Uvod u programski jezik Ruby

D400

priručnik za polaznike © 2008 Srce

TEČAJEVIsrca

srce
Sveučilište u Zagrebu
Sveučilišni računski centar

Ovu inačicu priručnika izradio je autorski tim Srca u sastavu:

Autor: Hrvoje Marjanović

Recenzent: Viktor Matić

Urednik: Vladimir Braus

Lektorica: Milvia Gulešić Machata

Sveučilište u Zagrebu

Sveučilišni računski centar

Josipa Marohnića 5, 10000 Zagreb

tecajevi@srce.hr

ISBN: 978-953-7138-52-3 (meki uvez)
ISBN: 978-953-7138-53-0 (PDF)

Verzija priručnika D400-20141205

Sadržaj

Uvod

Tečaj Uvod u programski jezik Ruby namijenjen je svima koji žele svladati osnove programskog jezika Ruby. Za uspješno pohađanje tečaja poželjno je poznavanje osnova programiranja.

U ovom je priručniku programski kôd otisnut **`neproporcionalnim fontom`**. Ključne riječi programskog jezika Ruby i imena varijabli **podebljana** su, a engleski izrazi te nazivi klasa i metoda pisani su *kurzivom*.

Oznaka $ na početku retka označava da se kôd treba upisati unutar komandne ljuske operacijskog sustava Linux:

```
$ gem -v
```

Interaktivna ljuska interpretera programskog jezika *Ruby* ima prompt `>>`, a ispred rezultata ispisuje se oznaka `=>`:

```
>> 2 + 2
=> 4
```

Ukoliko se naredba proteže kroz više redova, interaktivna ljuska rabi prompt `?>` kako bi označila da upisana naredba nije potpuna. Redovi bez ikakvih oznaka označavaju ispis programa:

```
>> puts "a" *
?> 8
aaaaaaaa
=> nil
```

Napomene su pisane *kurzivom* i označene sličicom  .

1. Općenito o programskom jeziku Ruby

Ruby je interpretirani programski jezik, što znači da se izvorni kôd prevodi u kôd razumljiv računalu prilikom svakog izvršavanja programa. Očiti je nedostatak interpretiranih programskih jezika to da su sporiji nego kompajlirani programski jezici kao što je na primjer C. Prednost je njihova veća fleksibilnost i kraće vrijeme izrade programa.

Ruby je potpuno objektno orijentiran programski jezik, što znači da se programski problemi rješavaju definiranjem klasa, a iz klasa nastaju pojedinačni objekti.

Jedna je od prednosti objektno orijentiranog pristupa to da on olakšava rad na velikim projektima jer se zadatak moguće podijeliti na više programera, od kojih svaki može zasebno raditi na jednoj klasi ili na više njih. Na početku rada se definiraju poruke koje se mogu slati određenom objektu, a nakon toga programeri imaju slobodu pisati i mijenjati kôd unutar klase pod uvjetom da poruke ostaju nepromijenjene.

Objektno orijentirana metoda rješavanja problema vrlo je učinkovita jer raščlanjuje probleme na prirodan način sličan onome koji se upotrebljava u rješavanju problema u stvarnom svijetu.

Više informacija o programskom jeziku Ruby može se naći na: http://en.wikipedia.org/wiki/Ruby_(programming_language).

Ruby pripada otvorenom kodu budući da je objavljen pod licencijom GPL. Vremenom je objavljeno više alternativnih interpretera programskog jezika Ruby. Najpoznatiji je JRuby koji je napisan u programskom jeziku Java.

Alternativni interpreteri za Ruby mogu se naći na adresama: http://en.wikipedia.org/wiki/YARV, http://jruby.codehaus.org/ i http://rubini.us/.

Vrijedi spomenuti da je Ruby stekao popularnost putem programskog okvira (eng. *framework*) za razvoj *web*-aplikacija „Ruby on Rails“.

1.1. Prednosti i nedostaci programskog jezika Ruby

PRIRODNA SINTAKSA

Za Ruby se kaže da ima prirodnu sintaksu. Naime, sintaksa programskog jezika Ruby ne sadrži nepotrebne znakove i stoga kôd napisan u programskom jeziku Ruby ostavlja dojam jasnoće i jednostavnosti.

Više o prirodnoj sintaksi programskoj jezika Ruby može se naći na adresi: http://www.ruby-lang.org/en/about.

Sljedeća je usporedba je dobar primjer za prirodnu sintaksu programskog jezika Ruby. Pretpostavimo da želimo postati poruku `napuni_gorivo`

objektu `moj_automobil` ukoliko spremnik nije pun. Sljedeći je kôd napisan u PHP-u:

```
if( ! $moj_auto->spremnik_pun()) {
  $moj_auto->napuni_gorivo;
}
```

Uočite niz suvišnih znakova koje sadrži gore napisani kôd, poput znakova `(`, `!`, `$`, `->`, `)`, `{`, `;`, i `}`.

Istovrsni kôd napisan u programskom jeziku Ruby izgleda ovako:

```
moj_auto.napuni_gorivo unless
                  moj_auto.spremnik_pun?
```

RUBY – „PROGRAMEROV NAJBOLJI PRIJATELJ“

Ruby je zamišljen kao programski jezik koji olakšava rad programeru lakoćom korištenja, jednostavnošću i fleksibilno. Usmjerenost na programera očita je u svim dijelovima programskog jezika.

Najočitiji je primjer za to mogućnost pisanja koda na više različitih načina. Evo nekoliko primjera:

	Prva mogućnost	Druga mogućnost
Zagrade prilikom pozivanja metoda	`puts "Hello, world"`	`puts("Hello, world")`

Skraćeni **if**	`if 2 > 1`
	`  puts "2 je veće od 1"`
	`end`
		`puts "2 je veće od 1" if 2 > 1`

Početak i kraj bloka	`while true do`
	`puts "Stalno"`
	`end`
		`while true {`
		`puts "Stalno"`
		`}`

Nizovi	`["jedan", "dva", "tri"]`	`%w[ jedan dva tri ]`

Instalacija programskih proširenja i dodataka jednostavna je:

```
$ gem install dodatak
```

Pronalaženje proširenja i dodataka na Internetu, njihovo preuzimanje i instalacija odvijaju se potpuno automatski.

Interaktivna ljuska interpretera programskog jezika Ruby omogućuje trenutno isprobavanje koda, koji bismo u suprotnom morali prvo snimiti u neku datoteku, nakon toga izvršiti i onda izbrisati datoteku. Također je podržano bojanje koda (eng. *syntax highlighting*), pozivanje prethodne linije i slično.

TESTOVI

Ispravljanje grešaka u kodu (eng. *bug*) vrlo je težak posao, a osobito nakon što se završi razvoj i krajnjim korisnicima program preda na uporabu. Najbolje je otkriti što veći broj grešaka prilikom same izrade programa.

Ispravljanje grešaka u kodu sastoji se od dvije faze:

1. Pronalaženje točnog mjesta u kodu gdje je greška nastala.
2. Ispravljanje izvornoga koda.

Ispravljanje izvornog koda najčešće je trivijalno jednom kada smo našli mjesto gdje je greška nastala. Međutim, pronalaženje točnog mjesta u kodu gdje se nalazi greška puno je dugotrajniji i mukotrpniji proces.

Ruby omogućava pisanje testova koji testiraju izvorni kôd. To je nova programerska paradigma koja omogućava brži razvoj i samim tim manje troškove razvoja programa.

Pisanje i izvršavanje testova nije samo nova, egzotična mogućnost programskog jezika, već vrlo koristan dodatak svakom programu. Programeri koji pišu programe u programskom jeziku Ruby vrlo se često koriste testovima i oni su integralni dio svakog ozbiljnog proširenja ili programa.

1.2. Priprema radnog okruženja

Da bi se počelo pisati programe u programskom jeziku Ruby, potrebno je napraviti radno okruženje koje se sastoji od interpretera i dokumentacije. Potrebno je imati i editor u kojem ćemo pisati svoj kôd.

KOMPONENTE PROGRAMSKOG JEZIKA RUBY

Važno

Okruženje za razvoj programa u programskom jeziku Ruby sastoji se od:

- interpretera
- sustava upravljanja proširenjima
- interaktivne ljuske.

Okruženje za razvoj programa u programskom jeziku Ruby sastoji se od tri komponente:

- interpretera
- sustava upravljanja proširenjima (eng. *Ruby gems*)
- interaktivne ljuske (**irb**).

Da bismo se koristili jezikom Ruby, dovoljno je instalirati interpreter. Ostale dvije komponente nisu neophodne, ali su vrlo korisne za ozbiljniji rad.

Interpreter omogućava izvršavanje koda i glavni je dio radnog okruženja.

Sustav Ruby gems služi za upravljanje proširenjima jezika Ruby. Pokreće se naredbom **gem** i omogućava automatsko skidanje i instalaciju ekstenzija. Dovoljno je znati ime ekstenzije.

Interaktivna ljuska omogućava jednostavno i brzo izvršavanje koda bez potrebe da se taj kôd prvo upisuje u datoteku, a zatim izvršava na standardni način.

INSTALACIJA NA OPERACIJSKOM SUSTAVU LINUX

Standardna implementacija programskog jezika Ruby može se instalirati i izvršavati na različitim operacijskim sustavima (Linux, Windows, Solaris i Mac OS X). Ipak, Ruby se najčešće koristi na Linuxu, osobito ako se upotrebljava i popularan programski okvir za izradu *web*-aplikacija pod nazivom Ruby on Rails.

Prije instalacije dobro je provjeriti jesu li komponente jezika Ruby već instalirane. Na Linuxu (distribucija Fedora) to možemo provjeriti sljedećim naredbama:

```
$ ruby -v
$ gem -v
$ irb -v
```

Ruby se instalira na sljedeći način:

```
$ yum install ruby
$ yum install rubygems
$ yum install ruby-irb
```

Prva naredba instalira interpreter, a ostale dvije sustav za upravljanje ekstenzijama i interaktivnu ljusku.

