3

Upravljanje memorijom pomoću automatskog brojanja referenci

U ovom poglavlju ćete naučiti kako se upravlja memorijom u operativnom sistemu iOS i šta se podrazumeva pod *automatskim brojanjem referenci* (engl. *automatic reference counting*, *ARC*). Počećemo od osnovnih pojmova koji se odnose na memoriju aplikacije.

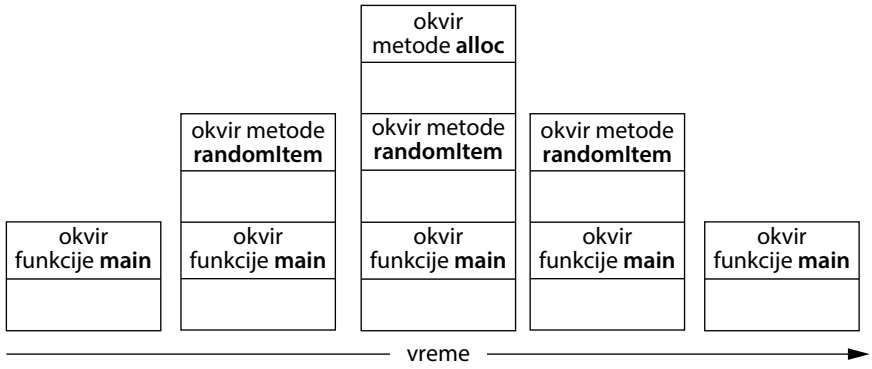
Stek

Kada se izvršava metoda (ili funkcija), dodeljuje joj se jedinica memorije u delu memorije koji se naziva *stek* (engl. *stack*). Dodeljena jedinica memorije naziva se *okvir* (engl. *frame*) i skladišti vrednosti promenljivih koje se deklarišu u metodi. Promenljiva deklarisana u metodi naziva se *lokalna promenljiva* (engl. *local variable*).

Kada se aplikacija pokrene i izvrši funkciju **`main()`**, okvir dodeljen funkciji **`main()`** stavlja se na stek. Kada funkcija **`main()`** pozove drugu metodu (ili funkciju), okvir dodeljen toj metodi stavlja se na vrh steka. Metoda može da pozove drugu metodu i tako redom, sve dok se u steku ne dobije niz okvira naslaganih jedan povrh drugog. Čim se metoda ili funkcija izvrše, njihov okvir se „skida“ s vrha steka i uništava. Ako se metoda pozove ponovo, dodeljuje joj se nov okvir u memoriji i postavlja na stek.

Na primer, u aplikaciji RandomItems, funkcija **`main`** izvršava metodu **`randomItem`** klase **`BNRItem`**, koja zatim izvršava metodu **`alloc`**. Stek bi u tom slučaju izgledao kao na slici 3.1. Okvir funkcije **`main()`** ostaje u memoriji dok se izvršavaju ostale metode, jer izvršavanje same funkcije još nije završeno.

Metoda **`randomItem`** izvršava se u petlji unutar funkcije **`main()`**. U svakoj novoj iteraciji petlje, stek se povećava ili smanjuje, zavisno od toga da li se okviri na njega postavljaju, ili se s njega uklanjaju.



Slika 3.1 Kako se stek povećava i smanjuje.

Hip

Postoji još jedan deo memorije, koji se naziva *hip* (engl. *heap*) i odvojen je od steka. Stek možete zamisliti kao uredno naslagan skup okvira, a hip kao mesto gde su svi objekti jezika Objective-C nabacani u velikoj i neurednoj gomili. Da bi se odredilo gde se objekti nalaze u hipu, koriste se pokazivači.

