Architektura Swift

Na początku przyda się ogólne wyobrażenie o tym, jak skonstruowany jest język Swift i jak wygląda program na iOS oparty na Swift. Tu przyjrzymy się ogólnej architekturze i naturze języka Swift. Kolejne sekcje uzupełnią szczegóły.

Podstawa bytu

Kompletne polecenie Swift to instrukcja. Plik tekstowy Swift składa się z wielu wierszy tekstu. Łamanie wierszy ma znaczenie. Typowy układ programu to jedna instrukcja, jedna linia:

```
print("hello")

print("world")
```

(Polecenie print zapewnia natychmiastową informację zwrotną w konsoli Xcode.) Możesz połączyć więcej niż jedną instrukcję w wierszu, ale wtedy musisz umieścić między nimi średnik:

```
print("hello"); print("world")
```

Możesz wstawić średnik na końcu wyrażenia, które jest ostatnie lub samotne w wierszu, ale nikt tego nigdy nie robi (chyba że z przyzwyczajenia, ponieważ C i Objective-C wymagają średnika):

```
print("hello");

print("world");
```

I odwrotnie, pojedyncza instrukcja może zostać podzielona na wiele wierszy, aby długie instrukcje nie stały się długimi wierszami. Ale powinieneś starać się to robić w rozsądnych miejscach, aby nie zmylić Swifta. Po nawiasie otwierającym jest dobre miejsce:

```
print(

"world")
```

Komentarze to wszystko po dwóch ukośnikach w linii (tzw. komentarze w stylu C++):

```
print("world") // to jest komentarz, więc Swift go ignoruje
```

Możesz także dołączyć komentarze w /*...*/, tak jak w C. W przeciwieństwie do C, komentarze w stylu C mogą być zagnieżdżane. Wiele konstrukcji w języku Swift używa nawiasów klamrowych jako ograniczników:

```
class Dog {

func bark() {

print("woof")

}

}
```

Zgodnie z konwencją zawartość nawiasów klamrowych jest poprzedzona i poprzedzona podziałami wierszy. Xcode pomoże narzucić tę konwencję, ale prawda jest taka, że Swift to nie obchodzi, a takie układy są legalne (i czasami są wygodniejsze):

```
class Dog { func bark() { print("hau") }}
```

Swift to język kompilowany. Oznacza to, że Twój kod musi zostać zbudowany - przechodząc przez kompilator i przekształcany z tekstu w jakąś formę niższego poziomu, zrozumiałą dla komputera - zanim będzie mógł działać i faktycznie robić to, co mówi. Kompilator Swift jest bardzo rygorystyczny; w trakcie pisania programu często będziesz próbował budować i uruchamiać, tylko po to, by odkryć, że w ogóle nie możesz nawet zbudować, ponieważ kompilator zasygnalizuje jakiś błąd, który będziesz musiał naprawić, jeśli chcesz kod do uruchomienia. Rzadziej kompilator da ci ostrzeżenie; kod może się uruchomić, ale ogólnie powinieneś traktować ostrzeżenia poważnie i naprawić wszystko, o czym ci mówią. Ścisłość kompilatora jest jedną z największych zalet Swift i zapewnia kodowi dużą miarę sprawdzanej poprawności, nawet przed jego uruchomieniem. Komunikaty o błędach i ostrzeżenia kompilatora Swift wahają się od wnikliwych przez tępe do wręcz mylących. Czasami będziesz wiedział, że coś jest nie tak z wierszem kodu, ale kompilator Swift może nie mówić dokładnie, co jest nie tak, a nawet w którym miejscu wiersza, aby skupić twoją uwagę. Moja rada w takich sytuacjach jest taka, aby podzielić wiersz na kilka wierszy prostszego kodu, aż dojdziesz do punktu, w którym możesz ustalić, na czym polega problem. Spróbuj pokochać kompilator, nawet jeśli jego wiadomości wydają się tajemnicze; pamiętaj, wie więcej niż ty.

Wszystko jest obiektem?

W Swift „wszystko jest obiektem”. To przechwałka wspólna dla wielu nowoczesnych języków obiektowych, ale co to oznacza? Cóż, to zależy od tego, co rozumiesz przez „obiekt” – i co rozumiesz przez „wszystko”. Zacznijmy od ustalenia, że obiekt, mówiąc z grubsza, jest czymś, do czego możesz wysłać wiadomość. Wiadomość, z grubsza rzecz biorąc, jest imperatywną instrukcją. Na przykład możesz wydawać komendy psu: „Kora!” "Siedzieć!" W tej analogii te frazy są wiadomościami, a pies jest obiektem, do którego wysyłasz te wiadomości. W Swift składnia wysyłania wiadomości to notacja z kropkami. Zaczynamy od obiektu; potem jest kropka (kropka); potem jest wiadomość. (Niektóre wiadomości są również poprzedzone nawiasami, ale na razie je zignoruj; pełna składnia wysyłania wiadomości jest jednym z tych szczegółów, które uzupełnimy później.) Jest to poprawna składnia Swift:

```
fido.kora()

rover.sit()
```

Nawiasem mówiąc, kropka jest również dobrym miejscem do przełamania długiej linii (przed kropką):

```
fido

.bark()
```

Idea, że wszystko jest obiektem, jest sposobem sugerowania, że nawet „prymitywne” jednostki językowe mogą być przesyłane wiadomościami. Weźmy na przykład 1. Wydaje się, że jest to dosłowna cyfra i nic więcej. Nie zaskoczy Cię, jeśli kiedykolwiek używałeś dowolnego języka programowania, że możesz mówić takie rzeczy w Swift:

```
let sum = 1 + 2
```

Zaskakujące jest jednak odkrycie, że po 1 może następować kropka i komunikat. Jest to legalne i znaczące w Swift (nie martw się, co to właściwie oznacza):

```
let s = 1.description
```

Ale możemy iść dalej. Wróć do niewinnie wyglądającego 1 + 2 z naszego wcześniejszego kodu. Okazuje się, że jest to właściwie rodzaj sztuczki składniowej, wygodny sposób wyrażania i ukrywania tego, co naprawdę się dzieje. Tak jak 1 jest w rzeczywistości przedmiotem, + jest w rzeczywistości wiadomością;

ale jest to wiadomość o specjalnej składni (składnia operatora). W Swift każdy rzeczownik jest dopełnieniem, a każdy czasownik jest wiadomością. Być może ostatecznym sprawdzianem, czy coś jest obiektem w Swift, jest to, czy możesz to zmodyfikować. Typ obiektu może być rozszerzony w Swift, co oznacza, że możesz zdefiniować własne komunikaty dla tego typu. Na przykład nie możesz normalnie wysłać wiadomości sayHello na numer, ale możesz zmienić typ numeru, aby:

```
extension Int {

func sayHello() {

print("Hello, I'm \(self)")

}

}

1.sayHello() // outputs: "Hello, I'm 1"
```

W Swift 1 jest więc obiektem. W niektórych językach, takich jak Objective-C, wyraźnie tak nie jest; jest to wbudowany typ danych „podstawowy" lub skalarny. Więc rozróżnienie, jakie tu przedstawiamy, dotyczy z jednej strony typów obiektów, a z drugiej skalarów. W Swift nie ma skalarów; wszystkie typy są ostatecznie typami obiektowymi. To właśnie oznacza „wszystko jest obiektem".