Naredbe za instalaciju razlikuju se na različitim distribucijama Linuxa (u primjeru su navedene one koje se odnose na distribuciju Fedora i njoj slične). Instalacija na operacijske sustave Microsoft Windows vrši se pomoću programa RubyInstaller koje se može naći na adresi http://rubyinstaller.org/. Instalacijski paketi za ostale operacijske sustave mogu se naći na adresi http://www.ruby-lang.org/en/downloads/.

1.3. Pokretanje koda

Najjednostavniji je način pokretanja koda kreiranje datoteke s programskim kodom i pokretanje te datoteke naredbom **`ruby`**.

```
$ ruby moj_program.rb
```

Datoteke koje sadrže kôd napisan u programskom jeziku Ruby obično (po dogovoru) imaju ekstenziju **.rb**.

Ukoliko želimo isprobati samo nekoliko linija koda, možemo rabiti interaktivnu ljusku (eng. *interactive ruby shell*) koja se može pokrenuti naredbom:

```
$ irb
```

Također postoji mogućnost izvršavanja koda koji se izravno upisuje u komandnoj liniji zajedno s pozivom interpretera. U tom slučaju interpreter pozivamo koristeći se opcijom **`-e`**.

```
$ ruby -e "puts 'hello world'"
```

1.4. Dokumentacija

Priručnik (eng. *reference manual*) za programski jezik Ruby sadrži navedene i opisane sve glavne objekte i njihove metode koje se koriste tijekom programiranja. Dostupan je putem Interneta, na adresi http://www.ruby-doc.org/core/.

Dokumentacija za dodatne klase nalazi se na adresi http://www.ruby-doc.org/stdlib/.

Najpoznatija knjiga o programskom jeziku Ruby je „Programming Ruby“. Prva verzija ove knjige može se besplatno pročitati na adresi http://www.ruby-doc.org/docs/ProgrammingRuby/.

1.5. Vježba: Instalacija programskog jezika Ruby

1. Prijavite se na sustav koristeći se podacima koje ćete dobiti od predavača.
2. Instalirajte Ruby koristeći naredbu **yum**.
3. Instalirajte sustav upravljanja proširenjima Ruby gems.
4. Instalirajte interaktivnu ljusku.
5. Pokrenite interaktivnu ljuski i upišite `puts "Ovo je irb."`. Izađite iz interaktivne ljuske naredbom `exit`.
6. Kreirajte datoteku koja sadrži programsku liniju `puts "Ovo je datoteka."`. Pohranite je pod imenom *prvi_program.rb*. Izvršite je.
7. Koristeći se opcijom `-e` programa `ruby` izvršite kôd `puts "Ovo je komandna linija."`.
8. Nađite u *on-line* dokumentaciji opis metode **upcase** koja pripada klasi *String.*

1.6. Pitanja za ponavljanje

1. Koje su osnovne osobine programskog jezika Ruby?
2. Na koja tri načina možemo pokrenuti kôd napisan u programskom jeziku Ruby?
3. Koje su dva najbitnija dijela dokumentacije programskog jezika Ruby i gdje ih možemo naći?

U ovom je poglavlju obrađeno:

- osobine i prednosti programskog jezika Ruby
- instalacija i korištenje interpretera za programski jezik Ruby
- korištenje dokumentacije.

2. Osnove programskog jezika Ruby

2.1. Ispis

Tekst se ispisuje rabeći naredbu **puts**. Ona ispisuje tekst koji je zadan kao argument i automatski na kraj ispisa dodaje znak za novi red:

```
>> puts "Ispis"
Ispis
```

Važno

Tekst se ispisuje naredbama **puts** i **print**.

Ako ne želimo da se na kraj ispisa dodaje znak za novi red onda trebamo upotrijebiti naredbu **print**.

2.2. Struktura programskog jezika Ruby

U programskom jeziku Ruby svaka linija predstavlja jednu naredbu. Ukoliko se želi staviti više naredbi unutar jedne linije, mora ih se odvojiti točkom sa zarezom:

```
>> puts "prva naredba"; puts "druga naredba"
prva naredba
druga naredba
```

Važno

Svaka linija predstavlja jednu naredbu.

Komentari se umeću tako da se postavi oznaka # prije početka komentara.

Komentari u programskom jeziku Ruby umeću se tako da se postavi oznaka # prije početka komentara. Komentar se može staviti na početku linije ili nakon koda:

```
>> # ovo je komenar
>> puts "Hello, world" # Ovo je također komentar
Hello, world
```

2.3. Varijable

Varijabla je simboličko ime kojem je pridružena neka vrijednost.

Varijabla je svojevrsno skladište (dio memorije računala) u kojem se nalazi neki podatak.

```
>> moja_varijabla = 7
```

Tom je naredbom varijabli koja se zove **moja_varijabla** pridružena vrijednost 7. Ako se nakon toga u nekom izrazu upotrijebi ime **moja_varijabla**, interpreter će iz memorije računala uzeti vrijednost varijable **moja_varijabla** (koja u tom slučaju iznosi 7) i uključit će je u izraz na mjestu na kojem se spominje.

```
>> moja_varijabla = 7
=> 7
>> puts moja_varijabla * 2
14
```

Vrijednost varijable može se tijekom rada mijenjati:

```
>> moja_varijabla = moja_varijabla * 3
=> 21
```

Pokušaj uporabe varijable kojoj nije prethodno pridružena neka vrijednost rezultirat će porukom o greški:

```
>> nepoznata_varijabla * 2
NameError: undefined local variable or method
'nepoznata_varijabla' form main:Object
```

2.4. Tipovi podataka

Svaki podatak je određenog tipa.

Osnovni tipovi podataka u programskom jeziku Ruby jesu brojevi, nizovi znakova (eng. *string*) i polja (eng. *array*).

BROJEVI

Brojevi se dijele na cjelobrojne i decimalne (eng. *float*).

```
>> 2
=> 2
>> 2.1
=> 2.1
```

Točku rabimo pri unosu decimalnih brojeva. Cjelobrojni dio ne smije se ispustiti:

```
>> .1
SyntaxError: compile error
(irb):62: no .<digit> floating literal anymore;
put 0 before dot
(irb):62: syntax error, unexpected '.'
        from (irb):62
>> 0.1
=> 0.1
```

Negativni brojevi imaju minus ispred broja. Između minusa i broja može se, ali i ne mora, staviti razmak:

```
>> -1
=> -1
>> - 1
=> -1
```

Heksadecimalno zapisani brojevi počinju s "0x":

```
>> 0x10
=> 16
```

Nula ispred broja označava da je broja zapisan oktalno:

```
>> 010
=> 8
```

Notacija sa znakom "e" također se može rabiti:

```
>> 1e10
=> 10000000000.0
>> 1.3e5
=> 130000.0
```

Prilikom upisivanja velikih brojeva možemo se poslužiti znakom "_" kako bismo povećali preglednost koda. Taj znak ima isključivo vizualno značenje:

```
>> 1_000_000_000
=> 1000000000
```

Rezultati dijeljenja ovise o tipu brojeva koje dijelimo. Ako su i djeljenik i djelitelj cjelobrojni brojevi, rezultat će također biti cjelobrojan. Ako je barem jedan od njih decimalan, rezultat će biti decimalan broj:

```
>> 7 / 2
=> 3
>> 7 / 2.0
=> 3.5
>> 7.0 / 2
=> 3.5
```

NIZOVI ZNAKOVA

Nizovi znakova (eng. *string*) mogu se u programskom jeziku Ruby napisati na više načina. Najčešće se rabe dvostruki navodnici:

```
>> "ovo je string"
=> "ovo je string"
```

Dvostruki navodnici omogućavaju umetanje i izvršavanje izraza unutar samog niza znakova:

```
>> "Ja imam #{100 + 100} kuna"
=> "Ja imam 200 kuna"
```

Dvostruki navodnici omogućavaju i upisivanje posebnih znakova kao što je znak za novi red (`\n`):

```
>> puts "prvi red\ndrugi red"
prvi red
drugi red
```

Nizovi znakova mogu se napisati i rabeći jednostruke navodnike. U tom slučaju nije moguće u nizove znakova umetati i izvršavati izraze te upisivati posebne znakove:

```
>> puts 'Ja imam #{100 + 100} kuna\ndrugi red'
Ja imam #{100 + 100} kuna\ndrugi red
```

NIZOVI

Niz je skup podataka poredanih određenim redoslijedom. Nizovi se zapisuju uglatim zagradama unutar kojih se nalaze poredani pojedini članovi niza, međusobno odvojeni zarezom:

```
>> c = [ "prvi", "drugi", 100 ]
=> ["prvi", "drugi", 100]
```

Svaki član niza može biti proizvoljnog tipa.

Nizovi mogu biti prazni:

```
>> a = []
=> []
```

Važno

Indeksi u nizovima počinju nulom.

Niz se indeksira isključivo cijelim brojevima koji počinju nulom. Vrijednosti dodajemo specificirajući indeks u uglatim zagradama:

```
>> a[0] = "Prva vrijednost"
=> "Prva vrijednost"

>> a[1] =  "Druga vrijednost"
=> "Druga vrijednost"
```

Nakon što smo definirali prva dva člana niza, ispisat ćemo sadržaj niza:

```
>> a
=> ["Prva vrijednost", "Druga vrijednost"]
```

NEPOSTOJEĆA ILI NEDEFINIRANA VRIJEDNOST - NIL

Prilikom definiranja niza u zadnjem primjeru rabili smo indekse nula i jedan. Sljedeći je indeks koji bismo trebali upotrijebiti indeks dva, ali taj ćemo indeks u sljedećem primjeru preskočiti i postaviti vrijednost na indeks tri:

```
>> a[3] = "Vrijednost na indeksu tri"
=> "Vrijednost na indeksu tri"
```

Provjerit ćemo što se nalazi na trećem mjestu u nizu (na indeksu dva):

```
>> a
=> ["Prva vrijednost", "Druga vrijednost", nil,
"Vrijednost na indeksu tri"]
```

Vidimo da se na indeksu dva pojavila vrijednost **nil**. U programskom jeziku Ruby **nil** predstavlja nepostojeću ili nedefiniranu vrijednost.