Kada klasi šaljete poruku **`alloc`**, jedinica memorije dodeljuje se iz hipa. Ta jedinica memorije predstavlja objekat i sadrži prostor za promenljive instance tog objekta. Instanca klase **`BNRItem`** ima pet promenljivih instance: četiri pokazivačke promenljive (`isa`, `_itemName`, `_serialNumber` i `_dateCreated`) i jednu celobrojnu – `int` (`_valueInDollars`). Tako dodeljena jedinica memorije sadrži prostor za jednu vrednost `int` i četiri pokazivača. Pokazivači skladište adrese drugih objekata u hipu.

iOS aplikacija definiše objekte u vreme pokretanja, i najčešće nastavlja da ih definiše dok god se izvršava. Da je hip memorija beskonačna, aplikacija bi mogla da definiše beskonačan broj objekata i oni bi mogli da postoje dok god se aplikacija izvršava.

Međutim, veličina hip memorije za aplikaciju je ograničena, pogotovo memorija na iOS uređaju. Stoga se ovom vrstom resursa mora upravljati, i veoma je važno da se redovno uništavaju objekti koji se više ne koriste. Tako se hip memorija oslobađa da bi se mogla ponovo koristiti za definisanje novih objekata, pri čemu se mora voditi računa o tome da se ne unište objekti koji su i dalje potrebni za rad.

ARC i upravljanje memorijom

Dobro je to što ne morate voditi računa o tome koji objekti treba da ostanu u memoriji, a koji ne. Umesto vas, memorijom aplikacije upravlja automatsko brojanje referenci (ARC), koje će koristiti sve aplikacije u ovoj knjizi. Pre nego što je ova mogućnost postala dostupna, aplikacije su koristile *ručno brojanje referenci* (engl. *manual reference counting*). Više informacija o njemu pronaći ćete na kraju poglavlja.

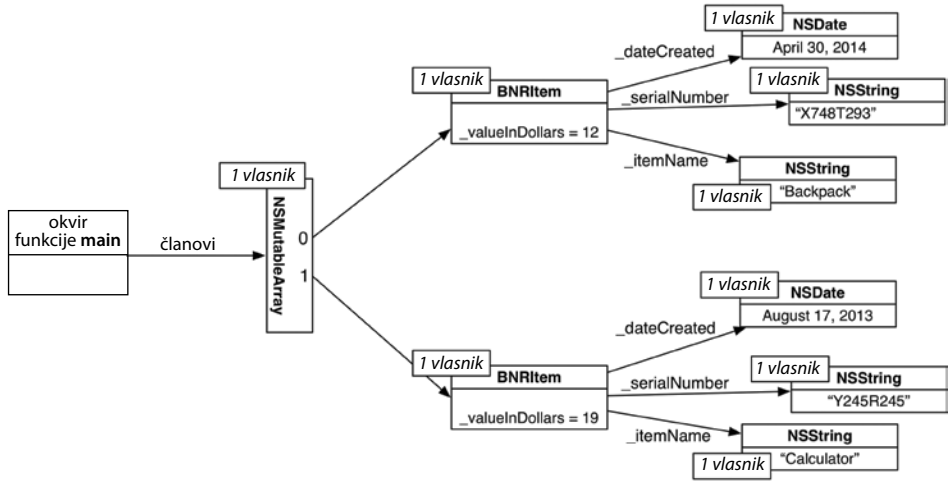
U većini slučajeva, ARC će automatski upravljati memorijom aplikacije. I pored toga, važno je da razumete na čemu se ova osobina zasniva, kako biste mogli da intervenišete ukoliko to bude potrebno. Započećemo konceptom vlasništva nad objektima.

Pokazivačke promenljive i vlasništvo nad objektima

Pokazivačke promenljive podrazumevaju *vlasništvo* (engl. *ownership*) nad objektima na koje pokazuju.

- Kada metoda ili funkcija imaju lokalnu promenljivu koja pokazuje na objekat, za promenljivu se kaže da *poseduje* objekat na koji pokazuje.
- Kada objekat ima promenljivu instance koja pokazuje na drugi objekat, za objekat s pokazivačem se kaže da *poseduje* pokazani objekat.

