**UNIVERSITATEA HYPERION din BUCURESTI**

**FACULTATEA de ŞTIINŢE EXACTE ŞI INGINEREŞTI**

**Specializarea: INFORMATICĂ**

**PROIECT DE LICENȚĂ**

**COORDONATOR STIINŢIFIC:**

**Conf. Univ. Dr. Ing. LIVIU ȘERBĂNESCU**

**ABSOLVENT:**

**ALBERT VICENȚIU LEONARD PETREACĂ**

**BUCUREŞTI – 2019**

**UNIVERSITATEA HYPERION din BUCURESTI FACULTATEA de ŞTIINŢE EXACTE ŞI INGINEREŞTI Specializarea: INFORMATICĂ**

**DEZVOLTAREA UNEI APLICAȚII MOBILE FOLOSIND REALITATEA AUGMENTATĂ ÎN DOMENIUL ASTRONOMIEI**

**COORDONATOR STIINŢIFIC:**

**Conf. Univ. Dr. Ing. LIVIU ȘERBĂNESCU**

**ABSOLVENT:**

**ALBERT VICENȚIU LEONARD PETREACĂ**

**BUCUREŞTI – 2019**

**TEMA PROIECTULUI**

Această lucrare are ca obiectiv crearea aplicației 3D cu numele „ARSolarSystem” pentru platforma iOS. Implementarea acestui obiectiv este realizată prin îmbinarea conceptelor şi tehnologiilor prezentate în capitolele anterioare. Astfel, aplicația are la bază SDK-ul ARKit prin intermediul căruia sunt controlate şi expuse utilizatorului aspectele vizuale, sunt tratate input-urile dispozitivului şi sunt administrate activităţile şi scenele componente ale aplicaţiei.

Utilizatorul va putea:

* să vadă planetele rotindu-se în jurul propriului ax;
* să vadă planetele rotindu-se în jurul soarelui;
* să apese pe una dintre planete;
* să vadă mai multe detalii despre planeta pe care a apăsat;
* să fie trimis către pagina de wikipedia corespunzătoare planetei respective

**CUPRINS**

[1. Limbajul Swift 1](#__RefHeading___Toc3922_3357734315)

[1.1 Bazele limbajului Swift 1](#__RefHeading___Toc3596_3357734315)

[1.1.1 Constante și variabile 2](#__RefHeading___Toc3300_3357734315)

[1.1.2 Adnotări de tip 2](#__RefHeading___Toc3344_3357734315)

[1.1.3 Inferența tipului și siguranță de tip 3](#__RefHeading___Toc3342_3357734315)

[1.1.4 Aliasuri de tip 4](#__RefHeading___Toc3340_3357734315)

[1.1.5 Tupluri 5](#__RefHeading___Toc3338_3357734315)

[1.1.6 Opționale 6](#__RefHeading___Toc3336_3357734315)

[1.1.6.1 nil 7](#__RefHeading___Toc3346_3357734315)

[1.1.6.2 Instrucțiunea *if* și „despachetarea” forțată 8](#__RefHeading___Toc3382_3357734315)

[1.1.6.3 Optional Binding 9](#__RefHeading___Toc3380_3357734315)

[1.1.6.4 Opționale implicit „despachetate” 10](#__RefHeading___Toc3378_3357734315)

[1.2 Tipuri de colecții 11](#__RefHeading___Toc3608_3357734315)

[1.2.1 Array (sau vector) 12](#__RefHeading___Toc3334_3357734315)

[1.2.1.1 Accesul și modificarea unui array 13](#__RefHeading___Toc3376_3357734315)

[1.2.1.2 Interarea printr-un array 14](#__RefHeading___Toc3374_3357734315)

[1.2.2 Set 15](#__RefHeading___Toc3332_3357734315)

[1.2.2.1 Accesarea și modificarea unui Set 16](#__RefHeading___Toc3372_3357734315)

[1.2.2.2 Iterarea printr-un Set 17](#__RefHeading___Toc3370_3357734315)

[1.2.2.3 Calitatea de membru și egalitate 19](#__RefHeading___Toc3368_3357734315)

[1.2.3 Dictionary 20](#__RefHeading___Toc3330_3357734315)

[1.2.3.1 Accesarea și modificarea unui Dicționar 22](#__RefHeading___Toc3366_3357734315)

[1.2.3.2 Iterarea printr-un Dicționar 24](#__RefHeading___Toc3364_3357734315)

[1.3 Construcțiile pentru control fluxului 25](#__RefHeading___Toc3606_3357734315)

[1.3.1 Buclele for-in 25](#__RefHeading___Toc3328_3357734315)

[1.3.2 Buclele cu număr necunoscut de pași pre-condiționate (while) 28](#__RefHeading___Toc3326_3357734315)

[1.3.3 Buclele cu număr necunoscut de pași post-condiționate (repeat-while) 28](#__RefHeading___Toc3324_3357734315)

[1.3.4 Structuri condiționale 28](#__RefHeading___Toc3322_3357734315)

[1.3.4.1 If 29](#__RefHeading___Toc3362_3357734315)

[1.3.4.2 Switch 30](#__RefHeading___Toc3360_3357734315)

[1.3.4.3 Asocierea valorilor 34](#__RefHeading___Toc3358_3357734315)

[1.3.4.4 Clauza where 35](#__RefHeading___Toc3356_3357734315)

[1.3.5 Instrucțiuni pentru transferul controlului 36](#__RefHeading___Toc3320_3357734315)

[1.3.5.1 Continue 36](#__RefHeading___Toc3354_3357734315)

[1.3.5.2 Break 37](#__RefHeading___Toc3352_3357734315)

[1.3.5.3 Fallthrough 37](#__RefHeading___Toc3350_3357734315)

[1.3.5.4 Ieșirea timpurie din scop 38](#__RefHeading___Toc3348_3357734315)

[1.4 Funcții (metode) 39](#__RefHeading___Toc3604_3357734315)

[1.4.1 Funcții fără tip de retur 40](#__RefHeading___Toc3318_3357734315)

[1.4.2 Funcții ce returnează mai multe valori 40](#__RefHeading___Toc3316_3357734315)

[1.4.3 Etichetarea argumentelor și numele parametrilor 41](#__RefHeading___Toc3314_3357734315)

[1.4.4 Parametrii variadici 43](#__RefHeading___Toc3312_3357734315)

[1.5 Closures (funcții anonime) 43](#__RefHeading___Toc3602_3357734315)

[1.6 Enumerația 45](#__RefHeading___Toc3600_3357734315)

[1.6.1 Valori asociate 45](#__RefHeading___Toc3310_3357734315)

[1.7 Structuri și clase 47](#__RefHeading___Toc3598_3357734315)

[1.7.1 Instanțele de clase și structuri 48](#__RefHeading___Toc3308_3357734315)

[1.7.3 Structurile și enumerările sunt Value Types 49](#__RefHeading___Toc3306_3357734315)

[1.7.3 Clasele sunt Reference Types 50](#__RefHeading___Toc3304_3357734315)

[1.8 Protocoale 52](#__RefHeading___Toc3592_3357734315)

[1.8.1 Protocoalele folosite ca tipuri 54](#__RefHeading___Toc3302_3357734315)

[2. Tehnologii folosite 55](#__RefHeading___Toc4089_3357734315)

[2.1 Sistemul de operare iOS 55](#__RefHeading___Toc4197_3357734315)

[2.2 ARKit 56](#__RefHeading___Toc4195_3357734315)

[2.3 CoreData 56](#__RefHeading___Toc4199_3357734315)

[3. Descrierea aplicației 56](#__RefHeading___Toc4209_3357734315)

[3.1 Mediul virtual al aplicației 57](#__RefHeading___Toc4234_3357734315)

[3.2 Adăugarea unei planete în spațiu 57](#__RefHeading___Toc4329_3357734315)

[3.3 Detaliile unei planete 62](#__RefHeading___Toc4327_3357734315)

[3.4 Încărcarea planetelor din baza de date 64](#__RefHeading___Toc4325_3357734315)

[3.5 Ecranul de detalii 64](#__RefHeading___Toc4323_3357734315)

[4. Concluzii 67](#__RefHeading___Toc4321_3357734315)

[5. Bibliografie 68](#__RefHeading___Toc3117_3602344575)

# 1. Limbajul Swift

1.1 Bazele limbajului Swift

Swift este un nou limbaj de programare pentru dezvoltarea aplicațiilor iOS, macOS, watchOS și tvOS. Cu toate acestea, multe părți ale Swift vor fi similare cu cele din C și Objective-C.

Swift oferă propriile versiuni ale tuturor tipurilor fundamentale C și Obiectiv-C, inclusiv Int pentru numere întregi, Double și Float pentru valori în virgulă mobilă, Bool pentru valori booleene și String pentru date textuale. Swift oferă, de asemenea, versiuni puternice ale celor trei tipuri de colecții primare, Array, Set și Dictionary.

La fel ca și C, Swift folosește variabilele pentru stocarea și referirea la valori printr-un nume de identificare. Swift utilizează, de asemenea, extensiv variabile ale căror valori nu pot fi modificate. Acestea sunt cunoscute sub numele de constante și sunt mult mai puternice decât constantele în C. Constantele sunt folosite pe tot parcursul scrieri codului Swift pentru a face codul mai sigur și mai clar cu privire la intenția codului atunci când lucrați cu valori care nu trebuie să se schimbe.

În plus față de tipurile familiare, Swift introduce tipuri avansate care nu sunt găsite în Objective-C, cum ar fi tuplul. Tuplul vă permite să creați și să pasați mai departe grupuri de valori. Puteți folosi un tuplu pentru a returna mai multe valori dintr-o funcție ca o singură valoare.

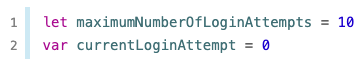
Swift introduce de asemenea tipuri opționale care se ocupă de absența unei valori. Opționalele spun fie "există o valoare și este egală cu x" sau "nu există o valoare". Utilizarea opționalelor este similară cu utilizarea valorii nil atribuita unui pointer în Objective-C, dar aceștia lucrează pentru orice tip, nu doar pentru clase. Nu numai că opționalele sunt mai sigure și mai expresive decât pointerii nuli în Objective-C, ele sunt în centrul multor caracteristici ale lui Swift.

Swift este un limbaj sigur, ceea ce înseamnă că limbajul vă ajută să vă lămuriți cu privire la tipurile de valori pe care le poate utiliza codul. Dacă o parte din codul dvs. necesită un String, tipul de siguranță vă împiedică să îl transmiteți din greșeală unui Int. De asemenea, tipul de siguranță vă împiedică să transmiteți accidental un String opțional unei bucăți de cod care necesită un String ne-opțional. Siguranța de tip vă ajută să prindeți și să remediați erorile cât mai curând posibil în procesul de dezvoltare.

1.1.1 Constante și variabile

Constantele și variabilele asociază un nume (cum ar fi maximumNumberOfLoginAttempts sau welcomeMessage) cu o valoare a unui anumit tip (cum ar fi numărul 10 sau șirul "Hello"). Valoarea unei constante nu poate fi schimbată odată setată, în timp ce o variabilă poate fi setată la o valoare diferită în viitor.

Constantele și variabilele trebuie declarate înainte de a fi utilizate. Declarați constantele folosind cuvântul cheie *let* și variabilele folosind cuvântul cheie *var*. Iată un exemplu despre modul în care constantele și variabilele pot fi utilizate pentru a urmări numărul de încercări de conectare pe care le-a făcut un utilizator:

  
Figura 1

Acest cod poate fi citit ca: "Declarați o nouă constantă numită maximumNumberOfLoginAttempts și dați-i o valoare de 10. Apoi, declarați o nouă variabilă numită currentLoginAttempt și dați-i o valoare inițială de 0."

În acest exemplu, numărul maxim de încercări de conectare permise este declarat ca o constantă, deoarece valoarea maximă nu se modifică niciodată. Contorul actual de încercare de autentificare este declarat ca o variabilă, deoarece această valoare trebuie incrementată după fiecare încercare de conectare nereușită.

1.1.2 Adnotări de tip

Puteți oferi o adnotare de tip atunci când declarați o constantă sau o variabilă, pentru a fi mai clar cu privire la tipul de valori pe care constanta sau variabila o poate stoca. Scrieți o adnotare de tip plasând două puncte după numele constantei sau variabilei, urmat de un spațiu, urmat de numele tipului utilizat.

Acest exemplu oferă o adnotare de tip pentru o variabilă numită welcomeMessage, pentru a indica faptul că variabila poate stoca valorile String:

  
Figura 2

Cele două puncte din declarație înseamnă "... de tip ...", astfel încât codul de mai sus poate fi citit ca: "Declarați o variabilă numită welcomeMessage, care este de tip String." Variabila welcomeMessage poate fi acum setată la orice valoare de șir fără eroare.

Rareori va fi nevoie să scrieți adnotări de tip în practică. Dacă furnizați o valoare inițială pentru o constantă sau o variabilă în punctul definit, Swift poate deduce aproape întotdeauna tipul care va fi utilizat pentru acea constantă sau variabilă. În exemplul welcomeMessage de mai sus, nu este furnizată nici o valoare inițială, deci tipul variabilei welcomeMessage este specificat cu o adnotare de tip, mai degrabă decât dedus dintr-o valoare inițială.

1.1.3 Inferența tipului și siguranță de tip

Swift este un limbaj sigur pentru tip. Un limbaj sigur de tip vă încurajează să vă lămuriți cu privire la tipurile de valori cu care codul dvs. poate funcționa. Dacă o parte din codul dvs. necesită un String, nu îi puteți trece din greșeală un Int.

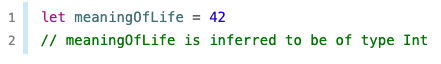
Deoarece Swift prezintă siguranța de tip, acesta efectuează verificări de tip atunci când compilează codul dvs. și scoate în evidență orice tip necompatibil ca și eroare. Acest lucru vă permite să prindeți și să remediați erorile cât mai curând posibil în procesul de dezvoltare.

Verificarea tipului vă ajută să evitați erorile atunci când lucrați cu diferite tipuri de valori. Cu toate acestea, acest lucru nu înseamnă că trebuie să specificați tipul fiecărei constante și variabile pe care le declarați. Dacă nu specificați tipul de valoare de care aveți nevoie, Swift utilizează inferența de tip pentru a elabora tipul corespunzător. Inferența de tip permite unui compilator să deducă automat tipul unei expresii în mod automat atunci când compilează codul dvs., pur și simplu examinând valorile pe care le furnizați.

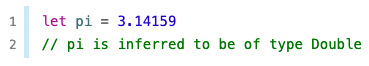
Din cauza inferenței de tip, Swift necesită mai puține declarații de tip decât limbajele precum C sau Objective-C. Constantele și variabilele sunt încă inscripționate în mod explicit, dar o mare parte din munca de specificare a tipului lor este făcută pentru dvs.

Inferența de tip este utilă în special atunci când declarați o constantă sau o variabilă cu o valoare inițială. Acest lucru se face deseori atribuindu-se o valoare literală constantă sau variabilă în punctul în care o declarați. (O valoare literală este o valoare care apare direct în codul dvs. sursă, cum ar fi 42 s 3.14159 în exemplele de mai jos.)