HASH-TABLICE (POLJA)

Tip podataka pod nazivom polje (eng. *hash*) sličan je nizu, ali za ključeve možemo rabiti bilo koju vrijednost. Niz ima svoj redoslijed, pa ako preskočimo indeks, stvaraju se vrijednosti **nil**. Kod polja to nije slučaj budući da indeks može biti bilo koji tip podataka pa ne može biti riječi o očekivanom redoslijedu indeksa.

Polje se definira upotrebom vitičastih zagrada:

```
>> a = {}
=> {}
```

Vrijednosti unutar polja dodaju se na isti način kao i kod nizova:

```
>> a["kljuc"] = "Vrijednost"
=> "Vrijednost"

>> a
=> {"kljuc"=>"Vrijednost"}
```

Polje možemo definirati i tako da mu odmah pridijelimo vrijednost:

```
>> boja_kose = {
>>   'ivan' => 'plava',
>>   'marija' => 'crna',
>>   'suzana' => 'plava',
>>   'petar' => 'crvena'
>>   }
```

SIMBOLI

Simbol najčešće predstavlja neki objekt iz realnog svijeta. Sličan je nizu znakova, ali zauzima manje memorije. Naziv simbola počinje dvotočkom i zatim se nastavlja kao niz znakova i brojeva. Hash-tablicu iz prethodnog primjera možemo definirati tako da ključevi budu simboli:

```
>> boja_kose= {
>>   :ivan => 'plava',
>>   :marija => 'crna',
>>   :suzana => 'plava',
>>   :petar => 'crvena'
>>   }
```

ISTINA I NEISTINA

Istina i neistina (eng. *boolean*) tip je podataka koji može sadržavati samo dvije vrijednosti, istina ili neistina:

```
>> a = true
=> true
>> a
=> true
>> b = false
=> false
```

2.5. Operatori

Operatori u jeziku Ruby dani su u tabeli:

Operatori (od višeg prioriteta prema nižem)	
Operator	Opis
`[ ] [ ]=`	referenciranje elemenata
`**`	potenciranje
`! ~ + -`	*not*, *complement*, unarni plus i minus (imena metoda za zadnja dva jesu `+@` i `-@`)
`* / %`	množenje, dijeljenje i modul
`+ -`	plus i minus
`>> <<`	lijevi i desni *shift*
`&`	"i" u radu s bitovima
`^ |`	ekskluzivni "ili" i obični "ili" u radu s bitovima
`<= < > >=`	usporedba
`<=> == === != =~ !~`	jednakost i operatori za rad s regularnim izrazima (`!=` and `!~` ne mogu se definirati kao metode)
`&&`	logički "i"
`||`	logički "ili"
`.. ...`	*range* (*inclusive* i *exclusive*)
`? :`	ternarni ako-onda-inače
`= %= { /= -= += |= &= >>= <<= *= &&= ||= **=`	pridruživanje
`defined?`	provjera je li simbol definiran
`not`	logička negacija
`or and`	logička kompozicija
`if unless while until`	modifikatori izraza
`begin/end`	blok

Mnogi operatori samo skraćuju pisanje koda i temelje se na dva ili više osnovna operatora. Na primjer:

```
>> a = 0
```

Pretpostavimo da želimo povećati vrijednost varijable **a** za jedan. Standardni je način da se to učini:

```
>> a = a + 1
```

Isto to moguće je učiniti puno kraće:

```
>> a += 1
```

Proučimo sljedeći primjer:

```
>> a = [3]
=> [3]
>> b = a[0] + 7 * 3
=> 24
```

Prvo smo definirali niz s jednim elementom. Nakon toga smo izvršili izraz koji sadrži više operatora različitih prioriteta.

Sljedeća tablica prikazuje kako Ruby rabi prioritete prilikom izračunavanja vrijednosti izraza:

Izraz	Opis
`b = a[0] + 7 * 3`	Početni izraz, najviši prioritet ima referenciranje elemenata (navedeno u prvom redu tablice operatora).
`b = 3 + 7 * 3`	Sljedeće po prioritetu je množenje.
`b = 3 + 21`	Sljedeće po prioritetu je zbrajanje.
`b = 24`	Na kraju se izvršava operator dodjeljivanja.

Ukoliko želimo da se operatori izvrše drugim redoslijedom, trebamo se koristiti zagradama:

```
>> (2 + 3) *4
=> 20
```

2.6. Kontrolne strukture i petlje

Kontrole strukture predstavljaju raskrižje u kodu koje omogućava da se izvršavanje koda nastavi u različitim smjerovima. Ruby podržava sve osnovne kontrolne strukture koje se nalaze i u drugim programskim jezicima, ali također sadrži još neke mogućnosti koji olakšavaju programiranje.

KONTROLNA STRUKTURA "AKO"

Kontrolna struktura **if** može se rabiti na više načina.

Osnovni način:

```
>> if (2 > 1) then
>>  puts "istina"
>> end
istina
=> nil
```

Svaka kontrolna struktura mora završiti izrazom **end**. Zagrade i ključna riječ **then** nisu obvezni, a možemo sve napisati u jednom redu odvajajući izraze točkom sa zarezom:

```
>> if 2 > 1 ; puts "istina"; end
istina
=> nil
```

Ako imamo samo jednu naredbu koju želimo izvršiti u slučaju da je test istinit, možemo rabiti sljedeći način pisanja:

```
>> puts "istina" if 2 > 1
istina
=> nil
```

Kontrolnoj strukturi "ako" možemo dodati ključnu riječ **else** kao i u mnogim drugim programskim jezicima:

```
>> if 5 > 10
>>   puts "istina"
>> else
>>   puts "nije istina"
>> end
nije istina
=> nil
```

Ruby podržava i ključnu riječ **unless** koja ima suprotan učinak od ključne riječi **if** i namijenjena je poboljšavanju izgleda i čitljivosti koda:

```
>> puts "nije istina" unless 2 < 1
nije istina
=> nil
```

ISTINA I NEISTINA U KONTROLNIM STRUKTURAMA I PETLJAMA

Sve kontrolne strukture rabe kontrolni izraz prilikom odlučivanja. Kontrolni izraz najčešće ima za rezultat vrijednost **true** ili **false**.

Međutim, kontrolni izraz može imati bilo koji rezultat pa je stoga važno znati kako Ruby odlučuje što je istina, a što neistina. Neistiniti su jedino izrazi koji imaju rezultat **false** ili **nil**.

Na primjer:

```
>> puts "istina" if 0
istina
=> nil
>> puts "istina" if 100
istina
=> nil
```

Bilo koji broj predstavlja istinu, pa čak i nula.

Proučimo sljedeći primjer rabeći nizove znakova:

```
>> puts "istina" if "string"
(irb):1: warning: string literal in condition
istina
=> nil
>> puts "istina" if ""
(irb):2: warning: string literal in condition
istina
=> nil
```

U gornjem primjeru interpreter ispisuje upozorenje, ali kako bilo koji niz znakova predstavlja istinu, pa čak i prazan niz, Ruby je u oba slučaja ispisao riječ „istina“.

Isto vrijedi za nizove i polja:

```
>> puts "istina" if []
istina
=> nil
>> puts "istina" if {}
istina
=> nil
```

Dakle, samo se **false** i **nil** interpretiraju kao neistina:

```
>> puts "istina" if false
=> nil
>> puts "istina" if nil
=> nil
```

Na ta pravila posebno trebaju pripaziti programeri koji su rabili druge programske jezike u kojima su ta pravila drugačija. Tako na primjer u PHP-u nula označava neistinu dok u programskom jeziku Ruby označava istinu.

Važno

Samo se **false** i **nil** interpretiraju kao neistina.

Bilo koja druga vrijednost predstavlja istinu, pa čak i nula.

KONTROLNA STRUKTURA "DOK"

Kontrolne strukture **while** i **until** rabe se slično kao u ostalim programskim jezicima:

```
>> a = 0
=> 0
>> while a < 5
>>   puts a
>>   a = a + 1
>> end
0
1
2
3
4
=> nil
```

Kontrolna struktura **until** radi obratno od kontrolne strukture **while**:

```
>> a = 0
=> 0
>> until a > 5
>>   puts a
>>   a = a + 1
>> end
0
1
2
3
4
5
=> nil
```

2.7. Sistemske naredbe

Ruby omogućava izvršavanje sistemskih naredbi u ljusci operativnog sustava na sljedeći način:

```
>> system "echo Ovo je sistemska naredba unutar
ljuske"
Ovo je sistemska naredba unutar ljuske
=> true
```

Izlaz naredbe ispisuje se na ekran. Ako je naredba uspješno izvršena, tj. ako je povratni kôd jednak nuli, vraća se vrijednost **true**. U svim ostalim slučajevima vraća se vrijednost **false**.

Ako želimo pohraniti izlaz od sistemske naredbe u neku varijablu, to možemo učiniti na sljedeći način:

```
>> a = `echo Ovo je naredba unutar ljuske`
=> "Ovo je naredba unutar ljuske\n"
>> puts a
Ovo je naredba unutar ljuske
=> nil
```

Uspješnost izvršenja naredbe (izlazni kôd) možemo povjeriti na sljedeći način:

```
>> $?.exitstatus
=> 0
```

Nula znači da je naredba uspješno izvršena, a svi ostali kodovi označavaju pogrešku.