Razmislite ponovo o aplikaciji RandomItems. U njoj se instanca klase **`NSMutableArray`** definiše unutar funkcije **`main()`**, a zatim joj se dodaje deset instanci klase **`BNRItem`**. Na slici 3.2 prikazani su neki objekti u aplikaciji RandomItems i pokazivači koji ih referenciraju.



Slika 3.2 Dijagram objekata aplikacije RandomItems (sa samo dva člana).

Unutar funkcije **`main()`**, lokalna promenljiva `items` pokazuje na instancu klase **`NSMutableArray`**, pa funkcija **`main()`** poseduje tu instancu klase **`NSMutableArray`**.

S druge strane, niz poseduje instance klase **`BNRItem`**. Objekat kolekcije, kao što je instanca klase **`NSMutableArray`**, ne sadrži stvarne objekte nego pokazivače na njih, a ti pokazivači podrazumevaju vlasništvo: niz uvek poseduje objekte koji se nalaze „u“ nizu.

Na kraju, svaka instanca klase **`BNRItem`** poseduje objekte na koje pokazuju njene promenljive instance.

Zamisao vlasništva nad objektima korisna je da bi se odredilo da li će objekat biti uništen kako bi se memorija oslobodila za nešto drugo.

- *Objekat koji nema vlasnike biće uništen*. Objektu bez vlasnika ne mogu se slati poruke, on je izolovan i beskoristan za aplikaciju. Njegovim zadržavanjem zauzima se dragoceni prostor u memoriji, što se naziva *curenje memorije* (engl. *memory leak*).
- *Objekat koji ima jednog ili više vlasnika neće biti uništen*. Ako se objekat uništi, a neki drugi objekat ili metoda i dalje imaju pokazivač na njega (ili, preciznije rečeno, pokazivač koji skladišti adresu na kojoj se objekat *nekada* nalazio), situacija se zaoštrava: slanje poruke preko ovog pokazivača može dovesti do otkaza aplikacije. Uništavanje objekta koji je i dalje potreban naziva se *preuranjeno oslobađanje* (engl. *premature deallocation*), a poznato je i pod nazivima *neupareni pokazivač* (engl. *dangling pointer*) ili *neuparena referenca* (engl. *dangling reference*).

Kako objekti gube vlasnike

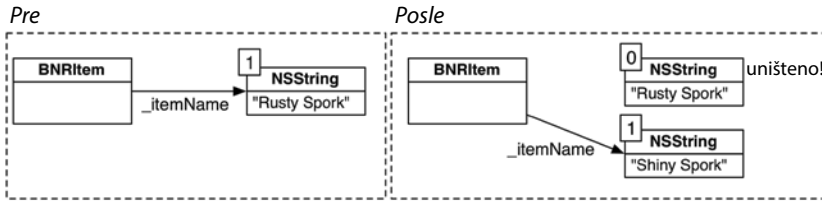
Objekat može izgubiti vlasnika na sledeće načine:

- Promenljiva koja pokazuje na objekat je promenjena, tako da pokazuje na drugi objekat.
- Za promenljivu koja pokazuje na objekat zadata je vrednost `nil`.
- Sam vlasnik objekta je uništen.
- Objekat u kolekciji, kao što je niz, uklonjen je iz te kolekcije.

Analiziraćemo svaku navedenu situaciju.

Menjanje pokazivača

Zamislite instancu klase **`BNRItem`**. Njena promenljiva instance `_itemName` pokazuje na instancu @"RustySpork" klase **`NSString`.** Ako pribor očistite od rđe, on više neće spadati u zarđali („Rusty Spork") nego u očišćeni pribor („Shiny Spork"), pa ćete vrednost promenljive instance `_itemName` promeniti tako da pokazuje na drugu instancu klase **`NSString`**.



Kada se vrednost promenljive instance `_itemName` promeni iz adrese znakovnog niza „Rusty Spork“ u adresu znakovnog niza „Shiny Spork“, niz „Rusty Spork“ će izgubiti svog vlasnika i, ako nema druge vlasnike, biće uništen.

