Properties | Documentation

Access stored and computed values that are part of an instance or type.

Properties associate values with a particular class, structure, or enumeration. Stored properties store constant and variable values as part of an instance, whereas computed properties calculate (rather than store) a value. Computed properties are provided by classes, structures, and enumerations. Stored properties are provided only by classes and structures.

Stored and computed properties are usually associated with instances of a particular type. However, properties can also be associated with the type itself. Such properties are known as type properties.

In addition, you can define property observers to monitor changes in a property's value, which you can respond to with custom actions. Property observers can be added to stored properties you define yourself, and also to properties that a subclass inherits from its superclass.

You can also use a property wrapper to reuse code in the getter and setter of multiple properties.

Stored Properties

In its simplest form, a stored property is a constant or variable that's stored as part of an instance of a particular class or structure. Stored properties can be either *variable stored properties* (introduced by the `var` keyword) or *constant stored properties* (introduced by the `let` keyword).

You can provide a default value for a stored property as part of its definition, as described in Default Property Values. You can also set and modify the initial value for a stored property during initialization. This is true even for constant stored properties, as described in Assigning Constant Properties During Initialization.

The example below defines a structure called `FixedLengthRange`, which describes a range of integers whose range length can't be changed after it's created:

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
```

```
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Instances of `FixedLengthRange` have a variable stored property called `firstValue` and a constant stored property called `length`. In the example above, `length` is initialized when the new range is created and can't be changed thereafter, because it's a constant property.

Stored Properties of Constant Structure Instances

If you create an instance of a structure and assign that instance to a constant, you can't modify the instance's properties, even if they were declared as variable properties:

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

Because `rangeOfFourItems` is declared as a constant (with the `let` keyword), it isn't possible to change its `firstValue` property, even though `firstValue` is a variable property.

This behavior is due to structures being *value types*. When an instance of a value type is marked as a constant, so are all of its properties.

The same isn't true for classes, which are *reference types*. If you assign an instance of a reference type to a constant, you can still change that instance's variable properties.

Lazy Stored Properties

A *lazy stored property* is a property whose initial value isn't calculated until the first time it's used. You indicate a lazy stored property by writing the `lazy` modifier before its declaration.

Lazy properties are useful when the initial value for a property is dependent on outside factors whose values aren't known until after an instance's initialization is complete. Lazy properties are also useful when the initial value for a property requires complex or computationally expensive setup that shouldn't be performed unless or until it's needed.

The example below uses a lazy stored property to avoid unnecessary initialization of a complex class. This example defines two classes called `DataImporter` and `DataManager`, neither of which is shown in full:

```
class DataImporter {
    /*
    DataImporter is a class to import data from an external file.
    The class is assumed to take a nontrivial amount of time to initialize.
    */
    var filename = "data.txt"
    // the DataImporter class would provide data importing functionality here
}

class DataManager {
    lazy var importer = DataImporter()
    var data: [String] = []
    // the DataManager class would provide data management functionality here
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// the DataImporter instance for the importer property hasn't yet been create
d
```

The `DataManager` class has a stored property called `data`, which is initialized with a new, empty array of `String` values. Although the rest of its functionality isn't shown, the purpose of this `DataManager` class is to manage and provide access to this array of `String` data.

Part of the functionality of the `DataManager` class is the ability to import data from a file. This functionality is provided by the `DataImporter` class, which is assumed to take a nontrivial amount of time to initialize. This might be because a `DataImporter` instance needs to open a file and read its contents into memory when the `DataImporter` instance is initialized.

Because it's possible for a `DataManager` instance to manage its data without ever importing data from a file, `DataManager` doesn't create a new `DataImporter` instance when the `DataManager` itself is created. Instead, it makes more sense to create the `DataImporter` instance if and when it's first used.