Trzy Smaki Obiektu

Jeśli znasz Objective-C lub inny język zorientowany obiektowo, możesz być zaskoczony wyobrażeniem Swifta o tym, czym jest obiekt 1. W wielu językach, takich jak Objective-C, obiekt jest klasą lub instancją klasy (później wyjaśnię, czym jest instancja). Swift ma klasy, ale 1 w Swift nie jest klasą ani instancją klasy: typ 1, mianowicie Int, jest strukturą, a 1 jest instancją struktury. A Swift ma jeszcze jeden rodzaj obiektu, do której możesz wysyłać wiadomości, zwany enum. Tak więc Swift ma trzy rodzaje typów obiektów: klasy, struktury i wyliczenia. Lubię je nazywać trzema smakami typu obiektu. Dokładnie to, czym różnią się od siebie, pojawi się w odpowiednim czasie. Ale wszystkie one są zdecydowanie typami obiektowymi, a ich podobieństwa do siebie są znacznie silniejsze niż ich różnice. Na razie pamiętaj tylko, że te trzy smaki istnieją. (Ok, skłamałem. Nowość w Swifcie 5.5, właściwie jest czwarty smak, aktorzy. Aktorzy są wysoce wyspecjalizowani i opowiem o nich później. Do tego czasu po prostu udawaj, że są trzy smaki.) Fakt, że struct lub enum to typ obiektu w Swift zaskoczy Cię, szczególnie jeśli znasz Obiective-C. Obiective-C ma struktury i wyliczenia, ale nie są obiektami. W szczególności konstrukcje Swift są znacznie ważniejsze i wszechobecne niż konstrukcje celu C. Ta różnica między tym, jak Swift widzi struktury i wyliczenia, a tym, jak Cel-C je widzi, może mieć znaczenie, gdy rozmawiasz z Cocoa.

Zmienne

Zmienna to nazwa obiektu. Technicznie odnosi się do obiektu; jest to odniesienie do obiektu. Nietechnicznie można o nim myśleć jako o pudełku na buty, w którym umieszcza się przedmiot. Przedmiot może ulec zmianie lub może zostać zastąpiony w pudełku po butach innym przedmiotem, ale nazwa ma swoją integralność. Obiekt, do którego odnosi się zmienna, jest wartością zmiennej. W Swift żadna zmienna nie pojawia się niejawnie; wszystkie zmienne muszą być zadeklarowane. Jeśli potrzebujesz nazwy dla czegoś, musisz powiedzieć „Tworzę nazwę". Robisz to za pomocą jednego z dwóch słów kluczowych: let lub var. W Swifcie deklaracji zwykle towarzyszy inicjalizacja - używasz znaku równości, aby od razu nadać zmiennej wartość, jako część deklaracji. Są to zarówno deklaracje zmiennych (i inicjalizacje):

```
let one = 1

var two = 2
```

Gdy nazwa istnieje, możesz jej używać. Możemy zmienić wartość dwóch tak, aby była taka sama jak wartość jedynki:

```
let one = 1

var two = 2

two = one
```

Ostatni wiersz tego kodu zawiera zarówno nazwę 1, jak i nazwę 2 zadeklarowane w pierwszych dwóch wierszach: nazwa 1, po prawej stronie znaku równości, służy jedynie do odniesienia się do wartości wewnątrz pudełka po butach (czyli 1); ale nazwa dwa, po lewej stronie znaku równości, służy do zastąpienia wartości w pudełku po butach dwa. Przed powiedzeniem dwa = jeden, wartość dwa wynosiła 2; potem jest to 1. Instrukcja z nazwą zmiennej po lewej stronie znaku równości nazywana jest przypisaniem, a znak równości jest operatorem przypisania. Znak równości nie jest stwierdzeniem równości, jak to mogłoby być w formule algebraicznej; to jest polecenie. Oznacza to: „Zdobądź wartość tego, co jest po mojej prawej stronie i użyj jej, aby zastąpić wartość tego, co jest po lewej stronie mnie”. Te dwa rodzaje deklaracji zmiennych różnią się tym, że nazwa zadeklarowana za pomocą let nie może mieć zastąpionej wartości początkowej. Zmienna zadeklarowana za pomocą let jest stałą; jego wartość jest przypisywana raz i pozostaje. To się nawet nie skompiluje:

```
let one = 1

var two = 2

one = two // compile error
```

Zawsze można zadeklarować nazwę za pomocą var, aby zapewnić sobie największą elastyczność, ale jeśli wiesz, że nigdy nie zamierzasz zastąpić początkowej wartości zmiennej, lepiej użyć let, ponieważ pozwala to Swiftowi zachowywać się wydajniej. W rzeczywistości kompilator Swift zwróci Twoją uwagę na każdy przypadek użycia var, w którym mogłeś użyć let, oferując zmianę tego za Ciebie. Zmienne mają również typ. Ten typ jest ustalany, gdy zmienna jest zadeklarowana i nigdy nie może ulec zmianie. To się nie skompiluje:

```
var two = 2

two = "hello" // compile error
```

Gdy dwa zostaną zadeklarowane i zainicjowane jako 2, jest to liczba (właściwie mówiąc, Int) i zawsze tak musi być. Możesz zastąpić jego wartość 1, ponieważ to również jest Int, ale nie możesz zastąpić jej wartości „hello”, ponieważ jest to string (prawidłowo mówiąc, String) - a String nie jest Int. Zmienne dosłownie żyją własnym życiem, a dokładniej, własnym życiem. Dopóki zmienna istnieje, utrzymuje swoją wartość przy życiu. Zatem zmienna może być nie tylko sposobem na wygodne nazwanie czegoś, ale także sposobem na jej zachowanie. Więcej na ten temat powiem później.

OSTRZEŻENIE

Zgodnie z konwencją nazwy typów, takie jak String lub Int (lub Dog) zaczynają się od dużej litery; nazwy zmiennych zaczynają się od małej litery. Nie łam tej konwencji. Jeśli to zrobisz, twój kod może się nadal dobrze kompilować i działać, ale osobiście wyślę agentów do twojego domu, aby zdjęli ci rzepki w środku nocy.