Zadavanje vrednosti nil za pokazivač

Zadavanje vrednosti `nil` za pokazivač predstavlja odsustvo objekta. Na primer, imate instancu klase **`BNRItem`** koja predstavlja televizor. Ako neko ošteti serijski broj televizora, za promenljivu instance `_serialNumber` zadaćete vrednost `nil`, a instanca klase **`NSString`**, na koju je promenljiva `_serialNumber` ranije pokazivala, izgubiće svog vlasnika.

Uništavanje vlasnika

Kada se objekat uništava, objekti koje on poseduje gube svog vlasnika. Na taj način, jedan objekat koji se oslobađa može da uzrokuje kaskadu oslobođenih objekata.

Metoda ili funkcija mogu da poseduju objekte preko svojih lokalnih promenljivih. Kada se metoda ili funkcija izvrše a njihov okvir skine sa steka, objekti koje poseduju izgubiće vlasnika.

Uklanjanje objekta iz kolekcije

Postoji još jedan način na koji objekat može izgubiti vlasnika. Objekat u objektu kolekcije, u vlasništvu je objekta kolekcije. Kada iz promenljivog objekta kolekcije uklonite neki objekat, kao što je instanca klase **`NSMutableArray`**, uklonjeni objekat gubi vlasnika.

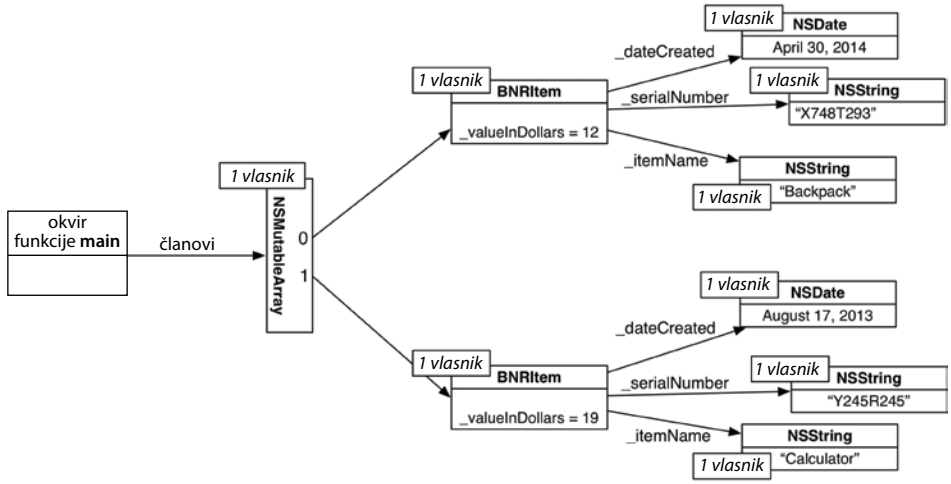
```
[items removeObject:item];    // Objekat na koji pokazuje član gubi
                              // vlasnika
```

Imajte na umu da gubljenje vlasnika na bilo koji od ovih načina neće nužno dovesti do uništavanja objekta; ako negde postoji još neki pokazivač na objekat, on će i dalje ostati u memoriji. Ukoliko objekat izgubi svog poslednjeg vlasnika, i on će prestati da postoji.

Lanci vlasništva

Budući da objekti poseduju objekte koji mogu da poseduju druge objekte, uništavanje jednog objekta može izazvati lančanu reakciju u pogledu gubitaka vlasništva, uništavanja objekata i oslobađanja memorije.

Ilustrovaćemo ovu pojavu pomoću aplikacije RandomItems. Pogledajte još jednom njen dijagram objekata.



Slika 3.3 Objekti i pokazivači u aplikaciji RandomItems.

