Dr.Dobb's

November 2014

Mobile Development

Swift * Objective-C * SQLite >>

Next

ALSO INSIDE

UBM Tech

www.drdobbs.com

Previous Next

Dr. Dobb's

CONTENTS

November 2014

FEATURES

More on DrDobbs.com

The C++14 Standard: What You Need to Know

Descriptions of the C++14 changes that will have significant impact in your coding work, along with working examples and discussions of when and why you would employ the features.

http://www.drdobbs.com/240169034

RESTful Web Services: A Tutorial

As REST has become the default for most Web and mobile apps, it's imperative to have the basics at your fingertips.

http://www.drdobbs.com/240169069

Software Development is Very, Very Hard — Even for Those Who Know It's Hard

Inherent software complexity is not the only important factor; so are team size, location and distribution, regulatory constraints, and requirements management. With no two teams working exactly alike, no single formula or methodology can address all needs.

http://www.drdobbs.com/240168984

Jolt Awards: The Best Books

The best programming books of the past year.

http://www.drdobbs.com/240169070

iOS 8 HealthKit: Working with Biometric Data

Once you begin collecting data on individuals in real time, permission management, data storage, and data access all become challenging issues. Apple's iOS8 attempts to balance rights and privacy with developers' desire to write apps.

http://www.drdobbs.com/240169153

Mailbag

Open Source, product manuals, and BASIC at 50.

OPEN SOURCE VULNERABILITIES

In response to our editorial that open source software makes no claim to be safer than other forms, only easier to fix ("Open Software's Thousand Eyes: A Misunderstood Claim," http://www.drdobbs.com/240169160), we were reminded:

"Please note that both Heartbleed (in OpenSSL) and Shellshock (in bash) were discovered by researchers who could examine the code. For example, CERT announcement TA-14-258 on the BASH bug was issued on September 25, 2014. However, a French researcher found the bug in July 2014. He sent a public announcement on the bug only on September 24, 2014, in order to provide enough time for software vendors to get a fix. So maybe there is some point in 'Given a thousand eyes, all bugs are shallow.'"

— Ami Shlezinger

"One relevant observation: Both recent high visibility vulnerabilities, Heartbleed and Shellshock, were discovered in code reviews, and the evaluation/remediation was decisive precisely because of open source. I contrast that with the closed software situation, where the process is entirely managed by the commercial entities owning the software. As a result, only wide disclosure and existing large scale exploitation forces quick reaction. Otherwise, we have extremely long vulnerability periods, such as in the recent Microsoft Sandworm catastrophe that apparently was being exploited since 2009."

— Przemek Klosowski

THE ABSENCE OF PRODUCT MANUALS

Our editorial bemoaning the lack of care in documentation today and the absence of options to get hard copy manuals ("Lack of Manual Labor," http://www.drdobbs.com/240169071) elicited several supportive comments and this interesting counterpoint:

"You make some good points in 'Lack of Manual Labor.' I've seen a lot of changes in the programming field over the past decades and lack of quality documentation for tools has hurt, I believe, for all the reasons you give.

However, I don't really want to see those old days come back. The idea of getting Visual Studio 2014 and a box of ten or twelve thick manuals sends a cold shiver down my spine. I don't want a large [part] of my time spent reading though those manuals. By the time I'm done, Visual Studio 2015 will be out and those manuals will go on my already cluttered shelf alongside my DOS 5.0 and Word Perfect books.

I would like to see the vacuum left by the lack of manuals filled with something more useful. Simply replacing the text manuals with electronic versions would be a start. They can be taken with you when you travel (hundreds of manuals if necessary) or accessed from the cloud when needed. I have found using electronic searching to be more useful than using the index — which I often find is less than helpful. I realize you did qualify your statement with 'a good index,' but not all indices are 'good.' Being able to find every

place in a document where 'ranged-based `for`' is used and being able to jump to each place it is used can be very useful. Also, as I get older, I appreciate being able to make the font larger.

However, replacing the hard copy manuals with electronic versions is far from nirvana. Less is more. I would much rather have fewer words with the same information. A Web page format — basically hypermedia — with concise overviews and short articles covering the details along with illustrations, videos, and examples would be much more useful.

While not perfect, MSDN's Web pages are superior in many ways to the old manuals. Also, Adobe has good training materials on their site. Other sites are also useful, but in the end, with the possible exception of MSDN, it is still harder to find answers to questions about tools today than it was when we had the manuals. This is because — as indicated in your title — the companies have not put as much effort into documenting their tools as they once did. I echo your concern, but would like to see them put that labor into more modern forms of documentation.

