Swift vs Scala 2.11

Denys Shabalin

June 2014

Control Flow

	Swift	Scala

for-in	for i in 1…5 {
	  println(“i = \(i)”)
	}
		for (i <– 1 to 5) {
		  println(s”i = $i”)
		}

for-yield	N/A	for (i <– 1 to 5)
		  yield i^2

for-increment	for var i = 0; i < 3; ++i {
	  println(“i = \(i)”)
	}
		N/A

while	while cond { … }	while(cond) { … }
do-while	do { … } while cond	do { … } while(cond)

Control Flow

	Swift	Scala
if-then	`if cond { … }`	`if (cond) { … }`
if-then-else	`if cond { … } else { … }`	`if (cond) { … } else { … }`

switch	`switch value {`
	`  case pattern where cond:`
	`    …`
	`}`
		`value match {`
		`  case pattern if cond =>`
		`    …`
		`}`

control transfer	`continue, break,`
	`fallthrough`
		`N/A`

labels	`label: while cond { … }`	`N/A`

Expressions

	Swift	Scala

unary op	`!expr`
	`* customizable`
		`!expr`
		`* limited to !, ~, +, -`

binary op	`a + b`	`a + b`
postfix op	`a++`	`a++`

assign	`a = b`
	`(a, b) = (1, 2)`
		`a = b`
		N/A

is	`a is T`	`a.isInstanceOf[T]`

as	`a as T`
	`a as? T`
		`a.asInstanceOf[T]`
		N/A

Expressions

	Swift	Scala
literals	1, 1.0, “foo”	1, 1.0, “foo”
interpo-lation	“\(x) + \(y) = \(x + y)” * not extensible	s”$x + $y = ${x + y}” * extensible
array literal	[a, b, c]	Array(a, b, c)
(mutable) map literal	[a: b, c: d]	s.c.m.Map(a -> b, c -> d) * scala.collection.mutable.Map

Expressions

	Swift	Scala

self	self
	self.foo
	self[foo]
	self.init(foo)
		this
		this.foo
		this(foo) // in exprs
		this(foo) // in ctors

super	super.foo super[foo] super.init(foo)	super.foo super(foo) N/A

closure	{ (params) -> ret in
	  …
	}
	* ret can be inferred
		{ (params) =>
		  …
		}

place-holders	f { $0 > $1 }	f { _ > _ }
implicit membership	.foo	N/A

Expressions

	Swift	Scala
block	`{ … }`	`{ … }`
return	`return foo`	`return foo`
throw	`N/A`	`throw expr`

try	`N/A`	`try expr`
		`catch { … }`
		`finally { … }`

imports	`import foo.bar`
	`import class foo.bar`
	`N/A`
		`import foo.bar`
		`N/A`
		`import foo._`

Declarations

	Swift	**Scala**

let	`let x: T = expr`
	`let y = 2`
	`let (x, y) = (1, 2)`
	`@lazy let z = f()`
		`val x: T = expr`
		`val y = 2`
		`val (x, y) = (1, 2)`
		`lazy val z = f()`

var	`var x: T = expr` …	`var x: T = expr` …

property	`var name: T {`
	`  get { stats1 }`
	`  set(v) { stats2 }`
	`}`
		`def name: T = stats1`
		`def name_=(v: T) = stats2`

observers	`var name: T = expr {`
	`  willSet { stats1 }`
	`  didGet(v) { stats 2 }`
	`}`
		N/A
		* can be emulated via macro annotations