Kada se završi ispisivanje niza u datoteci `main.m`, za vrednost promenljive `items` zadajete `nil`. Zbog toga će niz izgubiti svog jedinog vlasnika i biće uništen.

Međutim, tu nije kraj. Nakon niza, uništavaju se i svi pokazivači na instance klase **`BNRItem`**. Kada te promenljive nestanu, niko više nije vlasnik članova, pa se i oni uništavaju.

Uništavanje klase **`BNRItem`** na kraju uništava i njene promenljive instance, zbog čega niko više ne poseduje objekte na koje one pokazuju pa se i oni uništavaju.

Napišimo programski kôd koji će izvesti ovo o čemu govorimo. Klasa **`NSObject`** primenjuje metodu **`dealloc`** koja se šalje objektu pre nego što se on uništi. Metodu **`dealloc`** u klasi **`BNRItem`** možete redefinisati tako da ispiše nešto na konzoli kada se član uništi.

U projektu RandomItems otvorite datoteku `BNRItem.m` i redefinišite metodu **`dealloc`**.

```
- (void)dealloc
{
    NSLog(@"Destroyed: %@", self);
}
```

U datoteku `main.m` dodajte sledeći red koda:

```
NSLog(@"Setting items to nil...");
items = nil;
```

Prevedite i izvršite aplikaciju. Kada se članovi ispišu na ekranu, prikazaće se poruka da je za promenljivu `items` zadata vrednost `nil`. Nakon toga ćete videti kako se uništava svaka instanca klase **`BNRItem`** koja je prikazana na konzoli.

Na kraju, nijedan objekat više neće zauzimati memoriju i preostaće samo funkcija **main**. Do automatskog čišćenja i recikliranja memorije došlo je zato što je za promenljivu items zadata vrednost nil. U tome je moć ARC-a.

Jake i slabe reference

Kad god pokazivačka promenljiva pokazuje na neki objekat, to znači da on ima vlasnika i da ostaje u memoriji. To se naziva *jaka referenca* (engl. *strong reference*).

Promenljiva može i da *ne* preuzme vlasništvo nad objektom na koji pokazuje. Promenljiva koja nema vlasništvo nad objektom naziva se *slaba referenca* (engl. *weak reference*).

Slaba referenca je korisna za sprečavanje problema koji se naziva *ciklus jakih referenci* (engl. *strong reference cycle*), a poznat je i kao *ciklus zadržavanja* (engl. *retain cycle*). Takav ciklus nastaje kada dva ili više objekata imaju međusobne jake reference, što nije poželjno. Kada dva objekta poseduju jedan drugog, ne mogu se uništiti pomoću ARC-a. Čak i ako se svi ostali objekti odreknu vlasništva nad tim objektima, oni (kao i objekti koje poseduju) ostaju u svojoj mreži međusobnog vlasništva.

Tako ciklus jakih referenci predstavlja curenje memorije, i ARC-u treba vaša pomoć da bi rešio ovaj problem. To se radi tako što jednu od referenci pretvarate u slabu.

Da bi vam bilo jasno šta se dešava, uvešćemo ciklus jakih referenci u aplikaciju RandomItems. Prvo ćete instancu klase **BNRItem** definisati tako da sadrži još jednu instancu klase **BNRItem** (koja će predstavljati predmete kao što su ranac ili novčanik). Instanca će imati informaciju o tome koja ga instanca sadrži.

U datoteci BNRItem.h deklarišite dve promenljive instance i njihove pristupne metode.

```
@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    BNRItem *_container;
}

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                  valueInDollars:(int)value
                    serialNumber:(NSString *)sNumber;
```

```
- (instancetype)initWithItemName:(NSString *)name;

- (void)setContainedItem:(BNRItem *)item;
- (BNRItem *)containedItem;

- (void)setContainer:(BNRItem *)item;
- (BNRItem *)container;
```