—David Walker

BASIC AT 50

Months after our article on 50 years of BASIC ("BASIC Turns 50, "http://www.drdobbs.com/240168054) appeared, we received this lively anecdote:

"Over here in the UK, BASIC was a hugely significant part of the 8-bit micro revolution: Machines like the BBC Micro, C64, and Sinclair Spectrum fuelled our country's programming boom and our games industry. All of those machines came with versions of BASIC built in, most of them significantly more robust than the GOTO and 1 letter variable form you describe, which would be unrecognizable to most BASIC coders here. In this country, BASIC was primarily used on these 8-bit computers as opposed to IBM-style machines, which initially were mostly found in the business sector and not the home. Of course, most of us who started with these machines went on to use other languages outside of the classroom and hobbyist programmer community. One language in particular — BBC BASIC (which had functions, procedures, a built-in assembler, and many other advanced constructs) was used by millions because the BBC Micro was embedded in all of our schools. Most of us in the UK's game industry started with that language or something similar. BBC BASIC V was the language built into the first ARM-powered machines, and is enjoying something of a resurgence on the Raspberry PI, along with the first ARM-specific operating system. This doesn't negate many of the points you made in your article — BASIC probably isn't used very much in commercial applications these days, although I have found it turning up in stock control software. But even before VB, versions of BASIC were being used to write large scale projects, especially in the UK's education sector.

—Robin Jubber
Senior Programmer
Futurelab

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

The Uncompromising Economics of Being #3 In a Two-Product Race

[EDITORIAL]

In software development, being the #3 platform in the mobile market is as good as being last.

By Andrew Binstock @platypusguy

During the days when Blackberry was fighting for its place at the app development table, I met several times with the company's head of developer outreach. An energetic man, he had many views to express, most of which made a lot of sense. There were good reasons to code for the Blackberry platform: It was easier, in some senses, to develop for than other platforms; and users were more willing to pay for BlackBerry apps than for Apple iOS and Android apps, where the notion of "everything should be free" had already taken root.

However, as he clearly articulated one afternoon, none of these benefits meant much if app developers chose to write for the BlackBerry after writing for Android. The dynamic, he explained, went like this: Companies coming out with new apps almost always write the iOS app first. This is generally a native app because Apple users are finicky enough that they tend not to be satisfied with apps that don't exactly fit their guidelines. After writing the iPhone/iPad app, there comes the moment of decision.

The worst decision for the #3 player is that the vendor chooses to develop for Android next. Android development is a black hole due to the vast number of form factors and OS versions that must be accommodated. Said the fellow from BlackBerry, organizations don't ever truly exit the Android development process. They keep porting and testing the app on an endless list of devices until they finally decide they've covered enough of them to meet their goals. Only then do they stop porting on Android. In other words, the decision to stop is a purely economic one, rather than the naturally occurring end of project when all desired devices have the app.

After the Android ordeal, the company has already decided that further development of the app to other devices is no longer economically justified. So typically, they do not port to the #3 player. That player was, at the time of these conversations, BlackBerry. Today, it's Microsoft's Windows Phone.

There is, of course, another path vendors or businesses can choose after writing the iOS app. That is to write a multiplatform app, generally using HTML5 and some framework. Such an app can work well enough on Android and the subsidiary platforms that it becomes the overarching multiplatform solution. (In reality, there are often small tweaks made in the product for individual devices, so that minor vendors are still somewhat disfavored.)

The reality is that for vendors outside of the top two spots, this approach is the only path that gives them any hope of ever getting a big enough app ecosystem. For this reason, BlackBerry loudly touted its HTML5 support and Microsoft extols its own support for that strategy.

However, the HTML5 approach underscores the disenchanting but uncompromising reality: If you're not first or second in the mobile race, it really doesn't matter who has the bronze medal, you're in the same boat as all the vendors behind you.

— Andrew Binstock
Editor in Chief
alb@drdobbs.com

Comment

Swift: Introduction to Apple's New Programming Language

A decidedly simple alternative to Objective-C

By Gastón Hillar @GastonHillar

Apple began accepting Swift-coded applications in the App Store after the launch of both iPhone 6 and iPhone 6 Plus. In this article, I provide a brief introduction of the most interesting features included in this new programming language, which provides an alternative to Objective-C when you want to create native iOS apps with XCode 6.0.x.

A New Language, A New XCode Version

Swift is an object-oriented language that includes some functional programming concepts. I believe the most important benefit that Swift provides is that you can avoid working with Objective-C to develop a native iOS app. However, you still have to work with XCode, as it is the only IDE that allows you to work with Swift to develop iOS apps. Luckily, XCode 6.0.1 solved many of the bugs related to Swift that made it almost impossible to complete a working session without unexpected errors and crashes in the beta versions of the IDE that supported Swift. There are still many bugs that would require an update to get solved, but I was able to use XCode and the iOS emulator while working on complex apps without major problems for many days. You can install XCode 6.0.1 from the Mac

App Store. I don't recommend you work with earlier XCode versions that included support for Swift because they are very unstable.

If you have some experience with C#, Java, Python, Ruby, or JavaScript, you will find Swift's syntax easy to learn. You may still miss many advanced features included in those programming languages that aren't available in Swift, but you will benefit from the features that Swift did borrow from these and other modern programming languages. Swift doesn't require the use of header files. You can import any Objective-C module and C libraries to Swift with simple import statements.

XCode 6.0.1 offers a Swift interactive Playground that allows you to write Swift lines of code and check the results immediately. This Playground is really helpful for learning Swift and its interaction with the APIs because it provides nice code completion features. You simply need to start XCode, select File | New | Playground…, enter a name for the Playground, select iOS as the desired platform, click Next, select the desired location for the Playground file, and click Create. XCode will display a Playground window with the following lines of code:

```
// Playground - noun: a place where people can play
import UIKit
var str = "Hello, playground"
```

You can add your Swift lines of code and check the results as you enter them in the right hand side of the window. In fact, you can use the Playground to test the sample lines of code I will provide (see Figure 1).