De exemplu, dacă atribuiți o valoare literală de 42 unei constante noi, fără a spune ce tip este, Swift deduce că doriți ca constantul să fie un Int, deoarece ați inițializat-o cu un număr care arată ca un întreg:

  
Figura 3

De asemenea, dacă nu specificați un tip pentru un literal în virgulă mobilă, Swift deduce că doriți să creați un Double:

  
Figura 4

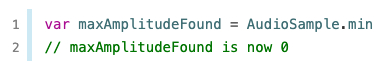
1.1.4 Aliasuri de tip

Aliasurile de tip definesc un nume alternativ pentru un tip existent. Definiți alias-uri de tip cu cuvântul cheie *typealias.*

Aliasurile de tip sunt utile atunci când doriți să vă referiți la un tip existent printr-un nume care este mai adecvat din punct de vedere contextual, cum ar fi atunci când lucrați cu date de o anumită dimensiune dintr-o sursă externă:

  
Figura 5

După ce definiți un alias de tip, puteți utiliza pseudonimul oriunde ați putea utiliza numele original:

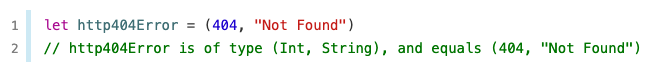
  
Figura 6

Aici, AudioSample este definit ca un alias pentru UInt16. Deoarece este un alias, apelul catre AudioSample.min este de fapt UInt16.min, care ofera o valoare initiala de 0 pentru variabila maxAmplitudeFound.

1.1.5 Tupluri

Tuplurile grupează valori multiple într-o singură valoare compusă. Valorile dintr-un tuplu pot fi de orice tip și nu trebuie să fie de același tip ca și celelalte.

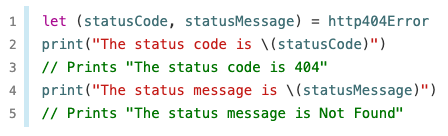
În acest exemplu, (404, "Not Found") este un tuplu care descrie un cod de stare HTTP. Un cod de stare HTTP este o valoare specială returnată de un server web ori de câte ori solicitați o pagină Web. Un cod de stare de 404 Nu a fost găsit este returnat dacă solicitați o pagină web care nu există.

  
Figura 7

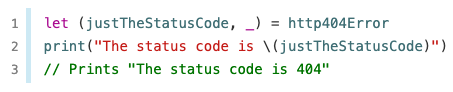
Tuplul (404, "Not Found") grupează un Int și un String pentru a da codului de stare HTTP două valori separate: un număr și o descriere care poate fi citită de om. Acesta poate fi descris ca "un tuplu de tip (Int, String)".

Puteți crea tupluri din orice permutare a tipurilor și pot conține cât mai multe tipuri diferite. Nu există nimic care să te oprească de a avea un tuplu de tip (Int, Int, Int) sau (String, Bool), sau chiar orice altă permutare pe care o doriți.

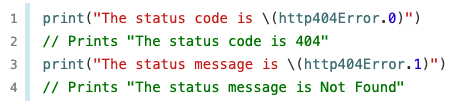
Puteți descompune conținutul unui tuplu în constante sau variabile separate, pe care le accesați ca de obicei:

  
Figura 8

Dacă aveți nevoie doar de unele dintre valorile tuplului, ignorați părți ale lui cu o subliniere (\_) atunci când descompuneți tuplul:

  
Figura 9

Alternativ, accesați valorile elementelor individuale dîntr-un tuplu folosind numerele index care încep de la zero:

  
Figura 10

Puteți numi elementele individuale într-un tuplu atunci când este il definiți:

  
Figura 11

Tuplurile sunt utile în special ca valori de întoarcere a funcțiilor. O funcție care încearcă să acceseze o pagină web ar putea să returneze tipul de tip (Int, String) pentru a descrie succesul sau eșecul accesării paginii. Prin returnarea unui tuplu cu două valori distincte, fiecare cu un tip diferit, funcția oferă informații mai utile despre rezultatul său decât dacă ar putea să returneze numai o singură valoare a unui singur tip.

Tuplurile sunt utile pentru grupuri simple de valori asociate. Nu sunt potrivite pentru crearea unor structuri complexe de date. Dacă structura de date este probabil complexă, modelați-o mai degrabă ca o clasă sau o structură, decât ca un tuplu.

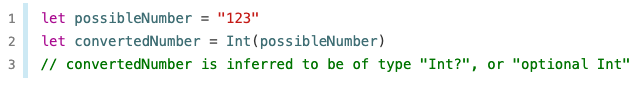
1.1.6 Opționale

Utilizați opționalele în situațiile în care o valoare poate fi absentă. Un opțional reprezintă două posibilități: Fie că există o valoare, și puteți să „despachetați” opționalul pentru a accesa acea valoare, fie nu există o valoare deloc.

Conceptul de opțional nu există în C sau în Objective-C. Cel mai apropiat lucru în Objective-C este abilitatea de a întoarce o valuare nulă dintr-o metodă care altfel ar întoarce un obiect, prin nil semnificând "absența unui obiect valid". Totuși, aceasta funcționează numai pentru obiecte - nu funcționează pentru structuri, tipuri primitive din C sau valori ale unei enumerații. Pentru aceste tipuri, metodele Objective-C returnează de obicei o valoare specială (cum ar fi NSNotFound) pentru a indica absența unei valori. Această abordare presupune că apelantul metodei știe că există o valoare specială pentru cazurile în care nu se gasește o valoare și are obligația de a compara rezultatul metodei cu acea valoare specială. Opționalele Swift vă permit să indicați absența unei valori pentru orice tip, fără a fi nevoie de constante speciale.

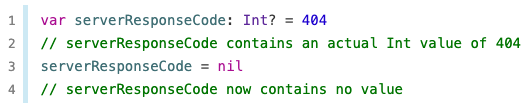
Iată un exemplu despre modul în care opționalele pot fi utilizate pentru a arăta înspre absenței unei valori. Tipul Int din Swift are un constructor care încearcă să convertească o valoare String într-o valoare Int. Cu toate acestea, nu fiecare șir poate fi convertit într-un număr întreg. Șirul "123" poate fi convertit în valoarea numerică 123, dar șirul "hello, world" nu are o valoare numerică evidentă pentru a se converti la.

Deoarece constructorul s-ar putea să eșueze, acesta returnează un Int opțional, mai degrabă decât un Int. Un opțional Int este scris ca Int?, nu Int. Semnul de întrebare indică faptul că valoarea pe care o conține este opțională, ceea ce înseamnă că ar putea conține o anumită valoare Int sau ar putea să nu conțină nicio valoare. (Nu poate conține nimic altceva, cum ar fi o valoare Bool sau o valoare String. Este fie un Int, fie nu este nimic.)

  
Figura 12

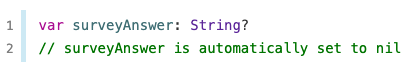
1.1.6.1 nil

Se poate seta o variabilă opțională la o stare fără valoare atribuindu-i valoarea specială nil:

  
Figura 13

Nu puteți folosi *nil* cu constante și variabile non-opționale. Dacă o constantă sau o variabilă din codul dvs. trebuie să funcționeze cu absența unei valori în anumite condiții, declarați-o întotdeauna ca o valoare opțională de tipul corespunzător.

Dacă definiți o variabilă opțională fără a furniza o valoare implicită, variabila este setată automat la nil pentru dvs.:

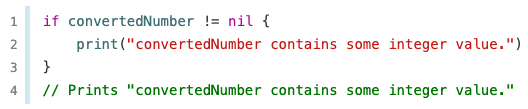
  
Figura 14

Nilul din Swift nu este același cu nilul în Objective-C. În Objective-C, nil este un pointer către un obiect inexistent. În Swift, nil nu este un pointer - este absența unei valori de un anumit tip. Opționalele de orice tip pot fi setate la nil, nu doar cele care sunt instanțe ale unor clase.

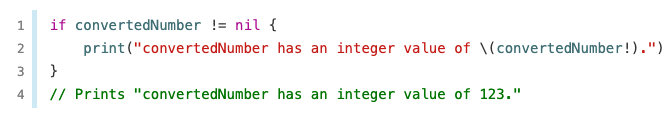
1.1.6.2 Instrucțiunea *if* și „despachetarea” forțată

Puteți utiliza o instrucțiune if pentru a afla dacă un opțional conține o valoare prin compararea opțiunii opționale cu nil. Efectuați această comparație cu operatorul "egal cu" (==) sau cu operatorul "diferit de" (!=).

Dacă un opțional are o valoare, el este considerat a fi "diferit de" nil:

  
Figura 15

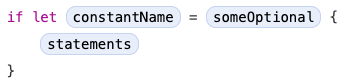
Odată ce sunteți sigur că opționalul conține o valoare, puteți accesa valoarea sa de bază adăugând un semn de exclamare (!) la sfârșitul numelui opționalului. Semnul de exclamare spune: "Știu că aceast opțional are cu siguranță o valoare; utilizați-o". Aceasta este cunoscută sub denumirea de despachetare forțată a valorii opționale:

  
Figura 16

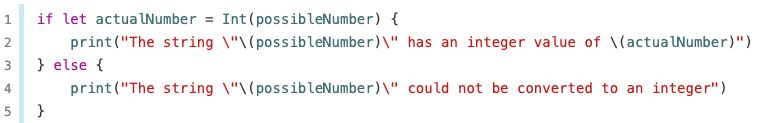
Încercarea de a folosi „!” pentru a accesa o valoare opțională inexistentă va face programul să încheie forțat. Asigurați-vă întotdeauna că opționalul conține o valoare non-nil înainte de a utiliza „!” pentru a-i despacheta valoarea.

1.1.6.3 Optional Binding

Utilizați Optional Binding pentru a afla dacă un opțional conține o valoare și, dacă da, pentru a face ca această valoare să fie disponibilă ca o constantă sau variabilă temporară. Optional binding-ul poate fi utilizat cu instrucțiunile *if* și *while* pentru a verifica valoarea din interiorul unui opțional și pentru a extrage acea valoare într-o constantă sau variabilă, ca parte a unei singure acțiuni. Optional binding-ul are următoarea structură:

  
Figura 17

Exemplul de mai sus poate fi rescris folosind optional binding în felul următor:

  
Figura 18

Acest cod poate fi citit ca: "Dacă Int-ul opțional returnat de Int(possibleNumber) conține o valoare, setați o nouă constantă numită actualNumber cu valoarea conținută de opțional."

Dacă conversia are succes, constanta actualNumber devine disponibilă pentru a fi utilizată în prima ramură a instrucțiunii if. Acesta a fost deja inițializată cu valoarea conținută în opțional, deci nu este nevoie să utilizați „!” la sfârșitul numelui constantei pentru a accesa valoarea sa. În acest exemplu, actualNumber este pur și simplu utilizat pentru a imprima rezultatul conversiei.

Puteți utiliza atât constante, cât și variabile cu optional binding. Dacă doriți să manipulați valoarea actualNumber în prima ramură a instrucțiunii if, ați putea scrie *if var actualNumber* și valoarea conținută în opțional ar fi disponibilă ca o variabilă în loc de constantă.

1.1.6.4 Opționale implicit „despachetate”

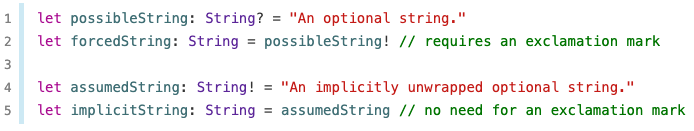
După cum este descris mai sus, opționalele arată că unei constante sau unei variabile îi este permis să aibă "nicio valoare". Opționalele pot fi verificate cu o instrucțiune if pentru a vedea dacă există o valoare și pot fi „despachetate” cu optional binding pentru a accesa valoarea opțională dacă aceasta există.

Uneori este clar din structura unui program că un opțional va avea întotdeauna o valoare, după ce valoarea este setată pentru prima dată. În aceste cazuri, este util să eliminați necesitatea de a verifica și de a „despacheta” valoarea opționalului de fiecare dată când este accesat, deoarece se poate presupune în siguranță că are o valoare tot timpul.

Aceste tipuri de opționale sunt definite ca fiind opțional implicit „despachetate”. Un opțional implicit „despachetat” se scrie prin plasarea unui semn de exclamare (String!) în locul unui semn de întrebare (String?) după tipul pe care doriți să-l faceți opțional.

Opționalele implicit „despachetate” sunt utile atunci când se confirmă existența unei valori imediat după ce opționalul este definit pentru prima dată și se poate presupune că există în orice moment după aceea.

Un opțional implicit „despachetat” este un opțional normal, dar poate fi de asemenea folosit ca o valoare non-opțională, fără a fi nevoie să despachetați valoarea opțională de fiecare dată când este accesat. Următorul exemplu prezintă diferența de comportament dintre un șir opțional și un șir opțional implicit „despachetat” atunci când accesează valoarea din el ca un String explicit:

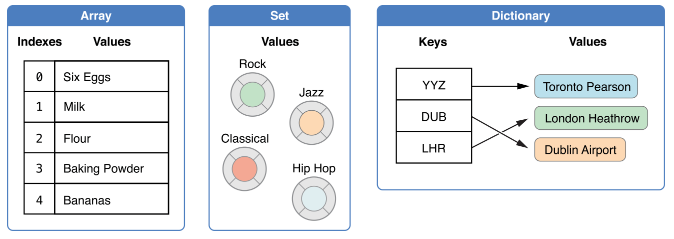
  
Figura 19

Vă puteți gândi la un opțional implicit „despachetat”, ca acordând permisiunea să fie „despachetat” automat ori de câte ori este folosit. În loc să introduceți un semn de exclamare după numele opționalului de fiecare dată când îl folosiți, plasați un semn de exclamare după tipul opțional când îl declarați.

Dacă un opțional implicit „despachetat” este nefolosit și încercați să accesați valoarea dinăuntrul lui, veți declanșa o eroare de runtime. Rezultatul este exact același ca și când ați plasa un semn de exclamare după un opțional obișnuit care nu conține o valoare.

1.2 Tipuri de colecții

Swift oferă trei tipuri primare de colecții, cunoscute sub formă de Array, Set și Dictionary, pentru stocarea colecțiilor de valori. Array-urile sunt colecții ordonate de valori. Seturile sunt colecții neordonate de valori unice. Dicționarele sunt colecții neordonate de asocieri cheie-valoare.

  
Figura 20

Array-urile, Set-urile și Dicționarele în Swift sunt întotdeauna clare cu privire la tipurile de valori și chei pe care le pot stoca. Aceasta înseamnă că nu puteți introduce o valoare de tip greșit într-o colecție din greșeală. De asemenea, înseamnă că puteți avea încredere în tipul de valori pe care le veți obține dintr-o colecție. Ele sunt implementate precum colecții generice. Mai multe detalii vom vedea în capitolul de generice.

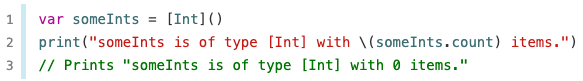
Dacă creați un Array, Set sau Dictionary și îl atribuiți unei variabile, colecția care este creată va fi mutabilă. Aceasta înseamnă că puteți schimba (sau muta) colecția după ce a fost creată prin adăugarea, eliminarea sau schimbarea elementelor din colecție. Dacă atribuiți Array, Set sau Dictionary la o constantă, acea colecție este imutabilă, iar dimensiunea și conținutul acesteia nu pot fi modificate ulterior.