U datoteci `BNRItem.m` primenite pristupne metode.

```
- (void)setContainedItem:(BNRItem *)item;
{
    _containedItem = item;

    // Kada se definiše član koji ulazi u sadržaj kontejnera,
    // tom članu se dodeljuje pokazivač na njegov kontejner
    item.container = self;
}

- (BNRItem *)containedItem
{
    return _containedItem;
}

- (void)setContainer:(BNRItem *)item
{
    _container = item;
}

- (BNRItem *)container
{
    return _container;
}
```

U datoteci `main.m` uklonite kôd koji popunjava niz slučajnim članovima. Zatim definišite dva nova člana, dodajte ih u niz i zadajte da pokazuju jedan na drugi.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        BNRItem *backpack = [[BNRItem alloc] initWithItemName:@"Backpack"];
        [items addObject:backpack];
```

```
        BNRItem *calculator = [[BNRItem alloc] initWithItemName:
                                                @"Calculator"];
        [items addObject:calculator];

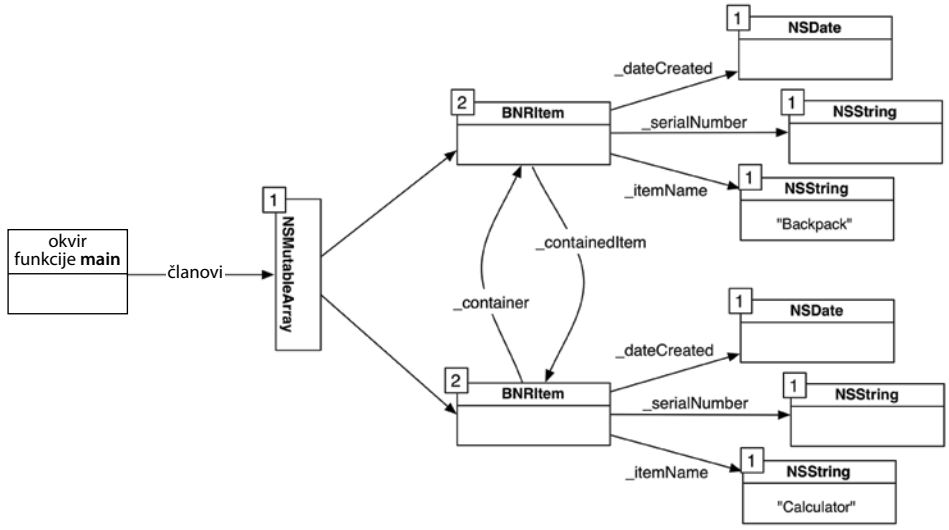
        backpack.containedItem = calculator;

        backpack = nil;
        calculator = nil;

        for (BNRItem *item in items)
            NSLog(@"%@", item);

