Class

Progress

An object that conveys ongoing progress to the user for a specified task.

iOS 7.0+ | iPadOS 7.0+ | Mac Catalyst 13.1+ | macOS 10.9+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
class Progress
```

Mentioned in

Downloading files from websites

Overview

The `Progress` class provides a self-contained mechanism for progress reporting. It makes it easy for code that performs work to report the progress of that work, and for user interface code to observe that progress for presentation to the user. Specifically, you can use a progress object to show the user a progress bar and explanatory text that update as you do work. It also allows the user to cancel or pause work.

Reporting Progress

Using the methods of this class, your code can report the progress it's currently making toward completing a task, including progress in related subtasks. You can create instances of this class using the `init(parent:userInfo:)` instance method or the `init(totalUnitCount:)` class method.

Progress objects have many properties that you can use to observe and report current progress. For instance, the `totalUnitCount` property represents the total number of units of work, and the `completedUnitCount` and `fractionCompleted` properties represent how much of that

work is complete. The `fractionCompleted` property is useful for updating progress indicators or textual descriptors. To check whether progress is complete, test the `isFinished` property.

The following code shows a sample method that reports the progress of performing an operation on a piece of data. When creating the progress object, you set the value of its `totalUnitCount` property to a suitable batch size for the operation, and the `completedUnitCount` count is `0`. Each time the loop executes and processes a batch of data, you increment the progress object's `completedUnitCount` property appropriately.

```
- (void)startTaskWithData:(NSData *)data {
    NSUInteger batchSize = ... use a suitable batch size
    NSProgress *progress = [NSProgress progressWithTotalUnitCount:batchSize];

    for (NSUInteger index = 0; index < batchSize; index++) {
        // Check for cancellation.
        if ([progress isCancelled]) {
             // Tidy up as necessary.
             break;
        }

        // Do something with this batch of data.

        // Report progress (add 1 because the work is complete for the current index)
        [progress setCompletedUnitCount:(index + 1)];
    }
}
```

Each of the properties of a progress object, including `totalUnitCount`, `completedUnitCount`, and `fractionCompleted`, support key-value observing (KVO). This makes it extremely easy for a view or window controller object to observe the properties, and update UI elements, such as progress indicators, when the values change. It also means that there is a nonzero cost to updating the values of those properties, so avoid using a unit count that is too granular. If you're iterating over a large dataset, for example, and each operation takes only a trivial amount of time, divide the work into batches so you can update the unit count once per batch rather than once per iteration.

Reporting Progress for Multiple Operations

Sometimes, your code may need to report the *overall* progress of an operation that consists of several suboperations. To accomplish this, your code can report the progress of each suboperation by building up a tree of progress objects.

The `Progress` reporting mechanism supports a loosely coupled relationship between progress objects. Suboperations don't need to know anything about the containing progress item — you can create new progress objects as suboperations of another progress instance. When you assign a progress instance, the system allocates a portion of the containing progress instance's pending unit count. When the suboperation's progress object completes, the containing progress object's `completedUnitCount` property automatically increases by a predefined amount.

> **Note**
>
> The `completedUnitCount` property for a containing progress object only updates when the suboperation is 100% complete. The `fractionCompleted` property for a containing progress object updates continuously as work progresses for all suboperations.

You add suboperation progress objects to your tree implicitly or explicitly.

Adding a Progress Operation Implicitly

Add a suboperation implicitly by setting a pending unit count for the containing progress object and creating a new `Progress` instance. When you create the new progress instance, the system sets it as a suboperation of the containing progress object, and assigns the pending unit count.

As an example, consider that you're tracking the progress of code downloading and copying files on disk. You can use a single progress object to track the entire task, but it's easier to manage each subtask using a separate progress object. You start by creating an overall progress object with a suitable total unit count, call `becomeCurrent(withPendingUnitCount:)`, then create your suboperation progress objects before finally calling `resignCurrent()`.

The system divides the pending unit count that you specify in the first method equally among the suboperation progress objects you create between these two method calls. Each suboperation progress object maintains its own internal unit count. When the suboperation object's `completedUnitCount` equals or exceeds its `totalUnitCount`, the system increases the containing progress object's `completedUnitCount` by the assigned portion of the original pending unit count.

In the following example, the overall progress object has 100 units. The two suboperation objects, therefore, get 50 pending units each, and keep track internally of 10 units of work each. When each suboperation completes its 10 units, the system increases the overall progress object's completed unit count by 50.

```
- (void)startLongOperation {
    self.overallProgress = [NSProgress progressWithTotalUnitCount:100];

    [self.overallProgress becomeCurrentWithPendingUnitCount:50];
```

```
    [self work1];
    [self.overallProgress resignCurrent];

    [self.overallProgress becomeCurrentWithPendingUnitCount:50];
    [self work2];
    [self.overallProgress resignCurrent];
}

- (void)work1 {
    NSProgress *firstTaskProgress = [NSProgress progressWithTotalUnitCount:10];
    // Perform first task.
}

- (void)work2 {
    NSProgress *secondTaskProgress = [NSProgress progressWithTotalUnitCount:10];
    // Perform second task.
}
```

If you don't create any suboperation progress objects between the calls to `becomeCurrent(withPendingUnitCount:)` and `resignCurrent()`, the containing progress object automatically updates its `completedUnitCount` by adding the pending units.

Adding a Progress Operation Explicitly

To add a progress operation explicitly, call `addChild(_:withPendingUnitCount:)` on the containing progress object. The value for the pending unit count is the amount of the containing progress object's `totalUnitCount` that the suboperation consumes, which conforms to the `ProgressReporting` protocol.

In the following example, the overall progress object has 10 units. The suboperation progress for the download gets eight units and tracks the download of a photo. The progress for the filter takes a lot less time and gets the remaining two units. When the download completes, the system updates the containing progress object's completed unit count by eight. When the filter completes, the system updates it by the remaining two units.