2.8. Konstante

Imena konstanti u programskom jeziku Ruby sastoje se isključivo od velikih slova. Ako pokušamo ponovno dodijeliti vrijednost već definiranoj konstanti, doći će do upozorenja:

```
>> MOJA_KONSTANTA = 1
=> 1
>> MOJA_KONSTANTA = 2
(irb):9: warning: already initialized constant
MOJA_KONSTANTA
=> 2
```

2.9. Vježba: Tipovi podataka, operatori, kontrolne strukture, ulaz/izlaz, sistemske komande

1. Napišite kod koji će podijeliti 15 sa 7 tako da rezultat bude decimalni broj.
2. Zapišite decimalni broj 10 u heksadecimalnom obliku.
3. Zapišite broj milijarda na najkraći način.
4. Zapišite niz znakova koji će u sebi imati izračun umnoška brojeva 456 i 32.
5. Definirajte varijablu x koja je niz koji se sastoji od niza znakova "Ruby" i od broja 2. Nakon toga na kraj tog niza dodajte broj 100.
6. Izračunajte treću potenciju broja 4 i provjerite je li rezultat veći od 50.
7. Napišite kontrolnu strukturu koja će ispisati niz znakova "istina" ako je vrijednost varijable **b** djeljiva sa 7 i manja od 100.
8. Ispišite niz znakova "Vjezba" bez da se na kraju ispisa ispiše znak za novi red.
9. Ispišite datum iz pozivajući odgovarajuću naredbu operacijskog sustava.

2.10. Pitanja za ponavljanje

1. Kako ispisati tekst u programskom jeziku Ruby?
2. Što je to varijabla?
3. Što predstavlja 10e6?
4. Kako umetnuti neku varijablu ili izraz unutar niza znakova?
5. Kako se u programskom jeziku Ruby označava nepostojeća vrijednost?
6. Po čemu se razlikuju polje i niz?
7. Koja je razlika između operatora = i ==?
8. Kako se u programskom jeziku Ruby može izvršiti sistemska naredba operativnog sustava?

U ovom je poglavlju obrađeno:

- varijable
- tipovi podataka
- operatori
- kontrolne strukture i petlje
- sistemske komande.

3. Izrazi

3.1. Vrijednost izraza

Definirajmo varijablu **c**:

```
>> c = 0
=> 0
```

Ovaj izraz postavlja nulu kao vrijednost varijable **c**, ali nula je također vraćena kao rezultat cijelog izraza. Oznaka "=>" unutar interaktivne ljuske označava vrijednost izraza.

Pretpostavimo da želimo postaviti nulu u varijable **a**, **b** i **c**. Standardni način bi bio:

```
>> a = 0
=> 0
>> b = 0
=> 0
>> c = 0
=> 0
```

Međutim, to možemo napraviti i jednostavnije u samo jednom retku:

```
>> a = b = c = 0
=> 0
```

3.2. Pridruživanje vrijednosti varijablama i usporedba

Pretpostavimo da želimo pohraniti u varijablu **a** zbroj brojeva pet i sedam, a ako je ta vrijednost veća od 9, onda ispisati tekst "Dvoznamenkasti broj". Standardni način je:

```
a = 5 + 7
if a > 9
  puts "Dvoznamenkasti broj"
end
```

Međutim, imajući na umu da svaki izraz ima svoju vrijednost te da postoji skraćeni oblik usporedbe **ako**, te četiri linije koda možemo pretvoriti u jednu:

```
puts "Dvoznamenkasti broj" if a = 5 + 7 > 9
```

3.3. Višestruko pridruživanje vrijednosti varijablama

Ponekad želimo postaviti više različitih vrijednosti u više različitih varijabli. To je moguće napraviti na sljedeći način:

```
>> a, b, c = 1, 2, 3
=> [1, 2, 3]
```

Ukoliko na desnoj strani znaka za dodjeljivanje postoji više vrijednosti koje su odvojene zarezima, iz njih se stvara niz. Nakon toga članovi niza pridružuju se redom varijablama na lijevoj strani.

Proučimo što se događa ako na desnoj strani izraza ima više vrijednosti nego što ima varijabli na desnoj strani:

```
>> a,b = 1,2,3
=> [1, 2, 3]
>> a
=> 1
>> b
=> 2
```

U ovom slučaju broj tri nije pridružen ni jednoj varijabli.

Proučimo što se događa u suprotnom slučaju – ako je lijevoj strani navedeno više varijabli nego što ima vrijednosti na desnoj strani:

```
>> c = 100
=> 100
>>  a, b, c = 1, 2
=> [1, 2]
>> c
=> nil
```

Budući da nije bilo dovoljno vrijednosti na desnoj strani, varijabli **c** pridruženo je vrijednost **nil**. Prethodna vrijednost varijable **c** (broj 100) nije sačuvana.

3.4. Zamjena sadržaja varijabli

Uobičajen način za zamjenu vrijednosti dviju varijabli je pomoću treće, privremene varijable:

```
privremena = a
a = b
b = privremena
```

U programskom jeziku Ruby to je moguće napraviti i na kraći način, rabeći višestruko pridruživanje:

```
a, b = b, a
```

3.5. Vježba: Izrazi

1. Pridružite varijablama **a**, **b**, **d** i **f** vrijednost 100 koristeći se samo jednom naredbom.
2. Rabeći samo jednu naredbu pridružite vrijednost 3 varijabli **a**, dodajte tom izrazu 5 i rezultat pohranite u varijablu **b**.
3. U varijablu **a** pohranite vrijednost 2 a u varijablu **b** pohranite niz znakova "pero" koristeći se samo jednom naredbom.
4. Pohranite vrijednost varijable **b** u varijablu **a**, vrijednost varijable **c** u varijablu **b** i vrijednost varijable **a** u varijablu **c** rabeći samo jednu naredbu.

3.6. Pitanja za ponavljanje

1. Što će ispisati naredba puts **a** = "string"?
2. Koja će biti vrijednost varijable **c** nakon naredbe **a,b,c = 1,2**?
3. Kako se može zamijeniti sadržaj varijabli samo jednom naredbom?

U ovom je poglavlju obrađeno:

- vrijednost izraza
- dodjeljivanje i usporedba
- višestruko dodjeljivanje
- zamjena sadržaja varijabli.

4. Klase i objekti

4.1. Klase

Klase su zaokružene cjeline koje se sastoje od koda koji je potreban za izvršavanje određenog zadatka. Definiranje klasa zapravo predstavlja definiranje koda.

Važno

Imena klasa u programskom jeziku Ruby počinju velikim slovom.

Klasu kreiramo rabeći ključnu riječ **class** i nakon toga navođenjem ime klase. Ime klase mora početi velikim slovom.

Nakon što smo naveli ime klase, definiramo sve metode koje klasa sadrži i na kraju dodamo ključnu riječ **end**.

Na primjer:

```
>> class Automobil
>> end
=> nil
```

Gore navedena klasa je praktično beskorisna jer ne sadrži nikakav kôd. Na sreću, programski jezik Ruby omogućava naknadno proširivanje klasa, što je svojstvo dinamičkih programskih jezika.

4.2. Metode

Metoda je dio koda koji izvršava određeni zadatak unutar klase. Metode su sastavni dio svake klase. Definiranje klase u stvari znači definiranje metoda.

Kreirat ćemo klasu *Automobil* koja se sastoji od metode *svjetla* koja ispisuje "Uključujem svjetla".

```
>> class Automobil
>>   def svjetla
>>     puts "Uključujem svjetla"
>>   end
>> end
```

Važno

Uobičajeno je da imena metoda koje nešto provjeravaju i vraćaju logičku vrijednost završavaju upitnikom.

Imena metoda koje mijenjaju sadržaj varijabli nad kojima su pozvane obično završavaju uskličnikom.

Postoje određena pravila pri imenovanju metoda. Imena metoda koje nešto provjeravaju i vraćaju logičku vrijednost najčešće završavaju upitnikom.

```
>> niz = ["a","b","c"]
=> ["a", "b", "c"]
>> niz.include?("a")
=> true
```

U ovom primjeru metoda **include?** provjerava je li neki objekt član niza ili nije.

Imena metoda koje mijenjaju vrijednost varijabli nad kojima su pozvane najčešće završavaju uskličnikom:

```
>> a = "Srce"
=> "Srce"
>> a.upcase
=> "SRCE"
>> a
=> "Srce"
>> a.upcase!
=> "SRCE"
>> a
=> "SRCE"
```

Metoda **upcase!** promijenila je sadržaj varijable **a** (za razliku od metode **upcase**).

Svaka metoda na mjesto poziva vraća neku vrijednost. Ta se vrijednost u metodi može specificirati ključnom riječi **return**, a ukoliko to ne učinimo, povratna vrijednost bit će rezultat zadnjeg izraza:

```
>> class Automobil
>>   def uvijek_istina
>>      return true
>>   end
>> end
```

U ovom primjeru definirali smo metodu *uvijek_istina* koja vraća vrijednost **true**. Koristili smo ključnu riječ **return**. Isti rezultat mogli smo postići na sljedeći način:

```
>> class Automobil
>>   def uvijek_istina
>>      true
>>   end
>> end
```

U programskom jeziku Ruby moguće je definirati metode bez navođenja ime klase kojoj metoda pripada. Sljedeći primjer pokazuje takav slučaj:

```
>> def moje_vrijeme
>> puts "Sada je #{Time.now}"
>> end
=> nil
```

Sada se tom metodom možemo služiti bilo gdje drugdje u programu:

```
>> moje_vrijeme
Sada je Sun Oct 26 13:49:33 -0500 2008
```

4.3. Objekti

Objekt je konkretna implementacija ili realizacija neke klase.

Definiranjem klasa definiramo metode i kôd unutar tih metoda. Nakon toga iz klasa kreiramo objekte i preko objekata pokrećemo kôd koji smo definirali u klasama.