1.2.1 Array (sau vector)

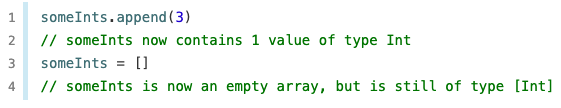
Un array stochează valori de același tip într-o listă ordonată. Aceeași valoare poate apărea într-un array de mai multe ori la poziții diferite.

Tipul unui array Swift este scris în întregime ca Array<Element>, unde Element este tipul de valori pe care array-ul îl stochează. De asemenea, puteți scrie tipul de array sub forma prescurtată [Element]. Deși cele două forme sunt identice din punct de vedere funcțional, forma preferată este cea prescurtată.

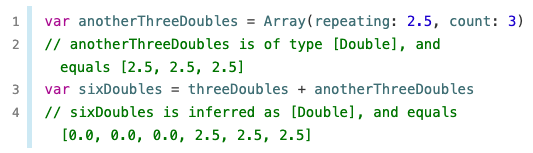
Se poate crea un array gol folosind următoarea sintaxă:

  
Figura 21

Alternativ, dacă contextul oferă deja informații de tip, cum ar fi un argument al funcției sau o variabilă sau o constantă deja declarată, puteți crea un array gol cu expresia literală de array gol, care este scrisă ca [] (o pereche goală de paranteze pătrate):

  
Figura 22

Puteți crea un array nou adăugând împreună două array-uri existente cu tipuri compatibile cu operatorul adițional (+). Tipul noului array este dedus din tipul celor două array-uri pe care le adăugați împreună:

  
Figura 23

De asemenea, puteți inițializa un array cu un array literal, care este o modalitate de prescurtată de a scrie una sau mai multe valori ca o colecție de tip array. Un array literal este scris ca o listă de valori, separate prin virgule, înconjurată de o pereche de paranteze pătrate:

  
Figura 24

  
Figura 25

Variabila shoppingList este declarată ca "un array de tip String", scris ca [String]. Deoarece aceast array are specificat tipul de valoare conținut, String, este permisă stocarea valorilor numai de tipul String. Aici, array-ul shoppingList este inițializat cu două valori String ("Eggs" și "Milk"), scrise într-un array literal.

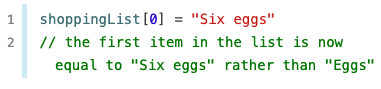
În acest caz, literalul array-ului conține două valori de tip String și nimic altceva. Acest lucru se potrivește cu tipul declarației variabilei shoppingList (un array care poate conține numai valori String), astfel încât alocarea literalului de array este permisă ca o modalitate de a inițializa shoppingList cu două elemente inițiale.

1.2.1.1 Accesul și modificarea unui array

Accesul și modificarea unui array se face pe bază de subscript. Accesul la o valoare din array utilizând subscriptul se face în felul următor, trecând index-ul valorii pe care doriți să o accesați în paranteze pătrate imediat după numele array-ului:

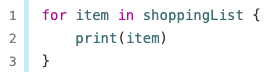
  
Figura 26

Subscriptul se poate folosi și pentru a modifica o valoare de la un anumit index:

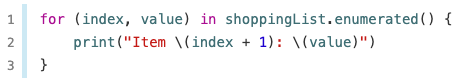


1.2.1.2 Interarea printr-un array

Se poate face printr-un for in:

  
Figura 27

Dacă aveți nevoie de indexul fiecărui element, precum și de valoarea acestuia, utilizați metoda enumerated() pentru a itera prin array. Pentru fiecare element din colectia, metoda enumerated() returnează un tuplu compus dintr-un întreg și elementul acelui index. Numerele întregi încep de la zero și sunt incrementate cu câte unul pentru fiecare element; dacă enumerați tot array-ul aceste numere întregi se potrivesc cu indicii elementelor. Puteți să descompuneți tuplul în constante sau variabile temporare ca parte a iterației:

  
Figura 28

1.2.2 Set

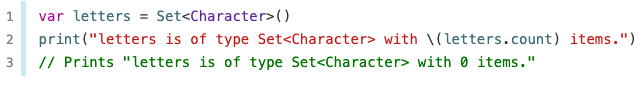
Un set stochează valori distincte de același tip într-o colecție fără o anume ordine. Puteți utiliza un set în locul unui array când ordinea articolelor nu este importantă sau când trebuie să vă asigurați că un element apare o singură dată.

Un tip trebuie să fie hashable pentru a fi stocat într-un set - adică tipul trebuie să ofere o modalitate de a calcula o valoare hash pentru el însăși. O valoare hash este o valoare int care este aceeași pentru toate obiectele care se compară egal, astfel încât dacă a == b, rezultă că a.hashValue == b.hashValue.

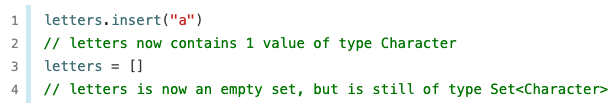
Toate tipurile de bază Swift (cum ar fi String, Int, Double și Bool) sunt hashable implicit și pot fi utilizate ca tipuri de valoare pentru un Set sau precum chei ale unui dicționar. Valorile cazurilor de enumerare fără valori asociate sunt, de asemenea, hashable implicit.

Puteți utiliza propriile tipuri ca tipuri de valoare pentru un Set sau precum chei ale unui dicționar, făcându-le să fie conforme cu protocolul Hashable din biblioteca standard Swift. Tipurile care sunt conforme cu protocolul Hashable trebuie să furnizeze o proprietate de tipul Int numită hashValue. Valoarea returnată de proprietatea hashValue a unui tip nu trebuie să fie aceeași pentru diferite execuții ale aceluiași program sau în diferite programe.

Puteți crea un set gol de anumit tip utilizând sintaxa constructorului:

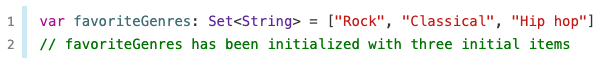
  
Figura 29

În mod alternativ, dacă contextul oferă deja informații de tip, cum ar fi un argument de funcție sau o variabilă sau o constantă deja declarată, puteți crea un set gol, cu un array literal gol:

  
Figura 30

De asemenea, puteți inițializa un set cu un array literal, ca o modalitate scurtă de a scrie una sau mai multe valori ca o colecție de tip Set. Exemplul de mai jos creează un set numit favoriteGenres ce stochează valori de tip String:

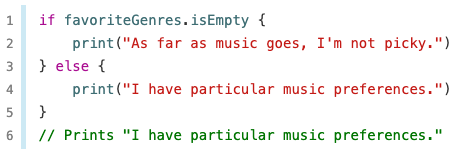
Variabila favoriteGenres este declarată ca "un set de valori String", scrisă ca Set<String>. Deoarece acest set are specificat ca tip de valoare String, este permisă numai stocarea valorilor String. Aici, setul favoriteGenres este inițializat cu trei valori String ("Rock", "Classical" și "Hip hop"), scrise într-un array literal.

  
Figura 31

1.2.2.1 Accesarea și modificarea unui Set

Accesul și modificarea unui set se face prin metodele și proprietățile sale.

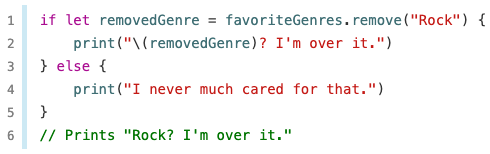
Utilizați proprietatea isEmpty ca o comandă rapidă pentru a verifica dacă proprietatea count este egală cu 0:

  
Figura 32

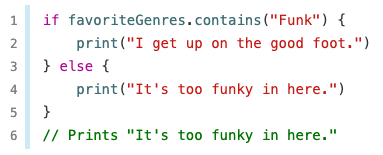
Puteți adăuga un element nou într-un set prin apelarea metodei insert(\_:) a setului:

  
Figura 33

Puteți elimina un element dintr-un set apelând metoda remove(\_:) a setului, care elimină elementul dacă este membru al setului și returnează valoarea eliminată sau nil dacă setul nu conține valoarea. În mod alternativ, toate elementele dintr-un set pot fi eliminate cu metoda removeAll().

  
Figura 34

Pentru a verifica dacă un set conține un anumit element sau nu, se poate utiliza metoda contains(\_:).

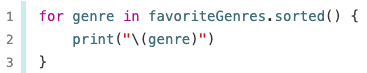
  
Figura 35

1.2.2.2 Iterarea printr-un Set

Puteți itera peste valorile dintr-un set cu o buclă for-in.

  
Figura 36

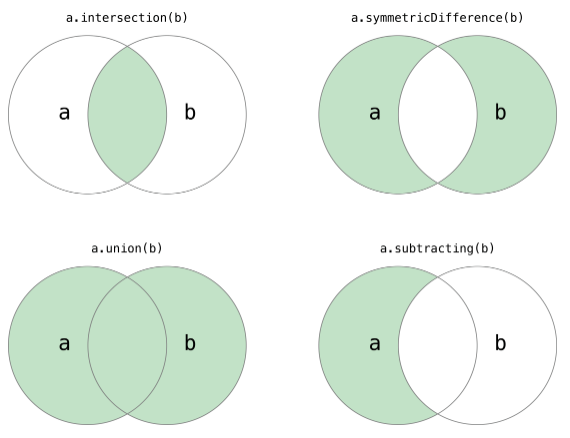
Tipul Set din Swift nu are o ordine definită. Pentru a itera peste valorile unui set într-o anumită ordine, utilizați metoda sorted(), care returnează elementele setului ca un array sortat folosind operatorul <.

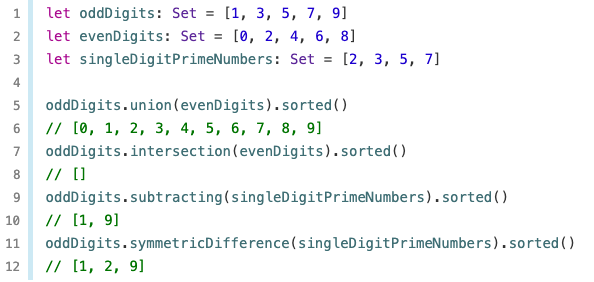
  
Figura 37

Puteți efectua eficient operațiile de Set fundamentale, cum ar fi combinarea a două seturi împreună, determinarea valorilor pe care două seturi le au în comun sau determinarea dacă două seturi conțin toate, unele sau nici una dintre aceleași valori. Set-ul din Swift este analogul mulțimii din matematică.

Ilustrația de mai jos descrie două seturi - a și b - cu rezultatele diferitelor operații de Set reprezentate de regiunile umbrite.

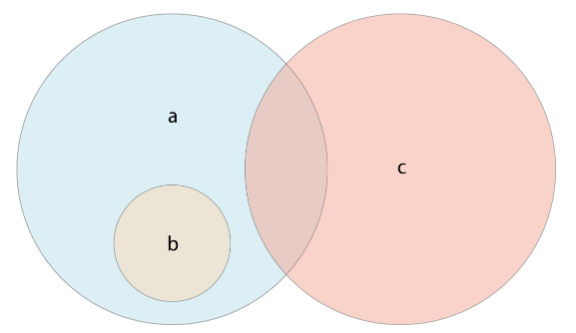
* Utilizați metoda intersection(\_:) pentru a crea un set nou, cu valori comune celor două seturi.
* Utilizați metoda symmetricDifference(\_:) pentru a crea un set nou cu valori în oricare dintre seturi, dar nu ambele.
* Utilizați metoda union(\_:) pentru a crea un set nou cu toate valorile din ambele seturi.
* Utilizați metoda de subtracting(\_:) pentru a crea un set nou cu valori care nu se află în setul specificat.

  
Figura 38

  
Figura 39

1.2.2.3 Calitatea de membru și egalitate

Ilustrația de mai jos descrie trei seturi - a, b și c - cu regiunile suprapuse reprezentând elemente partajate între seturi. Setul a este un superset al setului b, deoarece a conține toate elementele din b. În schimb, setul b este un subset al setului a, deoarece toate elementele din b sunt de asemenea conținute de a. Setul b și setul c sunt disjuncte unul cu celălalt, deoarece nu au elemente în comun.

  
Figura 40

* Utilizați operatorul "este egal" (==) pentru a determina dacă două seturi conțin aceleași valori.
* Utilizați metoda isSubset(of:) pentru a determina dacă toate valorile unui set sunt conținute în setul specificat.
* Utilizați metoda isSuperset(of:) pentru a determina dacă un set conține toate valorile dintr-un set specificat.
* Utilizați metodele isStrictSubset(of:) sau isStrictSuperset(of:) pentru a determina dacă un set este un subset sau superset, dar nu egal cu, un set specificat.
* Utilizați metoda isDisjoint(with:) pentru a determina dacă două seturi nu au valori comune.

  
Figura 41

1.2.3 Dictionary

Un dicționar stochează asociații între chei de același tip și valori de același tip într-o colecție fără o ordine definită. Fiecare valoare este asociată cu o cheie unică, care acționează ca un identificator pentru acea valoare în dicționar. Spre deosebire de elemente dintr-un array, elementele dintr-un dicționar nu au o ordine specificată. Utilizați un dicționar când trebuie să căutați valori pe baza identificatorului lor, în același mod în care un dicționar al lumii reale este folosit pentru a căuta definiția unui anumit cuvânt.

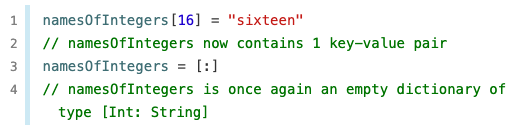
Tipul unui dicționar Swift este scris în întregime ca Dicționar<Cheie, Valoare>, unde Cheia este tipul de valoare care poate fi folosit ca o cheie în dicționar, iar Valoarea este tipul de valoare pe care dicționarul le stochează pentru acele chei. Cheia unui dicționar trebuie să implementeze protocolul Hashable, precum tipul de valoare al unui Set.

Ca și în cazul array-urilor, puteți crea un Dicționar gol al unui anumit tip utilizând sintaxa de inițializare:

  
Figura 42

Acest exemplu creează un dicționar gol de tipul [Int: String] pentru a stoca numele numerelor într-o manieră familiară oamenilor. Cheile sale sunt de tip Int, iar valorile sale sunt de tip String.

Dacă contextul conține deja informații de tip, puteți crea un dicționar gol, cu un literal gol, literal, care este scris ca [:] (două puncte în interiorul unei perechi de paranteze pătrate):

  
Figura 43

De asemenea, puteți inițializa un dicționar cu un dicționar literal, care are o sintaxă similară cu array-ul literal văzut mai devreme. Dicționarul literal este o modalitate scurtă de a scrie una sau mai multe perechi cheie-valoare ca o colecție de tip dicționar.

O pereche cheie-valoare reprezintă o combinație a unei chei și a unei valori. Cheia și valoarea fiecărei perechi cheie-valoare sunt separate de două puncte. Perechele cheie-valoare sunt scrise ca o listă, separate prin virgule, înconjurate de o pereche de paranteze pătrate:

  
Figura 44

Exemplul de mai jos creează un dicționar pentru a stoca numele aeroporturilor internaționale. În acest dicționar, cheile sunt codurile internaționale de transport aerian de trei litere, iar valorile sunt numele aeroporturilor:

  
Figura 45

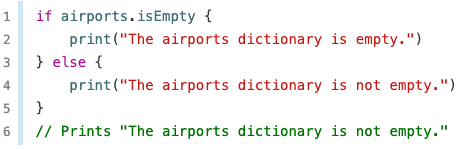
Dicționarul aeroporturilor este inițializat cu un dicționar literal care conține două perechi cheie-valoare. Prima pereche are o cheie "YYZ" și o valoare "Toronto Pearson". A doua pereche are o cheie "DUB" și o valoare "Dublin".