Funkcje

Kod wykonywalny, taki jak fido.bark() lub one = two lub print("hello"), nie może przejść gdziekolwiek w twoim programie. Brak zrozumienia tego faktu jest częstym błędem początkujących i może skutkować tajemniczym komunikatem o błędzie kompilacji, takim jak „Oczekiwana deklaracja". Ogólnie kod wykonywalny musi znajdować się w ciele funkcji. Funkcja to zestaw kodu, który można uruchomić jako zestaw. Jego ciało jest ograniczone kręconymi nawiasami klamrowymi. Zazwyczaj funkcja ma nazwę i pobiera tę nazwę poprzez deklarację funkcji. Składnia deklaracji funkcji to kolejny z tych szczegółów, które zostaną uzupełnione później, ale oto przykład:

```
func go() {

let one = 1

var two = 2

two = one

}
```

To opisuje sekwencję rzeczy do zrobienia - zadeklaruj jedną, zadeklaruj dwa, zmień wartość dwa, aby dopasować wartość jednego - i nadaje tej sekwencji nazwę, idź; ale nie wykonuje sekwencji. Sekwencja jest wykonywana, gdy ktoś wywoła funkcję. Można więc powiedzieć gdzie indziej:

```
go()
```

To jest komenda do funkcji, którą powinna faktycznie uruchomić. Ale znowu, to polecenie jest samo w sobie kodem wykonywalnym, więc nie może też żyć samodzielnie. Może żyć w ciele o innej funkcji:

```
func doGo() {

go()

}
```

Ale poczekaj! To robi się trochę szalone. To również jest tylko deklaracją funkcji; aby ją uruchomić, ktoś musi wywołać doGo, mówiąc doGo() - i to też jest kod wykonywalny.

Wygląda to na rodzaj nieskończonej regresji; wygląda na to, że żaden z naszych kodów nigdy się nie uruchomi. Jeśli cały kod wykonywalny musi znajdować się w funkcji, kto powie dowolnej funkcji, aby została uruchomiona? Początkowy impuls musi skądś pochodzić. W prawdziwym życiu na szczęście ten problem regresji nie pojawia się. Pamiętaj, że Twoim celem jest napisanie aplikacji na iOS. Twoja aplikacja zostanie uruchomiona na urządzeniu z systemem iOS (lub symulatorze) przez środowisko wykonawcze, które już chce wywołać określone funkcje. Więc zaczynasz od napisania specjalnych funkcji, o których wiesz, że samo środowisko wykonawcze będzie wywoływać. Daje to Twojej aplikacji sposób na rozpoczęcie pracy i zapewnia miejsca do umieszczania funkcji, które będą wywoływane przez środowisko wykonawcze w kluczowych momentach.

WSKAZÓWKA: Swift ma również specjalną regułę, że wyjątkowo plik o nazwie main.swift może mieć kod wykonywalny na najwyższym poziomie, poza treścią dowolnej funkcji, i jest to kod, który faktycznie jest uruchamiany podczas działania programu. Możesz skonstruować swoją aplikację za pomocą pliku main.swift, ale generalnie nie musisz. W dalszej części tej części założę, że nie znajdujemy się w pliku main.swift.

Struktura pliku Swift

Program Swift może składać się z jednego pliku lub wielu plików. W Swift plik jest znaczącą jednostką i istnieją określone zasady dotyczące struktury kodu Swift, który może się do niego dostać. Tylko niektóre rzeczy mogą znajdować się na najwyższym poziomie pliku Swift - głównie następujące:

Instrukcja importu modułów

Moduł jest jednostką jeszcze wyższego poziomu niż plik. Moduł może składać się z wielu plików, z których wszystkie mogą widzieć się nawzajem automatycznie. Pliki Twojej aplikacji należą do jednego modułu i mogą się nawzajem widzieć. Ale moduł nie może zobaczyć innego modułu bez instrukcji import. W ten sposób możesz rozmawiać z Cocoa w programie na iOS: pierwsza linia twojego pliku mówi import UIKit.

Deklaracje zmiennych

Zmienna zadeklarowana na najwyższym poziomie pliku jest zmienną globalną: cały kod w dowolnym pliku będzie mógł ją zobaczyć i uzyskać do niej dostęp, bez jawnego wysyłania wiadomości do dowolnego obiektu.

Deklaracje funkcji

Funkcja zadeklarowana na najwyższym poziomie pliku jest funkcją globalną: cały kod w dowolnym pliku będzie mógł ją zobaczyć i wywołać, bez jawnego wysyłania komunikatu do dowolnego obiektu.

Deklaracje typu obiektu

Deklaracja klasy, struktury lub wyliczenia. Jest to legalny plik Swift zawierający na swoim najwyższym poziomie (tylko po to, aby zademonstrować, że można to zrobić) instrukcję importu, deklarację zmiennej, deklarację funkcji, deklarację klasy, deklarację struktury i deklarację enum:

```
import UIKit

var one = 1

func changeOne() {

}

class Manny {

}

struct Moe {

}

enum Jack {

}
```

To bardzo głupi i w większości pusty przykład, ale pamiętaj, że naszym celem jest zbadanie części języka i struktury pliku, a przykład je pokazuje. Tyle jeśli chodzi o najwyższy poziom pliku. Ale teraz porozmawiajmy o tym, co może wejść do nawiasów klamrowych, które widzimy w naszym przykładzie. Okazuje się, że one również mogą mieć w sobie deklaracje zmiennych, deklaracje funkcji i deklaracje typu obiektu! Rzeczywiście, wszelkie strukturalne nawiasy klamrowe mogą zawierać takie deklaracje.

