copyable`
`CustomDebugStringConvertible`
`CustomStringConvertible`
`Equatable`
`Hashable`
`NSFetchRequestResult`
`NSObjectProtocol`
`ObservableObject`

See Also

Objects and entities

`class NSEntityDescription`

A description of a Core Data entity.