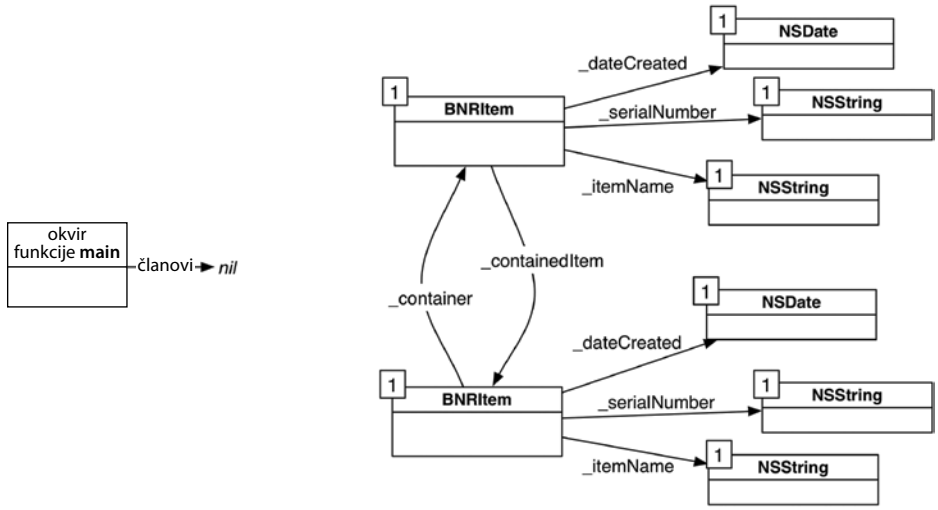
        NSLog(@"Setting items to nil...");
        items = nil;
    }
    return 0;
}
```

Evo kako aplikacija sada izgleda:



Slika 3.4 Aplikacija RandomItems sa ciklusom jakih referenci.

Prevedite i izvršite aplikaciju. Ovoga puta nećete videti nikakve poruke o uništavanju objekata **`BNRItem`**. Ovo je ciklus jakih referenci: ranac (backpack) i kalkulator (calculator) imaju jake međusobne reference i zato ne postoji način da se ovi objekti unište. Na slici 3.5 prikazani su objekti u aplikaciji, koji zauzimaju memoriju i nakon što je za promenljivu `items` zadata vrednost `nil`.



Slika 3.5 Curenje memorije.

Postoje dva člana kojima se ne može pristupiti ni iz jednog drugog dela aplikacije (u ovom slučaju, pomoću funkcije **`main()`**), a oni i dalje postoje i ne rade ništa korisno. Štaviše, budući da se ne mogu uništiti oni, ne mogu ni objekti na koje pokazuju njihove promenljive instanci.

Ovaj problem se rešava tako što se jedan od pokazivača između članova pretvara u slabu referencu. Da biste odlučili koji od njih treba da postane slab, zamislite da se objekti ciklusa nalaze u odnosu roditelj–dete: roditelj može da poseduje dete, ali dete nikada ne može da poseduje svog roditelja. U našem ciklusu jakih referenci, ranac je roditelj, a kalkulator dete. Zato ranac može da zadrži svoju jaku referencu na kalkulator (promenljivu instance `_containedItem`), ali referenca kalkulatora na ranac (promenljiva instance `_container`) mora da postane slaba.

Da biste promenljivu deklarisali kao slabu referencu, upotrebite atribut `__weak`. U datoteci `BNRItem.h` izmenite promenljivu instance `container` tako da postane slaba referenca.

