POGLAVLJE 1

Arhitektura jezika Swift

Biće korisno da na početku imate opšti osećaj o tome kako je jezik Swift konstruisan i o tome kako izgledaju iOS programi zasnovani na jeziku Swift. U ovom poglavlju biće prikazana uopštena arhitektura i priroda jezika Swift. U narednim poglavljima govorićemo o detaljima.

Osnova svega

Potpuna komanda u jeziku Swift je *iskaz* (engl. *statement*). Swift tekstualna datoteka sastoji se od više *redova* teksta. Prelomi redova imaju smisao. Uobičajeni izgled programa je jedan iskaz – jedan red:

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

Međutim, to nije čvrsto i obavezno pravilo. Možete da spojite više iskaza u jedan red, ali je potrebno da stavite znak ; između njih:

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

Znak ; možete da stavite slobodno i na kraj iskaza koji je poslednji u svom redu, ali to niko ne radi (osim iz navike, pošto jezici C i Objective-C *zahtevaju* znak ;):

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

Sa druge strane, jedan iskaz može da se podeli u više redova, kako bi se sprečilo da dugi iskazi postanu dugački redovi. To bi trebalo da radite na tačno određenim mestima kako ne biste zbunili Swift. Na primer, posle otvorene zagrade je dobro mesto:

```
print(
    "world")
```

Komentari su bilo šta posle dve kose crte u redu (takozvani komentari u stilu jezika C++):

```
print("world") // ovo je komentar, tako da Swift ne obraća pažnju na njega
```

Komentare možete zatvoriti u /*… */, kao u jeziku C. Za razliku od jezika C, komentari u stilu jezika C mogu da budu ugnježđeni.

Većina sklopova u jeziku Swift koristi vitičaste zagrade kao graničnike:

```
class Dog {
    func bark() {
        println("woof")
    }
}
```

Po pravilu, onome što se nalazi unutar vitičastih zagrada prethode i slede prelomi reda i oni su tu zbog jasnoće, kao što je prikazano u prethodnom kodu. Xcode će pomoći da poštujete ovo pravilo, ali je suština u tome da Swift ne brine o tome i kôd koji izgleda kao ovaj je sasvim ispravan (i ponekad zgodniji):

```
class Dog { func bark() { println("woof") }}
```

Trenutna povratna informacija u Xcode konzoli je obezbeđena komandama `print` i `println`. Razlika je u tome da komanda `println` dodaje novi red nakon ispisivanja.

Swift je *kompajlirani* jezik. To znači da vaš kôd mora da se *prevede* – prolaskom kroz kompajler i pretvaranjem teksta u neki niži oblik koji računar može da razume – pre nego što se *pokrene* i uradi ono što mu je rečeno da uradi. Swift kompajler je veoma strog; tokom pisanja programa, često ćete pokušati da ga prevedete i pokrenete, samo da biste otkrili da ne možete ni da ga prevedete, pošto kompajler prijavljuje neku *grešku*, koju ćete morati da ispravite ukoliko želite da vaš kôd proradi. Mnogo ređe, kompjaler će vas pustiti sa *upozorenjem*; kôd će moći da se izvrši, ali je obično najbolje da to upozorenje shvatite ozbiljno i popravite sve ono o čemu ste upozoreni. Strogost kompajlera je jedna od najvećih snaga jezika Swift pošto u velikoj meri proverava ispravnost vašeg koda, čak i pre početka izvršavanja.



U trenutku dok ovo pišem, poruke i upozorenja o greškama Swift kompajlera se kreću od veoma korisnih do onih koji više zbunjuju nego što pomažu. Obično ćete znati da *nešto* nije u redu sa linijom koda, ali Swift kompajler vam neće tačno reći *šta* je pogrešno ili čak *gde* je red na koji treba da obratite pažnju. Moj savet u takvim situacijama je da redove koda podelite u nekoliko redova sa jednostavnijim kodom sve dok ne dođete do mesta gde možete da nagađate u čemu je problem. Nemojte da mrzite kompajler uprkos povremeno beskorisnoj prirodi njegovih poruka. Zapamtite, on zna više od vas, mada ponekad ne ume da na najbolji način iskaže svoje znanje.

Sve je objekat?

U jeziku Swift „sve je objekat". To je hvalisanje zajedničko za razne savremene objektno orijentisane jezike, ali šta to znači? To zavisi od toga šta podrazumevate pod „objekat" – a šta podrazumevate pod „sve".

Počnimo od tvrdnje da je objekat, grubo rečeno, nešto čemu možete da pošaljete poruku. Poruka je, grubo rečeno, zapovedna naredba. Na primer, psu možete zadati komande: „Laj!" i „Sedi!". Po ovoj analogiji, ovi izrazi su poruke, a pas je objekat kome šaljete te poruke.

Sintaksa za slanje poruka u jeziku Swift koristi *notaciju sa tačkom* (engl. *dot-notation*). Počinjemo objektom; zatim nailazi tačka (*engl. dot*), a potom sledi poruka. (Iza nekih poruka takođe slede zagrade, ali ih za sada igorišemo; potpuna sintaksa slanja poruke je nešto o čijim detaljma ćemo govoriti kasnije). Ovo je valjana Swift sintaksa:

```
fido.bark()
rover.sit()
```

Ideja da je *sve* objekat je način da se ukaže na to da se čak i „primitivnim" jezičkim celinama može poslati poruka. Uzmimo, na primer, `1`. Izgleda da je to obična cifra i ništa više. Ne bi trebalo da vas iznenadi, ukoliko ste ikad koristili bilo koji programski jezik, da u jeziku Swift možete reći nešto slično ovome:

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

Ali *jeste* iznenađujuće ustanoviti da iza `1` mogu da slede tačka i poruka. Ovo je dozvoljeno i ima smisla u jezku Swift (ne brinite o tome šta ovo zaista znači):

```
let x = 1.successor()
```

Slično tome, deo teksta iskazan nizom znakova – *string* – je objekat. Na primer, `"hello"` je niz slovnih znakova i ovo je ispravan Swift kôd (i ovog puta ne obraćajte pažnju na to šta ovaj kôd znači):

```
let y = "hello".generate()
```

Možemo da idemo i dalje. Vratimo se našem naivno izgledajućem `1 + 2` iz našeg prethodnog koda. Ovo navodi na to da je u stvari u pitanju sintaksna prevara, zgodan način da se iskaže i sakrije šta se zaista događa. Pošto je `1` u stvari objekat, `+` je u stvari poruka, ali to je poruka sa posebnom sintaksom (sintaksom *operatora*). U jeziku Swift, svaka imenica je objekat, a svaki glagol je poruka.

Možda najvažnija provera da li je nešto objekat u jeziku Swift jeste u tome da li to nešto možete da promenite. U jeziku Swift tip objekta može da se *proširi*, što znači da možete da definišete sopstvene poruke za taj tip. Na primer, uobičajeno nekom broju ne možete da pošaljete poruku `sayHello`. Ali, možete da promenite tip za broj:

```
extension Int {
    func sayHello() {
```

```
            println("Hello, I'm \(self)")
        }
    }
    1.sayHello() // dobija se: "Hello, I'm 1"
```

Sve je rečeno.

U jeziku Swift, prema tome, `1` jeste objekat. U nekim jezicima, kao što je Objective-C, jasno je da nije; to je „prost tip" (engl. *primitive*) ili *skalarni* ugrađeni tip podataka. Razlika koja je ovde jasno navedena, dakle, kada kažemo „sve je objekat", jeste između tipova objekata sa jedne strane i skalara sa druge. U jeziku Swift, nema prostih tipova, *svi* tipovi su u krajnjoj liniji tipovi objekata. To je ono šta „sve je objekat" zaista znači.