Because it's marked with the `lazy` modifier, the `DataImporter` instance for the `importer` property is only created when the `importer` property is first accessed, such as when its `filename` property is queried:

```
print(manager.importer.filename)
// the DataImporter instance for the importer property has now been created
// Prints "data.txt"
```

Stored Properties and Instance Variables

If you have experience with Objective-C, you may know that it provides *two* ways to store values and references as part of a class instance. In addition to properties, you can use instance variables as a backing store for the values stored in a property.

Swift unifies these concepts into a single property declaration. A Swift property doesn't have a corresponding instance variable, and the backing store for a property isn't accessed directly. This approach avoids confusion about how the value is accessed in different contexts and simplifies the property's declaration into a single, definitive statement. All information about the property — including its name, type, and memory management characteristics — is defined in a single location as part of the type's definition.

Computed Properties

In addition to stored properties, classes, structures, and enumerations can define *computed properties*, which don't actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
// initialSquareCenter is at (5.0, 5.0)
```

```
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// Prints "square.origin is now at (10.0, 10.0)"
```

This example defines three structures for working with geometric shapes:

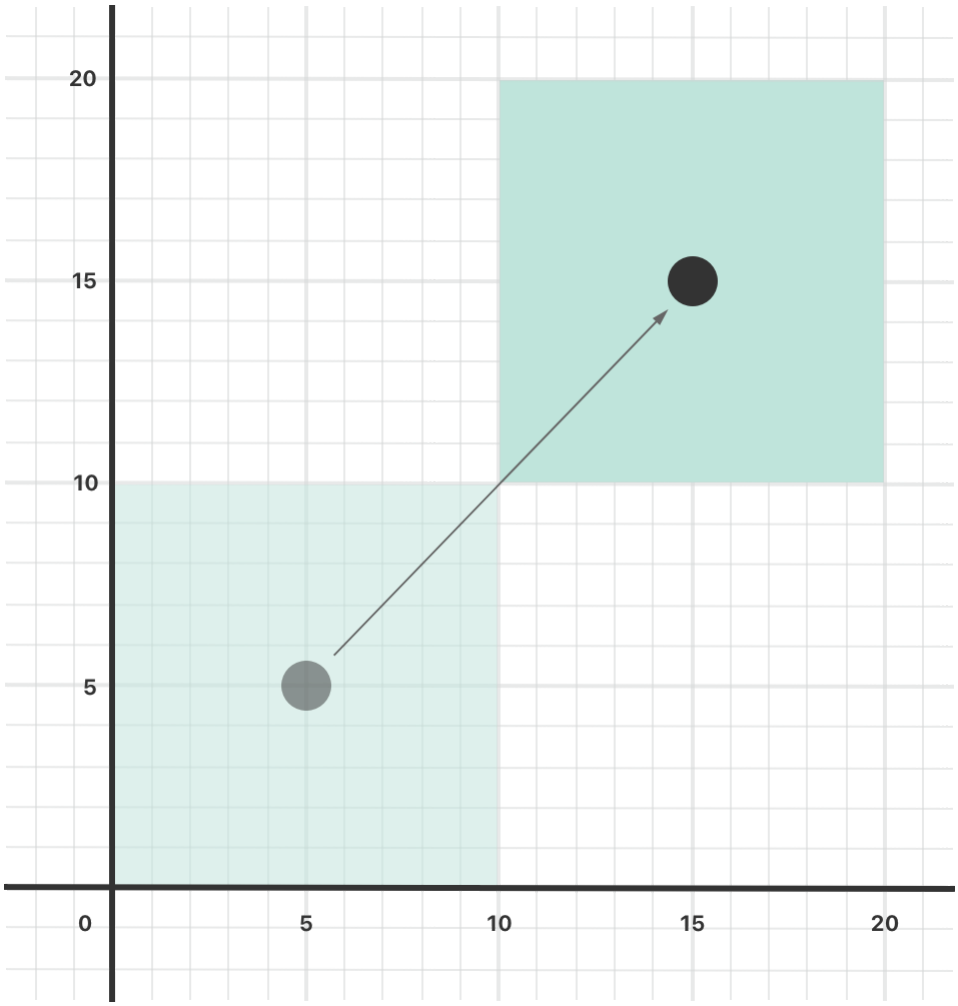
- `Point` encapsulates the x- and y-coordinate of a point.
- `Size` encapsulates a `width` and a `height`.
- `Rect` defines a rectangle by an origin point and a size.

The `Rect` structure also provides a computed property called `center`. The current center position of a `Rect` can always be determined from its `origin` and `size`, and so you don't need to store the center point as an explicit `Point` value. Instead, `Rect` defines a custom getter and setter for a computed variable called `center`, to enable you to work with the rectangle's `center` as if it were a real stored property.

The example above creates a new `Rect` variable called `square`. The `square` variable is initialized with an origin point of `(0, 0)`, and a width and height of `10`. This square is represented by the light green square in the diagram below.

The `square` variable's `center` property is then accessed through dot syntax (`square.center`), which causes the getter for `center` to be called, to retrieve the current property value. Rather than returning an existing value, the getter actually calculates and returns a new `Point` to represent the center of the square. As can be seen above, the getter correctly returns a center point of `(5, 5)`.

The `center` property is then set to a new value of `(15, 15)`, which moves the square up and to the right, to the new position shown by the dark green square in the diagram below. Setting the `center` property calls the setter for `center`, which modifies the `x` and `y` values of the stored `origin` property, and moves the square to its new position.



Shorthand Setter Declaration

If a computed property's setter doesn't define a name for the new value to be set, a default name of `newValue` is used. Here's an alternative version of the `Rect` structure that takes advantage of this shorthand notation:

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
```

```
    }
}
```