Declarations

	Swift	**Scala**
typealias	typealias T = …	type T = …

methods	func f(x: A) -> B { … }
	func g(x: A) { … }
	func h<T>(x: T) -> T { … }
	func k<T: A>(x: T) -> T { … }
	func m(x: Int = 0) { … }
	func n(x: A)(y: B) -> C { … }
		def f(x: A): B = …
		def g(x: A) { … }
		def h[T](x: T): T = …
		def k[T <: A](x: T): T = …
		def m(x: Int = 0) { … }
		def n(x: A) = { (y: B) => … }

subscripts	subscript(key: A) -> B {
	  get {
	    stats1
	  }
	  set(value) {
	    stats2
	  }
	}
		def apply(key: A): B = {
		  stats1
		}
		def update(key: A, value: B): Unit = {
		  stats2
		}

Declarations

	Swift	**Scala**

enum case	`enum Foo {`
	`  case A(x: Int)`
	`  case B(y: Int)`
	`}`
		`sealed abstract class Foo`
		`final case class A(x: Int) extends Foo`
		`final case class B(x: Int) extends Foo`

enum with raw cases	`enum Foo {`
	`  case A, B = 1, 2`
	`}`
		`// roughly but not really`
		`class Foo private(value: Int)`
		`  extends AnyVal`
		`object Foo {`
		` val (A, B) = (new Foo(1), new Foo(2))`
		`}`

struct	`struct Foo {`
	`  …`
	`}`
	
	`* alocated on stack`
		`N/A`
		
		`* multi-parametric value classes?`

Declarations

	Swift	Scala

class with explicit and convenience inits	class Foo {
	  let x: Int
	  init(x: Int) {
	    self.x = x
	  }
	  convenience init(x: String) {
	    self.x = x.toInt()
	  }
	}
	Foo(0)
	Foo(“1”)
		class Foo(val x: Int) {
		  def this(x: String) = this(x.toInt)
		}
		new Foo(0)
		new Foo(“1”)

struct with default init	struct Foo {
	  let x = 0
	}
	Foo()
	Foo(x: 1)
		class Foo(val x: Int = 0)
		new Foo()
		new Foo(x = 1)

Declarations

	Swift	**Scala**

protocol	protocol Nameable {
	  func name() -> String
	}
	
	func f<T: Nameable> (x: T) {
	  …
	}
		trait Nameable {
		  def name(): String
		}
		
		def f[T <: Nameable](x: T) {
		  …
		}

extensions	extension Foo: Nameable {
	  func name() -> String { … }
	}
		implicit class RichFoo(foo: Foo)
		         extends Nameable {
		  def name(): String = …
		}

Declarations

	Swift	**Scala**

prefix operator	operator prefix + {}
	func +(x: T) {}
	
	* extensible
		// this: T
		def unary_+ = …
		
		* not extensible

postfix operator	operator postfix ++ {}
	func ++(x: T) { … }
		// this: T
		def ++ = …

infix operator	operator infix + {
	  precedence 100
	  associativity left
	}
	func +(left: A, right: B) { … }
		// this: A
		def +(value: B) = …
		
		* associativity and precedence via convention

Patterns

	Swift	Scala
wildcard	case _:	case _ =>
binding	case let x:	case x =>
tuple	case let (a, b):	case (a, b) =>
enum	case Foo(let a):	case Foo(a) =>

is/as	case x is Int:
	case x as Int:
		case x: Int =>
		not sure

expression	case “foo”:
	case x:
	case 2 + 2:
		case “foo” =>
		case `x` =>
		N/A
		* limited subset of expressions

extractor	N/A
	case B:
	* you can emulate nullary extractors that return booleans via custom comparator and expression patterns
		case A(x) =>
		case B() =>

Types

	Swift	Scala
identifier	A	N/A * swift types aren’t nullable
tuple	(A, B) (x: A, y: B)	(A, B) N/A * but similar to { def x: A; def y: B }
function	A -> B	A => B
array	A[] Array<A>	Array[A]
optional	A? Optional<A>	Option[A] * doesn’t directly map as swift types aren’t nullable by default
implicitly unwrapped optional	A! ImplicitlyUnwrappedOptional<A>	A
protocol composition	protocol<A, B>	A with B
metatype	A.Type B.Protocol	N/A