Tri ukusa tipa objekta

Ukoliko znate Objective-C ili neki drugi objektno orijentisani jezik, bićete iznenađeni kako Swift tumači koja *vrsta objekta* je *`1`*. U većini jezika, kao što je Objective-C, objekat je *klasa* ili instanca klase. Swift ima klase i instance i možete da im šaljete poruke; ali `1` nije nijedno od toga: ono je *struktura* (engl. *struct*[1]). A Swift ima još nešto čemu možete slati poruke, nazvano *enumeracija* (engl. *enum*[2]).

Tako Swift ima tri vrste tipa objekta: klase, strukture i enumeracije. O ova tri tipa volim da govorim kao o tri *ukusa* tipa objekta. Po čemu se oni tačno međusobno razlikuju biće jasnije tokom priče. Ali, svi oni su posebni tipovi objekata, a njihove sličnosti su mnogo veće nego njihove različitosti. Za sada, zapamtite samo to da ova tri ukusa postoje.

(Činjenica da su struktura ili enumeracija tip objekta u jeziku Swift biće iznenađenje za vas, posebno ukoliko znate Objective-C. Objective-C ima strukture i enumeracije, ali oni nisu objekti. Posebno su Swift strukture važnije i sveobuhvatnije nego Objective-C strukture. Razlika u tome kako Swift posmatra strukture i enumeracije, a kako ih posmatra Objective-C je važna kada komunicirate sa Cocoa okruženjem.)

Promenljive

Promenljiva je *naziv* nekog objekta. Tehnički, ona *upućuje* (engl. *refer*) na neki objekat; ona je *referenca* objekta. Ne-tehnički, možete je zamisliti kao kutiju unutar koje je neki objekat smešten. Objekat može da pretrpi promene ili može da bude zamenjen unutar kutije nekim drugim objektom, ali naziv ima svoj sopstveni integritet.

U jeziku Swift nijedna promenljiva se ne pojavljuje sama po sebi; sve promenljive moraju da se *deklarišu* (engl. *declare*). Ukoliko vam je potreban naziv za nešto, morate da kažete „Napravio sam naziv". To radite jednom od dve ključne reči: `let` ili `var`. U jeziku

[1] *Struct* (skraćeno od *structure*) je računarski termin za zapis koji se koristi za čuvanje više vrednosti različitih tipova (*nap. prev.*).

[2] *Enum* (skraćeno od *enumeration*) je tip podataka koji se sastoji od skupa imenovanih vrednosti poznatih kao elementi, članovi ili prebrojivi spisak (*nap. prev.*).

Swift, deklaracija je obično praćena *inicijalizacijom* (engl. *initialization*) – koristite znak jednakosti da biste promenljivoj dali *vrednost,* koja postaje deo deklaracije. Oba ova primera su deklaracije promenljivih (i inicijalizacije):

```
let one = 1
var two = 2
```

Pošto naziv postoji, slobodno možete da ga koristite. Na primer, možete da promenite vrednost onoga što je u `two` tako da bude isto kao vrednost onoga što je u `one`.

```
let one = 1
var two = 2
two = one
```

Poslednji red ovog koda koristi nazive `one` i `two` deklarisane u prva dva reda: naziv `one`, sa desne strane znaka jednakosti, koristi se samo za *upućivanje* na vrednost unutar kutije (tačnije `1`); ali naziv `two`, sa leve strane znaka jednakosti, koristi se da bi se *zamenila* vrednost unutar kutije. Iskazi poput ovog, sa nazivom promenljive sa leve strane znaka jednakosti, nazivaju se *dodeljivanje* (engl. *assignment*), a znak jednakosti je *operator dodele*. Znak jednakosti ne znači da je nešto jednako, kao što je to u algebarskim jednačinama; to je komanda. Ona znači: „Uzmi vrednost onoga što je sa moje desne strane i to koristi da bi zamenio vrednost onoga što je sa moje leve strane".

Dve vrste za deklarisanje promenljivih razlikuju se po tome da nazivu deklarisanom sa `let` ne mogu da se menjaju objekti. Promenljiva deklarisana sa `let` je *konstanta*; vrednost joj se jednom dodeljuje i ne menja se. Ovo se ne bi čak ni kompajliralo:

```
let one = 1
var two = 2
one = two // greška u kompajliranju
```

Uvek je moguće deklarisati naziv sa `var`, čime sebi dajte više slobode, ali ako znate da nikad nećete zameniti početnu vrednost neke promenljive, bolje je da koristite `let`, pošto je zbog toga Swift mnogo efikasniji.

Promenljive takođe imaju tip. Taj tip se uspostavlja kada se određena promenljiva deklariše i *nikad ne može da se promeni*. Na primer, ovo se ne bi kompajliralo:

```
var two = 2
two = "hello"
```

Pošto je `two` deklarisano i inicijalizovano kao `2`, to je broj (tačnije rečeno, tip Int) i tako uvek mora da bude. Možete da zamenite njegovu vrednost sa `1` pošto je to takođe ceo broj, ali njegovu vrednost ne možete da zamenite sa `"hello"` pošto je to niz znakova (tačnije rečeno, tip String) – a tip String nije isto što i tip Int.

Promenljive bukvalno imaju sopstveni život – tačnije, sopstveni *vek trajanja*. Sve dok neka promenljiva postoji, njena vrednost ostaje živa. Prema tome, promenljive ne samo da su uobičajeni način za davanje naziva nečemu, već su način da se to sačuva. O tome ću više govoriti kasnije.



Po pravilu nazivi tipova kao što su String ili Int (ili Dog ili Cat) počinju velikim slovom; nazivi promenljivih počinju malim slovom. *Ne kršite ovo pravilo*. Ukoliko to uradite, vaš kôd će biti kompajliran i možda će lepo raditi, ali ću ja lično poslati par huligana da vam polome noge.

Funkcije

Izvršni kôd, kao što je `fido.bark()` ili `one = two` ne može da bude bilo gde. Uopšteno, on mora da živi unutar tela *funkcije*. Funkcija je grupisan kôd, niz redova koda kome može biti rečeno, kao grupi, da se izvršava. Uobičajeno, funkcija ima naziv, a taj naziv dobija preko deklarisanja funkcije. Sintaksa deklarisanja funkcije je takođe nešto o čemu ću kasnije detaljnije da govorim, ali ovde dajem primer:

```
func go() {
    let one = 1
    var two = 2
    two = one
}
```

Ovim se opisuje niz stvari koje treba uraditi – deklarisati `one`, deklarisati `two`, promeniti vrednost `two` da se podudara sa vrednošću od `one` – i toj funkciji dati *naziv,* `go`, ali se ta funkcija ne *izvršava*. Ova funkcija se izvršava tek kada je neko *pozove* (engl. *call*). Prema tome, mogli bismo, bilo gde, da kažemo:

```
go()
```

Ovo je komanda da funkcija `go` treba da se stvarno pokrene. Ali i ovog puta, ova komanda sama po sebi je izvršan kôd, tako da ni ona ne može da postoji sama. Mora da oživi u drugoj deklaraciji funkcije:

```
func doGo() {
    go()
}
```

Čekajte! Ovo postaje pomalo uvrnuto. Ovo je, takođe, samo deklaracija funkcije; da bi se ova funkcija pokrenula, neko mora da pozove `doGo`, a to je takođe izvršni kôd. Ovo izgleda kao neka vrsta beskonačnog kretanja u krug, izgleda da ništa od našeg koda *nikada* neće da se pokrene. Ako sav izvršni kôd mora da postoji unutar funkcije, ko će reći *bilo kojoj* funkciji da se pokrene? Početni udarac mora da dođe odnekud.

