# Laziness in Swift

Maciej Konieczny
narf.pl · macoscope.com

Python 😍

Django 😋

JavaScript 😩

## CoffeeScript 😉

## Objective-C 😕

Swift 😮

Python 😍

## Laziness

delaying computation until necessary

never necessary
never computed

## removing needless computation

## reducing memory footprint

## infinite data structures

*Laziness allows the expression of programs that would otherwise not terminate*

— Matt Might

not one pattern

## Swift

`lazy var`

## SequenceType

@autoclosure

`lazy var`

```
class BlogPost {
    var filename: String
}
```

```
class BlogPost {
    var filename: String

    init(filename: String) {
        self.filename = filename
    }
}
```

```swift
class BlogPost {
    var filename: String
    var image = Image()

    init(filename: String) {
        self.filename = filename
    }
}
```

```swift
class BlogPost {
    var filename: String
    lazy var image = Image()

    init(filename: String) {
        self.filename = filename
    }
}
```

```swift
class BlogPost {
    var filename: String
    lazy var image = Image()

    init(filename: String) {
        self.filename = filename
    }
}

var post = BlogPost(filename: "sw2.md")
post.image
```

```
class BlogPost {
    var filename: String
    lazy var image = Image()

    init(filename: String) {
        self.filename = filename
    }
}
```

```
class BlogPost {
    var filename: String
    lazy var image = \
        Image(forFilename: self.filename)

    init(filename: String) {
        self.filename = filename
    }
}
```

```
class BlogPost {
    var filename: String
    lazy var image = {
        Image(forFilename: self.filename)
    }()

    init(filename: String) {
        self.filename = filename
    }
}
```

# Swift ≠ ObjC

nil ≠ nil

```
- (Image *)image {
    if (!_image) {
        _image = [[Image alloc]
            imageForFilename:self.filename];
    }

    return _image;
}
```

## SequenceType

```
for x in xs {
    // ...
}
```

```
for x in xs {
    // ...
}

var _g = xs.generate()
while let x = _g.next() {
    // ...
}
```

awesome

```
class Integers: SequenceType {
    func generate() -> GeneratorOf<Int> {
        var n = -1
        return GeneratorOf { ++n }
    }
}
```

```
class Integers: SequenceType {
    func generate() -> GeneratorOf<Int> {
        var n = -1
        return GeneratorOf { ++n }
    }
}

for i in Integers() {
    println(i)  // 0, 1, 2, 3, ...
}
```

```
lazy()
lazy()

var xs = [1, 2, 3]
xs.lazy()
```

```
lazy()

var xs = [1, 2, 3]
xs.lazy()

LazySequence
LazyForwardCollection
LazyRandomAccessCollection
```

```
var integers = lazy(Integers())
var integers = lazy(Integers())

integers.filter
integers.map
```

```
var x = integers
var x = integers \
    .filter { $0 % 2 == 1 }
```

```
var x = integers \
    .filter { $0 % 2 == 1 } \
    .map { $0 * $0 }
```

```
var x = integers \
    .filter { $0 % 2 == 1 } \
    .map { $0 * $0 } \
    .filter { $0 > 100 }
```

```
var x = integers \
    .filter { $0 % 2 == 1 } \
    .map { $0 * $0 } \
    .filter { $0 > 100 } \
    .first!
```

```
var x = integers \
    .filter { $0 % 2 == 1 } \
    .map { $0 * $0 } \
    .filter { $0 > 100 } \
    .first!

println(x)  // 121
```

## call order

```
var x = integers \
    .filter { $0 % 2 == 1 } \
    .map { $0 * $0 } \
    .filter { $0 > 100 } \
    .first!

println(x)  // 121
```

```
var x = integers.filter {
    return $0 % 2 == 1
}.map {
    return $0 * $0
}.filter {
    return $0 > 10
}.first!

println(x)  // 25
```

```
var x = integers.filter {
    println("\n\($0)")
    println("odd?")
    return $0 % 2 == 1
}.map {
    println("square")
    return $0 * $0
}.filter {
    println("threshold")
    return $0 > 10
}.first!

println(x)  // 25
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!

0
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!

0 odd?
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!

0 odd?
1
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!
```

```
0 odd?
1 odd?
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!

0 odd?
1 odd? square
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!

0 odd?
1 odd? square threshold
```

```
integers.filter { $0 % 2 == 1 } \
        .map { $0 * $0 } \
        .filter { $0 > 10 } \
        .first!

0 odd?
1 odd? square threshold
2 odd?
3 odd? square threshold
4 odd?
5 odd? square threshold
```

declarative

```
extension LazySequence {
    var first: LazySequence.Generator.Element? {
        for x in self {
            return x
        }

        return nil
    }
}

integers.first!  // 0
```

`@autoclosure`

```
// without @autoclosure:
f({ x })
```

```
// without @autoclosure:
f({ x })

// with @autoclosure:
f(x)
```

```swift
func f() -> Bool {

    return true
}
```

```
func f() -> Bool {

    return true
}

func g() -> Bool {

    return false
}
```

```
func f() -> Bool {
    println("f")
    return true
}

func g() -> Bool {
    println("g")
    return false
}
```

```
func or
func or(left: Bool
func or(left: Bool, right: Bool)
func or(left: Bool, right: Bool) -> Bool
func or(left: Bool, right: Bool) -> Bool {
    if left {
        return left
    }
```

```
func or(left: Bool, right: Bool) -> Bool {
    if left {
        return left
    } else {
        return right
    }
}
```

```
func or(left: Bool,
        right: Bool)
-> Bool {

    if left {
        return left
    } else {
        return right
    }
}
```

```
func or(left: Bool,
        right: Bool)
-> Bool {

    if left {
        return left
    } else {
        return right
    }
}

println(or(f(), g()))
// f, g, true
```

```
func or(left: Bool,
        right: () -> Bool)
-> Bool {

    if left {
        return left
    } else {
        return right()
    }
}

println(or(f(), { g() }))
// f, true
```

```
func or(left: Bool,
        right: @autoclosure () -> Bool)
-> Bool {

    if left {
        return left
    } else {
        return right()
    }
}

println(or(f(), g()))
// f, true
```

powerful

```
f() || g()
```

```
f() || { g() }
```

## Laziness

not one pattern

removing
needless
computation

## reducing memory footprint

# infinite data structures

expressiveness

```
lazy var image = Image()
lazy var image = Image()

lazy var image = {
    Image(forFilename: self.filename)
}()
```

```
for x in xs {
    // ...
}
```

```
for x in xs {
    // ...
}

var _g = xs.generate()
while let x = _g.next() {
    // ...
}
```

```
// without @autoclosure:
f({ x })
```

```
// without @autoclosure:
f({ x })

// with @autoclosure:
f(x)
```

```
// without @autoclosure:
f({ x })

// with @autoclosure:
f(x)
```

POWER!

*That's all folks!*

narf.pl

## Questions?

## References (1 of 2)

- *Understand and implement laziness*, Matt Might http://matt.might.net/articles/implementing-laziness/
- *WWDC 2014, Session 404: Advanced Swift* https://developer.apple.com/videos/wwdc/2014/

## References (2 of 2)

- *Lazy by name, lazy by nature*, airspeedvelocity http://airspeedvelocity.net/2014/07/26/lazy-by-name-lazy-by-nature/
- */r/aww* http://www.panoptikos.com/r/aww/top