Closures

Mobile Computing - iOS

Objectives:

- Students will be able to:
  - explain the purpose of closures
  - write closures
  - write the signatures for methods that accept closures

First Class Citizens

- In programming languages, a type is said to be a "first class citizen" if it can be **all** of these:

  1. the value of a variable
  2. input to a procedure (function or method)
  3. returned from a procedure
  4. a member of a data aggregate
  5. anonymous (not named)

Q: Is an Int a first class citizen?

- First class *functions* are blocks of code that behave as first class citizens, i.e., they can be stored in a variable; input to a procedure; etc. Note that we are not talking about the *value* of a function, but the function itself, the code.

- The last bullet point — that functions can be anonymous — may seem unusual. How can you have a function without a name? Read on to find out!

Introducing Closures

- A closure in Swift is a **self-contained block of code** that is a first class function. It can be either:
  - a global or nested **named** function, or
  - a closure expression, an **unnamed** code block
- We have already worked with named functions, so in this keynote we will concentrate on closure expressions

Why the Name?

- Closures can capture and store references to constants and variables that are in the scope in which it is defined. They are said to "close over" the variables, hence the name.

Syntax

- Closures are created by enclosing them in { } (because code, of course, should *always* be in { }).
- It includes a parameter list, return type, and body. The latter is separated from the other two by keyword **in.**
- {(x:Int) -> Int in return x*x}
    - This is a function that computes the square of its argument.

A Small Example

```
var changer:(Int)->Int = {(x:Int)->Int in return x*x}

changer(5)

func printTableFrom(start:Int, end:Int, calc:(Int) -> Int) {

     for i in start ... end {
        let result = calc(i)
        print("\(i)     \(result)")
    }

}

printTableFrom(start: 5, end: 10, calc: changer)
```

NB: In defining the closure, we specify parameter names. But as a type, we do not.

Another Example

- In the previous example, we stored our closure in a variable. Often, however, closure expressions are simply written as needed, without wasting an extra variable

```
printTableFrom(start: 5, end: 20, calc: {(x:Int)->Int in return x*3})
printTableFrom(start: 3, end: 5,
                   calc: {(x:Int)->Int in return 3*x*x + 5*x - 7})
```

Syntactic Sugar

- Closures can be written more concisely, leaving out some information, when provided **as an argument**. Since the function's parameter already specifies what is needed, the compiler can deduce what we omit.

  1. Types -- parameter and/or return types -- can be omitted

  2. If a closure contains a single statement, return can be omitted — the value of the statement is returned

  3. If the closure is the last argument, you can place it after the parentheses (omitting the name of the last parameter as well)

  4. Parameters can be referred to by number rather than name ($0, $1, etc.)

```
var changer:(Int)->Int = {(x:Int)->Int in return x*x}
```

Syntactic Sugar in Action

```
func printTableFrom(start:Int, end:Int, calc:(Int) -> Int)
```

```
// These statements are all equivalent
printTableFrom(start: 5, end: 10, calc: {(x:Int)->Int in return x*x})
printTableFrom(start: 5, end: 10, calc: {(x) -> Int in return x*x})
printTableFrom(start: 5, end: 10, calc: { x in return x*x})
printTableFrom(start: 5, end: 10, calc: { x in  x*x})
printTableFrom(start: 5, end: 10) { x in  x*x}
printTableFrom(start: 5, end: 10) { $0 * $0 }
```

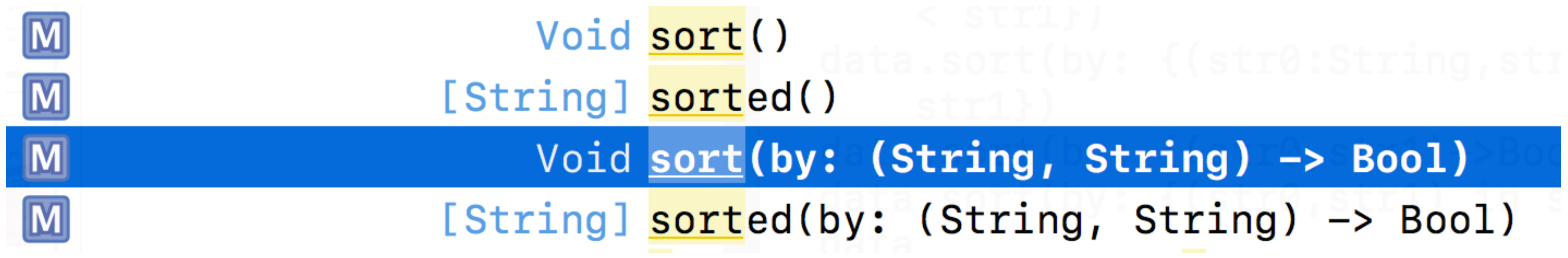
```
1.Parameter types can be eliminated
2.( ) can be eliminated
3.return type can be eliminated
4.return can be eliminated
5.argument label can be eliminated (if the closure is the last parameter)
6.parameters can be replaced with $0, $1
```