U stvarnom životu, srećom, ovaj problem sa vrtenjem u krug ne postoji. Sećate se da je vaš osnovni cilj da napišete neku iOS aplikaciju. Prema tome, vaša aplikacija će biti pokrenuta na nekom iOS uređaju (ili u simulatoru) izvršnim programom (engl. *runtime*) koji već treba da poziva izvesne funkcije. Zbog toga počinjete sa pisanjem posebnih funkcija za koje znate da će ih sam izvršni program pozvati. Ovo vašim aplikacijama daje način da se pokrenu i daje vam prostor za smeštanje funkcija koje će biti pozivane od strane izvršnog programa u ključnim trenucima – kao što su kada se aplikacija pokreće ili kada korisnik pritisne neko dugme na interfejsu vaše aplikacije.



Swift takođe ima posebno pravilo da datoteka pod nazivom *main.swift*, izuzetno, *može* da ima izvršni kôd na svom najvišem nivou, izvan tela svih funkcija, a ovo je kôd koji se u stvari izvršava kada se program izvršava. Možete da napravite svoje aplikacije sa datotekom *main.swift*, ali u opštem slučaju to nije potrebno. Takođe, Xcode omogućava da pravite *playground* datoteke. Playground datoteka ima posebnu osobinu da *deluje* kao *main.swift* datoteka, tako da vam je dozvoljeno da izvršni kôd stavite na najviši nivo ove datoteke, izvan tela bilo koje funkcije. Ali, playground datoteke ne mogu da budu deo iOS aplikacije i o njima se ne govori u ovoj knjizi.

Struktura Swift datoteke

Swift program može da se sastoji od jedne datoteke ili od više njih. U jeziku Swift, datoteka je smislena celina i postoje čvrsta pravila o strukturi Swift koda koji može da bude u njoj. (Podrazumevam da u pitanju nisu *main.swift* ili playground datoteka.) Zbog toga samo određene stvari mogu da se nađu na najvišem nivou ove datoteke:

Iskazi `import` modula
: Modul je celina čak i višeg nivoa nego što je to datoteka. Modul može da se sastoji od više datoteka, a u jeziku Swift, datoteke unutar modula mogu da se sve automatski vide međusobno; ali modul ne može da vidi drugi modul bez iskaza `import`. Na primer, na ovaj način možete da se obratite Cocoa u nekom iOS programu: prvi red vaše datoteke kaže `import UIKit`.

Deklaracije promenljivih
: Promenljiva deklarisana na najvišem nivou datoteke je *globalna* promenljiva: ona traje sve dok se program izvršava.

Deklaracije funkcija
: Funkcija deklarisana na najvišem nivou datoteke je *globalna* funkcija: čitav kôd može da se vidi i poziva, bez slanja poruke bilo kom objektu.

Deklaracije tipa objekata
: Deklaracije za klasu, strukturu ili enumeraciju.

Na primer, ovo je valjana Swift datoteka koja sadrži (samo da bi se pokazalo kako to može da se uradi) iskaz `import`, deklaraciju promenljive, deklaraciju funkcije, deklaraciju klase, deklaraciju strukture i deklaraciju enumeracije.

```
import UIKit
var one = 1
func changeOne() {
}
class Manny {
}
struct Moe {
```

```
}
enum Jack {
}
```

Ovo je prilično besmislen i većinom isprazan primer, ali upamtite, naš cilj je da prikažemo delove jezika i strukturu datoteke, a ovaj primer ih pokazuje.

Štaviše, unutar svih vitičastih zagrada svake stvari u ovom primeru moguće je imati deklaracije promenljivih, deklaracije funkcija i deklaracije tipa objekata! Zaista, *bilo koja* vitičasta zagrada unutar ove strukture može da sadrži takve deklaracije. Tako, na primer, ključna reč `if` (koja je deo kontrole toka u jeziku Swift, o čemu govorimo u poglavlju 5) praćena je strukturnim vitičastim zagradama, a one mogu da sadrže deklaracije promenljivih, deklaracije, funkcija i deklaracije tipa objekata. Ovaj kôd je, mada besmislen, valjan:

```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

Ali, samo deklaracija funkcije, sećate se, može da sadrži izvršni kôd. Ona može da sadrži izvršni kôd na bilo kojoj dubini unutar sebe, red `one = one + 1`, što je izvršni kôd, jeste valjan pošto je unutar konstrukcije `if`, koja je unutar deklaracije funkcije. Ali red `one = one + 1` ne može da ide na najviši nivo datoteke i *ne može* da ide neposredno unutar vitičastih zagrada deklaracije klase Cat.

Primer 1-1 je valjana Swift datoteka koja šematski prikazuje mogućnosti strukture datoteka. (Zanemarite podvalu sa deklaracijom promenljive `name` unutar deklaracije enumeracije Jack; promenljive enumeracije na najvišem nivou imaju neka posebna pravila koja ću kasnije objasniti.)

Primer 1-1 Šematska struktura valjane Swift datoteke

```
import UIKit
var one = 1
func changeOne() {
    let two = 2
    func sayTwo() {
        println(two)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
    one = two
}
class Manny {
    let name = "manny"
    func sayName() {
```

```
            println(name)
        }
        class Klass {}
        struct Struct {}
        enum Enum {}
    }
    struct Moe {
        let name = "moe"
        func sayName() {
            println(name)
        }
        class Klass {}
        struct Struct {}
        enum Enum {}
    }
    enum Jack {
        var name : String {
            return "jack"
        }
        func sayName() {
            println(name)
        }
        class Klass {}
        struct Struct {}
        enum Enum {}
    }
```

Očigledno, možemo ići u dubinu koliko god želimo: mogli bismo da imamo deklaraciju klase koja sadrži deklaraciju klase koja sadrži deklaraciju klase... i tako dalje. Ali nema smisla prikazati *tako nešto*.

Opseg i životni vek

U jeziku Swift, stvari imaju *opseg* (engl. *scope*). Ovo se odnosi na njihovu mogućnost da ih vide druge stvari. Stvari su ugneždene unutar drugih stvari, praveći ugneždenu hijerarhiju stvari. Pravilo je da stvari mogu videti stvari *na svom sopstvenom nivou i višim nivoima*. Nivoi su:

- modul je opseg,
- datoteka je opseg,
- deklaracija objekta je opseg,
- vitičaste zagrade su opseg.

Kada se nešto deklariše, deklariše se na nekom nivou unutar ove hijerarhije. Njegovo mesto u hijerarhiji – njegov opseg – određuje da li ga druge stvari mogu videti.

Na primer, pogledajmo ponovo primer 1-1. Unutar deklaracije klase Manny je deklaracija promenljive `name` i deklaracija funkcije `sayName`; kôd *unutar* vitičastih zagrada za sayName može da vidi stvari *izvan* tih vitičastih zagrada na višem nivou i prema

tome vidi promenljivu `name`. Slično tome, kôd unutar tela deklaracije funkcije `changeOne` može da vidi promenljivu `one` deklarisanu na najvišem nivou datoteke; *bilo šta* iz ove datoteke može da vidi promenljivu `one` deklarisanu na najvišem nivou datoteke.

Opseg je stoga veoma važan način *deljenja informacija*. Dve različite funkcije deklarisane unutar Manny bi, *obe,* mogle da vide `name` deklarisano na najvišem nivou klase Manny. *Kako k*ôd unutar Jack, *tako* i kôd unutar Moe mogu da vide `one` deklarisane na najvišem nivou datoteke.

Stvari takođe imaju *životni vek*, koji je u suštini istovetan njihom opsegu. Stvari traju sve dok ih opseg kojih ih okružuje traje. Prema tome, u primeru 1-1, promenljiva `one` traje sve dok traje ova datoteka – tačnije, sve dok se program izvršava. Opšta je i trajna. Ali, promenljiva `name` deklarisana na najvišem nivou klase Manny postoji samo onoliko dugo koliko Manny postoji (uskoro ću da objasnim šta to znači). Stvari deklarisane na dubljem nivou imaju čak i kraći životni vek; na primer, vratimo se na ovaj kôd:

```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

U ovom kodu, klasa Cat i promenljiva `one` postoje samo tokom kratkog trena tokom koga putanja izvršavanja koda prolazi kroz `if` konstrukciju. Kada se pozove funkcija `silly`, deklariše se i nastaje Cat; zatim se deklariše i nastaje `one`; zatim se izvršava izvršni red `one = one + 1`, a zatim se opseg završava i Cat i `one` nestaju o obliku dima.