Acest dicționar literal conține două perechi de tip String: String. Acest tip de cheie-valoare corespunde tipului de declarație a variabilei aeroportului (un dicționar care conține numai chei de tip String și numai de tip String), astfel încât alocarea dicționarului literal este permisă ca o modalitate de a inițializa dicționarul aeroporturilor cu două elemente inițiale.

1.2.3.1 Accesarea și modificarea unui Dicționar

Accesați și modificați un dicționar prin metodele și proprietățile acestuia sau utilizând sintaxa de subscript.

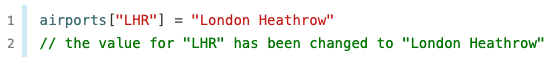
Utilizați proprietatea de tip Bool isEmpty ca o comandă rapidă pentru a verifica dacă proprietatea count este egală cu 0.

  
Figura 46

Puteți adăuga un element nou într-un dicționar folosind subscriptul. Utilizați o nouă cheie de tipul corespunzător index-ului și asociați o nouă valoare de tipul corespunzător valorilor:

  
Figura 47

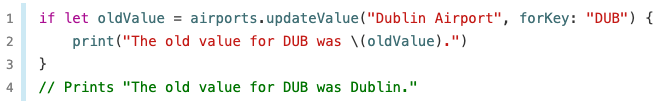
De asemenea, puteți utiliza subscriptul pentru a modifica valoarea asociată unei anumite chei:

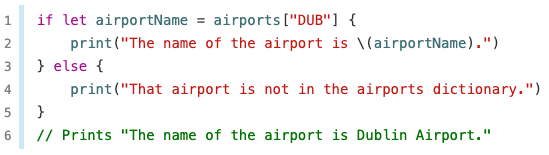
  
Figura 48

Ca alternativă la subscript, utilizați metoda updateValue(\_:forKey:) a dicționarului pentru a seta sau a actualiza valoarea unei anumite chei. Ca și în exemplele de subscript de mai sus, metoda updateValue(\_:forKey:) stabilește o valoare pentru o cheie dacă nu există sau actualizează valoarea dacă cheia există deja. Spre deosebire de un subscript, totuși, metoda updateValue(\_:forKey:) returnează vechea valoare după efectuarea unei actualizări. Aceasta vă permite să verificați dacă a avut loc sau nu o actualizare.

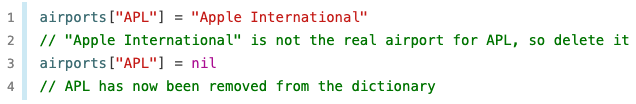
Metoda updateValue(\_:forKey:) returnează o valoare opțională de tipul corespunzător valorilor dicționarului. Pentru un dicționar care stochează valori de tip String, de exemplu, metoda returnează o valoare de tip String? Sau "String opțional". Această valoare opțională conține valoarea veche pentru acea cheie, dacă exista o asemenea valoare, sau nil dacă nu exista o valoare pentru cheia respectiva:

De asemenea, puteți utiliza subscriptul pentru a extrage valoarea din dicționar pentru o anumită cheie. Deoarece este posibilă solicitarea unei chei pentru care nu există nici o valoare, subscriptul unui dicționar returnează o valoare opțională a tipului de valoare stocat. Dacă dicționarul conține o valoare pentru cheia solicitată, subscriptul returnează o valoare opțională care conține valoarea existentă pentru acea cheie. În caz contrar, va fi returnat nil:

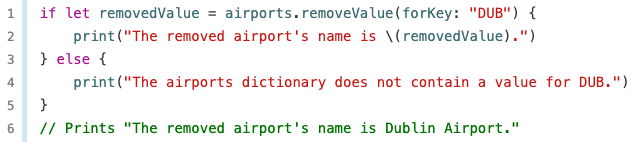
  
Figura 49

  
Figura 50

Puteți utiliza subscriptul pentru a elimina o pereche cheie-valoare dintr-un dicționar prin atribuirea valorii nil pentru acea cheie:

  
Figura 51

Alternativ, puteți elimina o pereche cheie-valoare dintr-un dicționar folosind metoda removeValue(forKey:). Această metodă elimină perechea cheie-valoare dacă există și returnează valoarea eliminată sau returnează valoarea nil dacă nu exista nici o valoare:

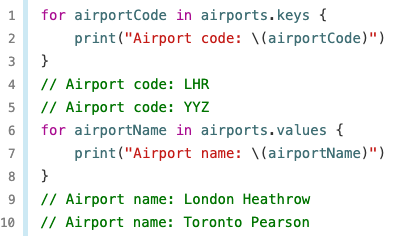
  
Figura 52

1.2.3.2 Iterarea printr-un Dicționar

Puteți itera prin perechile cheie-valoare dintr-un dicționar cu o buclă for-in. Fiecare element din dicționar este returnat ca un tuplu (cheie, valoare) și puteți descompune membrii tuplului în constante sau variabile temporare ca parte a iterației:

  
Figura 53

De asemenea, puteți obține o colecție iterabilă de chei sau valori ale unui dicționar accesând proprietățile keys și values:

  
Figura 54

1.3 Construcțiile pentru control fluxului

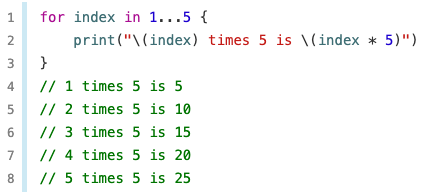
Swift oferă o varietate de construcții pentru controlul fluxului. Acestea includ buclele repetitive cu număr necunoscut de pași, „while”; „if”, „guard” și „switch” pentru a executa diferite ramuri ale codului pe baza anumitor condiții; și instrucțiuni precum „break” și „continue” pentru a transfera fluxul de execuție într-un alt punct al codului.

Swift furnizează, de asemenea, o buclă for-in care ușurează interarea peste array-uri, dicționare, intervale, șiruri de caractere și alte secvențe.

Declarația switch-ului în Swift este considerabil mai puternică decât omoloaga sa în multe limbaje ce au la baza C. Cazurile se pot potrivi cu multe modele diferite, incluzând intervale, tupluri și castarea la tipuri specifice. Valorile potrivite într-un switch pot fi asignate unor constante temporare sau variabile pentru utilizarea lor în corpul cazului, iar condițiile complexe de potrivire pot fi exprimate cu o clauză „where” pentru fiecare caz în parte.

1.3.1 Buclele for-in

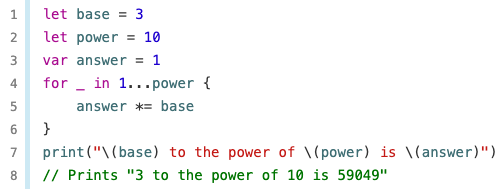
Pe lânga exemplele de mai sus în care am folosit bucle de tipul for-in pentru a itera prin valorile unor colecții precum array-uri sau dicționare, buclele for-in pot fi folosite și pentru a itera printr-un interval numeric:

  
Figura 55

Secvența care este iterată este o serie de numere de la 1 la 5, inclusiv, după cum este indicat de utilizarea operatorului de interval închis (...). Valoarea indexului este setată la primul număr din intervalul (1), iar instrucțiunile din bucla sunt executate. În acest caz, buclă conține o singură instrucțiune, care printează o intrare din tabla înmulțirii lui cinci pentru valoarea curentă a indexului. După executarea instrucțiunii, valoarea indexului este actualizată pentru a conține a doua valoare din intervalul (2), iar funcția print(\_:separator:terminator:) este chemată din nou. Acest proces continuă până la sfârșitul intervalului.

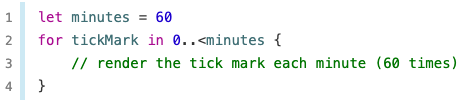
În exemplul de mai sus, indexul este o constantă a cărei valoare este setată automat la începutul fiecărei iterații a buclei. Ca atare, indicele nu trebuie declarat înainte de a fi utilizat. Este implicit declarată pur și simplu prin includerea sa în declarația de buclă, fără a fi nevoie de un cuvânt cheie cu declarație „let”.

Dacă nu aveți nevoie de fiecare valoare dintr-o secvență, puteți ignora valorile utilizând o subliniere (\_) în locul unui nume de variabilă.

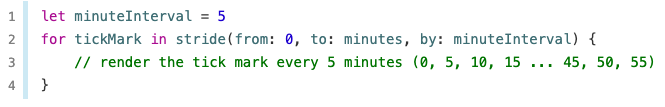
  
Figura 56

Exemplul de mai sus calculează valoarea unui număr la puterea altuia (în acest caz 3 la puterea de 10). Se multiplică o valoare de pornire de 1 (adică 3 la puterea de 0) cu 3, de zece ori, folosind un interval închis care începe cu 1 și se termină cu 10. Pentru acest calcul, valoarea contorului la fiecare trecere prin buclă, este redundantă - codul execută pur și simplu bucla de câte ori este nevoie. Caracterul de subliniere (\_) folosit în locul unei variabile de buclă determină ignorarea valorilor individuale și nu permite accesul la valoarea curentă în timpul iterației buclei.

În anumite situații, este posibil să nu doriți să utilizați intervale închise, care includ ambele capete ale intervalului. Luați în considerare desenarea marcajelor pentru fiecare minut pe o față de ceas. Doriți să desenați 60 de marcaje, începând cu minutul 0. Utilizați operatorul de interval semi-deschis (.. <) pentru a include limita inferioară, dar nu limita superioară.

  
Figura 57

Unii utilizatori ar putea dori mai puține marcaje pe interfața lor utilizator. Ar putea prefera o linie la fiecare 5 minute. Utilizați funcția stride(from:to:by:) pentru a sări peste marcajele nedorite.

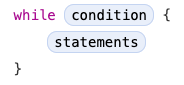
  
Figura 58

Intervalele închise sunt, de asemenea, disponibile, folosind stride(from:through:by:) în schimb:

  
Figura 59

1.3.2 Buclele cu număr necunoscut de pași pre-condiționate (while)

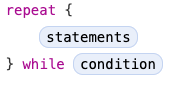
O buclă de tip while efectuează un set de instrucțiuni până când o condiție devine falsă. Aceste tipuri de bucle sunt utilizate cel mai bine atunci când numărul de iterații nu este cunoscut înainte de începerea primei iterații. Ele au forma generală:

  
Figura 60

1.3.3 Buclele cu număr necunoscut de pași post-condiționate (repeat-while)

Cealaltă variantă a buclei while, cunoscută sub numele de repeat-while, efectuează o trecere prin blocul buclei, înainte de a lua în considerare condiția buclei. Apoi continuă să repete bucla până când condiția este falsă. Bucla repeat-while în Swift este analoaga buclei do-while din alte limbaje. Această buclă prezintă dezavantajul executării codului o data chiar dacă condiția era falsă incă de la

început, pentru asta se recomandă verificarea condiției înainte de a intra în bucla propriu-zisă. Ea având forma generală:

  
Figura 61

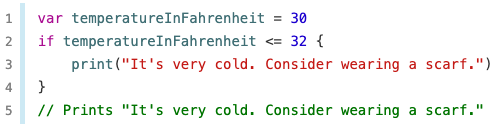
1.3.4 Structuri condiționale

Este adesea utilă executarea unor diferite bucăți de cod pe baza anumitor condiții. S-ar putea să doriți să executați o bucată suplimentară de cod atunci când apare o eroare sau să afișați un mesaj atunci când o valoare devine prea mare sau prea mică. Pentru a face acest lucru, faceți ca anumite părți ale codului să fie condiționate.

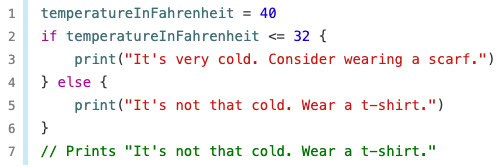
Swift oferă două moduri de a adăuga ramificații condiționate la codul dvs.: instrucțiunea if și instrucțiunea switch. De obicei, utilizați instrucțiunea if se folosește pentru a evalua condițiile simple, cu doar câteva rezultate posibile. Instrucțiunea switch este mai potrivită pentru condițiile mai complexe cu multiple permutări posibile și este utilă în situațiile în care potrivirea unui tipar poate ajuta la selectarea unei ramuri de cod corespunzătoare pentru a fi executată.

1.3.4.1 If

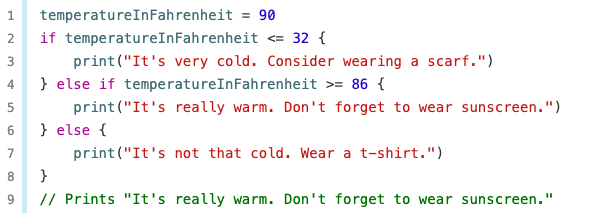
În forma sa cea mai simplă, instrucțiunea if are o singură condiție. Execută un set de instrucțiuni numai dacă această condiție este adevărată.

  
Figura 62

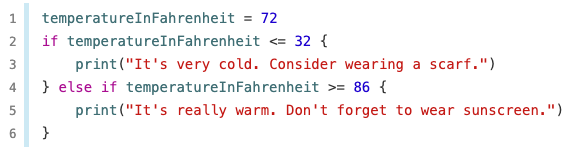
Instrucțiunea if poate oferi un set alternativ de declarații, cunoscut sub numele de clauză else, pentru situațiile în care condiția if este falsă.

  
Figura 63

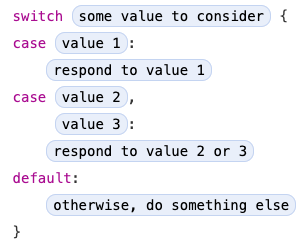
Aveți posibilitatea de a înlănțui mai multe instrucțiuni if, pentru a lua în considerare clauze suplimentare.

  
Figura 64

Cu toate acestea, clauza else este opțională și poate fi exclusă dacă setul de condiții nu trebuie să fie complet.

  
Figura 65

1.3.4.2 Switch

  
Figura 66

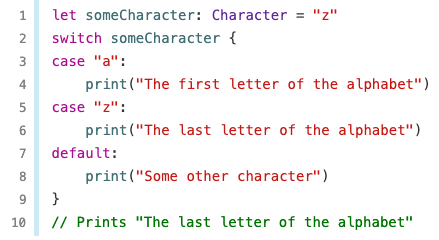
Un switch ia în considerare o valoare și o compară cu mai multe modele posibile de potrivire. Apoi execută un bloc corespunzător de cod, bazat pe primul model care se potrivește. Un switch oferă o alternativă la instrucțiunea if pentru a răspunde la mai multe potențiale stări. În cea mai simplă formă, un switch compară o valoare cu una sau mai multe valori de același tip.

Fiecare instrucțiune switch constă din mai multe cazuri posibile, fiecare dintre acestea începând cu cuvântul cheie „case”. În plus față de compararea cu valori specifice, Swift oferă mai multe moduri pentru fiecare caz pentru a specifica modele mai complexe de potrivire.