Ale co z kodem wykonywalnym? Zauważysz, że nie powiedziałem, że kod wykonywalny może znajdować się na najwyższym poziomie pliku. To dlatego, że nie może! Tylko treść funkcji może zawierać kod wykonywalny. Instrukcja taka jak jeden = dwa lub print("hello") jest kodem wykonywalnym i nie może przejść na najwyższy poziom pliku. Ale w naszym poprzednim przykładzie funkcja func changeOne() jest deklaracją funkcji, więc kod wykonywalny może zostać umieszczony w nawiasach klamrowych, ponieważ stanowią one treść funkcji:

```
var one = 1

// executable code can't go here!

func changeOne() {

let two = 2 // executable code

one = two // executable code

}
```

Podobnie kod wykonywalny nie może znajdować się bezpośrednio w nawiasach klamrowych towarzyszących deklaracji klasy Manny; to jest najwyższy poziom deklaracji klasy, a nie treść funkcji. Ale deklaracja klasy może zawierać deklarację funkcji, a ta deklaracja funkcji może zawierać kod wykonywalny:

```
class Manny {

let name = "manny"

// executable code can't go here!

func sayName() {

print(name) // executable code

}

}
```

To bardzo głupi i w większości pusty przykład, ale pamiętaj, że naszym celem jest zbadanie części języka i struktury pliku, a przykład je pokazuje. Tyle jeśli chodzi o najwyższy poziom pliku. Ale teraz porozmawiajmy o tym, co może wejść do nawiasów klamrowych, które widzimy w naszym przykładzie. Okazuje się, że one również mogą mieć w sobie deklaracje zmiennych, deklaracje funkcji i deklaracje typu obiektu! Rzeczywiście, wszelkie strukturalne nawiasy klamrowe mogą zawierać takie deklaracje. Ale co z kodem wykonywalnym? Zauważysz, że nie powiedziałem, że kod wykonywalny może znajdować się na najwyższym poziomie pliku. To dlatego, że nie może! Tylko treść funkcji może zawierać kod wykonywalny. Instrukcja taka jak jeden = dwa lub print("hello") jest kodem wykonywalnym i nie może przejść na najwyższy poziom pliku. Ale w naszym poprzednim przykładzie funkcja func changeOne() jest deklaracją funkcji, więc kod wykonywalny może zostać umieszczony w nawiasach klamrowych, ponieważ stanowią one treść funkcji. Podobnie kod wykonywalny nie może znajdować się bezpośrednio w nawiasach klamrowych towarzyszących deklaracji klasy Manny; to jest najwyższy poziom deklaracji klasy, a nie treść funkcji. Ale deklaracja klasy może zawierać deklarację funkcji, a ta deklaracja funkcji może zawierać kod wykonywalny. Podsumowując, Przykład 1 to legalny plik Swift, schematycznie ilustrujący możliwości konstrukcyjne.

```
import UIKit
```

```
var one = 1
func changeOne() {
let two = 2
func sayTwo() {
print(two)
}
class Klass {}
struct Struct {}
enum Enum {}
one = two
}
class Manny {
let name = "manny"
func sayName() {
print(name)
}
class Klass {}
struct Struct {}
enum Enum {}
}
struct Moe {
let name = "moe"
func sayName() {
print(name)
}
class Klass {}
struct Struct {}
enum Enum {}
}
enum Jack {
var name : String {
```

```
return "jack"

}

func sayName() {

print(name)

}

class Klass {}

struct Struct {}

enum Enum {}

}
```

Oczywiście możemy rekursywać w dół tak daleko, jak tylko chcemy: możemy mieć deklarację klasy zawierającą deklarację klasy zawierającą deklarację klasy i tak dalej. Ale jestem pewien, że masz już pomysł, więc nie ma sensu dalej ilustrować.

Zakres i żywotność

W programie Swift rzeczy mają zakres. Odnosi się to do ich zdolności do bycia widzianymi przez inne rzeczy. Rzeczy są zagnieżdżone w innych rzeczach, tworząc zagnieżdżoną hierarchię rzeczy. Zasada jest taka, że rzeczy mogą widzieć rzeczy na swoim własnym poziomie i na wyższym poziomie je zawierającym. Poziomy to:

* Moduł to zakres.

* Plik to zakres.

* Nawiasy klamrowe są zakresem.

Kiedy coś jest deklarowane, jest to deklarowane na pewnym poziomie w tej hierarchii. Jego miejsce w hierarchii – jego zakres – decyduje o tym, czy może być widziany przez inne rzeczy. Spójrz ponownie na Przykład 1. Wewnątrz deklaracji Manny'ego znajduje się deklaracja zmiennej name i deklaracja funkcji sayName; kod wewnątrz nawiasów klamrowych sayName może widzieć elementy poza tymi nawiasami klamrowymi na wyższym poziomie zawierającym i dlatego może zobaczyć zmienną name. Podobnie kod wewnątrz ciała funkcji changeOne może zobaczyć jedną zmienną zadeklarowaną na najwyższym poziomie pliku; w rzeczywistości wszystko w tym pliku może zobaczyć jedną zmienną zadeklarowaną na najwyższym poziomie pliku. Zakres jest zatem bardzo ważnym sposobem udostępniania informacji. Dwie różne funkcje zadeklarowane w Manny'm byłyby w stanie zobaczyć nazwę zadeklarowaną na najwyższym poziomie Manny'ego. Kod wewnątrz Jacka i kod wewnątrz Moe widzą ten zadeklarowany na najwyższym poziomie pliku. Rzeczy mają też czas życia, który jest w rzeczywistości równoznaczny z ich zakresem. Rzecz żyje tak długo, jak żyje jej otaczający zakres. W przykładzie 1-1 zmienna jeden żyje tak długo, jak długo żyje plik — czyli tak długo, jak działa program. Jest globalny i trwały. Ale nazwa zmiennej zadeklarowana na najwyższym poziomie Manny'ego istnieje tylko tak długo, jak istnieje instancja Manny'ego (za chwilę opowiem, co to oznacza). Rzeczy zadeklarowane na głębszym poziomie żyją jeszcze krócej. Rozważ ten kod:

```
func silly() {

if true {
```

```
class Cat {}

var one = 1

one = one + 1

}

}
```

Ten kod jest głupi, ale jest legalny: pamiętaj, powiedziałem, że deklaracje zmiennych, deklaracje funkcji i deklaracje typu obiektu mogą pojawiać się w dowolnych strukturalnych nawiasach klamrowych. W tym kodzie klasa Cat i zmienna jeden nawet nie zaistnieją, dopóki ktoś nie wywoła funkcji głupiej, a nawet wtedy będą istnieć tylko w krótkiej chwili, w której ścieżka wykonania kodu przechodzi przez konstrukcję if. Załóżmy, że wywoływana jest funkcja głupia; ścieżka wykonania następnie wchodzi do konstrukcji if. Tutaj Kot zostaje zadeklarowany i powstaje; wtedy zostaje się ogłoszony i powstaje; następnie wykonywana jest linia wykonywalna jeden = jeden + 1; a potem luneta się kończy - i zarówno Cat, jak i jeden znikają w obłoku dymu. I przez całe swoje krótkie życie Cat i jeden byli całkowicie niewidoczni dla reszty programu. (Widzisz dlaczego?)

Składowe obiektu

Wewnątrz trzech typów obiektów (class, struct i enum) rzeczy zadeklarowane na najwyższym poziomie mają specjalne nazwy, głównie ze względów historycznych. Posłużmy się przykładem klasy Manny:

```
class Manny {

let name = "manny"

func sayName() {

print(name)

}

}
```

W tym kodzie

* name jest zmienną zadeklarowaną na najwyższym poziomie deklaracji obiektu, dlatego nazywana jest właściwością tego obiektu.