Članovi objekata

Unutar tri tipa objekata (klasa, struktura i enumeracija), stvari koje su deklarisane na najvišem nivou imaju posebne nazive, uglavnom iz istorijskih razloga. Uzmimo klasu Manny kao primer:

```
class Manny {
    let name = "manny"
    func sayName() {
        println(name)
    }
}
```

U ovom kodu:

- `name` je promenljiva deklarisana na najvišem nivou deklaracije objekta, zbog toga se ona naziva *svojstvo* (engl. *propety*) tog objekta,
- `sayName` je funkcija deklarisana na najvišem nivou deklaracije objekta, zbog toga se ona naziva *metoda* (engl. *method*) tog objekta.

Stvari deklarisane na najvišem nivou deklaracije objekta – svojstva, metode i bilo koji objekat deklarisan na tom nivou – jesu svi zajedno *članovi* tog objekta. Članovi imaju poseban značaj, pošto oni definišu koje *poruke* je dozvoljeno da šaljete tom objektu!

Prostori imena

Prostor imena (engl. *namespace*) je imenovana oblast programa. Prostor imena ima svojstvo da nazivima stvari unutar njega ne mogu da pristupaju stvari izvan njega, a da pre toga na neki način ne prođu kroz prepreku izgovarajući naziv te oblasti. To je dobro pošto se time omogućava da se isti naziv koristi na različitim mestima bez sukobljavanja. Očigledno je da su prostori imena i oblasti blisko povezani pojmovi.

Prostori imena pomažu objašnjavanje zašto je značajno deklarisanje objekta na najvišem nivou objekta, poput ovoga:

```
class Manny {
    class Klass {}
}
```

Ovaj način deklarisanja Klass u suštini ga „skriva" unutar Manny. Manny je prostor imena! Kôd *unutar* Manny može da neposredno vidi (i da se obrati) Klass. Ali, kôd izvan Manny to ne može da uradi. On mora da *izričito* navede prostor imena kako bi prošao prepreku koju taj prostor imena predstavlja. Da bi to uradio, mora prvo da kaže naziv koji ima Manny, posle koje sledi tačka, posle čega sledi izraz Klass. Ukratko, mora da kaže: `Manny.Class`.

Prostor imena, sam po sebi, ne nudi bezbednost niti privatnost; on je pogodnost. Na taj način, u primeru 1-1, klasi Manny sam dao klasu Klass i takođe sam klasi Moe dao klasu Klass. Ali, one nisu u sukobu, pošto su u različitim prostorima imena i mogu da ih razlikujem, ukoliko je potrebno, kao: `Manny.Klass` i `Moe.Klass`.

Vašoj pažnji nije izmaklo to da sintaksa za izričito ulaženje u prostor imena jeste sintaksa za slanje poruka koja koristi notaciju sa tačkom. To su, u suštini, iste stvari.

U suštini, slanje poruka omogućava da vidite unutar oblasti koje inače ne biste mogli da vidite. Kôd unutar Moe ne može *automatski* da vidi klasu Klass deklarisanu unutar Manny, ali *može da je vidi* preuzimanjem još jednog dodatnog koraka, tačnije izgovarajući `Manny.Klass`. *To* može da uradi pošto *može* da vidi Manny (pošto je Manny deklarisana na nivou koji kôd unutar Moe može da vidi).

Moduli

Prostori imena na najvišem nivou su *moduli* (engl. *modules*). Podrazumevano, vaša aplikacija je modul i stoga je prostor imena; naziv tog prostora imena je, grubo rečeno, naziv aplikacije. Na primer, ukoliko je neka moja aplikacija nazvana `MyApp`, tada ako ja

deklarišem klasu Manny na najvišem nivou datoteke, *stvarni* naziv te klase je `MyApp.Manny`. Ali, obično ne moram da koristim taj stvarni naziv, pošto je moj kôd već unutar istog prostora imena i neposredno mogu da vidim naziv Manny.

Softverski kosturi (engl. *framework*) su takođe moduli i stoga su takođe prostori imena. Na primer, softverski kostur Foundation u razvojnom okruženju Cocoa , gde NSString živi, je modul. Kada programirate iOS, reći ćete `import Foundation` (ili ćete, verovatnije, reći `import UIKit`, što samo po sebi uvozi Foundation), što omogućava da se obratite NSString, a da ne morate da kažete `Foundation.NSString`. Ali, *mogli* biste da kažete `Foundation.NSString`, a ukoliko ste bili toliko glupi da deklarišete drugačiji NSString u svom sopstveno modulu, *morali* biste da kažete `Foundation.NSString` kako biste ih razlikovali. Možete takođe da sami napravite sopstvene softverske kosture i oni će, takođe, biti moduli.

Prema tome, iznad i ispod nivoa datoteke, kao što je prikazano u primeru 1-1, nalaze se neke biblioteke (moduli) koje ta datoteka uvozi. Vaš kôd *uvek neizostavno uvozi Swift sam po sebi*. Mogli biste to da izričito uradite počinjući datoteku redom `import Swift`; nema potrebe da to radite, ali ne smeta ako uradite.

Ova činjenica je važna, pošto rešava najveću misteriju: odakle stvari kao što je `println` dolaze i zašto ih je moguće koristiti izvan bilo koje poruke bilo kom objektu? `Println` je u stvari funkcija deklarisana na najvišem nivou datoteke *Swift.h* – koju vaša datoteka može da vidi bez greške pošto uvozi Swift. Ona je stoga najobičnija funkcija najvišeg nivoa poput bilo koje druge. Mogli biste da kažete nešto poput `Swift.println("hello")`, ali verovatno nikad nećete, pošto nema sukoba naziva koji se time rešava.



U stvari, *možete* da vidite datoteku *Swift.h* i da je čitate i proučavate, a to može da bude i korisno. Da biste to uradili, pritisnite taster Command i mišem pritisnite izraz `println` u svom kodu. Drugi način je da izričito navedete `import Swift` i da pritisnite taster Command i mišem pritisnite izraz `Swift`. Pogledajte, tu je! Ovde nećete da vidite bilo kakav izvršni Swift *kôd*, ali ćete videti *deklaracije* za sve raspoložive Swift izraze, uključujući funkcije najvišeg nivoa kao što je `println`, operatore poput `+` i deklaracije ugrađenih tipova kao što su Int i String (potražite `struct Int`, `struct String` i tako redom).

Instance

Tipovi objekata – klase, strukture i enumeracije – imaju jednu važnu zajedničku osobinu: mogu da naprave instancu (engl. *instantiate*) objekta. U suštini, kada deklarišete tip objekta, samo definišete *tip*. Da biste uspostavili tip, potrebno je da napravite nešto – *instancu* – od tog tipa.

