eero

objective-c, evolved

Andy Arvanitis

What is Eero?

- A programming language
  - It is a dialect of Objective-C
  - It has a streamlined syntax
  - It is binary- and header-compatible with Objective-C
  - Its toolchain is implemented using a continuously updated fork of LLVM/Clang (trunk)

Why did I do this?

- My background: I've mostly worked for large companies, large organizations, often large teams
- Whether an architect or individual contributor, I end up reading lots and lots of other people's code
- Even reading your own code after a few months (or years) is like reading someone else's code

Code readability

- Code is read much more often than it's written
- *Human* "parsability" of code is critical
- Readability matters A LOT! And when it comes to a programming language, it's **not** "*just syntactic sugar*"
- Eero is an unrepentant project about syntactic sugar

Eero goals

- To be a practical tool, not just an experiment
- To reuse ALL Objective-C frameworks out there, as seamlessly as possible
- To keep Objective-C strengths: named parameters, type checking, performance, interoperability with C libraries, tools

Eero principles

- DRY
- WYSIWYG (POLS)
- Make the compiler, not the human, do the work

Eero techniques

- Remove clutter without introducing ambiguity
- Avoid “syntactic saccharin”
- Employ sane defaults

General code-structure changes

Significant whitespace

- Offside-rule, à la Python
- Lots of controversy — strong opinions on both sides
- Personally, I wasn’t so sure about it at first, but grew to appreciate/love it

Significant whitespace — why?

- WYSIWYG / DRY:
  - most agree that blocks should be indented
  - Why should I have to indent *and* use braces?
- Readability! — less visual clutter

Optional semicolons

- Readability (less visual clutter)
- DRY
- Good compilers generally know where they should go. Why should I have to do it? (works best when newline is significant)
- Clang very good for this, made it easy to implement

Local type inference

- The **:=** operator specifies a variable definition and assignment, with the type inferred from the value
- Same behavior as C++11's "auto"

```
counter := 0  // infers an int

const title := "hello, world"
```

Local type inference

- Gains some of the conciseness advantages of Ruby/Python, but a bit safer
  - Compiler can still catch typos in subsequent assignments
- Normal = assignment operator remains unchanged

Namespace-like prefixes

- Eero does not introduce a true namespace facility, but instead works with existing Objective-C prefix conventions
- Prefixes can be “registered” within a scope
- If a symbol (class, type, function, etc.) is not found by the compiler, it tries again, with prefix
- The “NS” prefix is built-in, e.g., “String” resolves to “NSString”

Namespace-like prefixes, cont.

- User-defined prefixes
  - Eero introduces keyword *using* (borrowed from C++), and context keyword *prefix*
  - To register prefix: `using prefix XX`
  - Declared (and valid) within an enclosing scope: file, method, function, conditional block, etc.

Example

```
using prefix UI // file scope

int myFunction()

  using prefix AB // only in function scope

  AddressBook addressBook = nil

  Log(...) // resolves to NSLog()

  return 0
```

Objective-C–specific changes

Objective-C keywords

- Objective-C keywords “promoted” to first-class keywords
  - @ no longer needed in front of them

```
interface MyClass : Object
...
end
```

NSString literals

- No @, enclosed in single quotes
- C string literals remain in double-quotes

```
title := 'hello, world'
```

Array and dictionary literals

- Neither type of literal needs @

```
myArray := ['a', 'b', 'c']

myDict := {'A' : 'a', 'B' : 'b'}
```

Array and dictionary literals

- Since we're on the topic: empty object literals are mutable

```
myMutableArray := []

myMutableDict := {}

myMutableString := ''
```

NSRange literals

- Alternative to *NSMakeRange()* or C99's *(NSRange){ start, length }*
- Accepts any integer values, not just numeric literals

```
range := 1 .. 10

otherRange := kFirst .. kLast
```

Objects are always pointers

- In Objective-C, we never have objects on the stack, so it is never valid to declare a non-pointer object variable
- Eero assumes you mean a pointer; nothing else is valid anyway
  - Consistent with message-passing syntax

```
String title = 'Hello, World!'
```

Methods and message-passing

Message-passing sans brackets

- Like Smalltalk, message passing is done without square brackets
- However, commas are used to help humans (and the compiler) disambiguate selectors/ parameters
- Message-passing expressions are just like any other expression; they can be put in parentheses, if needed

Examples

```
names := []

names addObject: ’Saarinen’

names insertObject: ‘Eero’, atIndex: 0

names addObject: (otherNames objectAtIndex: 0)
```

Method declarations

- An observation: argument variable names are not really needed in interfaces
  - The users of the method don't care about them (it's an implementation detail)
  - The implementor of the method doesn't need to make variable naming decisions when designing just the interface

Method declarations, cont.

- Another observation: argument variable names tend to be very similar to (or derived from) their selector names
- Can we use sane defaults?
- When we do need to specify arg variable names, can we get something a little more consistent with normal variable declarations (maybe without those parentheses)?