Shorthand Getter Declaration

If the entire body of a getter is a single expression, the getter implicitly returns that expression. Here's an another version of the `Rect` structure that takes advantage of this shorthand notation and the shorthand notation for setters:

```
struct CompactRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            Point(x: origin.x + (size.width / 2),
                  y: origin.y + (size.height / 2))
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Omitting the `return` from a getter follows the same rules as omitting `return` from a function, as described in Functions With an Implicit Return.

Read-Only Computed Properties

A computed property with a getter but no setter is known as a *read-only computed property*. A read-only computed property always returns a value, and can be accessed through dot syntax, but can't be set to a different value.

You can simplify the declaration of a read-only computed property by removing the `get` keyword and its braces:

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
```

```
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// Prints "the volume of fourByFiveByTwo is 40.0"
```

This example defines a new structure called `Cuboid`, which represents a 3D rectangular box with `width`, `height`, and `depth` properties. This structure also has a read-only computed property called `volume`, which calculates and returns the current volume of the cuboid. It doesn't make sense for `volume` to be settable, because it would be ambiguous as to which values of `width`, `height`, and `depth` should be used for a particular `volume` value. Nonetheless, it's useful for a `Cuboid` to provide a read-only computed property to enable external users to discover its current calculated volume.

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.

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.

Property Wrappers

A property wrapper adds a layer of separation between code that manages how a property is stored and the code that defines a property. For example, if you have properties that provide thread-safety checks or store their underlying data in a database, you have to write that code on every property. When you use a property wrapper, you write the management code once when you define the wrapper, and then reuse that management code by applying it to multiple properties.

To define a property wrapper, you make a structure, enumeration, or class that defines a `wrappedValue` property. In the code below, the `TwelveOrLess` structure ensures that the value it wraps always contains a number less than or equal to 12. If you ask it to store a larger number, it stores 12 instead.

```
@propertyWrapper
struct TwelveOrLess {
    private var number = 0
    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, 12) }
    }
}
```

The setter ensures that new values are less than or equal to 12, and the getter returns the stored value.