Na primjer, klasa *Automobil* definira što automobil sve može učiniti, a iz te klase kreiramo konkretne objekte. Na primjer:

```
moj_auto = Automobil.new
```

Pozivanje metoda naziva se slanje poruke objektu. Poruka se sastoji od imena metode koju pozivamo i argumenata. Kada imamo konkretan objekt njime možemo upravljati pomoću poruka, na primjer:

```
moj_auto.vozi 60
```

Klase su osnova za svaki objekt. One definiraju predložak za objekte koji će iz njih nastati. To uključuje metode koje sadržavaju kôd za upravljanje tim objektom te interne varijable koje označavaju stanje objekta.

4.4. Atributi i varijable

Važno

Imena varijabli instancija započinju znakom @.

Svaki objekt koji kreiramo iz određene klase obično sadrži varijable koje odražavaju njegovo stanje. Takve varijable se zovu varijable instancije (eng. *instance variables*) i njihovo ime počinje znakom @.

Dakle, iako možemo napraviti više objekata iz klase *Automobil*, svaki od tih objekata će imati svoje vlastite varijable.

Pretpostavimo da želimo zabilježiti neke karakteristike automobila kao što su proizvođač, model, godina proizvodnje, snaga motora, boja ili broj prijeđenih kilometara. Sve te karakteristike određenog konkretnog automobila pohranit ćemo u varijable instancije.

Varijablama instancije ne možemo pristupiti izravno već putem metoda.

Na primjer:

```
>> class Automobil
>>  def postavi_boju(x)
>>    @boja = x
>>  end
>>  def ocitaj_boju
>>     @boja
>>  end
>> end
```

Našoj klasi *Automobil* dodali smo dvije nove metode, *postavi_boju* i *ocitaj_boju*. One nam omogućavaju da postavimo i očitamo vrijednost varijable instancije.

Programski jezik Ruby omogućava još jednostavnije definiranje takvih metoda. Metodu *ocitaj_boju* možemo jednostavno nazvati *boja*, a metodu

spremi_boju možemo nazvati *boja*=. Tada će definicija metoda izgledati ovako:

```
>> class Automobil
>>   def boja
>>     @boja
>>   end
>>   def boja= nova_boja
>>     @boja = nova_boja
>>   end
>> end
```

U gornjem primjeru klasi *Automobil* dodali smo dvije nove metode, *boja* i *boja*=. One nam omogućavaju da očitamo i postavimo vrijednost varijable instancije.

```
>> a = Automobil.new
=> #<Automobil:0xb7eaced4>
>> a.boja
=> nil
>> a.boja= "bijela"
=> "bijela"
>> a.boja
=> "bijela"
```

Uobičajeno je da imena metoda koje postavljaju vrijednost završavaju znakom jednakosti.

Dozvoljeno je staviti razmak ispred znaka **=**. Na primjer:

```
>> a.boja = "crvena"
=> "crvena"
>> a.boja
=> "crvena"
```

Iako smo napisali `boja =` (s razmakom), interpreter je prepoznao da je riječ metodi *boja*=.

Pogledajmo ponovno definiciju klase *Automobil*:

```
>> class Automobil
>>   def boja
>>     @boja
>>   end
>>   def boja= nova_boja
>>     @boja = nova_boja
>>   end
>> end
```

Kada bismo način na koji smo radi pristupa varijabli instancije *boja* definirali metode *boja* i *boja*= htjeli primijeniti na druge varijable instancije (kojima želimo zabilježiti proizvođača, model, godinu proizvodnje, snagu motora i broj prijeđenih kilometara), trebali bismo napisati dodatnih tridesetak linija koda.

Umjesto toga možemo koristi pristupnike atributima (eng. *attribute accessor*). Na primjer:

```
class Automobil
attr_accessor :proizvodjac, :model,
:godina_proizvodnje, :snaga_motora,
:prijedjeni_kilometri
end
```

Ključna riječ **attr_accessor** klasi dodaje metode koje omogućavaju čitanje i pisanje u imenovane varijable. Ispred imena varijabli navodimo znak **:** (na primjer: `:model`).

Sada možemo upotrijebiti pristupnike i za druge varijable:

```
>> a.proizvodjac = "Zastava"
=> "Zastava"
>> a.model = "Fićo"
=> "Fićo"
>> a.godina_proizvodnje = 1971
=> 1971
>> a.snaga_motora = 10
=> 10
>> a.prijedjeni_kilometri = 0
=> 0
>> a
=> #<Automobil:0xb7eaced4 @snaga_motora=10,
@godina_proizvodnje=1971, @model="Fićo",
@prijedjeni_kilometri=0, @boja="crvena",
@proizvodjac="Zastava">
```

4.5. Inicijalizacija objekata

Važno

Konstruktori su metode kojima se kreiraju novi objekti određene klase.

Metode koje služe za kreiranje objekata određene klase metode zovu se konstruktori.

Najčešće se koristi konstruktor **new** koji kreira novi objekt određene klase:

```
moj_auto = Automobil.new
```

Konstruktori se također mogu pozivati navodeći argumente.

Konstruktori nam omogućavaju i da postavimo početne vrijednosti na svakom novom objektu koji kreiramo.

Na primjer, prilikom kreiranja novog objekta klase *Automobil* možemo postaviti početne vrijednosti odgovarajućih varijabli instancije tako da pokazuju da je motor ugašen, automobil zaključan ili broj prijeđenih kilometara nula. To činimo definiranjem metode **initialize**:

```
>> class Automobil
>>   def initialize
>>     @motor = false
>>     @zakljucan = true
>>     @broj_kilometara = 0
>>   end
>> attr_accessor :motor, :zakljucan,
:broj_kilometara
>> end
=> nil
```

Iako se metoda zove **initialize**, novi objekt kreira se ključnom riječi **new**:

```
>> a = Automobil.new
=> #<Automobil:0xb7e057ec @zakljucan=true,
@motor=false, @broj_kilometara=0>
>> a.motor
=> false
```

Metoda **initialize** može primati argumente kao i sve ostale metode.

4.6. Sve je objekt

U programskom jeziku Ruby sve je objekt, uključujući i osnovne tipove podataka koji su spomenuti u drugom poglavlju.

Služeći se metodom **class,** možemo doznati kojoj klasi pripada neki objekt.

Decimalni brojevi pripadaju klasi *Float*:

```
>> 2.1.class
=> Float
```

Broj dva je objekt klase *Fixnum*:

```
>> 2.class
=> Fixnum
```

Koje sve metode možemo rabiti s nekom klasom možemo doznati na sljedeći način:

```
>> 2.methods.sort
=> ["%", "&", "*", "**", "+", "+@", "-", "-@",
"/", "<", "<<", "<=", "<=>", "==", "===", "=~",
">", ">=", ">>", "[]", "^", "__id__", "__send__",
"abs", "between?", "ceil", "chr", "class",
"clone", "coerce", "display", "div", "divmod",
"downto", "dup", "eql?", "equal?", "extend",
"floor", "freeze", "frozen?", "gem", "hash",
"id", "id2name", "inspect", "instance_eval",
"instance_of?", "instance_variable_defined?",
"instance_variable_get", "instance_variable_set",
```

```
"instance_variables", "integer?", "is_a?",
"kind_of?", "method", "methods", "modulo",
"next", "nil?", "nonzero?", "object_id", "po",
"poc", "prec", "prec_f", "prec_i",
"pretty_inspect", "pretty_print",
"pretty_print_cycle", "pretty_print_inspect",
"pretty_print_instance_variables",
"private_methods", "protected_methods",
"public_methods", "quo", "remainder", "require",
"respond_to?", "ri", "round", "send",
"singleton_method_added", "singleton_methods",
"size", "step", "succ", "taint", "tainted?",
"times", "to_a", "to_f", "to_i", "to_int",
"to_s", "to_sym", "truncate", "type", "untaint",
"upto", "zero?", "|", "~"]
```

Isprobajmo neke od tih metoda:

```
>> 2.zero?
=> false
>> 2.nonzero?
=> 2
>> 0.nonzero?
=> nil
>> 0.zero?
=> true
```

Ranije smo pokazali da je svaki broj zapravo objekt određenog tipa. Isto vrijedi i za nizove znakova. Svaki je niz znakova u stvari objekt klase *String* na koji možemo pozivati odgovarajuće metode:

```
>> "Ovo je string".class
=> String
>> "Ovo je string".size
=> 13
>> "Ovo je string".reverse
=> "gnirts ej ovO"
"Ovo je string" * 10
=> "Ovo je stringOvo je stringOvo je stringOvo je
 stringOvo je stringOvo je stringOvo je stringOvo
 je stringOvo je stringOvo je string"
>> "Ovo je string".upcase
=> "OVO JE STRING"
```

Čak je i **nil** objekt:

```
>> nil.class
=> NilClass
```

```
>> nil.methods
=> ["inspect", "&", "clone", "method",
"public_methods", "instance_variable_defined?",
"equal?", "freeze", "to_i", "gem", "methods",
"respond_to?", "dup", "moj_datum",
"instance_variables", "__id__", "|", "eql?",
"object_id", "require", "id", "to_f",
"singleton_methods", "send",
"pretty_print_cycle", "taint", "moje_vrijeme",
"frozen?", "pretty_print_inspect",
"instance_variable_get", "^", "__send__",
"instance_of?", "to_a", "type",
"protected_methods", "instance_eval", "==",
"display", "pretty_inspect", "===",
"pretty_print", "instance_variable_set",
"kind_of?", "extend", "to_s", "hash", "class",
"private_methods", "=~", "tainted?",
"pretty_print_instance_variables", "untaint",
"nil?", "is_a?"]
```

4.7. Vježba: Klase, metode, objekti, atributi i varijable

1. Kreirajte klasu *Automobil.*
2. Kreirajte klasu *Automobil* koja sadrži metodu *ukljuci_motor* koja ispisuje "Uključujem motor".
3. Kreirajte novi objekt klase *Automobil* i spremite ga u varijablu **a**.
4. Pokrenite metodu *ukljuci_motor* u novom objektu koji ste upravo kreirali.
5. Provjerite i zabilježite kojoj klasi pripadaju sljedeći objekti:

   "Niz znakova"
   102
   102.233
   10**100
   []
   {}

4.8. Pitanja za ponavljanje:

1. Što je klasa?
2. Je li moguće mijenjati klasu tijekom izvođenja programa?
3. Što je metoda?
4. Koji je najmanji broj metoda koji klasa može imati?
5. Što je objekt?
6. Što je varijabla instancije?