Tako, na primer, mogu da deklarišem klasu Dog:

```
class Dog {
}
```

I mojoj klasi mogu da dam metodu:

```
class Dog {
    func bark() {
        println("woof")
    }
}
```

Ali još uvek u suštini nemam nijedan objekat Dog u mom programu. Samo sam opisao *tip* stvari Dog kakav *bi bio* kada bih imao jednog. Da bih dobio stvarnog psa (engl. *Dog*), moram da *napravim* jednog. Postupak pravljenja stvarnog objekta Dog čiji tip je klasa Dog je postupak uspostavljanja objekta Dog. Rezultat je nov objekat – *instanca* Dog.

U jeziku Swift, instance se prave korišćenjem naziva tipa objekta kao naziva funkcije i pozivanjem te funkcije. To obuhvata korišćenje zagrada. Kada pridodate zagrade na naziv tipa objekta, šaljete veoma posebnu vrstu poruke tom tipu objekta: uspostavi se! Tako da sada nameravam da napravim instancu Dog:

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

Svašta se događa u ovom kodu! Uradio sam dve stvari. Uspostavio sam Dog, što je dovelo do toga da sam završio sa instancom Dog. Takođe sam stavio tu instancu Dog u kutiju nazvanu `fido` – deklarisao sam promenljivu i inicijalizovao tu promenljivu dodeljujući joj moju novu instancu Dog. Sada je `fido` *instanca klase Dog*. (Štaviše, pošto sam koristio `let`, `fido` će uvek biti ista instanca klase Dog. Mogao sam umesto toga da koristim `var`, ali čak i tad, inicijalizovanje `fido` kao instance Dog bi značilo da bi `fido` mogao biti samo neka instanca Dog-a posle toga.)

Sada kada imam instancu Dog, mogu da joj šaljem *poruke instance*. A šta mislite da je to? To su svojstva i metode objekta Dog! Na primer:

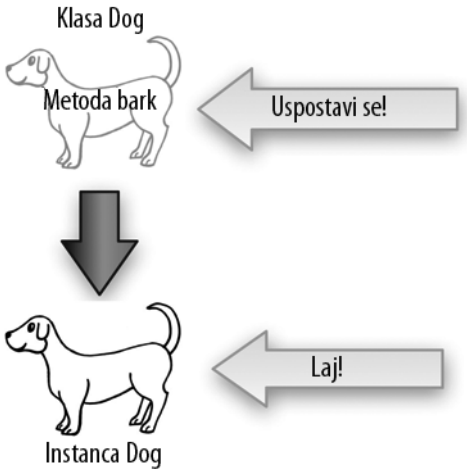
```
let fido = Dog()
fido.bark()
```

Ovaj kôd je valjan. Ne samo to, on radi: zapravo će dovesti do toga da se `"woof"` pojavi u konzoli. Napravio sam psa i naterao sam ga da laje (engl. *bark*)! (Pogledajte sliku 1-1.)

Naučili smo nešto veoma važno, pa mi dozvolite da zastanem i to pojasnim. Podrazumevano, svojstva i metode su *instance* svojstava i metoda. Ne možete da ih koristite kao poruke za sam tip objekta; morate da imate *instancu* da biste njoj slali te poruke. Kako stvari stoje, ovo ne valja i neće da se kompajlira:

```
Dog.bark() // greška pri kompajliranju
```

Moguće je definisati funkciju `bark` na takav način da `Dog.bark()` *jeste* valjano, ali bi to bila drugačija vrsta funkcije – funcija *klase* ili *statička* funkcija – i potrebno je da to kažete kada je deklarišete.



Slika 1-1. *Pravljenje instance i pozivanje metode instance*

Isto važi i za svojstva. Jedino mesto u kome je objekat Dog do sada imao ime je naziv promenljive koja mu je pridružena. Ali taj naziv nije svojstven objektu Dog *sam po sebi*. Dajmo objektu Dog svojstvo `name`:

```
class Dog {
    var name = ""
}
```

Ovo mi omogućava da postavim naziv objektu Dog, ali je potrebno da to bude *instanca* objekta Dog:

```
let fido = Dog()
fido.name = "Fido"
```

Moguće je deklarisati svojstvo `name` tako da reći `Dog.name` bude ispravno, ali bi to trebalo da bude drugačija vrsta svojstva – svojstvo *klase* ili *statičko* svojstvo – i potrebno je da to kažete kada ga deklarišete.

Zašto instance?

Čak i kada ne bi bilo stvari kao što je instanca, tip objekta je sam po sebi objekat. To znamo pošto je moguće poslati poruku tipu objekta: moguće je tip obekta posmatrati kao prostor imena i izričito ući u taj prostor imena (izraz `Manny.Klass` je takav slučaj). Štaviše, pošto postoje klasa i statički članovi, moguće je pozivati metodu neposredno na klasu, strukturu ili enumeraciju i pozivati se na klasu, strukturu ili enumeraciju. Zašto, onda, instance uopšte postoje?

Odgovor najvećim delom ima veze sa prirodom svojstava instance. Vrednost nekog svojstva instance definisana je uzimajući u obzir *tačno određenu instancu*. Ovo je to mesto gde instance dobijaju svoju pravu svrhu i moć.

Pogledajmo ponovo našu klasu Dog. Dao sam joj svojstvo `name` i metodu `bark`; sećate se, to su svojstvo instance i metoda instance:

```
class Dog {
    var name = ""
    func bark() {
        println("woof")
    }
}
```

Instanca Dog pojavljuje se sa praznim svojstvom `name` (prazan string). Ali njeno svojstvo `name` je `var`, tako da pošto imamo instancu Dog, njenom svojstvu `name` možemo da pridružimo novu vrednost String:

```
let dog1 = Dog()
dog1.name = "Fido"
```

Možemo takođe da tražimo svojstvo `name` instance Dog:

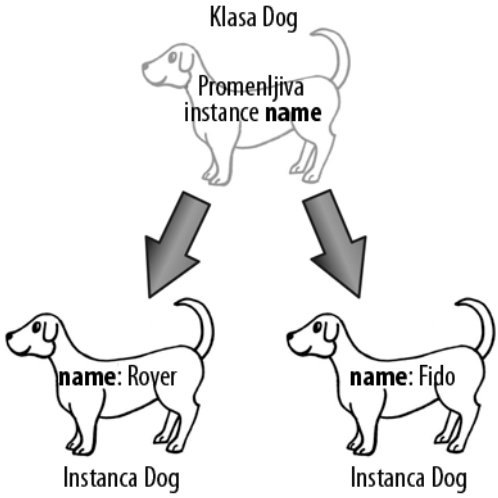
```
let dog1 = Dog()
dog1.name = "Fido"
println(dog1.name) // "Fido"
```

Važna stvar je da možemo napraviti više od jedne instance Dog, a da dve različite instance Dog imaju dve različite vrednosti svojstva `name` (slika 1-2):

```
let dog1 = Dog()
dog1.name = "Fido"
let dog2 = Dog()
dog2.name = "Rover"
println(dog1.name) // "Fido"
println(dog2.name) // "Rover"
```

Primećujete da svojstvo `name` instance Dog nema ništa sa nazivom promenljive koja je pridružena instanci Dog. Promenljiva je samo kutija. Možete da prenosite neku instancu iz jedne kutije u drugu. Ali, ta sama instanca zadržava svoj unutrašnji integritet:

```
let dog1 = Dog()
dog1.name = "Fido"
var dog2 = Dog()
dog2.name = "Rover"
println(dog1.name) // "Fido"
println(dog2.name) // "Rover"
dog2 = dog1
println(dog2.name) // "Fido"
```



Slika 1-2. *Dva psa sa različitim vrednostima svojstava*

Ovaj kôd nije promenio ime psa Rover; promenio je to koji je pas unutar kutije dog2, menjajući psa pod imenom Rover psom sa imenom Fido.