Closures in the Frameworks

- Most likely, apart from during quizzes/exams, you will not need to write methods with closure parameters in this class

- However, you will use closures extensively: they appear through the iOS frameworks, and in back end service APIs, so it is important that you know how to use them

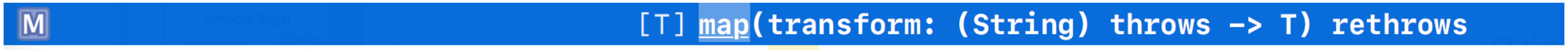
Sort Examples



```
Void sort()
[String] sorted()
Void sort(by: (String, String) -> Bool)
[String] sorted(by: (String, String) -> Bool)
```

```
var data:[String] = ["Trudeau","Obama","Modi","May","Turnbull", "Abe"]
data.sort(by: {(str0:String, str1:String)->Bool in return str0 < str1})
data.sort(by: {(str0:String, str1:String)->Bool in str0 < str1})
data.sort(by: {(str0, str1)->Bool in str0 < str1})
data.sort(by: {(str0, str1) in str0 < str1})
data.sort(by: {$0 < $1})
data.sort() {$0 < $1}
data.sort {$0 < $1}
```

A Simple Map Example



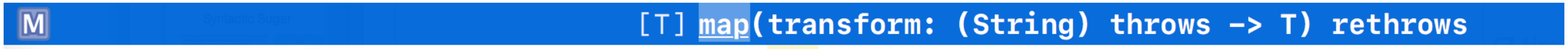
```
M    [T] map(transform: (String) throws -> T) rethrows
```

```
var celsius:[Double] = [10, 50, 0, 100]

var fahren = celsius.map( {(cel:Double)->Double in
                                return 9.0/5.0 * cel + 32.0
                           }
                        )

print(fahren)
```

A Mapnificent Map Example



```
[T] map(transform: (String) throws -> T) rethrows
```

```
var data:[String] = ["Trudeau","Obama","Modi","May","Turnbull","Netanyahu", "Abe"]


data = data.map(
    {(str:String)->String in
        var mystery:String = ""
        var i:Int = str.characters.count-1
        while i >= 0 {
            let anIndex = str.index(str.startIndex, offsetBy: i)
            mystery.append(str[anIndex])
            i = i - 1
        }
        return mystery
    } // end anonymous function
)

print(data)
```

Useful Array<T>Methods Involving Closures

- Array<T>.map((T) -> S) -> [S]
- Array<T>filter((T) -> Bool) -> [T]
- reduce() // see the docs

Should I use an Anonymous Function?

- If the function is small and easily understood.

- If you need to capture a reference (next slide!)

```swift
var data:[String] = ["Trudeau","Obama","Modi","May","Turnbull","Netanyahu", "Abe"]


let reverseString = {(str:String)->String in
    var mystery:String = ""
    var i:Int = str.count-1
    while i >= 0 {
        let anIndex = str.index(str.startIndex, offsetBy: i)
        mystery.append(str[anIndex])
        i = i - 1
    }
    return mystery
} // end function

data = data.map(reverseString)  // named function makes intent clear
print(data)
```

Putting the Close in Closure

- addUp returns an anonymous function that uses two variables.
- **value** is bound when we call the function **addUp** — The anonymous function that is returned is wrapped with a closure that records the values of the variables in the enclosing lexical scope.
- **x** is bound when **add2** or **add3** is called.

```
func addUp(value:Int) -> (Int)->Int {
    return {(x:Int)->Int in return x + value}
}

let add2 = addUp(value: 2)
let add3 = addUp(value: 3)

print(add2(4))
print(add3(4))
addUp(value: 10)(4)
```

We gave the closure a name and now we can use it like any other function.

Note: The arguments for an closure have no external name.

We can take the closure and immediately apply it to an argument.

Exercises

1. Create an array, **randy**, of 25 random Ints between 1-100.

2. Write a map to create an array of randy's elements, cubed.

3. Write a filter to remove the elements of randy that are perfect squares (e.g., 1, 3, 9, 16, 25, ...)

4. Write a function that behaves like the array's filter() method: it will be passed in an [Int] and an (Int) -> Bool closure, and apply that closure to return an array of filtered elements (those for which the closure returned true)

5. Demonstrate your method by filtering randy to only obtain even elements

Resources