Introduction to Swift
CMSC 436
Action Items

▶ Log into https://gitlab.cs.umd.edu

▶ Make sure you have access to a Mac. You can not complete this course by sharing w/ a friend.

▶ Start looking at the first quiz material. You can look at the material at the same time while taking the quizzes, quizzes are not timed.

▶ Start looking at the first project as soon as it becomes available this weekend.
What is Swift?

Apple’s language for iOS development

Descended from Objective-C

Has a lot of modern PL features

Compiles to native bytecode

Comes with a REPL (Read-Eval-Print-Loop)
  ▶ Easy to test basic language features
  ▶ You can use it as a calculator
Some Basics

All your favorite C operators (with some twists)

The usual comments: /* */ and //

Blocks: {}

Semicolons are optional!!!

Control flow: if, while, switch, for — but some differences
repeat-while, guard
break, continue

Optionals

Variables and constants
Variables and Constants

```swift
var x=1 — This defines a variable
let y=3 — This defines a constant
```

Where possible, types are *inferred*, but can be declared explicitly:
```swift
var z:Int
```

Variables are *mutable*, while constants are not:
```swift
x=2 // just fine
y=2 // error!
```

Integers can be specified with base:
- `11==11` (decimal)
- `0b11==3` (binary)
- `0o11==9` (octal)
- `0x11==17` (hexadecimal)

Variable and constant names can be unicode, so ☕️=3 will work
⇒ Use this power wisely!
Optionals

No NULL, but nil

If variable must be nilable ⇒ Optional

```swift
var w:Int?
```

```swift
w=3
w=nil
```

Dealing with Optionals:

```swift
if let v = w { ... } //conditional unwrapping
let v = w ?? 8 //nil-coalescing op ??
let v = (w != nil) ? w! : 8 //forced unwrapping op !
```
Operator Differences

Arithmetic ops *will not* overflow
⇒ runtime errors

Put an & in front if you *want* overflow
⇒ maybe you’re creating a trip odometer?

= does *not* result in a value
⇒ if x = y { ... } // error!
Control Flow

if, guard

```swift
if condition1 { // notice lack of parentheses
    statements1
} else if condition2 {
    statements2
} else {
    statements3
}

guard condition else {
    statements1 // if condition is false
    // must exit enclosing context
}
```
Control Flow

while, repeat

while condition {
    statements
}

repeat {
    statements
} while condition
Control Flow

`switch`

Like C, but:

- no break needed, `fallthrough` keyword for C-like behavior
- not limited to integers
- comma-separated values
- `ranges`

Cases *must* be exhaustive

⇒ you can use `default`
Control Flow

`for` and `ranges`

```plaintext
for i in collection { // i is declared here
    ...
}

collection can be anything iterable, like an array or a `range` (also called an `interval`)

1...5 // 1,2,3,4,5 (closed interval)
1..<5 // 1,2,3,4 (half-open interval)
1... // 1,2,3,... (one-sided interval)
...1 // not iterable!
```
What Use are One-Sided Ranges?

```swift
for i in 3... {
    // do stuff
    if done { break }
}

let a = ["a","b","c"]
for b in a[..<2] {
    print(a)
}

let r = ..<5
r.contains(-1) // true
r.contains(3) // true
r.contains(12) // false
```
Arrays are expandable:

```swift
var a = Array<Int>()
var a = [Int]()
a.append(3)
a += [4]
var b = a + [5,6,7]
```

Tuples are fixed-length:

```swift
var t = (3,4,5)
```

Also: `Set<T>` — like an array, but with unique elements and set operations like union and intersection
More Tuples

```swift
var t = (a:1, b:2) // named tuple
    t.a == t[0]
    t.b == t[1]

switch t {
    case (3,4):
        // something
    case let(x,y) where x==y:
        // something with x,y defined
    case (let(x),0):
        // something with x defined
    case (let(x),let(y)):
        // something with x,y defined
}
```
Strings

Unicode: let r = "®"

Interpolation: let s = "FooBar\(r)"

Indexing:
- s.startIndex  s.index(after:s.startIndex)
- s.endIndex  s.index(before:s.endIndex)
- s.index(s.startIndex, offsetBy:3)

⇒ These are not integers!

Substrings: s[..<s.firstIndex(of:"B")!]  
⇒ This is not a string!

s.insert("!", at: s.endIndex)

""""  
multi-line  
string  
""""
String Characters

The usual: `\n, \r, \", \', \t`

`\u{#}` — unicode (# is in hex)

Unicode can be entered directly

We don’t use single quotes for individual characters:

```swift
let a: Character = "a"
```
Dictionaries

Key-value stores

```swift
var d = [Int: String]()

d[1] = "foo"
d[38] = "bar"

var d2: [Int: String] = [1: "foo", 38: "bar"]
```

Key types must be *Hashable*
Type Aliasing

Like typedef in C

typealias Count = UInt16

var c:Count

print(Count.self) // UInt16