7. Kako se u programskom jeziku Ruby označava varijabla instancije?
8. Što je pristupnik atributu?

U ovom je poglavlju obrađeno:

- klase
- metode
- objekti
- atributi i varijable
- inicijalizacija objekata.

5. Rad s datotekama

5.1. Otvaranje datoteka

Datoteke se u programskom jeziku Ruby otvaraju na sljedeći način:

```
>> f = File.new("/etc/passwd","r")
=> #<File:/etc/passwd>
>> f.readline
=> "root:x:0:0:root:/root:/bin/bash\n"
>> f.close
=> nil
```

Prvom naredbom otvorena je datoteka */etc/passwd* u modu za čitanje i pridružena varijabli **f**. Nakon toga pročitan je prvi redak iz datoteke. Na kraju je datoteka zatvorena.

5.2. Čitanje i pisanje

Metoda **readline** čita jedan redak iz otvorene datoteke. Metoda **getc** čita sljedećih 8 bitova (jedan bajt) i pretvara ih u broj. Metode **lineno** i **pos** služe za određivanje mjesta na kojem se nalazimo u datoteci.

Na primjer:

```
>> file = File.new("/etc/passwd","r")
=> #<File: /etc/passwd>
>> puts file.lineno
0
=> nil
>> puts file.pos
0
=> nil
>> puts file.readline
root:x:0:0:root:/root:/bin/bash
=> nil
>> puts file.pos
32
=> nil
>> puts file.lineno
1
=> nil
>> puts file.getc
98
=> nil
>> file.close
file.close
```

U gornjem primjeru prvo smo otvorili datoteku */etc/passwd* i pridružili je varijabli **file**. Pozvali smo metodu **lineno** i ispisali dobiveni rezultat. Rezutat je bio nula što je značilo da se nalazimo u prvom retku datoteke. Nakon toga smo ispisali rezultat poziva metode **pos**. Rezultat je također bio nula.

Važno

Metoda **readline** čita sljedeći redak iz otvorene datoteke. Metoda **getc** čita sljedećih 8 bitova i pretvara ih u broj.

Metode **lineno** i **pos** služe za određivanje mjesta na kojem se nalazimo u datoteci.

Zatim smo pročitali prvi redak iz datoteke i ispisali ga na ekran. Nakon toga smo ponovno pozvali metode **lineno** i **pos**. Ispis je pokazao da se nalazimo na 32. znaku i prvom retku datoteke. Zatim smo pročitali sljedeći bajt koji je pretvoren u broj i ispisali ga. Na kraju smo zatvorili datoteku.

Važno

Metoda **puts** na kraj upisa dodaje znak za novi red, a metoda **print** ne.

U datoteku možemo pisati koristeći se metodama **puts** i **print**. Metoda **puts** na kraj upisa dodaje znak za novi red, a metoda **print** ne.

Na primjer:

```
>> file = File.new("/tmp/moja_datoteka", "w")
=> #<File:/tmp/moja_datoteka>
>> file.puts "Prvi redak"
=> nil
```

Datoteku možemo odjednom učitati kao niz znakova bez da je posebno otvaramo. U tu svrhu služi metoda **read**:

```
a = File.read("/etc/inittab")
```

Klasa *File* sadrži još mnoge druge metode koje se mogu izvršavati bez otvaranja datoteke. Na primjer, možemo doznati ime direktorija u kojem se nalazi datoteka, ime datoteke ili njezinu veličinu:

```
>> File.dirname("/tmp/moja_datoteka")
=> "/tmp"
>> File.basename("/tmp/moja_datoteka")
=> "moja_datoteka"
>> File.size("/etc/passwd")
=> 2057
```

5.3. Rad s direktorijima

Programski jezik Ruby sadrži i mnoge druge mogućnosti u radu s direktorijima i datotekama. Zato ga neki upotebljavaju kao zamjenu za programiranje u ljusci operativnog sustava (eng. *bash shell*).

Ukoliko želimo vidjeti u kojem direktoriju se nalazimo, koristit ćemo sljedeću naredbu:

```
>> Dir.pwd
=> "/root"
```

Ukoliko želimo napraviti niz od popisa datoteka u trenutnom direktoriju, koristit ćemo se sljedećom naredbom:

```
>> Dir["*"]
=> ["mbox", "changed_etc", "Mail",
"mailman.news.members", "all_databases.sql",
"weekly_backup", "bin", "mailman.bvks.members",
"lynx_bookmarks.html",
"weekly_linode_backup.zip", "Nessus-3.2.1-
es4.i386.rpm"]
```

Alternativni je način izlistavanja sadržaja direktorija sljedeći:

```
>> Dir.glob("*")
=> ["mbox", "changed_etc", "Mail",
"mailman.news.members", "all_databases.sql",
"weekly_backup", "bin", "mailman.bvks.members",
"lynx_bookmarks.html",
"weekly_linode_backup.zip", "Nessus-3.2.1-
es4.i386.rpm"]
```

Za promjenu trenutačnog direktorija koristit ćemo se metodom **chdir**:

```
>> Dir.chdir("/")
=> 0
>> Dir.pwd
=> "/"
```

Također možemo kreirati nove direktorije:

```
Dir.mkdir("/tmp/moj_direktorij")
```

5.4. Upravljanje imenima datoteka

Svaka datoteka ima svoje puno ime, ali joj također možemo pristupiti i navodeći samo njeno relativno ime uzimajući u obzir naš trenutačni direktorij.

U programskom jeziku Ruby postoji klasa *Pathname*, koja sadrži metode za upravljanje imenima datoteka.

Klasa *Pathname* je vanjska klasa pa je trebamo učitati prije početka rada:

```
>> require 'pathname'
=> true
```

Više ćemo o učitavanju dodatnog koda govoriti u poglavlju o modulima.

Sada možemo kreirati objekt klase *Pathname*. U ovom primjeru koristit ćemo realtivno ime datoteke (uz pretpostavku sa se trenutačno nalazimo u direktoriju */root*):

```
>> p = Pathname.new("Mail")
=> #<Pathname:Mail>
```

Pogledajmo puno ime datoteke:

```
>> p.realpath
=> #<Pathname:/root/Mail>
```

Možemo također provjeriti je li riječ o direktoriju, datoteci ili simboličnoj vezi:

```
>> p.directory?
=> true
>> p.file?
=> false
>> p.symlink?
=> false
```

Klasa *Pathname* također omogućava izdvajanje imena datoteke te direktorija u kojem se ta datoteka nalazi:

```
>> p.basename
=> #<Pathname:Mail>
>> p.dirname
=> #<Pathname:.>
```

Klasa Pathname omogućava više od 70 operacija nad imenima datoteka i samim datotekama. Više informacija možete naći na: http://www.ruby-doc.org/stdlib/libdoc/pathname/rdoc/index.html.

5.5. Program i osnovni direktorij

Programi najčešće dolaze zajedno s pomoćnim datotekama kao što su na primjer konfiguracijske datoteke.

Programski jezik Ruby posjeduje specijalnu varijablu koja sadržava ime datoteke koju trenutačno izvodimo. Ta varijabla zove se **__FILE__** i omogućava nam pozivanje ostalih datoteka koje se nalaze u tom istom direktoriju.

Važno

Naredba

File.dirname(__FILE__)

upućuje na direktorij gdje se nalazi program koji smo pokrenuli. Kada znamo ime tog direktorija, onda je lako odrediti ime bilo koje pomoćne datoteke koju želimo otvoriti.

Na primjer, možemo kreirati program *moj_program.rb* sa sljedećim naredbama:

```
puts __FILE__
puts File.dirname(__FILE__)
puts "#{File.dirname(__FILE__)}/config.txt"
```

Program će prvo ispisati vrijednost varijable **__FILE__** koja predstavlja puno ime programske datoteke, a nakon toga će ispisati ime direktorija u kojem se nalazi program. Na kraju će ispisati puno ime konfiguracijske datoteke koje se sastoji od imena direktorija u kojem se nalazi sam program i kratkog imena *config.txt*.

Program možemo pokrenuti na sljedeći način:

```
# ruby moj_program.rb
```

Program će ispisati sljedeće:

```
moj_program.rb
.
./config.txt
```

Ako se program nalazi u nekom drugom direktoriju od onog u kojem se trenutačno nalazimo pokrenut ćemo ga na sljedeći način:

```
# cd ..
# ruby /root/moj_program.rb
```

U tom će slučaju program ispisati sljedeće:

```
/root/moj_program.rb
/root
/root/config.txt
```

5.6. Pisanje programa za različite operacijske sustave

Ukoliko želimo u programskom jeziku Ruby pisati programe na način da se mogu izvoditi na različitim operacijskim sustavima, trebamo pažljivo slagati imena datoteka.

Jedan su od razloga različite oznake za odvajanje direktorija koje koriste različiti operacijski sustavi. Microsoftovi operacijski sustavi rabe oznaku **** (eng. *backslash*) za odvajanje direktorija, dok operacijski sustavi Unix rabe **/** (eng. *slash*).