You apply a wrapper to a property by writing the wrapper's name before the property as an attribute. Here's a structure that stores a rectangle that uses the `TwelveOrLess` property wrapper to ensure its dimensions are always 12 or less:

```
struct SmallRectangle {
    @TwelveOrLess var height: Int
    @TwelveOrLess var width: Int
}

var rectangle = SmallRectangle()
print(rectangle.height)
// Prints "0"

rectangle.height = 10
print(rectangle.height)
```

```
// Prints "10"

rectangle.height = 24
print(rectangle.height)
// Prints "12"
```

The `height` and `width` properties get their initial values from the definition of `TwelveOrLess`, which sets `TwelveOrLess.number` to zero. The setter in `TwelveOrLess` treats 10 as a valid value so storing the number 10 in `rectangle.height` proceeds as written. However, 24 is larger than `TwelveOrLess` allows, so trying to store 24 end up setting `rectangle.height` to 12 instead, the largest allowed value.

When you apply a wrapper to a property, the compiler synthesizes code that provides storage for the wrapper and code that provides access to the property through the wrapper. (The property wrapper is responsible for storing the wrapped value, so there's no synthesized code for that.) You could write code that uses the behavior of a property wrapper, without taking advantage of the special attribute syntax. For example, here's a version of `SmallRectangle` from the previous code listing that wraps its properties in the `TwelveOrLess` structure explicitly, instead of writing `@TwelveOrLess` as an attribute:

```
struct SmallRectangle {
    private var _height = TwelveOrLess()
    private var _width = TwelveOrLess()
    var height: Int {
        get { return _height.wrappedValue }
        set { _height.wrappedValue = newValue }
    }
    var width: Int {
        get { return _width.wrappedValue }
        set { _width.wrappedValue = newValue }
    }
}
```

The `_height` and `_width` properties store an instance of the property wrapper, `TwelveOrLess`. The getter and setter for `height` and `width` wrap access to the `wrappedValue` property.

Setting Initial Values for Wrapped Properties

The code in the examples above sets the initial value for the wrapped property by giving `number` an initial value in the definition of `TwelveOrLess`. Code that uses this property wrapper can't specify a different initial value for a property that's wrapped by `TwelveOrLess` — for example, the definition of `SmallRectangle` can't give `height` or `width` initial values. To support setting an initial value or

other customization, the property wrapper needs to add an initializer. Here's an expanded version of `TwelveOrLess` called `SmallNumber` that defines initializers that set the wrapped and maximum value:

```
@propertyWrapper
struct SmallNumber {
    private var maximum: Int
    private var number: Int

    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, maximum) }
    }

    init() {
        maximum = 12
        number = 0
    }
    init(wrappedValue: Int) {
        maximum = 12
        number = min(wrappedValue, maximum)
    }
    init(wrappedValue: Int, maximum: Int) {
        self.maximum = maximum
        number = min(wrappedValue, maximum)
    }
}
```

The definition of `SmallNumber` includes three initializers — `init()`, `init(wrappedValue:)`, and `init(wrappedValue:maximum:)` — which the examples below use to set the wrapped value and the maximum value. For information about initialization and initializer syntax, see Initialization.

When you apply a wrapper to a property and you don't specify an initial value, Swift uses the `init()` initializer to set up the wrapper. For example:

```
struct ZeroRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int
}

var zeroRectangle = ZeroRectangle()
print(zeroRectangle.height, zeroRectangle.width)
// Prints "0 0"
```

The instances of `SmallNumber` that wrap `height` and `width` are created by calling `SmallNumber()`. The code inside that initializer sets the initial wrapped value and the initial maximum value, using the default values of zero and 12. The property wrapper still provides all of the initial values, like the earlier example that used `TwelveOrLess` in `SmallRectangle`. Unlike that example, `SmallNumber` also supports writing those initial values as part of declaring the property.

When you specify an initial value for the property, Swift uses the `init(wrappedValue:)` initializer to set up the wrapper. For example:

```
struct UnitRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber var width: Int = 1
}

var unitRectangle = UnitRectangle()
print(unitRectangle.height, unitRectangle.width)
// Prints "1 1"
```

When you write `= 1` on a property with a wrapper, that's translated into a call to the `init(wrappedValue:)` initializer. The instances of `SmallNumber` that wrap `height` and `width` are created by calling `SmallNumber(wrappedValue: 1)`. The initializer uses the wrapped value that's specified here, and it uses the default maximum value of 12.