Prava moć objektno orijentisanog programiranja se sada pojavljuje. Postoji tip objekta Dog koji definiše *šta bi taj Dog trebalo da bude*. Naša deklaracija za Dog kaže da instanca Dog – *bilo koja* istanca Dog, *sve* instance Dog – ima svojstvo `name` i metodu `bark`. Ali i da *svaka* instanca Dog ima sopstvenu *vrednost* svojstva `name`. To su *različite* instance i održavaju svoje unutrašnje *stanje* (engl. *state*). Tako se više instanci istog tipa objekta *ponašaju* slično – i Fido i Rover mogu da laju, a to će uraditi kada im se pošalje poruka `bark` – ali su različite instance i imaju različite vrednosti svojstava: ime psa Fido je `"Fido"`dok je ime psa Rover `"Rover"`.

(Isto važi i za `1` i `2`, mada je ova činjenica malo teže razumljiva. Int ima svojstvo `value` (vrednost). `1` je Int čije svojstvo `value` je `1`, a `2` je Int čije svojstvo `value` je `2`. Međutim, ova činjenica nije toliko zanimljiva u stvarnom životu, pošto očigledno ne nameravate da *menjate* vrednost 1!)

Znači, instanca odražava metode instance njenog tipa, ali to nije *sve*, ona je takođe skup svojstava instance. Tip objekta je odgovoran za to *koja* svojstva određena instanca ima, ali ne neophodno i za *vrednosti* tih svojstava. Te vrednosti mogu da se menjaju u toku izvršavanja programa, a primenjuju se samo na tačno određenu instancu. Instanca je skup posebnih vrednosti svojstava.

Instanca je odgovorna ne samo za vrednosti, već takođe i za *vek trajanja* njenih svojstava. Pretpostavimo da smo uspostavili instancu Dog i njenom svojstvu `name` pridružili

vrednost `"Fido"`. Tada ova instanca Dog održava u životu string `"Fido"` sve dok ne zamenimo vrednost njenog svojstva `name` nekom drugom vrednošću *i onoliko dugo koliko ta instanca traje*.

Ukratko, instanca je istovremeno kôd i podatak. Kôd dobija od svog tipa i u tom smislu je zajednički za sve ostale instance tog tipa, ali podaci pripadaju samo njoj. Podaci traju sve dok instanca traje. Instanca ima, u svakom trenutku, stanje – potpuni skup njenih vlastitih vrednosti svojstava. Instanca je sredstvo za održavanje stanja. Ona je kutija za čuvanje podataka.

self

Instanca je objekat, a objekat je prijemnik podataka. Stoga je instanci potreban način da šalje poruke sama sebi. To je moguće učiniti magičnom rečju `self`. Ova reč može da se koristi uvek kad se očekuje naziv neke instance (instance odgovarajućeg tipa, to jest).

Na primer, recimo da želim da zadržim ono što Dog kaže kada laje – tačnije `"woof"` – u svojstvu. Tada mi je potrebno da se u mojoj primeni metode `bark` pozovem na to svojstvo. To mogu da uradim na sledeći način:

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
}
}
```

Slično, recimo da želim da napišem metodu instance `speak` koja je samo sinonim za `bark`. Moja primena metode `speak` može da se sastoji od jednostavnog pozivanja metode `bark` koju već imam. To mogu da uradim poput ovoga:

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        self.bark()
    }
}
```

Primećujete da se izraz `self` u ovom primeru pojavljuje samo u metodama instance. Kada kôd neke instance kaže `self` to se odnosi na *tu* instancu. Ako se iskaz `self.name` pojavljuje u kodu metode instance Dog, to znači naziv *ove* instance Dog, one čiji se kôd izvršava u tom trenutku.

Ispada da je svako korišćenje reči `self` koju sam upravo prikazao potpuno neobavezno. Možete da je izostavite i potpuno ista stvar će se dogoditi:

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(whatADogSays)
    }
    func speak() {
        bark()
    }
}
```

Razlog je u tome da ako izostavite primaoca poruke i poruka koju šaljete može da bude poslata za `self`, kompajler obezbeđuje `self` kao primaoca poruke bez vašeg uticaja. Međutim, *nikada* ne radim tako (osim greškom). U pitanju je samo stil i ja više volim da *izričito* koristim `self`. Mislim da je kôd u kome je reč `self` izostavljena teži za čitanje i razumevanje. A postoje i okolnosti u kojima *morate* da kažete reč `self`, tako da više volim da je koristim uvek kada mi je je dozvoljeno da je koristim.

Privatnost

Ranije sam rekao da prostor imena nije, sam po sebi, nepremostiva prepreka za pristupanje nazivima unutar njega. Ali *može* da se ponaša kao prepreka ukoliko to želite. Na primer, nisu svi podaci koje čuva neka instanca namenjeni tome da ih menja, ili čak da ih vidi, druga instanca. I nije svaka metoda instance namenjena za to da je pozivaju druge instance. Svakom pristojnom objektno orijentisanom programskom jeziku potreban je način da svojim objektima članovima podari *privatnost* – način kojim otežava drugim objektima da vide te članove ukoliko ne treba da budu viđeni.

Uzmimo, na primer:

```
class Dog {
    var name = ""
     whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        println(self.whatADogSays)
    }
}
```

U ovo slučaju drugi objekti mogu da dođu i promene svojstvo `whatADogSays`. Pošto ovo svojstvo koriste i `bark` i `speak`, mogli bismo da završimo sa psom koji, kada mu se kaže da laje, kaže „mjau" (engl. „*meow*"). Ovo ne izgleda baš naročito kao nešto poželjno:

```
let dog1 = Dog()
dog1.whatADogSays = "meow"
dog1.bark() // meow
```

Mogli biste da odgovorite: Pa dobro, zašto si `whatADogSays` deklarisao sa `var`? Deklariši ga sa `let` umesto toga. Napravi da bude nepromenljiv! Sada niko ne može da ga promeni:

```
class Dog {
    var name = ""
    let whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        println(self.whatADogSays)
    }
}
```

Ovo je dobar odgovor, ali nije dovoljno dobar. Postoje dva problema. Pretpostavimo da želim da sama instanca Dog može da menja `self.whatADogSays`. Tada `whatADogSays` *mora* da bude `var`; inače, čak ni sama instanca ne bi mogla da ga promeni. Takođe, pretpostavimo da ne želim da bilo koji drugi objekat *zna* šta taj Dog kaže, osim pozivanjem metoda `bark` ili `speak`. Čak i kada je deklarisano sa `let` ostali objekti i dalje mogu da *pročitaju* vrednost od `whatADogSays`. Možda mi se to ne sviđa.

Da bi rešio taj problem, Swift nudi ključnu reč `private`. Kasnije ću govoriti o svemu šta ta ključna reč znači, ali za sada je dovoljno da znate da rešava problem:

```
class Dog {
    var name = ""
    private var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        println(self.whatADogSays)
    }
}
```

Sada je `name` javno svojstvo, ali `whatADogSays` je privatno svojstvo: drugi objekti ne mogu da ga vide. Instanca Dog može da kaže `self.whatADogSays`, ali *drugi* objekat koji se poziva na instancu Dog kao, recimo, `dog1` ne može da kaže `dog1.whatADogSays`.

Ono što je važno što smo ovde naučili jeste da se podrazumeva da su članovi objekta javni, a ukoliko želite privatnost, morate to da tražite. Deklaracija klase definiše prostor imena; ovaj prostor imena zahteva da drugi objekti koriste još jedan nivo notacije sa tačkom da bi se pozvali na ono što se nalazi unutar tog prostora imena, ali drugi objekti *mogu* i dalje da se pozivaju na ono šta se nalazi unutar prostora imena; prostor imena, sam po sebi, ne zatvara vrata vidljivosti. Ključna reč `private` omogućava da zatvorite ta vrata.

Projektovanje

Sada znate šta je to objekat i šta je to instanca. Ali, koji tipovi objekata su potrebni za vaš program, koje metode i svojstva bi trebalo da imaju, kada i kako će biti uspostavljeni i šta bi trebalo da radite sa tim instancama kada ih budete imali? Nažalost, to ne mogu da vam kažem: to je umetnost – umetnost objektno orijentisanog programiranja. Ono što *mogu* da vam kažem je šta će biti ono o čemu ćete najviše voditi računa kada projektujete i primenjujete objektno orijentisani program – proces koji nazivam *narastanje programa*.

Projektovanje objektno orijentisanog programa mora da se zasniva na potpunom razumevanju prirode objekata. Želite da napravite tipove objekata koji u sebi sadrže pravu vrstu funkcionalnosti (metode) kojoj je pridružen pravi skup podataka (svojstva). Tada, pošto uspostavite te tipove objekata, želite da obezbedite da vaše instance traju koliko treba, budu raspoložive jedna drugoj i da imaju odgovarajuću sposobnost da međusobno komuniciraju.

Tipovi objekata i API interfejsi

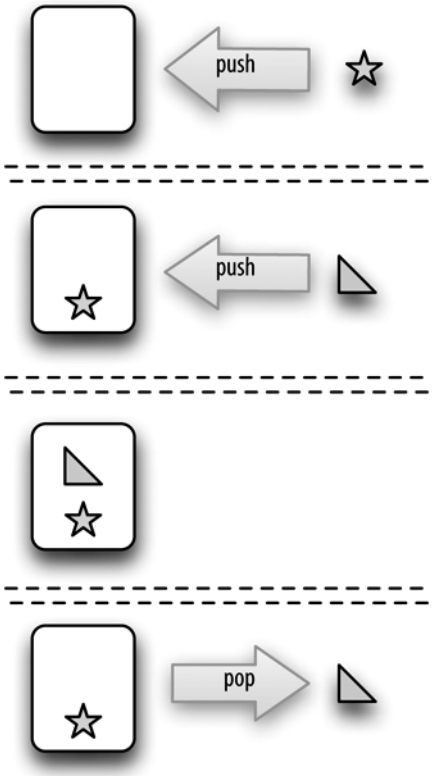
Datoteke vašeg programa imaće vrlo malo, ako ih uopšte budu imali, funkcija i promenljivih na najvišem nivou. Metode i svojstva tipova objekata – a posebno, metode instance i svojstva instance – biće ono gde će većina toga da se dešava. Tipovi objekata daju svakoj stvarnoj instanci njene posebne mogućnosti. Oni takođe pomažu da organizujete kôd svog programa tako da ima smisla i da se može održavati.

Prirodu objekata možemo ukratko svesti na dve fraze: enkapsulacija funkcionalnosti i održavanje stanja. (Ove fraze sam prvi put koristio pre mnogo godina u mojoj knjizi *REALbasic: Potpuni vodič*.)

Enkapsulacija funkcionalnosti
: Svaki objekat obavlja svoj vlastiti zadatak, a ostatku sveta – drugim objektima, a posebno se to odnosi na programera – predstavlja neprovidini zid čiji su jedini ulazi metode za koje je unapred rekao da će odgovarati i akcije za koje je unpared rekao da će vršiti kada mu se pošalju odgovarajuće poruke. Pojedinosti o tome kako, u pozadini, on stvarno primenjuje te akcije su sakrivene unutar njega; nijedan drugi objekat to ne treba da zna.

Održavanje stanja
: Svaka pojedinačna instanca je hrpa podataka koju ona održava. Često su ti podaci privatni, što u stvari znači da su enkapsulirani; nijedan drugi objekat ne zna koji su to podaci ili u kom obliku se čuvaju. Jedini način da se spolja otkrije koje podatke neki objekat održava je da postoji metoda ili javno svojstvo koji će ih obelodaniti.

Kao primer, zamislite objekat čiji je zadatak da popunjava skladište podataka – to bi mogla da bude instanca klase Stack. *Stek* (engl. *stack*) je struktura podataka u kome



Slika 1-3. *Stek podataka*

se podaci održavaju kao skup podataka po LIFO (last in, first out – prvi unutra, prvi napolje) redosledu. On odgovara na samo dve poruke: `push` i `pop`. Push (gurni) znači da dodaje dati deo podataka u skup. Pop (izvuci) znači da iz skupa podataka ukloni deo podataka koji je poslednji tamo stavljen i da ga isporuči. To liči na poređane tanjire: tanjiri se smeštaju na vrh gomile ili uzimaju sa vrha gomile jedan po jedan, tako da prvi tanjir koji je stavljen na dno ne može da se uzme dok se svi posle njega dodati tanjiri pre toga ne uklone (slika 1-3).

Objekat stek ilustruje enkapsulaciju funkcionalnosti pošto spoljašnji svet ne zna ništa o tome kako je stek u suštini ostvaren. To može da bude niz, povezana lista ili neki od brojnih drugih načina primene. Ali klijent objekat – objekat koji u stvari šalje poruke `push` ili `pop` objektu stek – ne zna ništa o tome i brine još manje, pod uslovom da objekat stek poštuje svoj dogovor o tome da se ponaša kao stek podataka. Ovo je takođe dobro za programera, koji može, kako se program razvija, da bezbedno zameni jednu primenu drugom ne oštećujući glomaznu mašineriju programa kao celine. A sa druge strane, objekat stek ne zna ništa i ne brine o tome ko mu kaže da ubaci ili izvuče podatke i zašto to radi. On se jednostavno mota okolo i svoj posao radi najbolje što može.

Objekat stek ilustruje održavanje stanja pošto nije samo prolaz do podataka u steku – on *jeste* stek podatka. Ostali objekti mogu da dobiju pristup do tih podataka, ali samo zato što imaju pristup do samog objekta stek i to samo na način na koji to objekat stek dozvoljava. Stek podataka je u suštini unutar objekta stek; niko drugi ne može da ih vidi. Sve što drugi objekti mogu da urade je da ih ubacuju ili izvlače. Ukoliko je u ovom trenutku tačno određeni objekat na vrhu steka podataka u našem objektu stek, tada bilo koji objekat koji pošalje poruku `pop` tom objektu stek dobiće taj objekat kao odgovor. Ukoliko nijedan objekat ne pošalje poruku `pop` tom objektu stek, tada objekat na vrhu steka ostaje tamo, čekajući.

Ukupan broj poruka koji je dozvoljen drugim objektima da pošalju svakom tipu objekta – njegov API (application programing interface – programski interfejs aplikacije) – sličan je spisku ili meniju stvari koje možete da tražite da taj tip objekta uradi. Vaši tipovi objekata dele vaš kôd – njihovi API interfejsi sačinjavaju osnovu za komunikaciju između ovih celina.

U stvarnom životu, kada programirate iOS, najveći deo tipova objekta sa kojima ćete raditi neće biti vaš, već ih je napravio Apple. Sam Swift ima nekoliko korisnih tipova objekata, kao što su String ili Int; takođe ćete koristiti `import UIKit`, koji obuhvata *prilično* broj tipova objekata, koji će svi naglo oživeti u vašem programu. Niste napravili nijedan od tih tipova objekata, a da biste naučili da ih koristite, pročitaćete javno objavljene API interfejse, poznate kao *dokumentacija*. Cocoa dokumentacija koju je objavio Apple sastoji se uglavnom od stranica gde svaka stranica navodi i opisuje svojstva i metode koji se nude jednim tipom objekta. Na primer, da biste znali koje poruke možete da pošaljete instanci NSString, počećete od proučavanja dokumentacije za klasu NSString. Odgovarajuća stranica je u stvari samo dugačak spisak svojstava i metoda, koji kaže šta sve objekat NSString može da uradi. To nije sve na svetu što se zna o objektu NSString, ali je veliki deo.