Ca și corpul unei declarații if, fiecare caz este o ramură separată a execuției codului. Instrucțiunea switch determină ce ramură ar trebui selectată.

Fiecare switch trebuie să fie exhaustiv. Adică, fiecare valoare posibilă a tipului luat în considerare trebuie să corespundă cu unul dintre cazurile de comutare. Dacă nu se poate să furnizați un caz pentru fiecare valoare posibilă, puteți defini un caz implicit pentru a acoperi orice valoare care nu este adresată în mod explicit. Acest caz implicit este indicat de cuvântul cheie „default” și trebuie să apară întotdeauna ultimul.

Acest exemplu folosește o instrucțiune switch pentru a lua în considerare un singur caracter minuscul numit someCharacter:

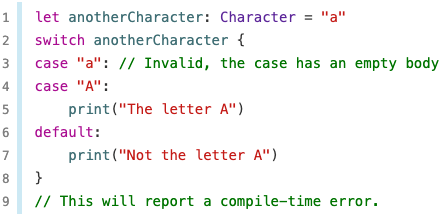
  
Figura 67

Primul caz al instrucțiunii switch se potrivește cu prima literă a alfabetului englez, a, iar al doilea caz se potrivește cu ultima literă, z. Deoarece switch-ul trebuie să aibă un caz pentru fiecare caracter posibil, nu doar pentru fiecare caracter alfabetic, această instrucțiune switch utilizează un caz implicit pentru a se potrivi cu toate caracterele, altele decât a și z. Această dispoziție asigură faptul că switch-ul este exhaustiv.

În contrast cu switch-urile din C și Objective-C, switch-urile din Swift nu trec în următorul caz al potrivirii în mod implicit. În schimb, întreagul switch termină executarea de îndată ce primul caz de potrivire este finalizat, fără a necesita o declarație „break” explicită. Aceasta face ca switch-ul să fie mai sigur și mai ușor de folosit decât cel din C și evită executarea a mai mult de un caz de potrivire din greșeală.

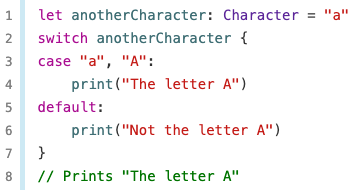
Deși instrucțiunea „break” nu este necesară în Swift, o puteți utiliza pentru a ignora un anumit caz sau pentru a ieși dintr-un caz potrivit înainte ca acel caz să-și fi finalizat execuția.

Corpul fiecărui caz trebuie să conțină cel puțin o declarație executabilă. Nu este valabil să scrieți următorul cod, deoarece primul caz este gol:

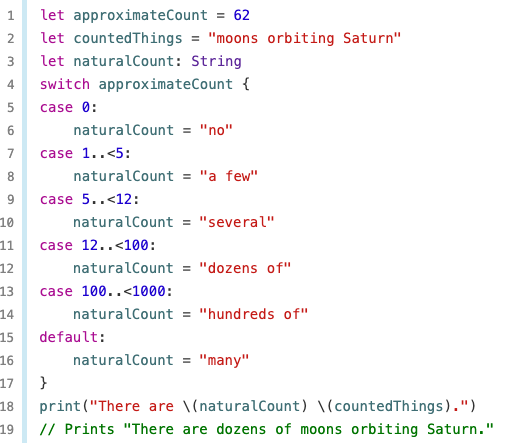
  
Figura 68

Spre deosebire de o instrucțiune switch în C, acest switch nu se potrivește atât cu "a" cât și cu "A". Mai degrabă, raportează o eroare de compilare în acel caz "a": nu conține nicio declarație executabilă. Această abordare evită căderi accidentale de la un caz la altul și face codul mai sigur și mai clar în intenția sa.

Pentru a face un switch cu un singur caz care se potrivește atât cu "a" cât și cu "A", combinați cele două valori într-un caz compus, separând valorile prin virgule.

  
Figura 69

Valorile unui switch pot fi verificate dacă aparțin unui interval.

  
Figura 70

În exemplul de mai sus, aproximateCount este evaluată într-o instrucțiune switch. Fiecare caz compară această valoare cu un număr sau un interval. Deoarece valoarea aproximateCount se situează între 12 și 100, naturalCount este atribuită valoarea "zeci de", iar execuția se oprește acolo.

Puteți utiliza tupluri pentru a testa mai multe valori în același switch. Fiecare element al tuplului poate fi testat cu o valoare diferită sau cu un interval de valori diferite. Alternativ, utilizați caracterul de subliniere (\_), cunoscut și „wildcard”, pentru a se potrivi cu orice valoare posibilă.

Exemplul de mai jos are un punct (x, y), exprimat ca un simplu tuplu de tip (Int, Int) și îl clasifică pe graficul care urmează exemplului.

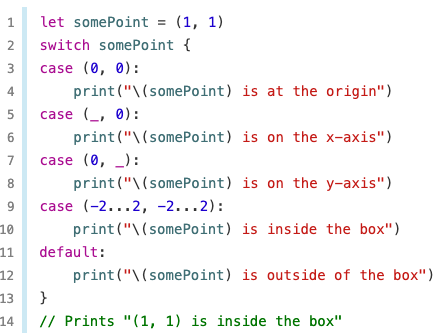
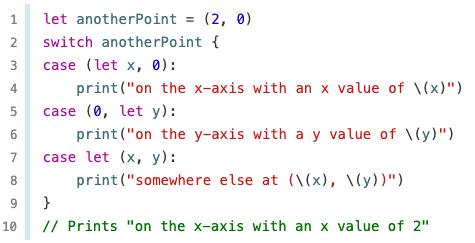
  
Figura 71

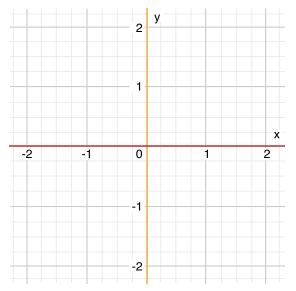
  
Figura 72

Spre deosebire de C, Swift permite mai multor cazuri de comutare să ia în considerare aceeași valoare sau valori. De fapt, punctul (0, 0) ar putea corespunde tuturor celor patru cazuri din acest exemplu. Cu toate acestea, dacă sunt posibile mai multe potriviri, primul caz de potrivire este întotdeauna utilizat. Punctul (0, 0) s-ar potrivi mai întâi cazului (0, 0), astfel încât toate celelalte cazuri de potrivire ar fi ignorate.

1.3.4.3 Asocierea valorilor

Un caz de comutare poate numi valoarea sau valorile care se potrivesc cu constantele sau variabilele temporare, pentru a fi utilizate în corpul cazului. Acest comportament este cunoscut ca asocierea valorilor, deoarece valorile sunt legate de constante temporare sau variabile în corpul cazului. Exemplul de mai jos are un punct (x, y), exprimat ca un tuplu de tip (Int, Int) și îl clasifică pe graficul care urmează:

  
Figura 73

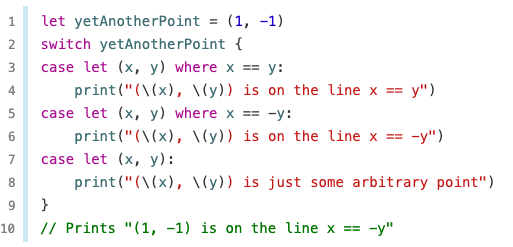
  
Figura 74

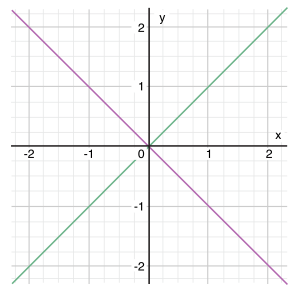
Cele trei cazuri de comutare declară constantele substituentului x și y, care iau temporar una sau ambele valori ale tuplului de la anotherPoint. Primul caz, case (let x, 0), se potrivește oricărui punct cu o valoare y de 0 și atribuie valoarea punctului x constantei temporare x. În mod similar, cel de-al doilea caz, case (0, let y), se potrivește oricărui punct cu o valoare x de 0 și atribuie valoarea y constantei temporare y.

Această instrucțiune switch nu are un caz implicit. Cazul final, case let (x, y), declară un tuplu de două constante de substituent care pot potrivi orice valoare. Deoarece anotherPoint este întotdeauna un tuplu de două valori, acest caz se potrivește cu toate valorile posibile rămase, iar un caz implicit nu este necesar pentru a face switch-ul exhaustiv.

1.3.4.4 Clauza where

Un caz de comutare poate utiliza o clauză „where„ unde să se verifice condiții suplimentare. Exemplul de mai jos categorizează un punct (x, y) pe următorul grafic:

  
Figura 75

  
Figura 76

Cele trei cazuri de comutare declară constantele x și y, care preiau temporar cele două valori ale tuplului din yetAnotherPoint. Aceste constante sunt folosite ca parte a unei clauze „where”, pentru a crea un filtru dinamic. Cazul de comutare se potrivește cu valoarea curentă a punctului numai dacă condiția clauzei „where” este adevărată pentru acea valoare. Ca și în exemplul precedent, cazul final se potrivește cu toate valorile posibile rămase, deci un caz implicit nu este necesar pentru a face declarația comutatorului exhaustivă.

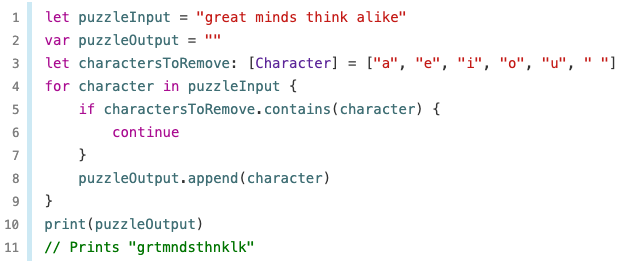
1.3.5 Instrucțiuni pentru transferul controlului

Comenzile pentru transferul de control schimbă ordinea în care codul dvs. este executat, transferând controlul de la o singură bucată de cod la alta. Swift are cinci declarații de transfer de control:

* continue
* break
* fallthrough
* return
* throw

1.3.5.1 Continue

Instrucțiunea „continue” spune unei bucle să oprească ceea ce face și să înceapă iterația următoare prin bucla. Spune "Am terminat cu iterația curentă a buclei", fără a părăsi cu totul bucla. Următorul exemplu elimină toate vocalele și spațiile dintr-un șir de caractere minuscule pentru a crea o frază criptată:

  
Figura 77

1.3.5.2 Break

Instrucțiunea break termină imediat executarea unei întregi instrucțiuni de controlul fluxlui. Instrucțiunea break poate fi utilizată în interiorul unui switch sau unei bucle atunci când doriți să terminați executarea instrucțiunii mai devreme decât s-ar întâmpla în caz contrar.

Atunci când se utilizează în interiorul unei bucle, instrucțiunea „break” termină execuția buclei imediat și transferă controlul codului după acolada de închidere a buclei (}). Nu se execută nici un cod suplimentar din iterația curentă a buclei și nu se pornesc alte iterații ale buclei.

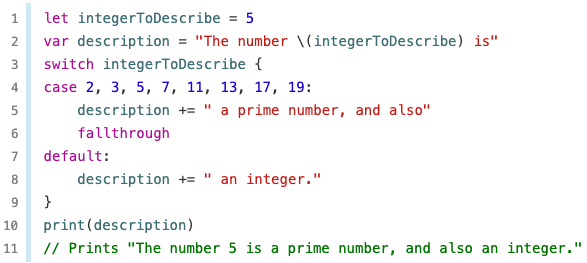
Atunci când se utilizează în interiorul unui switch, „break” determină switch-ul să-și încheie imediat execuția și să se transfere controlul la cod după acolada de închidere a instrucțiunii (}).

Acest comportament poate fi folosit pentru a ignora unul sau mai multe cazuri dintr-un switch. Deoarece un switch în Swift este exhaustiv și nu permite cazuri goale, este uneori necesar să se potrivească în mod deliberat și să se ignore un caz pentru a vă exprima intențiile. Faceți acest lucru prin scrierea instrucțiunii break ca întregul corp al cazului pe care doriți să îl ignorați. Atunci când acest caz este asociat cu o valoare, instrucțiunea break din interiorul cazului încheie imediat executarea instrucțiunii.

1.3.5.3 Fallthrough

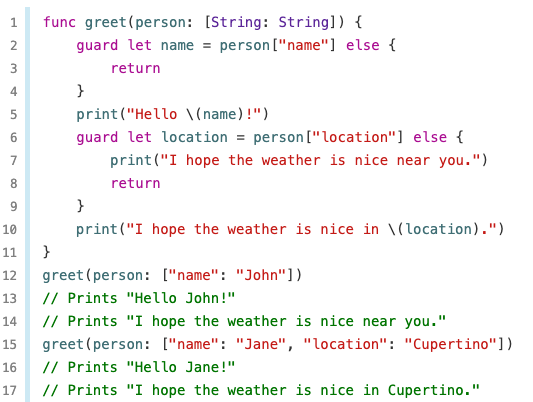
În Swift, switch-urile nu cad în următorul caz în mod implicit. Aceasta înseamnă că întregul switch își completează execuția de îndată ce se încheie primul caz de potrivire. În schimb, C vă cere să introduceți un break explicit la sfârșitul fiecărui caz pentru a preveni căderea în următorul caz. Evitarea căderii implicite înseamnă că switch-urile în Swift sunt mult mai concise și mai previzibile decât omoloagele lor în C și, prin urmare, evită executarea mai multor cazuri de potrivire din greșeală.

Dacă aveți nevoie de un comportament descendent în stil C, puteți opta pentru acest comportament de la caz la caz, cu ajutorul cuvântului cheie „falltrough”. Exemplul de mai jos folosește căderea pentru a crea o descriere textuală a unui număr.

  
Figura 78

1.3.5.4 Ieșirea timpurie din scop

O instrucțiune de tip „guard”, precum o instrucțiune if, execută declarații în funcție de valoarea Booleană a unei expresii. Utilizați un guard în scopul de a solicita ca o condiție să fie adevărată pentru ca codul după executarea instrucțiunii să fie executat. Spre deosebire de o declarație if, o declarație guard are întotdeauna o clauză else - codul din clauza else este executat dacă condiția nu este adevărată.

  
Figura 79

Dacă expresia din guard este îndeplinită, executarea de cod continuă după acolada de închidere al instrucțiunii guard. Orice variabile sau constante cărora le-au fost atribuite valori utilizând optional binding ca parte a condiției sunt disponibile pentru restul blocului de cod în care apare instrucțiunea guard.

Dacă această condiție nu este îndeplinită, codul din cadrul celeilalte ramuri este executat. Această ramură trebuie să transfere controlul pentru a ieși din blocul de cod în care apare declarația guard. Se poate face acest lucru cu o instrucțiune de transfer de control, cum ar fi return, break, continue sau throw, sau poate apela o funcție sau o metodă care nu întoarce vreun rezultat, cum ar fi fatalError(\_:[file:line](../../../line):).

Utilizarea unui guard pentru cerințe îmbunătățește citirea codului dvs., în comparație cu efectuarea verificării folosind o instrucțiune if. Vă permite să scrieți codul care este executat de obicei fără a îl înfășura într-un bloc else și vă permite să păstrați lângă cerință codul care se ocupă de cazul unde ea este falsă.