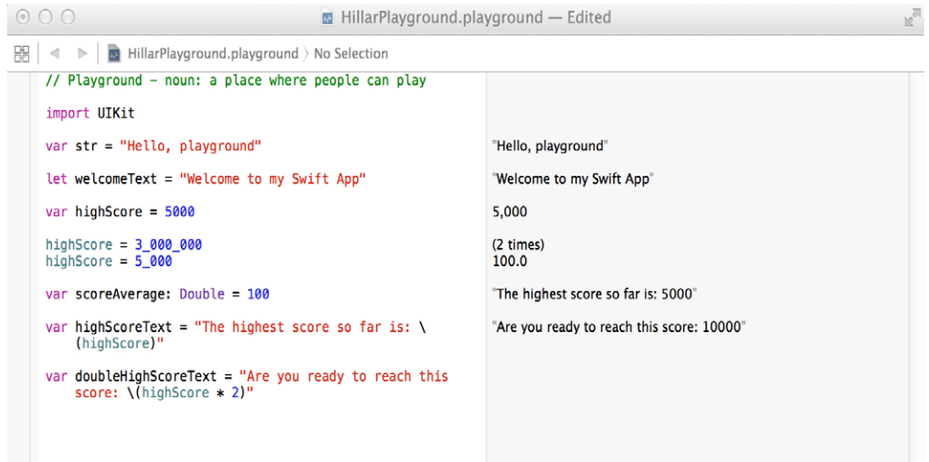

Figure 1: Checking the results of Swift code in the Playground..

Type Inference, Variables, and Constants

Swift doesn't require you to write semicolons at the end of every statement. The type inference mechanism determines the best type, and you can avoid specifying the type of value. You can declare constants with the `let` keyword and variables with `var`. For example, the following line creates an immutable `String` named `welcomeText`.

```
let welcomeText = "Welcome to my Swift App"
```

The following line uses the `var` keyword to create a `highScore` `Int` mutable variable.

```
var highScore = 5000
```

One nice feature is that you can use underscore (_) as a number separator to make numbers easier to read within the code. For example, the following lines assign 3000000 and 5000 to the previously declared `highScore` variable:

```
highScore = 3_000_000
highScore = 5_000
```

You can explicitly specify the desired type for a variable. The following line creates a `scoreAverage Double` mutable variable. In this case, the type inference mechanism would choose `Int` as the best type because the initial value is `100`. However, I want a `Double` to support future values, so I specify the type name in the variable declaration.

```
var scoreAverage: Double = 100
```

Swift makes it simple to include values in strings. You just need to use a backslash (\) and include an expression within parenthesis. For example, the following line:

```
var highScoreText = "The highest score so far is:
  \(highScore)"
```

Stores the following string in `highScoreText`:

```
"The highest score so far is: 5000"
```

The following line includes a more complex expression that doubles the value of `highScore`:

```
var doubleHighScoreText =
    "Are you ready to reach this score: \(highScore * 2)"
```

The line stores the following string in `doubleHighScoreText`:

```
"Are you ready to reach this score: 10000"
```

You can declare an optional value by adding a question mark after the type. The following lines declare the `optionalText` variable as an optional `String`. The initial value is `nil` and indicates that the value is missing. The next lines use `if` combined with `let` to retrieve the value in an operation known as optional binding. In the first case, the value is `nil` and the `println` line doesn't execute. Notice that `nil` means the absence of a value (and don't confuse it with the usage of `nil` in Objective-C). After the line that assigns a value to `optionalText` executes, the next `if` combined with `let` retrieves the value in text and prints an output with the retrieved `String`. Notice that the `Boolean` expression for the `if` statement doesn't use parenthesis because they are optional. However, braces around the body are always required even when the statement is just one line.

```
var optionalText: String?
if let text = optionalText {
    println("Optional text \(text)")
}

optionalText =
    "You must work harder to increase the score!"
if let text = optionalText {
    println("Optional text \(text)")
}
```

Generic Classes and Functions

Swift allows you to make generic forms of classes. The following lines show a simple example of a `Point3D` class that specifies `T` inside angle brackets to make a generic class with three variables: `x`, `y`, and `z` and a method that returns a `String` description. You can also use generics with functions, methods, enumerations, and structures.

```
class Point3D<T> {
    var x: T
    var y: T
    var z: T

    init(x: T, y: T, z: T) {
        self.x = x
        self.y = y
        self.z = z
    }

    func description() -> String {
        return "Point.X: \(self.x);
            Point.Y: \(self.y); Point.Z: \(self.z)."
    }
}
```

The initializer (`init`) sets up the class when you create an instance. In this case, the initializer assigns the values received as three arguments for `x`, `y`, and `z`. The arguments use the same name as the class variables, so it is necessary to use self to distinguish the `x`, `y`, and `z` properties from the arguments. If you need to add some cleanup code before the instance is deallocated, you can put some code in `deinit`.