* sayName to funkcja zadeklarowana na najwyższym poziomie deklaracji obiektu, dlatego nazywana jest metodą tego obiektu.

Obiekty zadeklarowane na najwyższym poziomie deklaracji obiektu - właściwości, metody i wszelkie obiekty zadeklarowane na tym poziomie - są zbiorczo składowymi tego obiektu. Składowe mają szczególne znaczenie, ponieważ definiują wiadomości, które możesz wysyłać do tego obiektu!

Przestrzenie nazw

Przestrzeń nazw to nazwany region programu. Nazwy rzeczy wewnątrz przestrzeni nazw nie mogą być osiągane przez rzeczy spoza niej bez uprzedniego przejścia przez barierę wypowiedzenia nazwy tego regionu. To dobrze, ponieważ pozwala na używanie tej samej nazwy w różnych miejscach bez konfliktu. Oczywiście przestrzenie nazw i zakresy są ściśle powiązanymi pojęciami. Przestrzenie nazw pomagają wyjaśnić znaczenie deklarowania obiektu na najwyższym poziomie obiektu, na przykład:

```
class Manny {

class Klass {}

}
```

Ten sposób deklarowania Klass sprawia, że Klass jest typem zagnieżdżonym. Skutecznie „ukrywa" Klassa wewnątrz Manny'ego. Manny to przestrzeń nazw! Kod wewnątrz Manny'ego może zobaczyć (i powiedzieć) bezpośrednio Klass. Ale kod spoza Manny'ego nie może tego zrobić. Musi jawnie określić przestrzeń nazw, aby przejść przez barierę, którą reprezentuje przestrzeń nazw. Aby to zrobić, musi najpierw wypowiedzieć imię Manny'ego, a następnie kropkę, a następnie termin Klass. Krótko mówiąc, musi to powiedzieć Manny.Klass. Przestrzeń nazw sama w sobie nie zapewnia tajemnicy ani prywatności; to wygoda. W naszym pierwszym przykładzie dałem Manny'emu klasę Klass, a także Moe klasę Klass. Ale nie są w konflikcie, ponieważ znajdują się w różnych przestrzeniach nazw i w razie potrzeby mogę je rozróżnić jako Manny.Klass i Moe.Klass. Nie umknie twojej uwadze, że składnia dla jawnego zanurzenia się w przestrzeni nazw to składnia z notacją kropkową wysyłania wiadomości. W rzeczywistości są tym samym. W efekcie wysyłanie wiadomości pozwala zajrzeć do zakresów, których nie można zobaczyć w inny sposób. Kod wewnątrz Moe nie może automatycznie zobaczyć Klass zadeklarowanej w Manny, ale może to zobaczyć, wykonując jeden prosty dodatkowy krok, a mianowicie mówiąc o Manny.Klass. Może to zrobić, ponieważ widzi Manny'ego (ponieważ Manny jest zadeklarowany na poziomie, który widzi kod wewnątrz Moe).

Moduły

Przestrzenie nazw najwyższego poziomu to moduły. Twoja aplikacja jest modułem, a zatem przestrzenią nazw; domyślną nazwą tej przestrzeni nazw jest nazwa aplikacji. Jeśli moja aplikacja nazywa się MyApp, to jeśli zadeklaruję klasę Manny na najwyższym poziomie pliku, jej prawdziwe imię to MyApp.Manny. Ale zwykle nie muszę używać tej prawdziwej nazwy, ponieważ mój kod znajduje się już w tej samej przestrzeni nazw i widzę bezpośrednio imię Manny. Kiedy importujesz moduł, wszystkie deklaracje najwyższego poziomu tego modułu stają się również widoczne dla twojego kodu, bez konieczności używania przestrzeni nazw modułu w celu odwoływania się do nich. Na przykład framework Cocoa's Foundation, w którym mieści ai1) NSString, jest modułem. Kiedy programujesz iOS, powiesz import Foundation (lub, co bardziej prawdopodobne, powiesz import UIKit, który sam importuje Foundation), co pozwoli ci mówić o NSString bez wypowiadania Foundation.NSString . Sam Swift jest zdefiniowany w module - module Swift. Ale nie musisz go importować, ponieważ Twój kod zawsze niejawnie importuje moduł Swift. Możesz to zrobić wprost, uruchamiając plik z wierszem import Swift; nie ma takiej potrzeby, ale też nie szkodzi. Ten fakt jest ważny, ponieważ rozwiązuje wielką zagadkę: skąd pochodzą rzeczy takie jak druk i dlaczego można ich używać poza jakimkolwiek przesłaniem do dowolnego obiektu? print jest w rzeczywistości funkcją zadeklarowaną na górze poziomu modułu Swift, a Twój kod może zobaczyć deklaracje najwyższego poziomu modułu Swift, ponieważ importuje on Swift. Funkcja drukowania staje się, jeśli chodzi o twój kod, zwykłą funkcją najwyższego poziomu, jak każda inna; jest globalny dla twojego kodu, a twój kod może o nim mówić bez określania jego przestrzeni nazw. Możesz określić jego przestrzeń nazw - jest całkowicie legalne mówienie takich rzeczy jak Swift.print("cześć") - ale prawdopodobnie nigdy tego nie zrobisz, ponieważ nie będziesz musiał. Jednak Twój własny moduł aplikacji przysłania wszystkie importowane moduły. Oznacza to, że jeśli zadeklarujesz termin identyczny z terminem importowanym, stracisz magiczną zdolność używania importowanego terminu bez określania przestrzeni nazw. Gdybyś miał zadeklarować własną funkcję drukowania, skutecznie ukryłoby to funkcję drukowania Swift; nadal możesz wywołać funkcję Swift pr int, ale teraz musisz jawnie użyć Swift.print z przestrzenią nazw. Podobnie, gdybyś zadeklarował własny typ String, cały kod, który odwołuje się do typu String Swifta,

wymagałby powiedzenia Swift.String. Prawie zawsze tego rodzaju rzeczy są przypadkowe i wolisz, aby Twoje własne terminy nie kolidowały z żadnymi importowanymi terminami.

WSKAZÓWKA: Możesz zobaczyć deklaracje najwyższego poziomu Swift, przeczytać je i przestudiować, co może być przydatne. Na przykład, aby zobaczyć deklarację print, kliknij z wciśniętymi klawiszami Command-Control termin print w kodzie. Oto niektóre deklaracje najwyższego poziomu Swift! Nie zobaczysz tutaj żadnego wykonywalnego kodu Swift, ale zobaczysz deklaracje dla różnych dostępnych terminów Swift, w tym do druku.

