Stored and Computed Variables

> **Warning**
>
> This document has not been updated since the initial design in Swift 1.0.

Variables are declared using the `var` keyword. These declarations are valid at the top level, within types, and within code bodies, and are respectively known as *global variables, member variables,* and *local variables.* Member variables are commonly referred to as *properties.*

Every variable declaration can be classified as either *stored* or *computed.* Member variables inherited from a superclass obey slightly different rules.

Stored Variables

The simplest form of a variable declaration provides only a type:

```
var count : Int
```

This form of `var` declares a *stored variable.* Stored variables cause storage to be allocated in their containing context:

- a new global symbol for a global variable
- a slot in an object for a member variable
- space on the stack for a local variable

(Note that this storage may still be optimized away if determined unnecessary.)

Stored variables must be initialized before use. As such, an initial value can be provided at the declaration site. This is mandatory for global variables, since it cannot be proven who accesses the variable first.

```
var count : Int = 10
```

If the type of the variable can be inferred from the initial value expression, it may be omitted in the declaration:

```
var count = 10
```

Variables formed during pattern matching are also considered stored variables.

```
switch optVal {
case .Some(var actualVal):
  // do something
case .None:
  // do something else
}
```

Computed Variables

A *computed variable* behaves syntactically like a variable, but does not actually require storage. Instead, accesses to the variable go through "accessors" known as the *getter* and the *setter.* Thus, a computed variable is declared as a variable with a custom getter:

```
struct Rect {
  // Stored member variables
  var x, y, width, height : Int

  // A computed member variable
  var maxX : Int {
    get {
      return x + width
    }
    set(newMax) {
      x = newMax - width
    }
  }

// myRect.maxX = 40
```

In this example, no storage is provided for `maxX`.

If the setter's argument is omitted, it is assumed to be named `value`:

```
var maxY : Int {
  get {
    return y + height
  }
  set {
    y = value - height
  }
}
```

Finally, if a computed variable has a getter but no setter, it becomes a *read-only variable.* In this case the `get` label may be omitted. Attempting to set a read-only variable is a compile-time error:

```
  var area : Int {
    return self.width * self.height
  }
}
```

Note that because this is a member variable, the implicit parameter `self` is available for use within the accessors.

It is illegal for a variable to have a setter but no getter.

Observing Accessors

Occasionally it is useful to provide custom behavior when changing a variable's value that goes beyond simply modifying the underlying storage. One way to do this is to pair a stored variable with a computed variable:

```
var _backgroundColor : Color
var backgroundColor : Color {
  get {
    return _backgroundColor
  }
  set {
    _backgroundColor = value
    refresh()
  }
}
```

However, this contains a fair amount of boilerplate. For cases where a stored property provides the correct storage semantics, you can add custom behavior before or after the underlying assignment using "observing accessors" `willSet` and `didSet`:

```
var backgroundColor : Color {
  didSet {
    refresh()
  }
}

var currentURL : URL {
  willSet(newValue) {
    if newValue != currentURL {
      cancelCurrentRequest()
    }
  }
  didSet {
    sendNewRequest(currentURL)
  }
}
```

A stored property may have either observing accessor, or both. Like `set`, the argument for `willSet` may be omitted, in which case it is provided as "value":

```
var accountName : String {
  willSet {
    assert(value != "root")
  }
}
```

Observing accessors provide the same behavior as the two-variable example, with two important exceptions:

- A variable with observing accessors is still a stored variable, which means it must still be initialized before use. Initialization does not run the code in the observing accessors.
- All assignments to the variable will trigger the observing accessors with the following exceptions: assignments in the init and destructor function for the enclosing type, and those from within the accessors themselves. In this context, assignments directly store to the underlying storage.

Computed properties may not have observing accessors. That is, a property may have a custom getter or observing accessors, but not both.

Overriding Read-Only Variables

If a member variable within a class is a read-only computed variable, it may be overridden by subclasses. In this case, the subclass may choose to replace that computed variable with a stored variable by declaring the stored variable in the usual way:

```
class Base {
  var color : Color {
    return .Black
  }
}

class Colorful : Base {
  var color : Color
}

var object = Colorful(.Red)
object.color = .Blue
```

The new stored variable may have observing accessors:

```
class MemoryColorful : Base {
  var oldColors : Array<Color> = []

  var color : Color {
    willSet {
      oldColors.append(color)
    }
  }
}
```

A computed variable may also be overridden with another computed variable:

```
class MaybeColorful : Base {
  var color : Color {
    get {
      if randomBooleanValue() {
        return .Green
      } else {
        return super.color
      }
    }
    set {
      print("Sorry, we choose our own colors here.")
    }
  }
}
```

Overriding Read-Write Variables

If a member variable within a class as a read-write variable, it is not generally possible to know if it is a computed variable or stored variable. A subclass may override the superclass's variable with a new computed variable:

```
class ColorBase {
  var color : Color {
    didSet {
      print("I've been painted \(color)!")
    }
  }
}

class BrightlyColored : ColorBase {
  var color : Color {
    get {
      return super.color
    }
    set(newColor) {
      // Prefer whichever color is brighter.
      if newColor.luminance > super.color.luminance {
        super.color = newColor
      } else {
        // Keep the old color.
      }
    }
  }
}
```

In this case, because the superclass's `didSet` is part of the generated setter, it is only called when the subclass actually invokes setter through its superclass. On the `else` branch, the superclass's `didSet` is skipped.

A subclass may also use observing accessors to add behavior to an inherited member variable:

```
class TrackingColored : ColorBase {
  var prevColor : Color?

  var color : Color {
    willSet {
      prevColor = color
    }
  }
}
```

In this case, the `willSet` accessor in the subclass is called first, then the setter for `color` in the superclass. Critically, this is *not* declaring a new stored variable, and the subclass will *not* need to initialize `color` as a separate member variable.

Because observing accessors add behavior to an inherited member variable, a superclass's variable may not be overridden with a new stored variable, even if no observing accessors are specified. In the rare case where this is desired, the two-variable pattern shown above can be used.