The following lines create an instance of a `Point3D` with `Int` values and another `Point3D` with `Double` values. Then, two lines print the result of calling the description method for each `Point3D` instance.

```
var pointInt = Point3D<Int>(x: 10, y: 5, z: 5)
println(pointInt.description())
var pointDouble = Point3D<Double>(x:
    15.5, y: 5.5, z: 32.5)
println(pointDouble.description())
```

I'll use a `Point3D<Int>` instance to clarify the use of the `let` keyword with instances. The following line declares `pointIntConst` as a constant.

```
let pointIntConst = Point3D<Int>(x: 5, y: 5, z: 5)
```

Thus, you cannot change the value for `pointIntConst`; that is, you cannot assign a different instance of `Point3D<Int>` to it. However, you can change the values for the instance properties. For example, the following line is valid and changes the value of `x`.

```
pointIntConst.x = 3
```

As happens in many modern programming languages, functions are first-class citizens in Swift. You can use functions as arguments for other functions or methods. The following lines declare the `applyFunction` function that receives an array of `Int` (`list`) and a func-

tion (`condition`) that receives an `Int` and returns a `Bool` value. The function executes the received function (`condition`) for each element in the input array and adds the element to an output array whenever the result of the called function is `true`. This way, only the elements that meet the specified condition are going to appear in the resulting array of `Int`.

```
func applyFunction(list: [Int], condition:
        Int -> Bool) -> [Int] {
    var returnList = [Int]()
    for item in list {
        if condition(item) {
            returnList.append(item)
        }
    }

    return returnList
}
```

The following line declares a `divisibleBy10` function that receives an `Int` and returns a `Bool` indicating whether the received number is divisible by 10.

```
func divisibleBy10(number: Int) -> Bool {
    return number % 10 == 0
}
```

The following two lines declare an array of `Int` initialized with 9 numbers and call the `applyFunction` function with the array of `Int` and the `divisibleBy10` function as the arguments. The `divisibleBy10Numbers` array of `Int` will have the following values after the `applyFunction` function runs: `[10, 20, 30, 40, 50, 60]`.

```
var numbers = [10, 20, 30, 40, 50, 60, 63, 73, 43]
var divisibleBy10Numbers =
    applyFunction(numbers, divisibleBy10)
```

For Loops

The previous lines showed simple examples of the use of arrays. If you want to declare an immutable array, you can use the `let` keyword, also known as a `let` introducer, when you declare the array. The `applyFunction` function used the following `for` loop to iterate through the elements of the array:

```
for item in list
```

The global `enumerate` function allows you to easily access both the index and the value for each item in an array. This function returns a tuple for each element composed by the index and the value. The following lines use this function to print the index and the value of the elements in the `divisibleBy10Numbers` array of `Int`.

```
for (index, value) in enumerate(divisibleBy10Numbers) {
    println("Item index #:\(index). Value: \(value)")
}
```

The use of `(index, value)` instead of a single variable name assigns the first element of the tuple to `index`, and the second one to `value`. The following lines would produce the same results by accessing the tuple elements with .0 and .1 for the first and second element:

```
for tuple in enumerate(divisibleBy10Numbers) {
    println("Item index #:\(tuple.0). Value: \(tuple.1)")
}
```

The following lines use a classic `for` loop to make the `i` variable go from 0 up to the number of elements in the array minus 1. The results are the same as with the previous two loops:

```
for var i = 0; i < divisibleBy10Numbers.count; i++ {
    println("Item index #:\(i). Value:
        \(divisibleBy10Numbers[i])")
}
```

The following lines use another syntax to make the `i` variable go from 0 up to the number of elements in the array minus 1:

```
for i in 0..<divisibleBy10Numbers.count {
    println("Item index #:\(i). Value:
        \(divisibleBy10Numbers[i])")
}
```

If you replace `..<` with `...`, the `for` loop increments the variable until it reaches the value specified after `...` (inclusive instead of exclusive). The following lines produce the same results with `...` instead of `..<`.

```
for i in 0...divisibleBy10Numbers.count - 1 {
    println("Item index #:\(i). Value:
        \(divisibleBy10Numbers[i])")
}
```

Closures

As happens in most modern languages, Swift also supports closures. The following lines use a closure as an argument for the filter to generate the array with the numbers divisible by 10. The closure is the code surrounded with braces (`{}`). It uses the `in` keyword to separate the argument (`number: Int`) and the return type (`Bool`) from the body.

```
var divisibleBy10NumbersFiltered = numbers.filter({
    (number: Int) -> Bool in
    let result = number % 10 == 0
    return result
})
```

In this case, it is possible to omit the type for the closure's parameter and its return type. The following lines show a simplified version of the previously shown code that generates the same result. Notice that the closure code is really bare-bones and doesn't even include the `return` statement.

```
numbers.filter({
    number in number % 10 == 0
})
```

Tuples and Dictionaries

As you've seen, Swift supports tuples that group multiple values into a single compound value. The following lines define a `returnPreviousAndNext` function that receives an `Int` and returns a tuple with two `Int` values.

```
func returnPreviousAndNext(x: Int) -> (Int, Int) {
    return (x - 1, x + 1)
}
```

"Swift allows you to easily decompose the contents of a tuple into either separate constants or variables."