Instancje

Typy obiektów - class, struct i enum - mają ważną funkcję wspólną: mogą być tworzone. W efekcie, kiedy deklarujesz typ obiektu, definiujesz tylko typ. Tworzenie instancji typu to tworzenie rzeczy - instancji - tego typu. Na przykład mogę zadeklarować klasę Dog i nadać mojej klasie metodę:

```
class Dog {

func bark() {

print("woof")

}

}
```

Ale w moim programie nie mam jeszcze żadnych obiektów Dog. Opisałem tylko, czym byłby pies, gdybym go miał. Aby mieć prawdziwego psa, muszę go stworzyć. Proces tworzenia rzeczywistego obiektu Dog, którego typem jest klasa Dog, jest procesem tworzenia instancji Dog. Rezultatem jest nowy obiekt - instancja Dog. W Swift instancje można tworzyć, używając nazwy typu obiektu jako nazwy funkcji i wywołując funkcję. Wiąże się to z użyciem nawiasów. Kiedy dołączasz nawiasy do nazwy typu obiektu, wysyłasz bardzo szczególny rodzaj wiadomości do tego typu obiektu: Utwórz wystąpienie! Więc teraz zrobię instancję Dog:

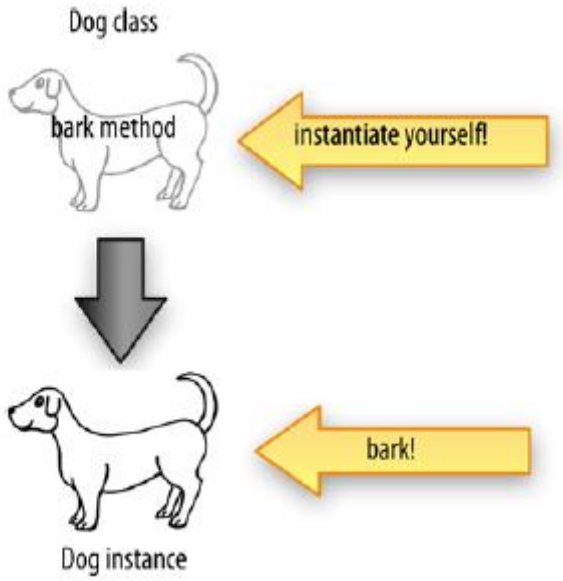
```
let fido = Dog()
```

W tym kodzie dużo się dzieje! Zrobiłem dwie rzeczy. Utworzyłem instancję Dog, powodując, że skończyłem z instancją Dog. Umieściłem również tę instancję Dog w pudełku na buty o nazwie fido - zadeklarowałem zmienną i zainicjowałem ją, przypisując do niej moją nową instancję Dog. Teraz fido jest instancją Dog. (Ponadto, ponieważ użyłem let, fido zawsze będzie tą samą instancją Dog. Mogłem zamiast tego użyć var, ale nawet wtedy inicjalizacja fido jako instancji Dog oznaczałaby, że fido może być później tylko instancją Dog.) Teraz, gdy ja mieć instancję Dog, mogę wysyłać do niej komunikaty instancji. Jak myślisz, czym one są? To właściwości i metody psa! Na przykład:

```
let fido = Dog()

fido.bark()
```

Ten kod jest legalny. Mało tego, jest skuteczny: faktycznie powoduje pojawienie się „woof” w konsoli. Zrobiłem psa i sprawiłem, że szczekał!



Jest tu ważna lekcja, więc pozwólcie, że zatrzymam się, aby to podkreślić. Domyślnie właściwości i metody są właściwościami i metodami instancji. Nie możesz ich używać jako wiadomości do samego typu obiektu; musisz mieć instancję, do której możesz wysyłać te wiadomości. W obecnej sytuacji jest to nielegalne i nie skompiluje się:

Dog.bark() // compile error

Możliwe jest zadeklarowanie funkcji szczekanie w taki sposób, że powiedzenie Dog.bark() jest dozwolone, ale byłby to inny rodzaj funkcji - funkcja klasy lub funkcja statyczna - i musiałbyś to powiedzieć podczas deklarowania to. To samo dotyczy właściwości. Aby to zilustrować, nadajmy Psu właściwość name:

class Dog {

var name = ""

}

To pozwala mi ustawić imię psa, ale musi to być instancja psa:

let fido = Dog()

fido.name = "Fido"

Możliwe jest zadeklarowanie nazwy właściwości w taki sposób, że powiedzenie Nazwa.Psa jest dozwolone, ale byłby to inny rodzaj właściwości - właściwość klasy lub właściwość statyczna - i musiałbyś to powiedzieć, kiedy ją deklarujesz.

Dlaczego instancje?

Nawet gdyby nie było czegoś takiego jak instancja, typ obiektu sam w sobie jest obiektem. Wiemy o tym, ponieważ można wysłać wiadomość do typu obiektu (przykładem jest fraza Manny.Klass). Dlaczego zatem w ogóle istnieją instancje? Odpowiedź dotyczy głównie natury właściwości instancji. Wartość właściwości instancji jest definiowana w odniesieniu do konkretnej instancji. To tutaj instancje

uzyskują swoją prawdziwą użyteczność i moc. Rozważ ponownie naszą klasę psów. Nadaję mu nazwę własności i metodę kory; pamiętaj, są to właściwość instancji i metoda instancji:

```
class Dog {

var name = ""

func bark() {

print("woof")

}

}
```

Instancja Dog powstaje z pustą nazwą (pustym ciągiem). Ale jego właściwość name to var, więc gdy mamy już instancję Dog, możemy przypisać jej nazwę nową wartość String:

```
let dog1 = Dog()

dog1.name = "Fido"
```

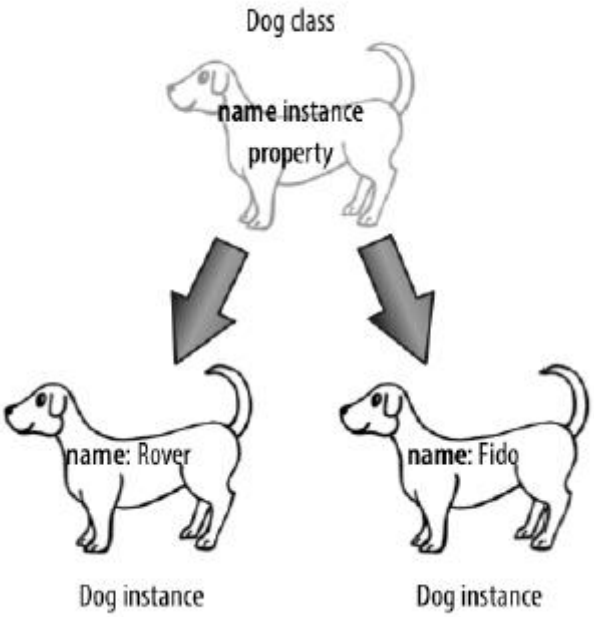
Możemy również poprosić o nazwę instancji Dog:

```
let dog1 = Dog()

dog1.name = "Fido"

print(dog1.name) // "Fido"
```

