orphan:

Immutability and Read-Only Methods

Abstract: Swift programmers can already express the concept of read-only properties and subscripts, and can express their intention to write on a function parameter. However, the model is incomplete, which currently leads to the compiler to accept (and silently drop) mutations made by methods of these read-only entities. This proposal completes the model, and additionally allows the user to declare truly immutable data.

The Problem

Consider:

```
class Window {

  var title: String { // title is not writable
    get {
      return somethingComputed()
    }
  }
}

var w = Window()
w.title += " (parenthesized remark)"
```

What do we do with this? Since += has an `inout` first argument, we detect this situation statically (hopefully one day we'll have a better error message):

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\[swift-main][docs]MutationModel.rst`, line 47)

Cannot analyze code. No Pygments lexer found for "swift-console".

```
.. code-block:: swift-console

 <REPL Input>:1:9: error: expression does not type-check
 w.title += " (parenthesized remark)"
 ~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Great. Now what about this? [1]

```
w.title.append(" (fool the compiler)")
```

Today, we allow it, but since there's no way to implement the write-back onto `w.title`, the changes are silently dropped.

Unsatisfying Approaches

We considered three alternatives to the current proposal, none of which were considered satisfactory:

1. Ban method calls on read-only properties of value type
2. Ban read-only properties of value type
3. Status quo: silently drop the effects of some method calls

For rationales explaining why these approaches were rejected, please refer to earlier versions of this document.

Proposed Solution

Terminology

Classes and generic parameters that conform to a protocol attributed `@class_protocol` are called **reference types**. All other types are **value types**.

Mutating and Read-Only Methods

A method attributed with `inout` is considered **mutating**. Otherwise, it is considered **read-only**.

```
struct Number {
  init(x: Int) { name = x.toString() }

  func getValue() {               // read-only method
    return Int(name)
  }
  mutating func increment() {   // mutating method
    name = (Int(name)+1).toString()
  }
```

```
    var name: String
  }
```

The implicit `self` parameter of a struct or enum method is semantically an `inout` parameter if and only if the method is attributed with `mutating`. Read-only methods do not "write back" onto their target objects.

A program that applies the `mutating` to a method of a class--or of a protocol attributed with `@class_protocol`--is ill-formed. [Note: it is logically consistent to think of all methods of classes as read-only, even though they may in fact modify instance variables, because they never "write back" onto the source reference.]

Mutating Operations

The following are considered **mutating operations** on an lvalue

1. Assignment to the lvalue
2. Taking its address

Remember that the following operations all take an lvalue's address implicitly:

- passing it to a mutating method:

```
var x = Number(42)
x.increment()         // mutating operation
```

- passing it to a function attributed with `@assignment`:

```
var y = 31
y += 3                // mutating operation
```

- assigning to a subscript or property (including an instance variable) of a value type:

```
x._i = 3             // mutating operation
var z: Array<Int> = [1000]
z[0] = 2             // mutating operation
```

Binding for Rvalues

Just as `var` declares a name for an lvalue, `let` now gives a name to an rvalue:

```
var clay = 42
let stone = clay + 100 // stone can now be used as an rvalue
```

The grammar rules for `let` are identical to those for `var`.

Properties and Subscripts

A subscript or property access expression is an rvalue if

- the property or subscript has no `set` clause
- the target of the property or subscript expression is an rvalue of value type

For example, consider this extension to our `Number` struct:

```
extension Number {
  var readOnlyValue: Int { return getValue()  }

  var writableValue: Int {
    get {
     return getValue()
    }
    set(x) {
      name = x.toString()
    }
  }

  subscript(n: Int) -> String { return name }
  subscript(n: String) -> Int {
    get {
      return 42
    }
    set(x) {
      name = x.toString()
    }
  }
}
```

Also imagine we have a class called `CNumber` defined exactly the same way as `Number` (except that it's a class). Then, the following table holds:

Declaration:
Expression
	`var x = Number(42)  // this`
	`var x = CNumber(42) // or this`
	`let x = CNumber(42) // or this`
		`let x = Number(42)`

`x.readOnlyValue`	**rvalue** (no `set` clause)	**rvalue** (target is an rvalue of value type)
`x[3]`		
`x.writeableValue`	**lvalue** (has `set` clause)	
`x["tree"]`		
`x.name`	**lvalue** (instance variables implicitly have a `set` clause)	

The Big Rule

> **Error**
>
> A program that applies a mutating operation to an rvalue is ill-formed

For example:

```
clay = 43           // OK; a var is always assignable
stone = clay * 1000 // Error: stone is an rvalue

swap(&clay, &stone) // Error: 'stone' is an rvalue; can't take its address

stone += 3          // Error: += is declared inout, @assignment and thus
                    // implicitly takes the address of 'stone'

let x = Number(42)  // x is an rvalue
x.getValue()        // ok, read-only method
x.increment()       // Error: calling mutating method on rvalue
x.readOnlyValue     // ok, read-only property
x.writableValue     // ok, there's no assignment to writableValue
x.writableValue++   // Error: assigning into a property of an immutable value
```

Non-`inout` Function Parameters are RValues

A function that performs a mutating operation on a parameter is ill-formed unless that parameter was marked with `inout`. A method that performs a mutating operation on `self` is ill-formed unless the method is attributed with `mutating`:

```
func f(_ x: Int, y: inout Int) {
  y = x         // ok, y is an inout parameter
  x = y         // Error: function parameter 'x' is immutable
}
```

Protocols and Constraints

When a protocol declares a property or `subscript` requirement, a `{ get }` or `{ get set }` clause is always required.

```
protocol Bitset {
  var count: Int { get }
  var intValue: Int { get set }
  subscript(bitIndex: Int) -> Bool { get set }
}
```

Where a `{ get set }` clause appears, the corresponding expression on a type that conforms to the protocol must be an lvalue or the program is ill-formed:

```
struct BS {
  var count: Int    // ok; an lvalue or an rvalue is fine

  var intValue : Int {
    get {
      return 3
    }
    set {             // ok, lvalue required and has a set clause
      ignore(value)
    }
  }

  subscript(i: Int) -> Bool {
    return true   // Error: needs a 'set' clause to yield an lvalue
  }
}
```

---

[1] String will acquire an `append(other: String)` method as part of the formatting plan, but this scenario applies equally to any method of a value type