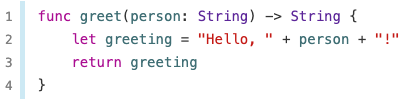
1.4 Funcții (metode)

Funcțiile sunt bucăți autonome de cod care îndeplinesc o sarcină specifică. I se dă unei funcții un nume care identifică ceea ce face, iar acest nume este folosit la "apelul" funcției pentru a-și îndeplini sarcina atunci când este necesar.

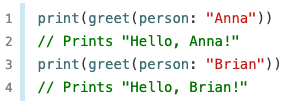
Fiecare funcție din Swift are un tip, constând din tipurile de parametri ale funcției și tipul de returnare. Puteți utiliza acest tip ca orice alt tip în Swift, ceea ce face ușor să pasați funcțiile ca parametri altor funcții și să returnați funcții din funcții. Funcțiile pot fi, de asemenea, scrise în cadrul altor funcții pentru a încapsula o funcționalitate utilă într-un mediu privat.

Când definiți o funcție, puteți defini opțional una sau mai multe valori numite, pe care funcția le are ca intrare, cunoscute sub numele de parametri. De asemenea, puteți să definiți opțional un tip de valoare pe care funcția o va transmite înapoi atunci când se termină, cunoscută ca tipul de retur.

Funcția din exemplul de mai jos se numește greet(person:), pentru că asta e ceea ce face - este nevoie de numele persoanei ca intrare și returnează un salut pentru acea persoană. Pentru a realiza acest lucru, definiți ca parametru de intrare o valoare String numită person, și un tip de retur de tip String, care va conține salutul pentru acea persoană:

  
Figura 80

Definiția descrie ce face funcția, ce așteaptă să primească și ce se întoarce atunci când este executată. Definiția face mai ușoră sarcina de a o apela fără echivoc în altă parte a codului dvs.:

  
Figura 81

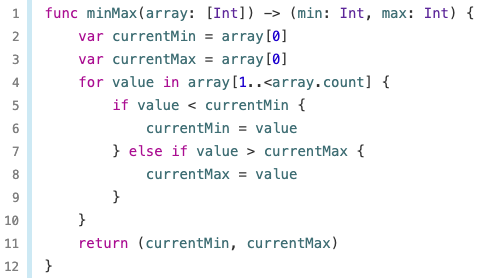
1.4.1 Funcții fără tip de retur

Funcțiile nu sunt obligate să definească un tip de retur. Deoarece nu este necesară returnarea unei valori, definiția funcției nu include săgeata de întoarcere (->) sau un tip de returnare. Iată o versiune a funcției greet(person:), care printează valoarea String-ului în loc să o returneze:

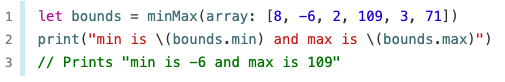
  
Figura 82

1.4.2 Funcții ce returnează mai multe valori

Puteți utiliza un tuplu ca tip de returnare pentru ca o funcție să returneze mai multe valori ca o singură valoare compusă.

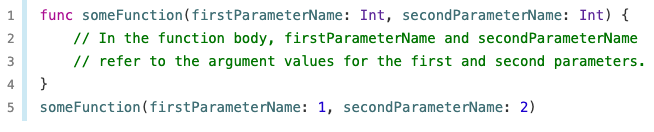
  
Figura 83

Funcția minMax(array:) returnează un tuplu care conține două valori Int. Aceste valori sunt etichetate min și max, astfel încât să poată fi accesate prin nume atunci când valoarea de retur a funcției este folosită.

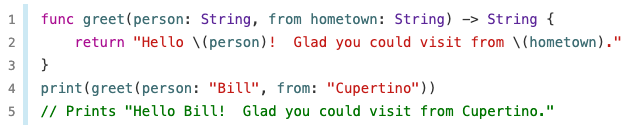
  
Figura 84

1.4.3 Etichetarea argumentelor și numele parametrilor

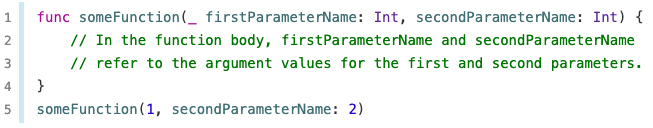
Fiecare parametru de funcții are atât o etichetă de argument, cât și un nume de parametru. Eticheta de argument este utilizată la apelarea funcției; fiecare argument este scris în apelul funcției cu eticheta sa de argument înaintea acestuia. Numele parametrului este utilizat în implementarea funcției. În mod prestabilit, parametrii utilizează numele parametrului ca etichetă argument.

  
Figura 85

Se scrie o etichetă de argument înainte de numele parametrului, separat de un spațiu. Utilizarea etichetelor de argumentare poate permite unei funcții să fie chemată într-o manieră expresivă, asemănătoare unei fraze, oferind în același timp un corp de funcție care poate fi citit ușor și este clar în intenție. Iată o variantă a funcției greet(person:) care primește numele persoanei și orașul natal și returnează un salut:

  
Figura 86

Dacă nu doriți o etichetă de argument pentru un parametru, scrieți un subliniere (\_) în locul unei etichete de argument explicite pentru acel parametru. Dacă un parametru are o etichetă de argument, argumentul trebuie să fie etichetat când apelați funcția.

  
Figura 87

Puteți defini o valoare implicită pentru orice parametru dintr-o funcție atribuind o valoare parametrului după tipul respectivului parametru. Dacă este definită o valoare implicită, puteți omite acest parametru la apelarea funcției.

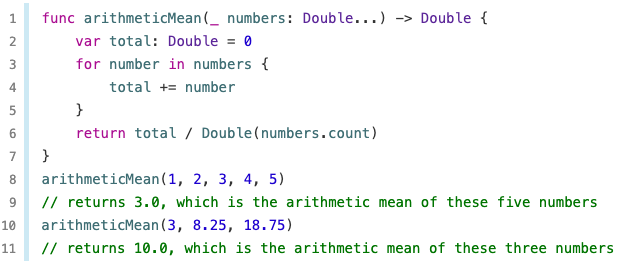
  
Figura 88

1.4.4 Parametrii variadici

Un parametru variadic acceptă zero sau mai multe valori ale unui tip specificat. Utilizați un parametru variadic pentru a specifica faptul că parametrul poate fi trecut cu un număr diferit de

valori de intrare atunci când este apelată funcția. Scrieți parametrii variadic introducând trei puncte (...) după numele tipului parametrului.

Valorile transmise unui parametru variadic sunt puse la dispoziție în corpul funcției ca o array de acel tip.

  
Figura 89

1.5 Closures (funcții anonime)

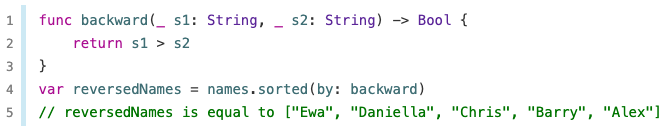
Closures sunt blocuri de funcționalitate autonome care pot fi transmise și utilizate în codul dvs. Closures din Swift sunt similare blocurilor în C și Objective-C sau funcții lambda în alte limbaje de programare. Closures pot capta și stoca referințe la orice constantă și variabilă din contextul în care sunt definite. Aceasta se numește „închidere” peste acele constante și variabile.

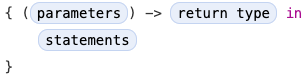
Biblioteca standard Swift oferă o metodă numită sorted(by:), care sortează o gamă de valori de un tip cunoscut, bazată pe rezultatul unui closure de sortare pe care îl furnizați. După finalizarea procesului de sortare, metoda sorted(by:) returnează un nou array de același tip și dimensiune ca cel inițial, cu elementele sale în ordinea ordonată corectă.

Metoda sorted(by:) acceptă un closure care ia două argumente de același tip ca și conținutul array-ului și returnează o valoare Bool pentru a spune dacă prima valoare ar trebui să apară înainte sau după cea de-a doua valoare. Closure-urile de sortare trebuie să întoarcă true dacă prima valoare ar trebui să apară înaintea celei de a doua valor, altfel false.

O modalitate de a furniza un closure de sortare este aceea de a scrie o funcție normală de tip corect și de a o transmite drept argument metodei sorted(by:):

Totuși, aceasta este o modalitate destul de lungă de a scrie ceea ce este în mod esențial o funcție de exprimare unică (a> b). În acest exemplu, ar fi preferabil să scrieți closure-ul folosind sintaxa expresiei de closure, care are forma generală:

  
Figura 90

  
Figura 91

Rețineți că declarația parametrilor și tipul returului pentru acest closure este identică cu declarația din funcția backward(\_:\_:). În ambele cazuri, este scris ca (s1: String, s2: String) -> Bool. Cu toate acestea, pentru expresia inline closure, parametrii și tipul de returnare sunt scrise în interiorul acoladelor și nu în afara acestora.

  
Figura 92

Începutul corpului closure-ului este introdus de cuvântul cheie „in”. Acest cuvânt cheie indică faptul că definiția parametrilor și tipul de returnare s-au terminat și corpul closure-ului este pe cale să înceapă.

1.6 Enumerația

Enumerările din Swift sunt tipuri de primă clasă. Ele adoptă multe caracteristici care în mod tradițional erau acceptate numai de clase, cum ar fi proprietățile calculate pentru a furniza informații suplimentare despre valoarea curentă a enumerării și metode de instanța pentru a furniza funcționalități legate de valorile pe care enumerarea le reprezintă. Enumerările pot de asemenea să definească constructori pentru a furniza o valoare inițială; pot fi extinse pentru a extinde funcționalitatea acestora dincolo de implementarea lor inițială; și se pot conforma protocoalelor pentru a oferi funcționalități standard.

  
Figura 93

Cazurile dintr-o enumerare Swift nu au o valoare intregă setată implicit, spre deosebire de limbajele precum C și Objective-C. În exemplului CompassPoint, nordul, sudul, estul și vestul nu sunt implicit egale cu 0, 1, 2 și 3. În schimb, diferitele cazuri de enumerare sunt valori de sine stătătoare, cu un tip explicit de tipul CompassPoint.

1.6.1 Valori asociate

Puteți defini enumerări în Swift pentru a stoca anumite valori asociate unui caz, tipurile de valori putând fi diferite pentru fiecare caz al enumerării, dacă este necesar. În Swift, o enumerare pentru definirea codurilor de bare pentru un produs ar putea arăta astfel:

  
Figura 94

Această definiție nu oferă nicio valoare reală de tip Int sau String - definește doar tipul de valori asociate pe care cazurile tipului Barcode le pot stoca atunci când sunt egale cu Barcode.upc sau Barcode.qrCode. Apoi puteți crea noi coduri de bare utilizând oricare caz al enumerației:

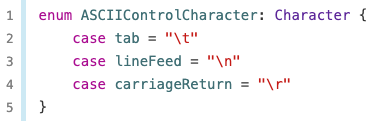
  
Figura 95

  
Figura 96

În acest moment, versiunea originală Barcode.upc și valorile sale întregi se înlocuiesc cu noul cod Barcode.qrCode și valoarea șirului acestuia. Constantele și variabilele de tip Barcode pot stoca fie un .upc sau un .qrCode (împreună cu valorile asociate), dar pot stoca numai unul dintre ele la un moment dat.

1.6.2 Valori de bază

Ca o alternativă la valorile asociate, cazurile enumerărilor pot veni prepopulate cu niște valorile implicite (numite valori de bază), care sunt toate de același tip.

  
Figura 97

Aici, valorile de baza pentru o enumerare numită ASCIIControlCharacter sunt definite ca fiind de tip Character, și sunt setate la unele dintre cele mai comune caractere ASCII de control.

Valorile de bază nu sunt același lucru cu valorile asociate. Valorile de bază sunt setate la definirea pentru prima dată a enumerării în codul dvs., cum ar fi cele trei coduri ASCII de mai sus. Valoarea de bază pentru un anumit caz de enumerare este întotdeauna aceeași. Valorile asociate sunt stabilite atunci când creați o nouă constantă sau variabilă pe baza unuia dintre cazurile enumerării și pot fi diferite de fiecare dată când faceți acest lucru.

1.7 Structuri și clase

Structurile și clasele sunt concepte generale, flexibile, care devin blocurile de cod pe care construiți programul dvs. Definiți proprietăți și metode pentru a adăuga funcționalitate structurilor și claselor folosind aceeași sintaxă pe care o utilizați pentru a defini constantele, variabilele și funcțiile.

Spre deosebire de alte limbaje de programare, Swift nu vă cere să creați fișiere separate pentru interfață si pentru implementare. În Swift, definiți o structură sau o clasă într-un singur fișier, iar interfața externă este automat pusă la dispoziție pentru a putea fi utilizată de către alte bucăți de cod.

Comparând clasele și structurile

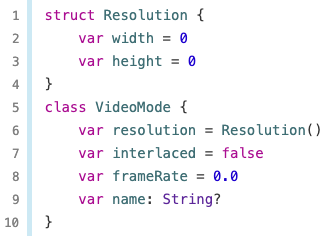
Structurile și clasele din Swift au multe lucruri în comun. Ambele pot:

* Defini proprietăți pentru a stoca valorile
* Defini metodele pentru a oferi funcționalitate
* Defini subscript-uri pentru a oferi acces la valorile lor utilizând sintaxa subscript
* Defini constructori pentru a configura starea lor inițială
* Extinde funcționalitatea acestora în afara unei implementări implicite
* Să se conforme protocoalelor pentru a oferi o funcționalitate standard de un anume tip

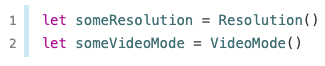
Clasele au capacități suplimentare pe care structurile nu le au:

* Moștenirea permite unei clase să moștenească caracteristicile altei clase.
* Castarea la tip vă permite să verificați și să interpretați tipul instanței în timpul rulării.
* Destructorii permit unei instanțe a unei clase să elibereze resursele pe care le-a atribuit.
* Numărătoarea referințelor permite mai mult de o referință la o instanță a unei clase.

Structurile și clasele au o sintaxă de definire similară. Declarati structuri folosind cuvântul cheie „struct” și clase cu cuvântul cheie „class”. Ambele își plasează întreaga definiție într-o pereche de acolade:

  
Figura 98

1.7.1 Instanțele de clase și structuri

  
Figura 99

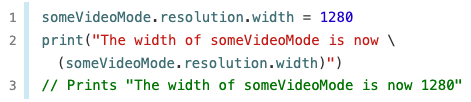
Structurile și clasele utilizează sintaxa inițializatorului pentru instanțe noi. Cea mai simplă formă de inițializare utilizează numele de tip al clasei sau structurii, urmat de paranteze goale, cum ar fi Resolution() sau VideoMode(). Aceasta creează o nouă instanță a clasei sau a structurii, cu proprietățile inițializate la valorile implicite.

1.7.2 Accesarea proprietăților

Puteți accesa proprietățile unei instanțe utilizând sintaxa punctului. În sintaxa punct, scrieți numele proprietății imediat după numele instanței, separate de un punct (.), fără spații:

  
Figura 100

De asemenea, puteți utiliza sintaxa punctului pentru a atribui o nouă valoare unei proprietăți variabile:

  
Figura 101

1.7.3 Structurile și enumerările sunt Value Types

Un value type este un tip a cărui valoare este copiată atunci când este atribuită unei variabile sau unei constante sau când este trimisă ca parametru către o funcție.