Ważne jest to, że możemy utworzyć więcej niż jedną instancję Dog, a dwie różne instancje Dog mogą mieć dwie różne właściwości name :




```
let dog1 = Dog()
```

```
dog1.name = "Fido"

let dog2 = Dog()

dog2.name = "Rover"

print(dog1.name) // "Fido"

print(dog2.name) // "Rover"
```

Zauważ, że właściwość name instancji Dog nie ma nic wspólnego z nazwą zmiennej, do której przypisana jest instancja Dog. Zmienna to po prostu pudełko na buty. Możesz przekazać instancję z jednego pudełka na buty do drugiego. Ale sama instancja zachowuje swoją wewnętrzną integralność:

```
let dog1 = Dog()

dog1.name = "Fido"

var dog2 = Dog()

dog2.name = "Rover"

print(dog1.name) // "Fido"

print(dog2.name) // "Rover"

dog2 = dog1

print(dog2.name) // "Fido"
```

Ten kod nie zmienił nazwy Rovera; zmieniło to, który pies był w pudełku po butach dog2, zastępując Rovera Fido. Teraz pojawiła się pełna moc programowania obiektowego. Istnieje typ obiektu Dog, który definiuje, co to znaczy być Psem. Nasza deklaracja z Dog mówi, że każda instancja Dog ma właściwość name i metodę kory. Ale każda instancja Dog może mieć własną wartość właściwości name. Tak więc wiele instancji tego samego typu obiektu zachowuje się podobnie — zarówno Fido, jak i Rover mogą szczekać i zrobią to po wysłaniu komunikatu o szczekaniu — ale są to różne instancje i mogą mieć różne wartości właściwości: Fido ma na imię „Fido”, podczas gdy Rover nazwa to „Rover”. Instancja odpowiada nie tylko za wartości, ale także za czas życia swoich właściwości. Załóżmy, że tworzymy instancję Dog i przypisujemy jej właściwości name wartość „Fido”. Wtedy ta instancja Dog utrzymuje przy życiu ciąg „Fido”, o ile nie zastąpimy wartości jego nazwy inną wartością - i tak długo, jak ta instancja żyje. Krótko mówiąc, instancja to zarówno kod, jak i dane. Kod, który otrzymuje ze swojego typu iw pewnym sensie jest współdzielony ze wszystkimi innymi instancjami tego typu, ale dane należą wyłącznie do niego. Dane mogą trwać tak długo, jak długo trwa instancja. Instancja ma w każdej chwili stan — kompletny zbiór własnych wartości własności osobistej. Instancja to urządzenie do utrzymywania stanu. To pudełko do przechowywania danych.

Słowo kluczowe self

Instancja to obiekt, a obiekt to odbiorca wiadomości. W związku z tym instancja potrzebuje sposobu na wysłanie wiadomości do siebie. Jest to możliwe dzięki słowu kluczowemu self. Tego słowa można używać wszędzie tam, gdzie oczekiwana jest instancja odpowiedniego typu. Powiedzmy, że chcę zachować to, co mówi pies, gdy szczeka, na przykład „hau”, w nieruchomości. Następnie w mojej realizacji kory muszę odwołać się do tej właściwości. Mogę to zrobić tak:

```
class Dog {

var name = ""

var whatADogSays = "woof"+

func bark() {

print(self.whatADogSays)

}

}
```

Podobnie, powiedzmy, że chcę napisać metodę instancji speak, która jest zaledwie synonimem kory. Moja implementacja mowy może polegać na prostym wywołaniu mojej własnej metody szczekania. Mogę to zrobić tak:

```
class Dog {

var name = ""

var whatADogSays = "woof"

func bark() {

print(self.whatADogSays)

}

func speak() {

self.bark()

}

}
```

Zauważ, że termin self w tym przykładzie pojawia się tylko w metodach instancji. Kiedy kod instancji mówi self, odnosi się do tej instancji. Jeśli wyrażenie self.name pojawia się w kodzie metody instancji Dog, oznacza to nazwę tej instancji Dog, której kod jest w tym momencie uruchomiony. Okazuje się, że każde użycie słowa self, które właśnie zilustrowałem, jest opcjonalne. Możesz to pominąć i wydarzą się te same rzeczy:

```
class Dog {

var name = ""var whatADogSays = "woof"

func bark() {

print(whatADogSays)

}

func speak() {

bark()

}
```

}

Powodem jest to, że jeśli pominiesz odbiorcę wiadomości, a wiadomość, którą wysyłasz, może zostać wysłana do siebie, kompilator dostarcza siebie jako odbiorcę wiadomości pod maską. Jednak nigdy tego nie robię (chyba że przez pomyłkę). Jeśli chodzi o styl, lubię wyraźnie wyrażać się w używaniu siebie. Kod, który pomija mnie, jest dla mnie trudniejszy do odczytania i zrozumienia. I są sytuacje, w których musisz powiedzieć siebie, więc wolę używać tego, kiedy mi na to wolno.

Prywatność

Wcześniej powiedziałem, że przestrzeń nazw sama w sobie nie jest barierą nie do pokonania w dostępie do zawartych w niej nazw. Ale taka bariera jest czasem pożądana. Nie wszystkie dane przechowywane przez instancję są przeznaczone do zmiany przez inną instancję lub nawet dla widoczności. I nie każda metoda instancji ma być wywoływana przez inne instancje. Każdy przyzwoity język programowania oparty na obiektach potrzebuje sposobu, aby zapewnić swoim elementom obiektowym prywatność - sposób na utrudnienie innym obiektom zobaczenia tych elementów, jeśli nie mają być widziane. Rozważ na przykład:

```
class Dog {

var name = ""

var whatADogSays = "woof"

func bark() {

print(self.whatADogSays)

}

func speak() {

print(self.whatADogSays)

}

}
```

Tutaj mogą pojawić się inne obiekty i zmienić moją właściwość whatADogSays. Ponieważ ta właściwość jest wykorzystywana zarówno przez szczekanie, jak i mówienie, możemy łatwo skończyć z psem, który po poproszeniu o szczekanie mówi „miau”. Wydaje się to jakoś niepożądane:

```
let fido = Dog()

fido.whatADogSays = "meow"

fido.bark() // meow
```

