Property Observers

Property observers observe and respond to changes in a property's value.

Property observers are called every time a property's value is set, even if the new value is the same as the property's current value.

You can add property observers in the following places:

- Stored properties that you define
- Stored properties that you inherit
- Computed properties that you inherit

For an inherited property, you add a property observer by overriding that property in a subclass. For a computed property that you define, use the property's setter to observe and respond to value changes, instead of trying to create an observer.

Overriding properties is described in Overriding.

You have the option to define either or both of these observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

If you implement a `willSet` observer, it's passed the new property value as a constant parameter. You can specify a name for this parameter as part of your `willSet` implementation. If you don't write the parameter name and parentheses within your implementation, the parameter is made available with a default parameter name of `newValue`.

Similarly, if you implement a `didSet` observer, it's passed a constant parameter containing the old property value. You can name the parameter or use the default parameter name of `oldValue`. If you assign a value to a property within its own `didSet` observer, the new value that you assign replaces the one that was just set.

Note:

The `willSet` and `didSet` observers of superclass properties are called when a property is set in a subclass initializer, after the superclass initializer has been called. They aren't called while a class is setting its own properties, before the superclass initializer has been called.

For more information about initializer delegation, see Initializer Delegation for Value Types and Initializer Delegation for Class Types.

Here's an example of `willSet` and `didSet` in action. The example below defines a new class called `StepCounter`, which tracks the total number of steps that a person takes while walking. This class might be used with input data from a pedometer or other step counter to keep track of a person's exercise during their daily routine:

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet (newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter ()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

The `StepCounter` class declares a `totalSteps` property of type `Int`. This is a stored property with `willSet` and `didSet` observers.
The `willSet` and `didSet` observers for `totalSteps` are called whenever the property is assigned a new value. This is true even if the new value is the same as the current value.
This example's `willSet` observer uses a custom parameter name of `newTotalSteps` for the upcoming new value. In this example, it simply prints out the value that's about to be set.
The `didSet` observer is called after the value of `totalSteps` is updated. It compares the new value of `totalSteps` against the old value. If the total number of steps has increased, a message is printed to indicate how many new steps have been taken. The `didSet` observer doesn't provide a custom parameter name for the old value, and the default name of `oldValue` is used instead.
Note

If you pass a property that has observers to a function as an in-out parameter, the `willSet` and `didSet` observers are always called. This is because of the copy-in copy-out memory model for in-out parameters: The value is always written back to the property at the end of the function. For a detailed discussion of the behavior of in-out parameters, see In-Out Parameters.