Method declarations in Eero

- Argument variable names default to last camel-case word in selector (see site for more details)
  - Similar to Apple's approach to properties/ setters
  - Override with variable name following type, just like a normal declaration

Example

```
@interface MyNotificationCenterClass

 -(void)addObserver:(id)observer
           selector:(SEL)selector
               name:(NSString *)notificationName
             object:(id)object;

@end
```

Example

```
interface MyNotificationCenterClass

        addObserver: id,
           selector: SEL,
               name: String,
             object: SEL

end
```

Example

```
implementation MyNotificationCenterClass

        addObserver: id,
           selector: SEL,
               name: String,
             object: SEL

           if observer == nil
               ...

end
```

Example

```
implementation MyNotificationCenterClass

        addObserver: id,
           selector: SEL,
               name: String notificationName,
             object: SEL

           if observer == nil
               ...

end
```

Method declarations in Eero

- Note: there are a couple of (subtle) departures from Objective-C
  - Parameter types are no longer optional (they do not default to *id*)
  - When absent, return type no longer defaults to *id*, it now means no return value at all, i.e., *void* — WYSIWYG

Optional and default parameters

- Eero's method prototype syntax allows specification of optional parameters (selector pieces) and default values
- Optional parameters are enclosed in square brackets in the interface (mirroring common software/tech-doc convention)
- Defaults are assigned in the implementation using "= value" (they are, in fact, an implementation detail)

Example

```
interface MyNotificationCenterClass
        addObserver: id,
           selector: SEL,
               name: String,
             object: SEL
end

implementation MyNotificationCenterClass
        addObserver: id,
           selector: SEL,
               name: String,
             object: SEL

            if name == nil
                ...
end
```

Example

```
interface MyNotificationCenterClass
        addObserver: id,
           selector: SEL,
              [name: String],
             object: SEL
end

implementation MyNotificationCenterClass
        addObserver: id,
           selector: SEL,
               name: String,
             object: SEL

            if name == nil
                ...
end
```

Example

```
interface MyNotificationCenterClass
        addObserver: id,
           selector: SEL,
              [name: String],
             object: SEL
end

implementation MyNotificationCenterClass
        addObserver: id,
           selector: SEL,
               name: String = nil,
             object: SEL

            if name == nil
                ...
end
```

Optional and default parameters

- This is all just sugar, and doesn't introduce any semantic changes
  - The compiler simply generates the appropriate method interfaces and implementations
- You can mix/match these with existing Objective-C classes

Object operators

Object subscript operators

- Same support recently added to Objective-C (although Eero had them first ;-)
  - Integers imply array operations; objects imply dictionary operations
- Eero introduces support for NSRange subscripts
  - If a string, slice via *substringWithRange*
  - Otherwise, slice via *subarrayWithRange*

Example

```
implementation MyClass

    + truncateString: String, return String
        if string.length > 10
            truncatedString := string[0 .. 9]
            return truncatedString
        return string



end
```

Example

```
implementation MyClass

    + truncateString: String, return String
        if string.length > 10
            truncatedString := string[0 .. 9]
            return truncatedString
        return string

    + truncateArray: Array, return Array
        if array.length > 100
            truncatedArray := array[0 .. 99]
            return truncatedArray
        return array
end
```

Overloadable binary operators

- Supported operators are **+ - * / < >**
  - Includes implicit support for operator/equal variants (**+=**, **-=**, **<=**, etc.)
- Each of these map to method selectors *plus*, *minus*, *multipliedBy*, *isLess*, etc.
  - Overloaded by implementing any of these methods

Example

```
// Defining some string comparison operators using
// a category

implementation String (operators)

    isLess: String string, return BOOL
        const comparison := self compare: string
        return (comparison == OrderedAscending)

    isGreater: String string, return BOOL
        const comparison := self compare: string
        return (comparison == OrderedDescending)

end
```

Example

```
// Using the string comparison operator ‘<‘ from
// the category

name := ‘Eames’
previousName := ‘Saarinen’

if name < previousName
    ...
```

Overloadable operators, cont.

- There are some built-in operators
  - **==** and **!=** via *isEqual* for all object types
  - If object is a String, **+** is concatenation via *stringWithString*
  - If object is a MutableString, **<<** is concatenation via *appendString*

Example

```
hello := 'hello'
world := 'world'

title := hello + ', ' + world

// title is now 'hello, world'

itemName := ''
itemName << 'tulip'
itemName << ' '
itemName << 'chair'

// itemName is now 'tulip chair'
```

Other features

- Optional parentheses around conditions
- Concise boxing/unboxing
- Enhanced blocks (including compact form)
- Stricter enum type checking
- Enhanced *switch/case*
- And more!

Questions?

Thank you!

- More information:
  - http://eerolanguage.org
  - https://github.com/eerolanguage
- My twitter: @andyarvanitis