Możesz odpowiedzieć: Cóż, głuptasie, dlaczego zadeklarowałeś coADogSays z var? Zadeklaruj to zamiast let . Niech to będzie stałe! Teraz nikt nie może tego zmienić:

```
class Dog {

var name = ""

let whatADogSays = "woof"

func bark() {
```

```
print(self.whatADogSays)

}

func speak() {

print(self.whatADogSays)

}

}
```

To dobra odpowiedź, ale niewystarczająca. Są dwa problemy. Załóżmy, że chcę, aby instancja Dog była w stanie zmienić swoje własne whatADogSays — poprzez przypisanie do elf.whatADogSays. Następnie whatADogSays musi być var; w przeciwnym razie nawet sama instancja nie może tego zmienić. Załóżmy też, że nie chcę, aby jakikolwiek inny obiekt wiedział, co mówi ten pies, z wyjątkiem wołania szczekania lub mówienia. Nawet po zadeklarowaniu z let inne obiekty mogą nadal czytać wartość whatADogSays. Może mi się to nie podoba. Aby rozwiązać ten problem, Swift udostępnia słowo kluczowe private. Porozmawiam później o wszystkich konsekwencjach tego słowa kluczowego, ale na razie wystarczy wiedzieć, że istnieje:

```
class Dog {

var name = ""

private var whatADogSays = "woof"

func bark() {

print(self.whatADogSays)

}

func speak() {

print(self.whatADogSays)

}

}
```

Teraz name jest własnością publiczną, ale whatADogSays jest własnością prywatną: nie może być widziana przez inne typy obiektów. Instancja Dog może powiedzieć self.whatADogSays, ale instancja Cat z odwołaniem do instancji Dog jako fido nie może powiedzieć fido.whatADogSays:

```
class Cat {

func tryToChangeWhatADogSays() {

let fido = Dog()

fido.whatADogSays = "meow" // compile error

}

}
```

Ważną lekcją tutaj jest to, że członkowie obiektów są domyślnie publiczni, a jeśli chcesz prywatności, musisz o to poprosić. Podsumowując: Deklaracja klasy definiuje przestrzeń nazw. Ta przestrzeń nazw

wymaga, aby inne obiekty używały dodatkowego poziomu notacji z kropkami, aby odwoływać się do tego, co znajduje się w przestrzeni nazw, ale inne obiekty mogą nadal odwoływać się do tego, co znajduje się w przestrzeni nazw; przestrzeń nazw sama w sobie nie zamyka żadnych drzwi widoczności. Słowo kluczowe private pozwala zamknąć te drzwi.

SŁOWA ZASTRZEŻONE

Niektóre terminy, takie jak class i func i var i let i if oraz private i import, są zastrzeżone w języku Swift; są częścią języka. Oznacza to, że nie możesz ich używać jako identyfikatorów — takich jak nazwa klasy, funkcja lub zmienna. Jeśli spróbujesz to zrobić, otrzymasz błąd kompilacji. Aby wymusić zastrzeżenie słowa jako identyfikatora, otocz je znakami (`). Ten (niezwykle mylący) kod jest legalny:

```
class `func` {

func `if`() {

let `class` = 1

}

}
```

Projekt

Instancje nie powstają za pomocą magii. Musisz utworzyć instancję typu, aby uzyskać instancję. W związku z tym duża część akcji twojego programu będzie polegać na tworzeniu instancji typów. I oczywiście będziesz chciał, aby te instancje trwały, więc przypiszesz także każdą nowo utworzoną instancję do zmiennej jako pudełko na buty, aby ją przechowywać, nazwać i dać jej życie. Instancja będzie trwać zgodnie z okresem istnienia zmiennej, która się do niej odwołuje. A instancja będzie widoczna dla innych instancji zgodnie z zakresem zmiennej, która się do niej odwołuje. Znaczna część sztuki programowania obiektowego polega na zapewnieniu instancjom wystarczającej żywotności i uczynieniu ich widocznymi dla siebie. Często umieszczasz instancję w konkretnym pudełku po butach – przypisując ją do konkretnej zmiennej, zadeklarowanej w określonym zakresie – dokładnie po to, aby dzięki regułom czasu życia i zasięgu zmiennej ta instancja przetrwała wystarczająco długo, aby nadal była użyteczna dla twojego program, póki nadal będzie potrzebny, aby inny kod mógł uzyskać odniesienie do tej instancji i porozmawiać z nią później. Planowanie, w jaki sposób zamierzasz tworzyć instancje, a także opracowywanie okresów życia i komunikacji między tymi instancjami, może wydawać się zniechęcające. Na szczęście w prawdziwym życiu, gdy programujesz iOS, framework zapewni Ci rusztowanie. Zanim napiszesz pojedynczy wiersz kodu, framework zapewnia, że Twoja aplikacja, podczas uruchamiania, otrzyma kilka instancji, które będą trwać przez cały okres istnienia aplikacji, zapewniając podstawę widocznego interfejsu Twojej aplikacji i dając pierwsze miejsce do umieszczenia własnymi instancjami i daj im wystarczająco długie życie. A co z pytaniem, jakich typów obiektów będzie potrzebował Twój program i jakie metody i właściwości powinien on mieć? Nie jest to aż tak duże zmartwienie, jak mogłoby się wydawać. Sam Swift dostarcza bibliotekę potężnych i użytecznych typów obiektów. Co więcej, znaczna część kodu podczas programowania systemu iOS będzie koncentrować się na szczegółach rzeczywistych obiektów interfejsu, takich jak etykiety i przyciski, które użytkownik może zobaczyć i dotknąć, a framework wyjaśni, jakie typy obiektów i udogodnienia oferuje w tym celu i zapewni sposoby zapewnienia odpowiedniej trwałości i widoczności powiązanych instancji. Struktura nie może Ci powiedzieć, jak zaprojektować podstawową logikę biznesową tego, co Twoja aplikacja robi za kulisami. W tym miejscu będziesz mieć największą swobodę - i najwięcej trudności w uzyskaniu odpowiedniej architektury typów obiektów, funkcjonalności i relacji. Nie będą to łatwe decyzje i nie ma jednoznacznych odpowiedzi. Programowanie obiektowe to sztuka; a

pozwalanie twojemu programowi (i twojemu myśleniu) ewoluować podczas pisania kodu, odkrywania nowych potrzeb i problemów, jest sztuką w tej sztuce, którą nazywam rozwijaniem programu. Wszystkie osoby i zespoły opracowują własny sposób sprostania wyzwaniom długoterminowym.