You can easily access the different elements of a tuple by adding a dot and the element number, starting with 0. For example, the following lines use `.0` and `.1` to display the `previous` and `next` values returned by the `returnPreviousAndNext` function.

```
var previousAndNext = returnPreviousAndNext(500)
println("The previous value is: \(previousAndNext.0)")
println("The next value is: \(previousAndNext.1)")
```

Swift allows you to easily decompose the contents of a tuple into either separate constants or variables. The following lines decompose the `previousAndNext` tuple into the `previous` and `next` constants, and then prints their values.

```
let (previous, next) = previousAndNext
println("The previous value is: \(previous)")
println("The next value is: \(next)")
```

You can also declare the names for the elements of the tuple in the `returnPreviousAndNext` function return type declaration. The following lines show a new version of the function that declares `previous` and `next` as the elements of the returning tuple.

```
func returnPreviousAndNext(x: Int) ->
        (previous: Int, next: Int) {
    return (x - 1, x + 1)
}
```

This way, you can easily add the following code to access the members of the tuple:

```
let previousAndNextTuple = returnPreviousAndNext(500)
println("The previous value is:
    \(previousAndNextTuple.previous)")
println("The next value is:
    \(previousAndNextTuple.next)")
```

The following lines declare a currencies dictionary with `String` key/value pairs.

```
var currencies = [
    "USD": "US Dollar",
```

```
    "EUR": "Euro",
    "CAD": "Canadian Dollar"
]
```

You can easily iterate through the dictionary key/value pairs by using a `for` loop (see Figure 2).

```
for (key, value) in currencies {
    println("Key: \(key), Value: \(value)")
}
```

Extensions

Extensions make it easy to add functionality to an existing type. The following lines add a `toPoint3DInt` method and a `logText` computed property to the `Int` type.

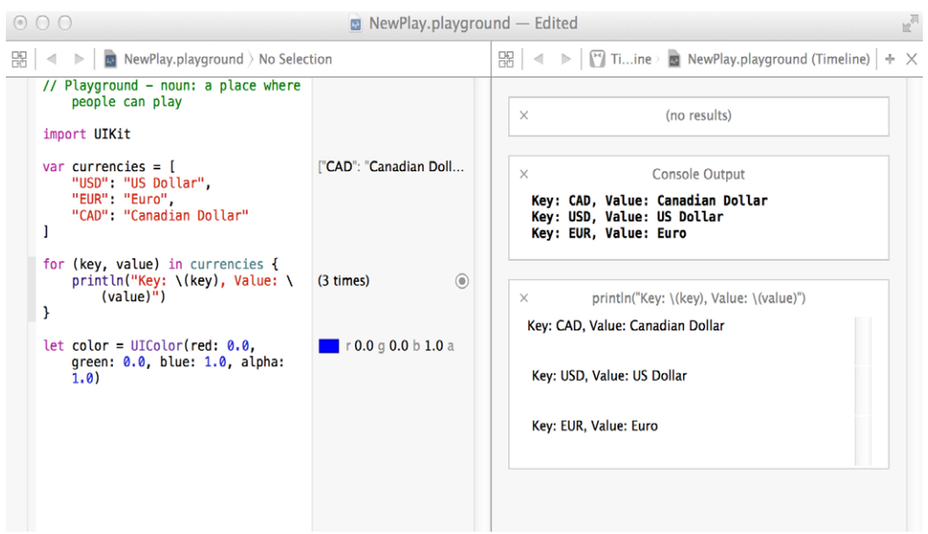

Figure 2: The Playground allows you to evaluate the execution of loops.

```
extension Int {
    func toPoint3DInt() -> Point3D<Int> {
        return Point3D<Int>(x: self, y: self, z: self)
    }
    var logText: String {
        return "Int value: \(self)"
    }
}
```

The following line creates a `Point3D<Int>` with `x`, `y`, and `z` set to `5` by using the `toPoint3DInt` extension method:

```
var point = 5.toPoint3DInt()
println(point.x)
```

The following line displays the contents of the `logText` computed property. The type inference makes `number` an `Int`.

```
var number = 7
println(7.logText)
```

Objective-C Factory Methods and Swift

Objective-C factory methods are mapped to initializers in Swift, so the creation of instances of Objective-C API objects feels natural. The following code shows an Objective-C line that initializes a `UIColor`.

```
UIColor *color =
    [UIColor colorWithRed:0.0 green:0.0
        blue:1.0 alpha:1.0];
```

The following line shows the equivalent code in Swift that initializes a `UIColor`:

```
let color = UIColor(red: 0.0, green: 0.0, blue: 1.0, alpha: 1.0)
```

The Swift compiler includes support for attributes that make it possible to enable Interface Builder features for Swift:

```
Outlets: @IBOutlet.
Actions: @IBAction.
Live and interactive custom view design:
    @IBDesignable and @IBInspectable.
```

The project structure for iOS apps for Swift is similar to the one used for Objective-C. The only difference is the programming language — and that Swift doesn't use headers.