When you write arguments in parentheses after the custom attribute, Swift uses the initializer that accepts those arguments to set up the wrapper. For example, if you provide an initial value and a maximum value, Swift uses the `init(wrappedValue:maximum:)` initializer:

```
struct NarrowRectangle {
    @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
    @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
}

var narrowRectangle = NarrowRectangle()
print(narrowRectangle.height, narrowRectangle.width)
// Prints "2 3"

narrowRectangle.height = 100
narrowRectangle.width = 100
print(narrowRectangle.height, narrowRectangle.width)
// Prints "5 4"
```

The instance of `SmallNumber` that wraps `height` is created by calling `SmallNumber(wrappedValue: 2, maximum: 5)`, and the instance that wraps `width` is created by calling `SmallNumber(wrappedValue: 3, maximum: 4)`.

By including arguments to the property wrapper, you can set up the initial state in the wrapper or pass other options to the wrapper when it's created. This syntax is the most general way to use a property wrapper. You can provide whatever arguments you need to the attribute, and they're passed to the initializer.

When you include property wrapper arguments, you can also specify an initial value using assignment. Swift treats the assignment like a `wrappedValue` argument and uses the initializer that accepts the arguments you include. For example:

```
struct MixedRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber(maximum: 9) var width: Int = 2
}

var mixedRectangle = MixedRectangle()
print(mixedRectangle.height)
// Prints "1"

mixedRectangle.height = 20
print(mixedRectangle.height)
// Prints "12"
```

The instance of `SmallNumber` that wraps `height` is created by calling `SmallNumber(wrappedValue: 1)`, which uses the default maximum value of 12. The instance that wraps `width` is created by calling `SmallNumber(wrappedValue: 2, maximum: 9)`.

Projecting a Value From a Property Wrapper

In addition to the wrapped value, a property wrapper can expose additional functionality by defining a *projected value* — for example, a property wrapper that manages access to a database can expose a `flushDatabaseConnection()` method on its projected value. The name of the projected value is the same as the wrapped value, except it begins with a dollar sign (`$`). Because your code can't define properties that start with `$` the projected value never interferes with properties you define.

In the `SmallNumber` example above, if you try to set the property to a number that's too large, the property wrapper adjusts the number before storing it. The code below adds a `projectedValue` property

to the `SmallNumber` structure to keep track of whether the property wrapper adjusted the new value for the property before storing that new value.

```
@propertyWrapper
struct SmallNumber {
    private var number: Int
    private(set) var projectedValue: Bool

    var wrappedValue: Int {
        get { return number }
        set {
            if newValue > 12 {
                number = 12
                projectedValue = true
            } else {
                number = newValue
                projectedValue = false
            }
        }
    }

    init() {
        self.number = 0
        self.projectedValue = false
    }
}
struct SomeStructure {
    @SmallNumber var someNumber: Int
}
var someStructure = SomeStructure()

someStructure.someNumber = 4
print(someStructure.$someNumber)
// Prints "false"

someStructure.someNumber = 55
print(someStructure.$someNumber)
// Prints "true"
```

Writing `someStructure.$someNumber` accesses the wrapper's projected value. After storing a small number like four, the value of `someStructure.$someNumber` is `false`. However, the projected value is `true` after trying to store a number that's too large, like 55.

A property wrapper can return a value of any type as its projected value. In this example, the property wrapper exposes only one piece of information — whether the number was adjusted — so it exposes that Boolean value as its projected value. A wrapper that needs to expose more information can return an instance of some other type, or it can return `self` to expose the instance of the wrapper as its projected value.