U prethodnom smo poglavlju stvorili ime datoteke koristeći se naredbom:

```
"#{File.dirname(__FILE__)}/config.txt"
```

Na ime direktorija dodali smo */config.txt*. Takav način kreiranja imena datoteka je prikladan za operacijske sustave Unix, međutim, nije prikladan za Microsoftove operacijske sustave.

Iz tog razloga klasa *File* ima metodu **join** koja je prikladna za kreiranje imena datoteka i direktorija na svim operacijskim sustavima:

```
File.join(File.dirname(__FILE__), "config.txt")
```

Ta naredba dodaje odgovarajući znak za razdvajanje direktorija ovisno o tome na kojem se operacijskom sustavu izvodi.

5.7. Brisanje i kopiranje datoteka

Ukoliko želimo izbrisati datoteku, upotrijebit ćemo metodu **delete**:

```
File.delete("/tmp/moja_datoteka")
```

Ovaj će kôd prijaviti grešku ukoliko datoteka koju želimo izbrisati ne postoji. Da bismo to izbjegli, možemo prvo provjeriti postoji li datoteka koristeći se metodom **exist?**:

```
if File.exist?("/tmp/moja_datoteka")
  File.delete("/tmp/moja_datoteka")
End
```

Za kopiranje datoteka služimo se metodom **cp** iz klase *FileUtils:*

```
FileUtils.cp("/etc/inittab","/tmp")
```

U gornjem primjeru iskopirali smo datoteku */etc/inittab* na odredište koje je u ovom slučaju direktorij */tmp*. Nova datoteka će imati ime */tmp/inittab*.

Odredište može biti ime nove datoteke:

```
FileUtils.cp("/etc/inittab","/tmp/inittab.bak")
```

Rezultat te naredbe bit će kopija polazne datoteke koja će se zvati */tmp/inittab.bak*.

Ukoliko želimo odjednom iskopirati više datoteka, kao prvi argument trebamo navesti niz:

```
FileUtils.cp(["/etc/file1","/etc/file2"],"/tmp")
```

U tom slučaju odredište mora biti neki direktorij (ne može biti ime datoteke).

Najčešće su korištene metode za rad s datotekama i direktorijima:

Metode	Opis
`cd`	promjena radnog direktorija
`chmod, chmod_R`	mijenjanje prava nad datotekom
`cp, cp_r`	kopiranje datoteka i direktorija
`link, ln, ln_s`	kreiranje simboličnih veza
`mkdir, mkdir_p`	kreiranje direktorija
`move, mv`	premještanje i preimenovanje datoteka
`pwd`	trenutačni direktorij
`rm, rm_f, rm_r, rm_rf`	brisanje datoteka i direktorija
`touch`	kreiranje i(li) osvježavanje datuma na datotekama

Klasa FileUtils podržava sve operacije nad datotekama uključujući rekurzivno kopiranje, rekurzivno brisanje, kreiranje veza, mijenjanje dozvola i drugo. Potpuna dokumentacija može se naći na: http://www.ruby-doc.org/stdlib/libdoc/fileutils/rdoc/index.html.

5.8. Vježba: Rad s datotekama

1. Napišite kôd koji će kreirati datoteku */tmp/moja_datoteka* i u nju upisati tekst "Moja datoteka".
2. Učitajte cijelu datoteku */tmp/moja_datoteka* kao niz znakova u varijablu ***a***.
3. Provjerite koji je trenutačni direktorij, preselite se u direktorij ***/*** te ponovno provjerite koji je trenutačni direktorij.
4. Napišite program koji će ispisati sadržaj konfiguracijske datoteke *config.txt*. Program mora raditi iz bilo kojeg direktorija.
5. Izbrišite datoteku */tmp/moja_datoteka*.

5.9. Pitanja za ponavljanje

1. Koja se klasa rabi za pristup datotekama?
2. Koja se klasa rabi za rad s direktorijima?
3. Što predstavlja __**FILE**__?
4. Kako pisati programski kod koji će radiit na različitim operacijskim sustavima?

U ovom je poglavlju obrađeno:

- otvaranje datoteka
- čitanje datoteka i pisanje u datoteke
- rad s direktorijima
- kopiranje i brisanje datoteka
- učitavanje i pretraživanje datoteka.

6. Više o klasama i objektima

6.1. Vrste metoda

Do sada smo spominjali samo metode instancije (eng. *instance methods)* koje se pozivaju nakon što smo kreirali objekt. Na primjer:

```
>> class Automobil
>>   def svjetla
>>     puts "Uključujem svjetla!"
>>   end
>> end
=> nil
>> a = Automobil.new
=> #<Automobil:0xb7eaced4>
>> a.svjetla
Uključujem svjetla!
=> nil
```

Postoji još jedna vrsta metoda koje se zovu metode klase ili statičke metode (eng. *class methods*). Takve se metode definiraju navođenjem ključne riječi **self.** ispred imena.

Na primjer:

```
>> class Automobil
>>   def self.boje
>>     %w( crvena zelena bijela plava crna )
>>   end
>> end
```

Tako definiranu metodu *boje* možemo isprobati na samoj klasi:

```
>> Automobil.boje
=> ["crvena", "zelena", "bijela", "plava",
"crna"]
```

Proučimo primjer koji slijedi.

Definirajmo dvije metode *pozdrav*: jednu kao metodu klase, a drugu kao metodu instancije. Neka klasa ima ime *Kvadrat*.

```
>> class Kvadrat
>>   def self.pozdrav
>>     puts "Pozdrav od klase Kvadrat!"
>>   end
>>   def pozdrav
>>     puts "Pozdrav od objekta klase Kvadrat!"
>>   end
>> end
```

Pozovimo metodu klase:

```
>> Kvadrat.pozdrav
Pozdrav od klase Kvadrat!
```

Pozovimo metodu instancije:

```
>> Kvadrat.new.pozdrav
Pozdrav od objekta klase Kvadrat!
```

U praksi se većina metoda definira kao metode instancije, a tek se u rijetkim slučajevima rabe metode klase. Na primjer, lista mogućih boja univerzalna je za svaki automobil pa smo zato metodu *boje* kreirali unutar klase *Automobil*.

Kao što postoje varijable instancije, tako u programskom jeziku Ruby postoje i varijable klasa. Imena varijabli klasa počinju s dva znaka @ (@@), za razliku od imena varijabli instancija čija imena počinju s jednim znamom @.

Važno

Imena varijabli klasa započinju s dva znaka @ (@@).

Varijable klasa su korisne ako želimo zabilježiti neki podatak koji se odnosi na cijelu klasu.

Doseg varijabli klasa je cijela klasa, za razliku od varijabli instancija čiji je doseg objekt kojem pripadaju. Varijable klasa su korisne ako želimo zabilježiti neki podatak koji se odnosi na cijelu klasu.

Na primjer, pretpostavimo da želimo brojati koliko je kreirano objekata određene klase. To ćemo napraviti rabeći odgovarajuću varijablu klase i odgovarajuću varijablu metode:

```
>> class Krug
>>   def initialize
>>     if defined?(@@broj_krugova)
>>       @@broj_krugova += 1
>>     else
>>       @@broj_krugova = 1
>>     end
>>   end
>>   def Krug.broj
>>     @@broj_krugova
>>   end
>> end
```

Iskoristimo našu novu klasu *Krug*:

```
>> a = Krug.new
=> #<Krug:0xb7eaced4>
>> puts Square.broj
1
=> nil
>> b = Krug.new
=> #<Krug:0xb7eafcd3>
>> puts Krug.broj
2
=> nil
```

6.2. Nasljeđivanje

Mnoge klase slične su jedna drugoj tako da nema potrebe definirati sve metode u svakoj klasi. Možemo prvo definirati generalnu klasu sa svim glavnim metodama. Nakon toga možemo kreirati specifične klase koje imaju još neke posebne funkcije.

Na primjer, ukoliko želimo definirati klasu *Kabriolet* možemo uzeti sve metode iz postojeće klase *Automobil* i dodati još neke posebne metode. Ako želimo da neka klasa naslijedi sve metode iz neke druge klase, u prvi redak definicije dodat ćemo znak **<** iza imena klase koju definiramo te navesti ime klase od koje se nasljeđuju metode.

U sljedećem primjeru definirat ćemo klasu *Kabriolet* koja nasljeđuje sve metode klase *Automobil* i ima jednu dodatnu metodu za spuštanje krova:

```
class Kabriolet < Automobil
  def spusti_krov
    puts "Spuštam krov!"
  end
end
```

Na ovaj način izbjegavamo umnožavanje koda. Ako su dvije klase slične, možemo kreirati zajedničke metode u zajedničkoj klasi i onda svaka od klasa može naslijediti metode iz zajedničke klase i imati još neke dodatne.

6.3. Ulančavanje metoda

Poput izraza i metode se u programskom jeziku Ruby mogu ulančavati.

Pretpostavimo da niz znakova želimo ispisati velikim slova obrnutim redoslijedom. Jedan od načina je sljedeći:

```
>> a = "Ovo je string"
=> "Ovo je string"
>> a = a.upcase!
=> "OVO JE STRING"
>> a = a.reverse!
=> "GNIRTS EJ OVO"
>> puts a
GNIRTS EJ OVO
=> nil
```

Rabeći ulančavanje metoda isti rezultat možemo postići u samoj jednom retku:

```
>> puts "Ovo je string".upcase.reverse
GNIRTS EJ OVO
=> nil
```

6.4. Blokovi i iteratori

Blokovi i iteratori jedna su od specifičnosti programskog jezika Ruby. U drugim programskim jezicima postoje funkcije i procedure koje pozivamo

jednim ili s više argumenata. Ti argumenti mogu biti brojevi, nizovi znakova i drugi tipovi podataka. Međutim, u programskom jeziku Ruby moguće je kao parametar zadati ne samo podatke već i sam programski kôd.

Zaokruženi kôd koji možemo promatrati kao cjelinu se zove blok.