De fapt, toate tipurile de bază din Swift – Int, Double, Bool, String, Array, Set, Dictionary - sunt value types și sunt implementate ca structuri.

Toate structurile și enumerările sunt value types în Swift. Aceasta înseamnă că orice instanțe de structură și enumerare pe care le creați - și orice tipuri de valori pe care le au ca proprietăți - sunt copiate întotdeauna când sunt pasate în codul dvs.

Considerați exemplul următor:

  
Figura 102

Deoarece Resoultion este o structură, se face o copie a instanței existente și această copie nouă este atribuită variabilei cinema. Chiar dacă hd și cinema au acum aceeași lățime și înălțime, acestea sunt două instanțe complet diferite. În continuare, proprietatea de lățime a cinematografiei este modificată astfel încât să fie lățimea standardului 2K, utilizat pentru proiecția cinematografică digitală (2048 pixeli și 1080 pixeli înălțime):

  
Figura 103

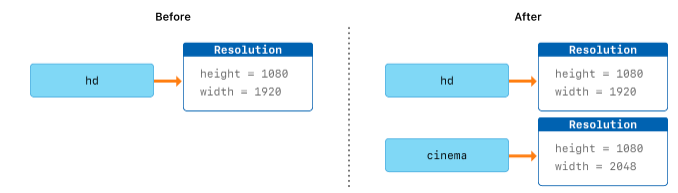
Verificarea proprietatii de lățime a cinematografului arată că într-adevăr sa schimbat la 2048:

  
Figura 104

Cu toate acestea, proprietatea lățimii instanței hd originale are încă valoarea veche de 1920:

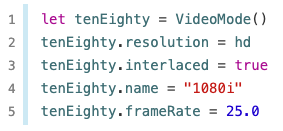
  
Figura 105

Când cinema a primit valoarea curentă a hd, valorile stocate în hd au fost copiate în noua instanță cinema. Rezultatul final au fost două cazuri complet separate care conțineau aceleași valori numerice. Cu toate acestea, deoarece acestea sunt cazuri separate, stabilirea lățimii cinematografului la 2048 nu afectează lățimea stocată în hd, după cum se arată în figura de mai jos:

  
Figura 106

1.7.3 Clasele sunt Reference Types

Spre deosebire de value types, reference types nu sunt copiate atunci când sunt atribuite unei variabile sau unei constante sau când sunt trimise unei funcții. În locul unei copii, este utilizată o referință la aceeași instanță existentă.

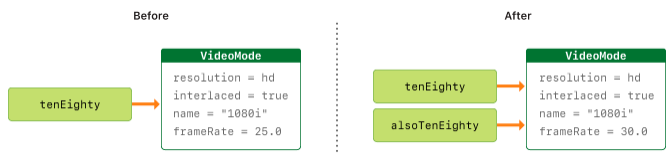
  
Figura 107

Acest exemplu declară o nouă constantă numită tenEighty și o stabilește pentru a se referi la o nouă instanță a clasei VideoMode. Modului video i se atribuie o copie a rezoluției HD de 1920 pe 1080. Este setat să fie intercalat, numele său este setat la "1080i", iar rata de cadre este setată la 25 de cadre pe secundă. Apoi, tenEighty este atribuit unei noi constante numite alsoTenEighty, iar rata cadrelor alsoTenEighty este modificata:

  
Figura 108

Deoarece clasele sunt reference types, tenEighty și alsoTenEighty se referă de fapt la aceeași instanță VideoMode. De fapt, acestea sunt doar două nume diferite pentru aceeași instanță unică, așa cum se arată în figura de mai jos:

Verificarea proprietatii frameRate a lui tenEighty arată că acesta raportează corect noua rată de cadre de 30.0 din instanța VideoMode care stă la baza:

  
Figura 109

Acest exemplu arată, de asemenea, modul în care reference types pot fi mai greu de explicat. În cazul în care tenEighty și alsoTenEighty ar fi fost folosite în puncte îndepărtate ale codului, ar fi fost foarte dificil să găsiți toate locurile în care se modifică modul video. Ori de câte ori folosiți tenEighty, trebuie să vă gândiți și la codul care utilizează, de asemenea, alsoTenEighty și invers. În schimb, value types sunt mai ușor de explicat, deoarece tot codul care interacționează cu aceeași valoare este împreună în fișierele sursă.

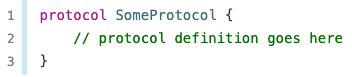
  
Figura 110

1.8 Protocoale

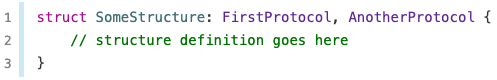
Un protocol definește un model de metode, proprietăți și alte cerințe care se potrivesc unei anumite sarcini sau unei anumite funcționalități. Protocolul poate fi apoi adoptat într-o clasă, structură sau enumerare pentru a asigura o punere în aplicare efectivă a acestor cerințe. Orice tip care satisface cerințele unui protocol este considerat conform cu acel protocol.

În plus față de specificarea cerințelor pe care trebuie să le implementeze tipurile conforme, puteți extinde un protocol pentru a implementa unele dintre aceste cerințe sau pentru a implementa funcționalități suplimentare pe care le pot beneficia tipurile conforme.

Definiția protocoalelor se face într-un mod foarte asemănător cu cel al claselor, structurilor și enumerărilor:

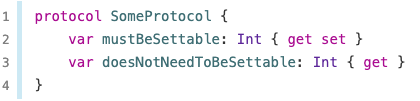
  
Figura 111

Tipurile personalizate afirmă că adoptă un anumit protocol prin plasarea numelui protocolului după numele tipului, separat prin două puncte, ca parte a definiției lor. Pot fi enumerate mai multe protocoale și sunt separate prin virgule:

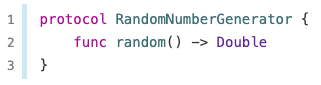
  
Figura 112

Un protocol poate impune ca orice tip ce adoptă acel protocol să furnizeze o proprietate de instanță sau o proprietate de tip cu un anumit nume și tip. Protocolul nu specifică dacă proprietatea ar trebui să fie o proprietate stocată sau o proprietate calculată - specifică numai numele și tipul proprietății necesare. Protocolul specifică de asemenea dacă fiecare proprietate trebuie să fie gettable sau gettable și settable.

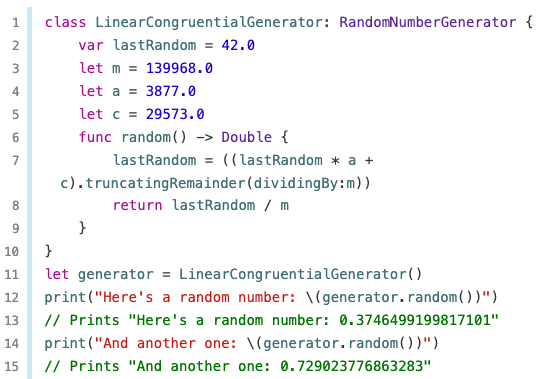
Cerințele de proprietate sunt întotdeauna declarate ca proprietăți variabile, prefixate cu cuvântul cheie var. Proprietățile gettable și settable sunt indicate prin scrierea { get set } după declarația de tip și proprietățile gettable sunt indicate prin scrierea { get }.

  
Figura 113

Protocoalele pot impune metode de instanță specifice și metode de tip pentru a fi implementate de către tipurile conforme. Aceste metode sunt scrise ca parte a definiției protocolului în exact același mod ca și în cazul metodelor normale și de tip, dar fără acolade sau un corp de metodă. Parametrii variadici sunt permiși, sub rezerva acelorași reguli ca și pentru metodele normale. Valorile implicite nu pot fi însă specificate pentru parametrii metodei în cadrul definiției protocolului.

  
Figura 114

Iată o implementare a unei clase care adoptă și se conformează protocolului RandomNumberGenerator. Această clasă implementează un algoritm generator de numere pseudo-aleatoare, cunoscut ca un generator congruențial liniar:

  
Figura 115

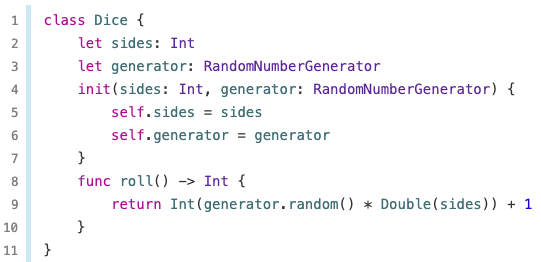
1.8.1 Protocoalele folosite ca tipuri

Protocoalele nu implementează de fapt nicio funcționalitate. Cu toate acestea, puteți utiliza protocoalele ca tipuri complete în codul dvs. Folosirea unui protocol ca tip este denumită uneori un tip existențial, care provine din expresia "există un tip T astfel încât T să fie conform cu protocolul".

Puteți folosi un protocol în multe locuri unde sunt permise alte tipuri, inclusiv:

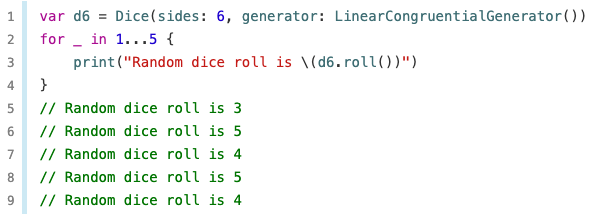
* Ca tip de parametru sau tip retur într-o funcție, metodă sau constructor
* Ca tip de o constantă, variabilă sau proprietate
* Ca tip de elemente dintr-un array, dicționar sau alt container

Iată un exemplu de protocol folosit ca tip

  
Figura 116

Proprietatea generator este de tip RandomNumberGenerator. Prin urmare, îl puteți seta la o instanță de orice tip care adoptă protocolul RandomNumberGenerator. Nimic altceva nu este necesar pentru instanța pe care o atribuiți acestei proprietăți, cu excepția faptului că instanța trebuie să adopte protocolul RandomNumberGenerator. Deoarece tipul său este RandomNumberGenerator, codul din clasa Dice poate interacționa numai cu generatorul în moduri care se aplică tuturor generatoarelor care se conformează acestui protocol. Aceasta înseamnă că nu poate folosi nici o metodă sau proprietate care sunt definite de tipul de bază al generatorului.

Iată cum poate fi folosită clasa Dice pentru a crea un zar cu șase fețe cu o instanță LinearCongruentialGenerator ca generator de numere aleatorii:

  
Figura 117

# 2. Tehnologii folosite

2.1 Sistemul de operare iOS

iOS este un sistem de operare de tip Unix, care încă în prima sa versiune a conținut multe elemente din Mac OS X, tot un sistem de operare de tip Unix de la Apple. Versiunea actuală (06 Mai 2019) este iOS 12.2.

Funcționalitatea iOS poate fi întregită de către utilizator prin procurarea de aplicații suplimentare specializate numite apps în prăvălia online ''App Store'' a lui Apple. În mai 2011 stăteau la dispoziție acolo cca 350.000 de apps, din care unele sunt chiar gratuite. Exemple de apps gratuite: cumpănă "cu apă"; mici animale casnice mișcătoare care se lasă alintate etc.; pahar cu bere virtual. La cealaltă extremă stă aplicația I'm rich („Sunt bogat”) care nu poate decât să afișeze pe ecran un diamant rotitor, dar care în schimb costă circa 800 euro. Desigur însă că majoritatea aplicațiilor oferă o utilitate reală. Odată cu iOS 11 o noua categorie și-a făcut loc in AppStore: AR (Augmented Reality). Aplicațiile AR au nevoie de acces la camera și de o suprafața plana. Aplicațiile AR necesita cel puțin un iDevice cu procesor Apple A7, iar jocurile AR necesita cel puțin un dispozitiv cu procesor Apple A9.

Din motive de politică a produsului, iOS nu sprijină aplicația multimedială Flash a companiei americane Adobe. Prin funcționalitatea sa iOS este unul din factorii de succes primordiali al telefoanelor iPhone pe piața mondială. Un concurent al lui iOS este sistemul de operare Android de la compania Google. [X]

2.2 ARKit

Realitatea augmentată (AR) descrie experiențele utilizatorilor care adaugă elemente 2D sau 3D la vizualizarea live din aparatul de fotografiat al unui dispozitiv, într-un mod care face ca aceste elemente să pară că locuiasc în lumea reală. ARKit combină urmărirea mișcării dispozitivului, captarea scenei camerei, procesarea avansată a scenelor și facilitățile de afișare pentru a simplifica sarcina de a construi o experiență AR. Puteți utiliza aceste tehnologii pentru a crea multe tipuri de experiențe AR, utilizând camera din spate sau camera frontală a unui dispozitiv iOS.

2.3 CoreData

Utilizați CoreData pentru a salva datele aplicației pentru utilizarea lor în mod offline, pentru a memora datele temporare și pentru a adăuga funcționalitatea de undo aplicației dvs. pe un singur dispozitiv.

Prin intermediul editorului de model al CoreData, definiți tipurile și relațiile datelor și generați definiții de clasă respective. CoreData poate apoi să gestioneze instanțele obiectului în timpul rulării pentru a oferi următoarele caracteristici.

Întreaga interacțiune cu CoreData se va realiza printr-un ManagedObjectContext, el reprezintă o „ciornă” într-o aplicație CoreData. Un ManagedObjectContext este o instanță a NSManagedObjectContext. Aceste ManagedObjects reprezintă interfața a unuia sau mai multor PersistentStore-uri. Contextul este un obiect puternic, cu un rol central în ciclul de viață al obiectelor gestionate, cu responsabilități de la gestionarea ciclului de viață (inclusiv faulting) la validare, manipulare inversă a relațiilor și anularea / refacerea schimbărilor.