Conclusion

There are many other interesting features in Swift that you might want to consider, such as the powerful and extensible enumerations and structures, the versatile switch statement, and the protocols. After years of dealing with Objective-C and suffering from its painful syntax, I'm sure Swift is not the best language you will discover, but you will want to learn it to replace Objective-C.

— Gastón Hillar is a senior contributing editor at Dr. Dobb's.

Comment

From the Vault

Using SQLite on Android

With a little care, SQLite can be used as a data store or full database on Android devices.

By Gaddo F. Benedetti @gadthecad

As with most platforms, Android gives us a few options to store data so that it persists even after we've terminated the application. Of the various ways we can do this, text files — either stored in the application's own files directory or to the phone's SD card — represent one approach. Preferences are also frequently used to store data because they can be both hidden from the user and persist as long as the application is installed. And while not strictly speaking in the same category, Assets can be useful for storing read-only data, too. Assets are essentially files that you bundle into the application package prior to compilation in the project assets folder, which can be accessed at runtime. I will take a closer look at these later.

Sometimes, however, we need to be able to carry out complex operations on persistent data, or the volume of data requires a more efficient method of management than a flat text file or preference entry will allow. This is where a mobile database comes in.

Android comes with SQLite (version 3.5.9+), which has been available on the platform since release 1.5 of the OS (the "Cupcake" release). For readers unfamiliar with SQLite (http://sqlite.org/), it is a self-contained, server-less, transactional SQL database. While it has its limitations, SQLite serves as a powerful tool in the Android developer's arsenal.

What I'll principally cover here is one way to use a SQLite database in Android, concentrating on its management; specifically, creation and update, rather than all the runtime operations.

Managing SQLite

To begin with, we can manage SQLite using a class that extends

This article originally appeared on drdobbs.com in 2012.

`SQLiteOpenHelper`, which comes with a constructor and two required methods; `onCreate` and `onUpgrade`.

Naturally, the first of these is executed when the constructor is instantiated; it is here that, via the superclass, we provide four important pieces of data:

- Context. This is the context of the application. It can be useful to set this in the constructor and store it locally for later use in other methods.
- Database name. This is the filename (as a `String`) of the physical database file being accessed.
- Cursor factory. The cursor factory in use, if any.
- Database Version. This is the version of your database (as an integer), which I'll discuss in more detail later. Your initial version should be 1.

For our example, we put these four pieces together and get the following:

```
class DB extends SQLiteOpenHelper {
    final static int DB_VERSION = 1;
        final static String DB_NAME = "mydb.s3db";
    Context context;

    public DB(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
        // Store the context for later use
        this.context = context;
    }
```

The constructor does two things. First, it checks whether the database exists and, if not, will call the `onCreate` method. Second, if the database does exist, it will check whether the existing database version number differs from the one implemented in the constructor, so as to determine if the database has been updated. If it has, the `onUpgrade` method will be called.

Additionally, as we now know that the `onCreate` method is called only when the database does not exist, it can be used as a handy way to determine if you're dealing with a first run of the application following installation. As such, you can use this method to call any other methods that you need executed only on the first run, such as EULA dialogs.

Let's look at the database itself. For the purposes of this article, I'm just going to use a very simple employee database with a SQL creation script as follows:

```
CREATE TABLE employees (
 _id INTEGER PRIMARY KEY AUTOINCREMENT,
 name TEXT NOT NULL,
 ext TEXT NOT NULL,
 mob TEXT NOT NULL,
 age INTEGER NOT NULL DEFAULT '0'
);
```

We can easily construct this by hard-coding and executing the creation SQL, line by line, in our code as follows:

```
@Override
public void onCreate(SQLiteDatabase database) {
```

```
    database.execSQL(
"CREATE TABLE employees ( _id INTEGER PRIMARY KEY "
    + "AUTOINCREMENT, name TEXT NOT NULL, ext TEXT NOT NULL, "
    + "mob TEXT NOT NULL, age INTEGER NOT NULL DEFAULT '0')");
}
```

As you can see, this can get quite unwieldy once the database reaches a certain size and complexity, so the ideal solution would be to bundle a SQL creation script as an asset file. To use this approach, you need to write a method that takes in a SQL script from the assets directory and parses it, executing it line by line:

```
@Override
public void onCreate(SQLiteDatabase database) {
executeSQLScript(database, "create.sql");
}

private void executeSQLScript(SQLiteDatabase database,
                              String dbname) {
ByteArrayOutputStream outputStream = new
                              ByteArrayOutputStream();
    byte buf[] = new byte[1024];
    int len;
        AssetManager assetManager = context.getAssets();
    InputStream inputStream = null;

    try{
        inputStream = assetManager.open(dbname);
        while ((len = inputStream.read(buf)) != -1) {
            outputStream.write(buf, 0, len);
        }
        outputStream.close();
        inputStream.close();

        String[] createScript =
            outputStream.toString().split(";");
        for (int i = 0; i < createScript.length; i++) {
                String sqlStatement =
                    createScript[i].trim();
            // TODO You may want to parse out comments here
            if (sqlStatement.length() > 0) {
                    database.execSQL(sqlStatement + ";");
                }
        }
    } catch (IOException e){
        // TODO Handle Script Failed to Load
    } catch (SQLException e) {
        // TODO Handle Script Failed to Execute
    }
}
```

While this is a more complex approach than simply executing each SQL statement for very simple databases, it quickly pays dividends once the database becomes more complex or you need to pre-populate it. You'll also see that I abstracted the creation into a separate method called `executeSQLScript` so that it can be reused in other situations, as I'll explain later on.