```
__weak BNRItem *_container;
```

Ponovo prevedite i izvršite aplikaciju. Ovoga puta, objekti će biti uništeni.

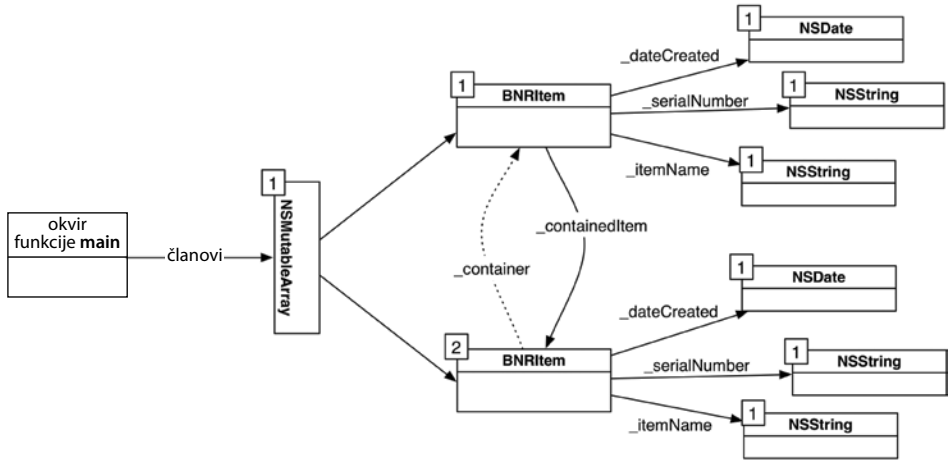
U većini slučajeva, ciklusi jakih referenci mogu se prevesti u odnos roditelj–dete. Roditelj obično zadržava jaku referencu na dete i zato pokazivač na roditelja, ukoliko je potreban detetu, mora biti slaba referenca kako bi se izbegao ciklus jakih referenci.

Dete s jakom referencom na *roditelja svog roditelja* takođe uzrokuje ciklus jakih referenci. Zato se i na ovu situaciju primenjuje isto pravilo: ako je detetu potreban pokazivač na roditelja njegovog roditelja (ili roditelja roditelja njegovog roditelja itd.), taj pokazivač mora predstavljati slabu referencu.

U Appleove razvojne alatke spada i alatka Leaks, koja vam pomaže da pronađete cikluse jakih referenci. U poglavlju 14 naučićete kako se ova alatka koristi.

Slaba referenca ima informaciju o tome kada je objekat na koji pokazuje uništen i odgovara na tu informaciju tako što zadaje `nil` kao sopstvenu vrednost. Shodno tome, ako se ranac uništi, za promenljivu instance `_container` kalkulatora automatski se zadaje vrednost `nil`, što je prikladno rešenje. Kada za promenljivu instance `_container` ne bi bila zadata vrednost `nil`, uništavanje objekta ostavilo bi neuparen pokazivač, što bi moglo izazvati pad aplikacije.

Sledi tekući dijagram aplikacije RandomItems. Strelica koja predstavlja pokazivačku promenljivu `container` prikazana je ovde isprekidanom linijom. Takva linija označava slabu referencu. Jake reference uvek se označavaju punim linijama.



Slika 3.6 U aplikaciji RandomItems izbegnut je ciklus jakih referenci.

Svojstva

Kad god ste deklarisali promenljivu instance u klasi **`BNRItem`**, deklarisali ste i primenili par pristupnih metoda. Sada ćete naučiti da koristite *svojstva* (engl. *properties*), koja predstavljaju zgodnu zamenu za pristupne metode, jer vam štede vreme potrebno za pisanje njihovog koda i omogućavaju da datoteke klase postanu preglednije.

Deklarisanje svojstava

Deklaracija svojstava ima sledeći oblik:

```
@property NSString *itemName;
```

Deklarisanjem svojstva podrazumevano dobijate jednu promenljivu instance i dve pristupne metode za promenljivu instance. Pogledajte tabelu 3.1, u kojoj je

na levoj strani prikazana klasa koja ne koristi svojstva, a na desnoj ista klasa sa svojstvima.

Tabela 3.1 Klasa sa svojstvima i klasa bez svojstava.

	Bez svojstava	**Sa svojstvima**

BNRThing.h	`@interface BNRThing : NSObject`
	`{`
	`  NSString *_name;`
	`}`
	`- (void)setName:(NSString *)n;`
	`- (NSString *)name;`
	`@end`
		`@interface BNRThing : NSObject`
		
		`@property NSString *name;`
		
		`@end`

BNRThing.m	`@implementation BNRThing`
	
	`- (void)setName:(NSString *)n`
	`{`
	`  _name = n;`
	`}`
	
	`- (NSString *)name`
	`{`
	`  return _name;`
	`}`
	
	`@end`
		`@implementation BNRThing`
		`@end`

Klase u tabeli 3.1 su iste: obe imaju po jednu promenljivu instance za ime instance, kao i metode za zadavanje i učitavanje vrednosti imena. Na levoj strani sami pišete ove deklaracije i promenljive instance, a na desnoj samo deklarišete svojstvo.

U klasi **BNRItem** zamenićete promenljive instance i pristupne metode svojstvima.

U datoteci BNRItem.h, obrišite oblast s promenljivama instance i deklaracije pristupne metode, a zatim dodajte deklaracije svojstava koja će ih zameniti.

```
@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    __weak BNRItem *_container;
}

@property BNRItem *containedItem;
@property BNRItem *container;

@property NSString *itemName;
@property NSString *serialNumber;
```