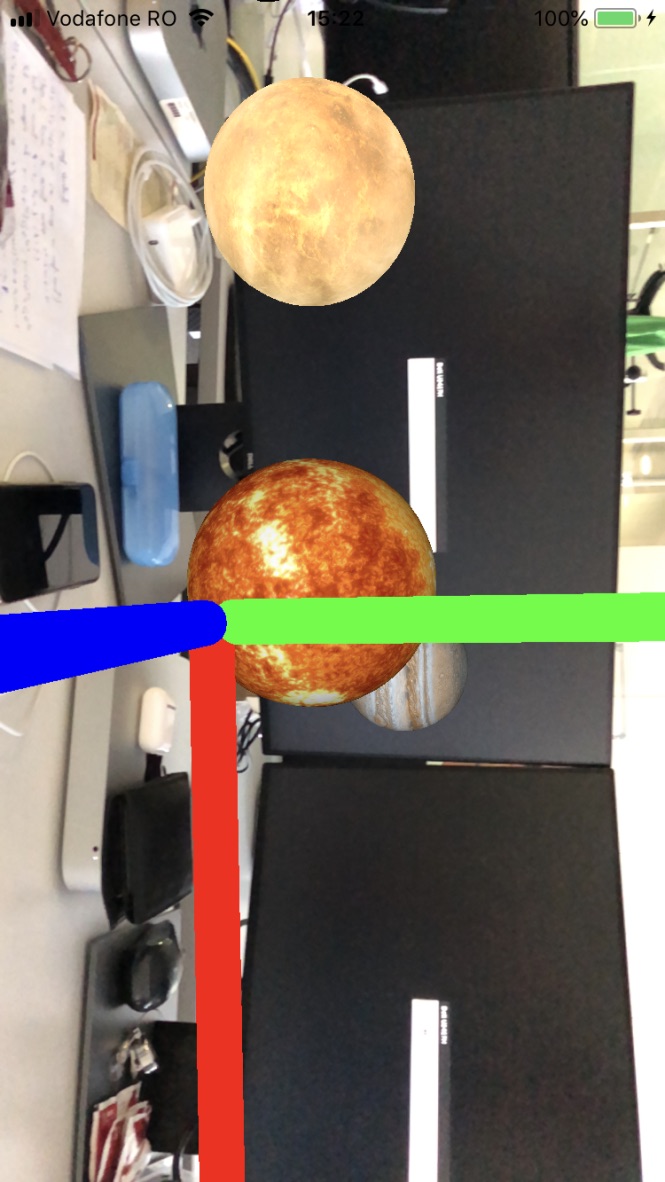
# 3. Descrierea aplicației

Această lucrare are ca obiectiv crearea aplicației 3D cu numele „ARSolarSystem” pentru platforma iOS. Implementarea acestui obiectiv este realizată prin îmbinarea conceptelor şi tehnologiilor prezentate în capitolele anterioare. Astfel, jocul are la bază SDK-ul ARKit prin intermediul căruia sunt controlate şi expuse utilizatorului aspectele vizuale, sunt tratate input-urile dispozitivului şi sunt administrate activităţile şi scenele componente ale aplicaţiei.

3.1 Mediul virtual al aplicației

Mediul virtual al aplicației constă în folosirea feed-ului de pe camera principală a telefonului mobil, în momentul în care feed-ul video este pasat framework-ului ARKit, se declanșează afișarea planetelor în spațiul tridimensional din spatele telefonului.

Framework-ul ARKit preia sarcina de a localiza dispozitivul mobil în spațiul tridimensional, el setând originea planului tridimensional ce urmează a fi folosit la poziția curenta a dispozitivului în spațiu, iar ulterior, orice nod adăugat în acest spațiu virtual va fi poziționat relativ la poziția inițială a dispozitivului. Aici putem observa cum punctul de origine a fost setat la poziția dispozitivului.

  
Figura 118

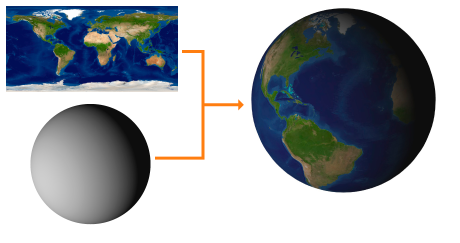
3.2 Adăugarea unei planete în spațiu

Pentru a adăuga planetele propriu-zise, s-a folosit subclasă, numită PlanetWrapper, a clasei predefinite în framework-ul ARKit numita SCNNode, ce este doar un element structural al unei scene grafice, reprezentând o poziție și o transformare într-un spațiu 3D, la care puteți atașa o anumită formă geometrică, lumini, camere sau alt conținut afișabil. Subclasa, pe lângă proprietățile părintelui, definește o proprietate nouă pentru a stoca modelul ce urmează a fi folosit în UI.

  
Figura 119

După cum se poate observa, constructorul clasei PlanetWrapper, primește ca parametrii cele 4 tipuri de imagini ce pot fi folosite pentru a reprezenta grafic o planetă, ele se folosesc în mod special pentru simularea luminozității planetelor.

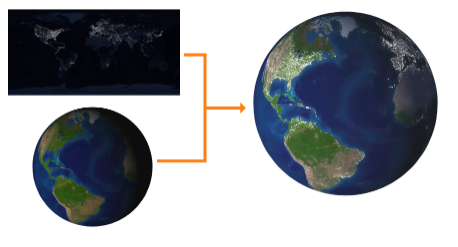
Prima dintre ele, cea de difuzare, descrie cantitatea și culoarea luminii reflectate în mod egal în toate direcțiile din fiecare punct de pe suprafața materialului. Figura de mai jos arată efectul de a seta conținutul proprietății diffuse la o textura pe un material ce are restul proprietăților cu conținut implicit.

  
Figura 120

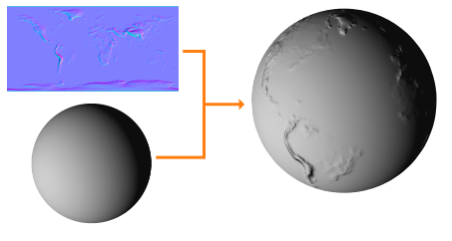
Cea de-a doua proprietate, cea de specular, descrie cantitatea și culoarea luminii reflectate de material direct spre vizualizator, formând o lumină strălucitoare pe suprafață și simulând un aspect lucios sau strălucitor. Figura de mai jos prezintă un material (cu o textură pentru proprietatea sa difuză) înainte și după furnizarea unei imagini de pe harta speculară. Observați că punctele luminoase speculare apar doar pe porțiuni ale suprafeței unde imaginea de pe hartă este albă.

  
Figura 121

Cea de-a treia proprietate, cea de emisie, simulează părți ale unei suprafețe care strălucesc cu propria lor lumină. Figura de mai jos prezintă un material (cu textură pentru proprietatea sa difuză) înainte și după furnizarea unei imagini de emise.

  
Figura 122

Ultima dintre cele patru proprietăți, este cea de normal. Simularea interacțiunii luminilor cu un material necesită informații despre orientarea suprafeței în fiecare punct. În mod obișnuit, vectorii normali sunt furnizați de un obiect cu o formă geometrică. Figura de mai jos arată efectul de a seta conținutul proprietății normale la o imagine cu o textură pe un material ale cărui proprietăți au conținut implicit.

  
Figura 123

Deși ideal ar fi ca unui SCNNode să i se aplice toate cele 4 imagini, nu toate sunt necesare, singura fără de care nu se poate proiecta un nod în spațiu este cea de difuzare, ea reprezintă „baza” unei figuri tridimensionale.

Pentru că în aplicație planetele nu se vor roti doar în jurul soarelui, ci și în jurul propriului lor ax, vom avea nevoie de un nod invizibil, ce va fi părintele nodului vizibil, părintele fiind folosit pentru rotația în jurul propriului ax, iar copilul, și anume planeta vizibilă, va fi responsabilă de rotația în jurul soarelui. Pentru asta, vom folosi un dicționar ce va avea ca și cheie un caz al unei enumerații, iar ca valoare un tuplu format din nodul părinte al planetei și planeta în sine.

  
Figura 124

Se poate observa că singurul nod ce nu se construiește în acest dicționar este cel al soarelui, asta fiindcă soarele nu are nevoie de un părinte, el nefiind nevoit să se rotească în jurul altui obiect, ci rotația să se întamplă doar în jurul propriului ax. Soarele este creat puțin mai sus, unde i se atribuie acțiunea de rotație precum se și adaugă scenei ca și copil al nodului rădăcină.

  
Figura 125

Adăugarea celorlalte planete se va face trecând prin acest dicționar, și adăugând fiecare nod părinte ca și copil al nodului rădăcină.

  
Figura 126

Se poate observa că luna este omisă, asta se datorează faptului că Luna va avea ca părinte Pământul, și nu va fi nod de sine stătător în spațiu, ea depinzând de poziția pământului în spațiu.

  
Figura 127

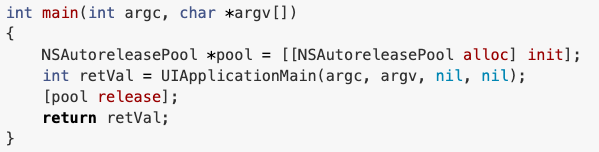
3.3 Detaliile unei planete

Detaliile planetelor sunt stocate în baza de date locala, folosind CoreData. De cele mai multe ori, aplicațiile sunt conectate la un server și își descarcă datele de acolo, însă în cazul de față, pentru simplitate, datele vor fi ținute într-un fisier static de tip JSON în interiorul aplicației, baza de date populându-se la prima lansare a aplicației cu acele date. Această populare a baze de date, se întâmplă în Singleton-ul AppDelegate, care implementează protocolul UIApplicationDelegate, și conține decoratorul de @UIApplicationMain.

  
Figura 128

Modelul singleton este un model de design care restricționează instanțiarea unei clase la un singur obiect, această clasă fiind responsabilă de instanțiere și de a se asigura ca altă instanță nu poate fi creată[x].

Decoratorul @UIApplicationMain se asigură de crearea fișierului main.swift, fișier ce nu este prezent în proiect, și acolo declară clasa AppDelegate ca punct de intrare în aplicație. În proiectele scrise folosind limbajul Objective-C, Xcode crează automat fișierul main.c, ce conține funcția main. Decoratorul @UIApplicationMain este analogul acestei funcții.

  
Figura 129

Salvarea propriu-zisă a datelor, se întâmpla în funcția application(\_:didFinishLaunchingWithOptions:), funcție ce se cheamă la fiecare lansare a aplicației.

După cum se poate observa, aici se verifică Singleton-ul UserDefaults, care este una dintre modalitățile de a persista date în iOS, acesta fiind un dicționar, verificăm valoarea corespunzătoare cheii „wasDatabasePopulated”, dacă ea este false înseamnă ca aplicația este lansată pentru prima oară, deci va trebui sa populăm baza de date, altfel, sărim peste salvarea datelor și returnăm true.

3.4 Încărcarea planetelor din baza de date

Odată salvate în baza de date, detaliile urmează a fi încărcate în ecranul principal, acesta fiind cel în care vom vedea planetele. Încărcarea se va face în metoda viewDidLoad(), aceasta chemându-se imediat după încărcarea view-ului în memorie.

Aici se întâmpla mai multe chestii, în primul rând, se accesează Singleton-ul AppDelegate de unde se extrage o referință către ManagedObjectContext-ul principal al aplicației (proprietatea viewContext); pasul numărul doi, se cere permisiunea user-ului asupra accesului la camera dispozitvului pentru modul de video, cu mențiunea că alerta pentru permiterea accesului se afișează o singură dată, sistemul de operare reținând alegerea făcuta de user, dacă userul acceptă, atunci se accesează baza de date prin acel ManagedObjectContext și se încarcă în memorie toate planetele salvate iar după se sortează folosind câmpul order; ultimul pas după încărcarea planetelor din baza de date, se va rula o sesiune de SceneKit folosind o configurație standard iar ulterior se va chema metoda addPlanets() care creează planetele, conținutul ei fiind descris la punctul 3.2.

  
Figura 131

3.5 Ecranul de detalii

Atunci când utilizatorul apasă pe ecran, se verifică poziția acelei apăsări, iar dacă ea intersectează cumva o planetă, se va lua primul nod care s-a intersectat cu apăsarea pe ecran, și se va face tranziția către ecranul de detalii, pasând id-ul planetei atinse, pentru a afișa datele aferente.

  
Figura 132

Pentru ca putem avea mai multe tipuri de noduri, aici verificăm prin optional binding, dacă nod-ul atins pe ecran este de tipul PlanetWrapper, atunci tranziția către ecranul de detalii se va realiza, în caz contrar, ne având o ramură else, nu vom face nimic. La apăsarea butonului „Mai multe detalii”, se va deschide browser-ul implicit direct pe pagina de wikipedia aferentă acelei planete.

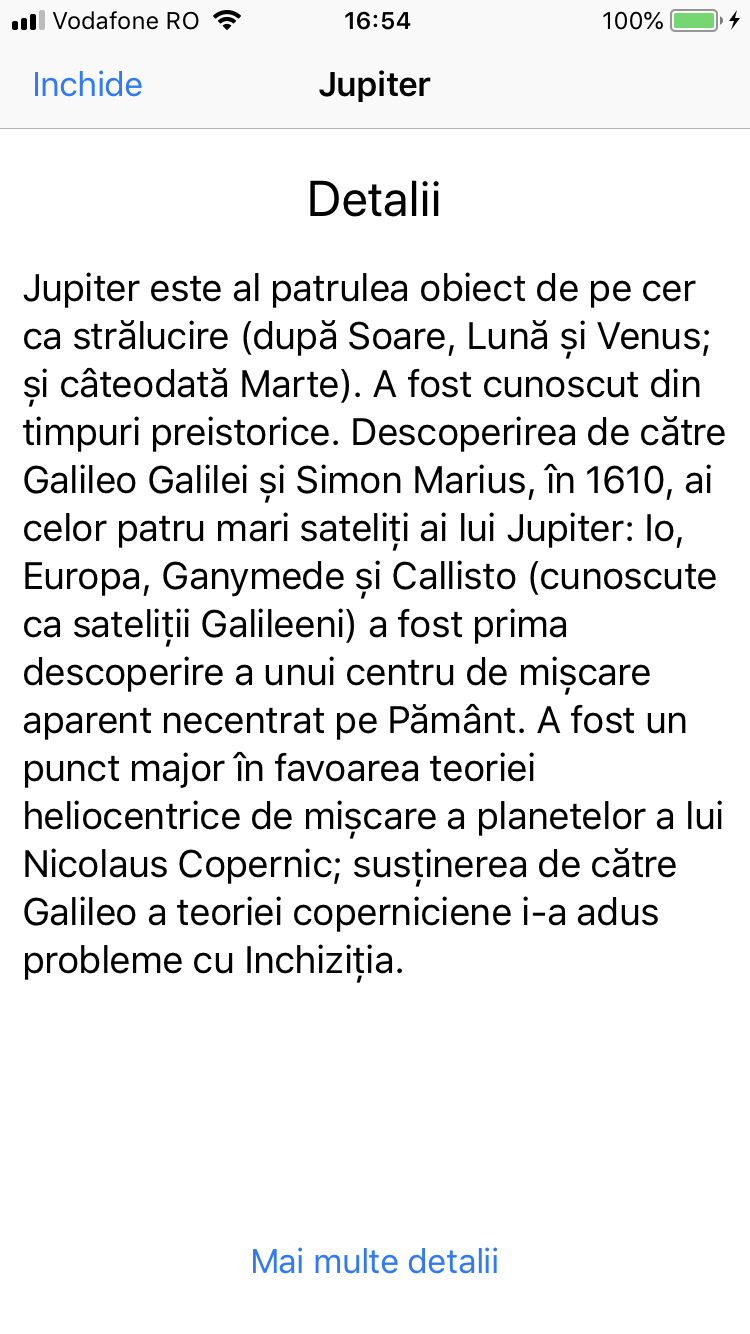
  
Figura 133

  
Figura 134

# 4. Concluzii

Folosind instrumentele prezentate anterior, aplicația ARSolarSystem a fost implementată astfel încât sa ofere utilizatorului o experiență plăcută vazând prin camera dispozitivului său mobil planetele rotindu-se precum în spațiu.

La un nivel transaprent utilizatorului s-au folosit concepte fizice și de matematică precum și de inginerie a programării, ele fiind necesare pentru a putea plasa dispozitivul mobil în spațiul tridimensional, pentru a putea adăuga obiecte în același spațiu 3D, pentru a salva datele în vederea utilizării lor pe parcursul rulării aplicației astfel obținându-se o aplicație robustă, scalabilă și eficientă din punct de vedere al memoriei ocupate și al algoritmilor folosiți.

# 5. Bibliografie