Interacting with the Database

Now that the database is created, I want to be able to interact with it. Here's a brief rundown:

The first step is to open the database and there are two ways to do this: using `getReadableDatabase()` and `getWritableDatabase()`. The former is faster and uses fewer resources. It should be used for anything that does not require writing or changing the database. The latter call is better suited to `INSERT`s, `UPDATE`s, and the like.

In Android, recordsets are returned from queries as `Cursor` objects. To carry out a query, use either the `query()` or `rawQuery()` method. For example, the following two calls return exactly the same data:

```
DB db = new DB(this);
SQLiteDatabase qdb = db.getReadableDatabase();
Cursor recordset1 = qdb.query("mytable", null, null, null,
     null, null, null);
Cursor recordset2 = qdb.rawQuery("SELECT * FROM mytable",
     null);
```

The first call uses a bewildering number of parameters. They are the table name, a string array of the column names, the `WHERE` clause, the selection arguments, the `GROUP BY` clause, the `HAVING` clause, and the `ORDER BY` clause, respectively. You'll note that setting many of these as null has the effect of their being treated as wildcards or omitted altogether.

Most of these parameters are fairly straightforward to anyone familiar with SQL. The selection arguments, however, require a little more explanation because they form a string array that interacts with the `WHERE` parameter, systematically replacing any "`?`" characters in the clause with a value from the array.

With the `rawQuery()` approach, there are only two parameters: the first is the SQL query, and the second is the selection argument — akin to those used in the query method. Selection arguments may be preferable to use with complex queries, such as those that use `JOIN`s.

Similarly, `INSERT`, `UPDATE`, `DELETE`, and a range of other common operations are handled with methods similar to `query()`; or, as with `rawQuery()`, they can be executed as raw SQL code using `execSQL()`.

"Once our application has reached a point whereby more than two versions have been released, we cannot presume that the user has been diligently upgrading all along."

Database Upgrades

Returning to management of the database, let's look at the tricky scenario of database upgrades. Over time, an app will likely change. New functionality may be added, or it may be better optimized. These changes, in turn, may lead to a requirement to upgrade the database schema and change the value for `DB_VERSION` in the updated application code to reflect this.

One potential problem is that replacing our database with a new version will end up invalidating the previous version and lead to the loss of data that was present in existing user installations. A second

problem is that, once our application has reached a point whereby more than two versions have been released, we cannot presume that the user has been diligently upgrading all along, so a simple upgrade from one version to the next may no longer work.

To deal with this, we already know that if we introduce a new database version, the `onUpgrade()` method will be called. So ideally, we can use our SQL script parser method and execute one or more update scripts.

Let's look at what we intend to change in Version 2 of our database in the example:

- Normalize the phone number data (extension, mobile) into a separate "numbers" table, which includes a numeric field to denote the type of phone number.
- Add a salary field to the employee table.

Using the Version 1 schema as a starting point, this can be handled by writing a SQL script that updates the schema and then populates it with the data from the older version:

```
CREATE TABLE numbers (
   _id INTEGER PRIMARY KEY AUTOINCREMENT,
   employid INTEGER NOT NULL,
   number TEXT NOT NULL,
   ntype INTEGER NOT NULL DEFAULT '0'
);
CREATE INDEX employid ON numbers(employid);

INSERT INTO numbers (employid, number, ntype) SELECT _id,
ext, 0 FROM employees;
INSERT INTO numbers (employid, number, ntype) SELECT _id,
      mob, 1
 FROM employees;

CREATE TABLE temp (
     _id INTEGER PRIMARY KEY AUTOINCREMENT,
     name TEXT NOT NULL,
     salary INTEGER NOT NULL DEFAULT '0'
);
INSERT INTO temp (_id, name) SELECT _id, name FROM
     employees;

DROP TABLE employees;
ALTER TABLE temp RENAME TO employees;
```

Naturally, the more complex the changes in your database schema, the more complex the script you'll need to write to handle this. SQLite has more limited support for SQL than many databases, so sometimes you'll need to devise workarounds for these limitations. For example, in the sample update script, I had to employ a temporary table as a workaround for the lack of `DROP COLUMN` functionality. Now that I have the SQL upgrade script, I need to handle how it is executed when the `onUpgrade` method is called. One approach is to do the following:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
if (newVersion > oldVersion) {
```