Stoga, u stvarnom životu, „mudar programer" o kome sam govorio nešto ranije jeste, većim delom, Apple. *Vaša* mudrost neće ležati u stvaranju novih tipova objekata, već u *korišćenju* tipova objekata koje vam je Apple već dao. Možete *takođe* da pravite nove tipove objekata, i to ćete da radite, ali srazmerno to ćete raditi neuporedivo ređe nego što ćete koristiti tipove objekata koji već postoje.

Pravljenje instance, oblasti i životni vek

Najvažnije pokretačke celine u Swift programu su svakako instance. Tipovi objekata definišu koje *vrste* instanci mogu da postoje i kako se svaka vrsta instance ponaša. Ali, stvarne instance tih tipova su nosioci stanja pojedinih „stvari" o kojima se određeni program odnosi, a metode i svojstva instanci su poruke koje mogu biti poslate instancama. Stoga je neophodno da *postoje* instance da bi program *uradio* bilo šta.

Same po sebi, međutim, instance *ne postoje*! Pogledajte ponovo primer 1-1, gde smo definisali neke tipove objekata, ali od njih nismo napravili instance. Ukoliko bismo pokrenuli ovaj program, naši tipovi objekata bi postali ni iz čega, ali to je sve što bi nastalo. Napravili smo svet čistih mogućnosti – neke tipove objekta koji *mogu* da postoje. U tom svetu se u stvari ništa *ne događa*.

Instance ne nastaju magijom. Morate da uspostavite neki tip kako biste dobili instancu. Većina onoga što se radi u vašem programu, prema tome, sastojaće se od uspostavljanja tipova. A pošto naravno želite da te instance opstanu, takođe ćete svakoj novoj napravljenoj instanci pridružiti promenljivu kao kutiju u kojoj ćete je držati, daćete joj naziv i odrediti vek trajanja. Određena instanca će *opstati* saglasno veku trajanja promenljive koja se odnosi na nju. I instanca će biti *vidljiva* drugim instancama saglasno oblasti promenljive koja se odnosi na nju.

Najveća umetnost objektno orijentisanog programiranja ovde izlazi na videlo, koja se sastoji od toga da se instancama dodeli dovoljno dug vek trajanja i od toga da se naprave međusobno vidljivim. Često ćete neku instancu smeštati u posebnu kutiju – pridružujući joj posebnu promenljivu, deklarisanu za tačno određenu oblast – baš tako da, zahvaljući pravilima o veku trajanja i oblasti promenljivih, ta instanca će *opstati* dovoljno dugo da bude korisna u vašem programu dok mu je potrebna, a tako da ostali kôd može da *dobije referencu* na tu instancu i kasnije joj se obrati.

Planiranje kako ćete praviti instance i raditi na njihovom veku trajanja i komunikaciji između tih instanci može da zvuči zastrašujuće. Srećom, u stvarnom životu, kada programirate iOS, samo radno okruženje Cocoa će vam ponuditi početni skelet.

Na primer, znaćete od početka da vam je, za neku iOS aplikaciju, potreban tip delegata aplikacije (engl. *app delegate*) i tip kontrolera prikaza (engl. *view controller*), a u stvari kada pravite iOS projekat aplikacije, Xcode će vam ih dati. Štaviše, kada se vaša aplikacija pokrene, izvršni program će za vas uspostaviti te tipove objekata i smestiće te instance u čvrstu i korisnu međusobnu vezu. Izvršni program će napraviti instancu app delegate i odrediti tako da ona traje tokom veka trajanja aplikacije; napraviće instancu window (prozor) i pridružiti je svojstvu instance app delegate; a napraviće instancu view controller i pridružiti je svojstvu instance window. Na kraju, instanca view controller imaće prikaz, koji će se automatski pojaviti u prozoru.

Na taj način, a da ništa još niste uradili, već ćete imati neke objekte koji će opstati tokom veka trajanja aplikacije, uključujući onaj koji je osnova na kojoj se vidi interfejs vaše aplikacije. Što je još važnije, imate dobro definisan opšte dostupan način za pozivanje svih tih objekata. To znači da, bez pisanja bilo kakvog koda, već imate pristup do nekih važnih objekata i imate početno mesto za smeštanje bilo kakvih drugih objekata sa dugim vekovima trajanja i drugih vidljivih delova interfejsa koji su možda potrebni vašoj aplikaciji.

Rezime i zaključak

Kada zamišljamo konstruisanje nekog objektno orijentisanog programa za obavljanje određenog zadatka u mislima nam je priroda objekta. Postoje tipovi i instance. Tip je skup metoda koje opisuju šta sve instance tog tipa mogu da urade (enkapsulacija funkcionalnosti). Instance istog tipa razlikuju se samo po vrednosti svojih svojstava (održavanje stanja). Saglasno tome pravimo plan. Objekti su organizaciona alatka, skup kutija za enkapsulaciju koda koji ispunjava određeni zadatak. Takođe su konceptualna alatka. Programer, budući da primoran da razmišlja u smislu izdvojenih objekata, mora da podeli ciljeve i ponašanje programa u izdvojene zadatke, pri čemu se svaki zadatak pridružuje odgovarajućem objektu.

U isto vreme, nijedan objekat nije izolovan od ostalih. Objekti mogu međusobno da sarađuju, odnosno da međusobno komuniciraju – to jest, šalju poruke između sebe. Putevi kojima odgovarajuće veze za komunikaciju mogu da se razmešte su bezbrojni. Postaviti odgovarajući razmeštaj – *arhitekturu* – za uspešan i nesmetan međusoban odnos između objekata je jedan od najizazovnijih elemenata programiranja zasnovanog na objektima. U iOS programiranju, podršku dobijate od Cocoa radnog okruženja, koje nudi početni skup tipova objekata i praktičan osnovni arhitektnoski kostur.

Korišćenje objektno orijentisanog programiranja za uspešno pravljenje programa koji će da radi ono što želite da radi, a da program bude jasan i lak za održavanje je samo po sebi umetnost; vaša umešnost će se povećavati povećavanjem iskustva. Pre ili kasnije možda ćete poželeti da pročitate još nešto o uspešnom planiranju i konstruisanju arhitekture objektno orijentisanog programa. Posebno preporučujem dve izuzetne, omiljene knjige. *Refaktorisanje*, koju je napisao Martin Flower (u izdanju CET-a, 2003), opisuje šta je možda potrebno da biste prepakovali metode koje pripadaju određenim klasama (i kako da se izborite sa svojim strahovima da to uradite). *Gotova rešenja*, koju su napisali Erich Gamma, Richard Helm, Ralph Jonson i John Vilssides (poznati kao „četvoročlana banda"), predstavlja bibliju građenja objektno orijentisanih programa, u kojoj su dati svi načini na koje možete razmeštati objekte sa pravom moći i pravim znanjem svakog od njih (CET, 2002).