Mobile Application Development
Aileen Pierce

ADVANCED SWIFT

Optionals

- Defining a variable or constant as an optional says it might have a value or it might not
- If it does not have a value it has the value nil
  - nil is the absence of a value
- Optionals of any type can have the value nil
- A ‘?’ after the type indicates it’s an optional
- If you define an optional without a value it will automatically be set to the value of nil

Optionals

- You can use an if statement to find out if an optional has a value

```
if score != nil {
    print(“There is a score”)
}
```

- Once you’re sure the optional contains a value you can access it by adding a ‘!’. This is called forced unwrapping.

```
if score != nil {
    print(“The score is \(score!)”)
}
```

Optionals

- You can conditionally unwrap an optional and if it contains a value, assigns it to a temporary variable or constant
  - Called optional binding

```
if let currentScore = score {
    print(“My current score is
    \(currentScore)”)
}
```

Optionals

- Sometimes it's clear that an optional will always have a value, after the first value is set
- We can unwrap these optionals without the need to check it each time
- These are called implicitly unwrapped optionals and are indicated by '!' after the type
- An implicitly unwrapped optional is giving permission for the optional to be unwrapped automatically each time it's used

Closures

- Functions provide a way to group a set of instructions that perform a specific task
- Functions are really just closures with a name
- Closures are blocks of code that can be passed around and used in your code

```
{(parameters) -> return type in
    statements
}
```

Classes

- Classes are the building blocks of object-oriented programming
  - Properties define the characteristics
  - Methods define the behavior
- Enumerations and Structures in Swift have many features that in other languages only exist in classes

Enumerations

- An enumeration defines a type for a group of related values

```
enum MyEnumeration{
    //enumeration member values
    case member1
    case member2
    ...
}
```

- Use dot notation to access the member values

Classes and Structures

- Classes and structures can both
  - Define properties
  - Define methods
  - Define subscripts
  - Define initializers
  - Be extended
  - Conform to protocols

Classes

- Classes have additional functionality structures do not
  - Support inheritance
  - Type casting
  - Support deinitializers
  - Use reference counting

Structures

```
struct MyStructure{
    //data members
    //methods
}
```

- Structures and classes are definitions of a type, you need to create an instance in order to assign values and call the methods

```
let x = MyStructure()
```

Structures

- Structures and classes both use initializer syntax to create instances
  - A default initializer method that lets you initialize all the data members is automatically created for structs, but not for classes
- Classes are reference types
  - A reference to the instance is assigned or passed
- Structures and enumerations are value types
  - values are copied when passed around in your code

Type Casting

- Type casting is a way to check the type of an instance
- Use the “is” type check operator to test whether an instance is of a certain class type
  - Returns **`true`** if it is of that type
  - Returns **`false`** if is is not of that type

Type Casting

- Type casting lets you treat an instance as if its is a different class in its class hierarchy
- When you believe an instance refers to the subclass type use the “as” type cast operator to try to downcast to the subclass type
  - Use the conditional form “as?” when you’re not sure if the downcast will succeed
    - Returns an optional
    - Returns `nil` if the downcast wasn’t possible

Type Casting

  - – Use the forced form “as!” when you are sure the downcast will always succeed
    - Attempts the downcast and force-unwraps the result
    - You will get a runtime error if you try to downcast to an incorrect class type
- Casting treats the instance being cast as an instance of the type to which it has been cast
- Casting does not actually modify the instance or change its value

Type Casting

- AnyObject can represent an instance of any class type
  - Objective-C does not have typed arrays so the SDK APIs often return an array of [AnyObject]
  - If you know the type of objects in the array you can use the force form to downcast to that class type
- Any can represent an instance of any type at all, including function types and non-class types