```
    switch (oldVersion) {
        case 1:
            executeSQLScript(database, "update_v2.sql");
        case 2:
            executeSQLScript(database, "update_v3.sql");
    }
}
}
```

"Each time you upgrade the database, you will only need to replace the create script with one that reflects the new schema and an update script to handle all possible upgrades."

There are two things to note in this code. The first is that I check to see whether the new database version is greater than the old one. I wrote the code this way because the `onUpgrade()` method will be called any time these two versions are different, leading to a situation in which version downgrades are also possible. Our code does not envisage this ever occurring, but an `else` clause and accompanying code could be added to handle this.

The second thing is that there are no break statements between the cases in our switch. This is because each script simply updates from one version to the next, meaning that an upgrade from Version 1 to 3 will first run the upgrade script from Version 1 to 2, then continue on to run the upgrade script from Version 2 to 3. If the database is already at Version 2, it will simply skip the first script and only run the upgrade script from Version 2 to 3.

Thus, each time you upgrade the database, you will only need to replace the create script with one that reflects the new schema (for new installs), and an update script (that handles only an update from the previous version) to handle all possible upgrades. Meanwhile, in our Java code, we need to update the value for `DB_VERSION` and, naturally, any operations that may be affected by the new database schema.

Conclusion

SQLite can be a very useful means to store and manage data in a persistent manner on Android devices. However, as with any database, care needs to be taken to administer it correctly, particularly with regard to version changes.

Using the script parser solution and saving this part of the application logic as a series of SQL scripts is an efficient and simple management technique to avoid having to write complex methods to the core application to handle each upgrade, thus allowing you to concentrate on the application's business logic instead.

— Gaddo F. Benedetti is presently an independent analyst-engineer who consults for various European clients in the telecom and Internet industries.

Comment

Dr. Dobb's

Andrew Binstock Editor In Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobbs
amy.stephens@ubm.com

Adrian Bridgwater News Editor, Dr. Dobb's
adrianbridgwater@gmail.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

CONTRIBUTING EDITORS

Scott Ambler
Mike Riley
Herb Sutter
Gastón Hillar

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

INFORMATIONWEEK

Rob Preston VP and Editor In Chief InformationWeek
rob.preston@ubm.com 516-562-5692

Chris Murphy Editor, InformationWeek
chris.murphy@ubm.com 414-906-5331

Lorna Garey Content Director, Reports, InformationWeek
lorna.garey@ubm.com 978-694-1681

Brian Gillooly VP and Editor In Chief, Events
brian.gillooly@ubm.com

INFORMATIONWEEK.COM

Laurianne McLaughlin Editor
laurianne.mchlaughlin@ubm.com
516-562-5336

Roma Nowak Senior Director, Online Operations and Production
roma.nowak@ubm.com 516-562-5274

Atiff Malik Director, Web Development
attif.malik@ubm.com

MEDIA KITS

http://createmarketingservices.com/

UBM TECH AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, karen.mcaleer@ubm.com

SALES CONTACTS—WEST

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, sandra.kupiec@ubm.com

Account Manager, Vesna Beso
(415) 947-6104, vensa.beso@ubm.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, matthew.meyer@ubm.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, winnie.ng@ubm.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, angele.leemoll@ubm.com

Marketing Manager, Monique Luttrell
(949) 223-3609, monique.luttrell@ubm.com

Program Manager, Nicole Schwartz
516-562-7684, nicole.schwartz@ubm.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mary.hyland@ubm.com

Account Manager, Tara Bradeen
(212) 600-3387, tara.bradeen@ubm.com

Account Manager, Jennifer Gambino
(516) 562-5651, jennifer.gambino@ubm.com

Account Manager, Elyse Cowen
(212) 600-3051, elyse.cowen@ubm.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kathleen.jurina@ubm.com

BUSINESS OFFICE

General Manager, Marian Dujmovits
United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030
(516) 562-5000

UBM Tech

UBM AMERICAS

Tim Cobbold, CEO, UBM

Sally Shankland, CEO, UBM Americas

Brian Field. COO, UBM Americas

Stacey Lisowski. SVP, People & Culture, UBM Americas

Aharon Shamash, Sr Managing Director, UBM Medica, UBM Canon & UBM CNY

Joshua Dome, EVP & Managing Director, UBM Canon

Lori Silva, Managing Director, UBM Connect NY

Jennifer Day, EVP & Managing Director, UBM Medica

Marco Pardi, President, Events, UBM Tech

Simon Carless, EVP, Game Network, Black Hat & Electronic Events, UBM Tech

Jaime Salazar, Managing Director, UBM Mexico

UBM TECH ONLINE COMMUNITIES

Bank Systems & Tech
Dark Reading
DataSheets.com
Designlines
Dr. Dobb's
EBN
EDN
EE Times
EE Times University
Embedded
Gamasutra
GAO
Heavy Reading
InformationWeek
IW Education
IW Government
IW Healthcare
Insurance & Technology
Light Reading
Network Computing
Planet Analog
Pyramid Research
TechOnline
Wall Street & Tech

UBM TECH EVENT COMMUNITIES

4G World
App Developers Conference
ARM TechCon
Big Data Conference
Black Hat
Cloud Connect
DESIGN
DesignCon
E2
Enterprise Connect
ESC
Ethernet Expo
GDC
GDC China
GDC Europe
GDC Next
GTEC
HDI Conference
Independent Games Festival
Interop
Mobile Commerce World
Online Marketing Summit
Telco Vision
Tower & Cell Summit

http://createmarketingservices.com/