When you access a projected value from code that's part of the type, like a property getter or an instance method, you can omit `self.` before the property name, just like accessing other properties. The code in the following example refers to the projected value of the wrapper around `height` and `width` as `$height` and `$width`:

```
enum Size {
    case small, large
}

struct SizedRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int

    mutating func resize(to size: Size) -> Bool {
        switch size {
        case .small:
            height = 10
            width = 20
        case .large:
            height = 100
            width = 100
        }
        return $height || $width
    }
}
```

Because property wrapper syntax is just syntactic sugar for a property with a getter and a setter, accessing `height` and `width` behaves the same as accessing any other property. For example, the code in `resize(to:)` accesses `height` and `width` using their property wrapper. If you call `resize(to: .large)`, the switch case for `.large` sets the rectangle's height and width to 100. The wrapper prevents the value of those properties from being larger than 12, and it sets the projected value to `true`, to record the fact that it adjusted their values. At the end of `resize(to:)`, the return statement checks `$height` and `$width` to determine whether the property wrapper adjusted either `height` or `width`.

Global and Local Variables

The capabilities described above for computing and observing properties are also available to *global variables* and *local variables*. Global variables are variables that are defined outside of any function, method, closure, or type context. Local variables are variables that are defined within a function, method, or closure context.

The global and local variables you have encountered in previous chapters have all been *stored variables*. Stored variables, like stored properties, provide storage for a value of a certain type and allow that value to be set and retrieved.

However, you can also define *computed variables* and define observers for stored variables, in either a global or local scope. Computed variables calculate their value, rather than storing it, and they're written in the same way as computed properties.

You can apply a property wrapper to a local stored variable, but not to a global variable or a computed variable. For example, in the code below, `myNumber` uses `SmallNumber` as a property wrapper.

```
func someFunction() {
    @SmallNumber var myNumber: Int = 0

    myNumber = 10
    // now myNumber is 10

    myNumber = 24
    // now myNumber is 12
}
```

Like when you apply `SmallNumber` to a property, setting the value of `myNumber` to 10 is valid. Because the property wrapper doesn't allow values higher than 12, it sets `myNumber` to 12 instead of 24.

Type Properties

Instance properties are properties that belong to an instance of a particular type. Every time you create a new instance of that type, it has its own set of property values, separate from any other instance.

You can also define properties that belong to the type itself, not to any one instance of that type. There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called *type properties*.

Type properties are useful for defining values that are universal to *all* instances of a particular type, such as a constant property that all instances can use (like a static constant in C), or a variable property that stores

a value that's global to all instances of that type (like a static variable in C).

Stored type properties can be variables or constants. Computed type properties are always declared as variable properties, in the same way as computed instance properties.

Type Property Syntax

In C and Objective-C, you define static constants and variables associated with a type as *global* static variables. In Swift, however, type properties are written as part of the type's definition, within the type's outer curly braces, and each type property is explicitly scoped to the type it supports.

You define type properties with the `static` keyword. For computed type properties for class types, you can use the `class` keyword instead to allow subclasses to override the superclass's implementation. The example below shows the syntax for stored and computed type properties:

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

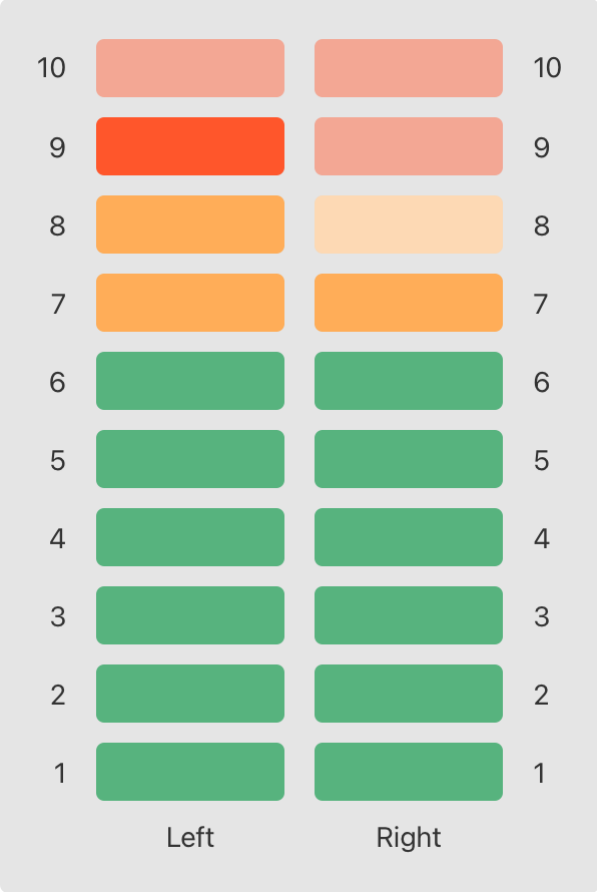
Querying and Setting Type Properties

Type properties are queried and set with dot syntax, just like instance properties. However, type properties are queried and set on the *type*, not on an instance of that type. For example:

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27"
```

The examples that follow use two stored type properties as part of a structure that models an audio level meter for a number of audio channels. Each channel has an integer audio level between `0` and `10` inclusive.

The figure below illustrates how two of these audio channels can be combined to model a stereo audio level meter. When a channel's audio level is `0`, none of the lights for that channel are lit. When the audio level is `10`, all of the lights for that channel are lit. In this figure, the left channel has a current level of `9`, and the right channel has a current level of `7`:



The audio channels described above are represented by instances of the `AudioChannel` structure:

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {
                // cap the new audio level to the threshold level
                currentLevel = AudioChannel.thresholdLevel
            }
            if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                // store this as the new overall maximum input level
                AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}
```

The `AudioChannel` structure defines two stored type properties to support its functionality. The first, `thresholdLevel`, defines the maximum threshold value an audio level can take. This is a constant value of `10` for all `AudioChannel` instances. If an audio signal comes in with a higher value than `10`, it will be capped to this threshold value (as described below).

The second type property is a variable stored property called `maxInputLevelForAllChannels`. This keeps track of the maximum input value that has been received by *any* `AudioChannel` instance. It starts with an initial value of `0`.

The `AudioChannel` structure also defines a stored instance property called `currentLevel`, which represents the channel's current audio level on a scale of `0` to `10`.

The `currentLevel` property has a `didSet` property observer to check the value of `currentLevel` whenever it's set. This observer performs two checks:

- If the new value of `currentLevel` is greater than the allowed `thresholdLevel`, the property observer caps `currentLevel` to `thresholdLevel`.
- If the new value of `currentLevel` (after any capping) is higher than any value previously received by *any* `AudioChannel` instance, the property observer stores the new `currentLevel` value in the `maxInputLevelForAllChannels` type property.

You can use the `AudioChannel` structure to create two new audio channels called `leftChannel` and `rightChannel`, to represent the audio levels of a stereo sound system:

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

If you set the `currentLevel` of the *left* channel to `7`, you can see that the `maxInputLevelForAllChannels` type property is updated to equal `7`:

```
leftChannel.currentLevel = 7
print(leftChannel.currentLevel)
// Prints "7"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "7"
```

If you try to set the `currentLevel` of the *right* channel to `11`, you can see that the right channel's `currentLevel` property is capped to the maximum value of `10`, and the `maxInputLevelForAllChannels` type property is updated to equal `10`:

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)
// Prints "10"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "10"
```

- Properties
- Stored Properties
- Stored Properties of Constant Structure Instances
- Lazy Stored Properties
- Stored Properties and Instance Variables
- Computed Properties
- Shorthand Setter Declaration
- Shorthand Getter Declaration
- Read-Only Computed Properties
- Property Observers
- Property Wrappers
- Setting Initial Values for Wrapped Properties
- Projecting a Value From a Property Wrapper
- Global and Local Variables
- Type Properties

- Type Property Syntax
- Querying and Setting Type Properties