```
- (void)startLongOperation {
    self.overallProgress = [NSProgress progressWithTotalUnitCount:10];

    [self.overallProgress addChild:download.progress withPendingUnitCount:8];
    // Do the download.

    [self.overallProgress addChild:filter.progress withPendingUnitCount:2];
    // Perform the filter.
```

```
}
```

Topics

Creating Progress Objects

`init(parent: Progress?, userInfo: [ProgressUserInfoKey : Any]?)`

Creates a new progress instance.

`class func discreteProgress(totalUnitCount: Int64) -> Progress`

Creates and returns a progress instance with the specified unit count that isn't part of any existing progress tree.

`init(totalUnitCount: Int64)`

Creates and returns a progress instance.

`init(totalUnitCount: Int64, parent: Progress, pendingUnitCount: Int64)`

Creates a progress instance for the specified progress object with a unit count that's a portion of the containing object's total unit count.

Accessing the Current Progress Object

`class func current() -> Progress?`

Returns the progress instance, if any.

`func becomeCurrent(withPendingUnitCount: Int64)`

Sets the progress object as the current object of the current thread, and assigns the amount of work for the next suboperation progress object to perform.

`func addChild(Progress, withPendingUnitCount: Int64)`

Adds a process object as a suboperation of a progress tree.

`func performAsCurrent<ReturnType>(withPendingUnitCount: Int64, using: () throws -> ReturnType) rethrows -> ReturnType`

Retrieves the current thread's progress object, executes the specified block, and increments the progress object by the specified units of work.

`func resignCurrent()`

Restores the previous progress object to become the current progress object on the thread.

Reporting Progress

`var totalUnitCount: Int64`

The total number of tracked units of work for the current progress.

`var completedUnitCount: Int64`

The number of completed units of work for the current job.

`var localizedDescription: String!`

A localized description of tracked progress for the receiver.

`var localizedAdditionalDescription: String!`

A more specific localized description of tracked progress for the receiver.

`var isCancellable: Bool`

A Boolean value that indicates whether the receiver is tracking work that you can cancel.

`var isCancelled: Bool`

A Boolean value that Indicates whether the receiver is tracking canceled work.

`var cancellationHandler: (() -> Void)?`

The block to invoke when canceling progress.

`var isPausable: Bool`

A Boolean value that indicates whether the receiver is tracking work that you can pause.

`var isPaused: Bool`

A Boolean value that indicates whether the receiver is tracking paused work.

`var pausingHandler: (() -> Void)?`

The block to invoke when pausing progress.

Observing Progress

`var isIndeterminate: Bool`

A Boolean value that indicates whether the tracked progress is indeterminate.

`var fractionCompleted: Double`

The fraction of the overall work that the progress object completes, including work from its suboperations.

`var isFinished: Bool`

A Boolean value that indicates the progress object is complete.

Controlling Progress

`func cancel()`

Cancels progress tracking.

`func pause()`

Pauses progress tracking.

`func resume()`

Resumes progress tracking.

`var resumingHandler: (() -> Void)?`

The block to invoke when progress resumes.

Inspecting Progress Information

`var kind: ProgressKind?`

An object that represents the kind of progress for the progress object.

`var estimatedTimeRemaining: TimeInterval?`

A value that indicates the estimated amount of time remaining to complete the progress.

`var throughput: Int?`

A value that represents the speed of data processing, in bytes per second.

`func setUserInfoObject(Any?, forKey: ProgressUserInfoKey)`

Sets a value in the user info dictionary.

`var userInfo: [ProgressUserInfoKey : Any]`

A dictionary of arbitrary values for the receiver.

`struct ProgressKind`

An object that represents the kind of progress.

`struct ProgressUserInfoKey`

Keys for the user info dictionary that affect the autogenerated localized additional description string.

Inspecting File Operation Progress Information

`var fileOperationKind: Progress.FileOperationKind?`

The kind of file operation for the progress object.

`var fileURL: URL?`

A URL that represents the file for the current progress object.

`var fileTotalCount: Int?`

The total number of files for a file progress object.

`var fileCompletedCount: Int?`

The number of completed files for a file progress object.

`struct FileOperationKind`

The kind of file operation.

Reporting Progress to Other Processes

`func publish()`

Publishes the progress object for other processes to observe it.

`func unpublish()`

Removes a progress object from publication, making it unobservable by other processes.

Observing and Controlling File Progress by Other Processes

`class func addSubscriber(forFileURL: URL, withPublishingHandler: Progress.PublishingHandler) -> Any`

Registers a file URL to hear about the progress of a file operation.

`class func removeSubscriber(Any)`

Removes a proxy progress object that the add subscriber method returns.

`var isOld: Bool`

A Boolean value that indicates when the observed progress object invokes the publish method before you subscribe to it.

`typealias PublishingHandler`

A block that the system calls when an observed progress object matches the subscription.

`typealias` UnpublishingHandler

A block that the system calls when an observed progress object terminates the subscription.

Relationships

Inherits From

NSObject

Conforms To

CVarArg
CustomDebugStringConvertible
CustomStringConvertible
Equatable
Hashable
NSObjectProtocol
Sendable
SendableMetatype

See Also

Progress

`protocol` ProgressReporting

An interface for objects that report progress using a single progress instance.