Definirajmo jednostavnu metodu:

```
>> def novi_blok
>> puts "Početak ponavljanja."
>> yield
>> yield
>> puts "Kraj ponavljanja."
>> end
=> nil
```

U gornjem smo primjeru definirali metodu *novi_blok*. Posebnost je te metode da u sadrži ključnu riječ **yield** koja izvršava blok koji je poslan u pozivu te metode. Pozvat ćemo metodu koju smo upravo definirali navodeći kao argument pri pozivanju te metode drugi programski kôd - blok. Taj se blok sastoji od samo jedne naredbe koja ispisuje "Pozdrav svima!":

```
>> novi_blok { puts "Pozdrav svima!" }
Početak ponavljanja.
Pozdrav svima!
Pozdrav svima!
Kraj ponavljanja.
```

Dakle, blokovi se ne tretiraju na isti način kao drugi argumenti koji se prenose funkcijama. Argumenti koji nisu blokovi evaluiraju se prije poziva funkcije dok se blokovi evaluiraju tek prilikom poziva metode **yield** u tijelu funkcije.

Blok može primati argumente svaki put kada ga izvršimo, pa se blokova rabe najčešće za implementaciju iteratora. Iterator je specijalna metoda koja izvršava blok na svakom članu niza.

Na primjer:

```
>> ["zgrada", "ulica", "cesta"].each do |a|
puts a.upcase
>> end
ZGRADA
ULICA
CESTA
=> ["zgrada", "ulica", "cesta"]
```

Prvo smo definirali niz koji se sastoji od tri člana. Svaki od tih članova niz je znakova. Nakon toga smo pozvali metodu **each** koja je iterator koji redom za svaki član niza izvršava zadani blok predajući mu navedenog člana niza kao parametar.

U gornjem primjeru, iterator **each** će prvo pozvati blok s parametrom "zgrada". Unutar bloka varijabla *a* poprimit će vrijednost "zgrada" i blok će se izvršiti. Nakon toga blok će se pozvati ponovno, a ovaj put će vrijednost varijable *a* biti "ulica". I tako dalje.

Isto smo mogli napisati i na kraći način, rabeći vitičaste zagrade umjesto ključnih riječi **do** i **end**:

```
>> ["zgrada", "ulica", "cesta"].each { |a| puts
a.upcase }
```

Iteratorima se može značajno smanjiti količina koda koju je potrebno napisati. Ipak u programskom jeziku Ruby i dalje se programski kôd može pisati na „stari način“.

Na primjer, umjesto koda koji je napisan u stilu programskog jezika Ruby:

```
[1, "test", 2, 3, 4].each { |element| puts
element.to_s + "X" }
```

može se u stilu manje dinamičnih programskih jezika također napisati:

```
a = [1, "test", 2, 3, 4]
i = 0
while (i < a.length)
puts a[i].to_s + "X"
i += 1
end
```

6.5. Vježba: Klase i nasljeđivanje

1. Definirajte metodu *broj_kotaca* unutar klase *Automobil* koja vraća broj 4. Provjerite ispravnost koda koji ste napisali na nekom primjeru.
2. Definirajte klasu *Terenac* koja nasljeđuje metode iz klase *Automobil* te dodajte joj metodu *pogon_na_4_kotaca* (koja ne radi ništa). Zatim kreirajte novi objekt klase *Terenac* u varibli a i pokrenite metodu *pogon_na_4_kotaca*.
3. Obradite niz znakova "Ovo je string" tako da ima samo mala slova te nakon toga obrnite redoslijed znakova.

6.6. Pitanja za ponavljanje

1. Kako dijelimo metode u programskom jeziku Ruby?
2. Što je to nasljeđivanje klasa i koje su njegove prednosti?
3. Što je ulančavanje metoda?

U ovom je poglavlju obrađeno:

- vrste metoda
- nasljeđivanje klasa
- ulančavanje metoda
- blokovi i iteratori.

7. Upravljanje greškama

7.1. Klasa Exception

Programski jezik Ruby omogućava pisanje programskog koda koji će se izvršiti u slučaju da program tijekom izvršavanja naiđe na neku programsku grešku.

Za upravljanje greškama sluzi klasa **Exception**. U njoj su definirane metode koje se izvršavaju u slučaju neke programske greške.

Ukoliko sami ne odredimo što će program napraviti u slučaju da se dogodi neka programska greška, ispisat će se poruka s opisom greške i prekinut će se izvođenje programa.

Na primjer, pokušajmo varijabli čija je vrijednost **nil** pristupiti kao da je niz:

```
>> a = nil
=> nil
>> a[0]
NoMethodError: undefined method `[]' for
nil:NilClass
        from (irb):2
=>
```

Obavijest o greški ispisana je na ekranu i program je nakon toga prekinuo izvođenje.

7.2. Upravljanje greškama

Kôd kojim se želi upravljati greškom uvijek počinje ključnom riječi **begin**, a nakon toga se navodi kôd za koji želimo osigurati poseban tretman u slučaju greške. Sljedeća je ključna riječ **rescue** i ime klase ili podklase koja je nadležna za vrstu greške za koju želimo poseban tretman. Ako želimo da poseban tretman vrijedi za bilo koju vrstu greške, navest ćemo klasu *Exception* (u suprotnom ćemo navesti neku od njenih 30-tak podklasa, poput *NoMemoryError*, *RuntimeError*, *SecurityError, ZeroDivisionError* i drugih). Nakon toga slijedi kôd koji će se izvršiti ukoliko program naiđe na programsku grešku.

Proučimo kako možemo upravljati greškom iz prethodnog primjera:

```
>> begin
>>   a[0]
>>   rescue Exception
>>   puts "Doslo je do greske!"
>> end
Doslo je do greske!
=> nil
```

U ovom smo primjeru samo ispisali da je došlo do greške i nastavili dalje izvršavati kôd. Mogli smo također zapisati grešku u *log*-datoteku, poslati poruku putem elektroničke pošte i tako dalje.

Ukoliko umjesto ključne riječi **rescue** upotrijebimo ključnu riječ **raise**, izvršit će se programski kôd koji slijedi, ali nakon toga će se greška nastaviti obrađivati na standardan predefiniran način.

U sljedećem primjeru rabit ćemo ključnu riječ **raise**. Ispisat će se poruka *Greska!* te nakon standardna poruka o grešci uz prekid izvršavanja programa:

```
>> begin
>>   0.upcase
>>   rescue Exception
>>   puts "Greska!"
>>   raise
>> end
Greska!
NoMethodError: undefined method `upcase' for
0:Fixnum
        from (irb):2
```

Ukoliko želimo da se izvrši neki kôd bez obzira na to je li došlo do greške ili ne, rabit ćemo ključnu riječ **ensure**:

```
>> begin
>>   0.upcase
>>   rescue Exception
>>   puts "Greska!"
>>   ensure Exception
>>   puts "Zatvaram datoteke"
>> end
Greska!
Zatvaram datoteke
=> nil
```

Gornji primjer je primjer uobičajene uporabe ključne riječi **ensure** (radi se o zatvaranju datoteka neovisno o tome je li došlo do greške ili nije).

Proučimo što bi se dogodilo u gornjem primjeru da prilikom izvršavanja nije pokrenut neispravan kôd:

```
>> begin
>>   0.zero?
>>   rescue Exception
>>   puts "Greska!"
>>   ensure Exception
>>   puts "Zatvaram datoteke"
>> end
Zatvaram datoteke
=> true
```

Obzirom da nije došlo do programske greške (naredba *0.zero?* je ispravna), ne ispisuje se poruka *Greska!*, ali se izvršava dio koda u kojem piše *Zatvaram datoteke*.

Kod za upravljanje greškama možemo uključiti i u definicije klasa. Na primjer:

```
>> class Student
>>  def initialize(a)
>>   raise ArgumentError, "Nedostaje ime!" if
a.empty?
>>  end
>> end
=> nil
```

Pokušajmo stvoriti novi objekt klase *Student* navodeći kao argument prazan niz znakova:

```
>> kandidat=Student.new('')
ArgumentError: Nedostaje ime!
```

Pokušajmo sada stvoriti novi objekt ispravno navodeći kao argument niz znakova „Marko“:

```
>> kandidat=Student.new('Marko');
=> #<Student:0x3d8285>
```

7.3. *Catch* i *throw*

Programski jezik Ruby putem metoda **catch** i **throw** omogućava prekidanje izvođenja nekog programskog bloka, na primjer petlje.

Metodom **catch** definira se programski blok koji se normalno izvršava do mjesta poziva metode **throw**. Kad interpreter naiđe na **throw**, prekida se izvođenje izvođenje odgovarajućeg programskog bloka.

Na primjer:

```
a=[1, 2 , 3 , nil ,5, 6]
sum = 0
catch :element_nil do
  a.each do |n|
    throw :element_nil unless n
    sum += n
    p sum
  end
end
```

Ako izvršimo ovaj programski kôd, on će rezultirati sljedećim ispisom:

```
1
3
6
=> nil
```

Ispis pokazuje da se petlja izvršila samo za prva tri člana niza. Kad je petlja naišla na četvrti član niza čija je vrijednost **nil**, što odgovara logičkoj vrijednosti i false, izvršila se metoda **throw** i prekinulo se izvođenje cijelog programskog bloka.

Gore navedeni primjer mogao se napisati i na sljedeći, više tradicionalan način:

```
a=[1, 2 , 3 , nil ,5, 6]
sum = 0
catch  :element_nil do
  a.each do |n|
    if not n
       throw :element_nil
    end
    sum += n
    p sum
  end
end
```

7.4. Vježba: Upravljanje greškama

1. Napišite program koji sadrži naredbu **`0.upcase`** tako da se ne prekine izvršavanje programa nego se ispiše poruka "Greška". U nastavku program treba ispisati poruku "Kraj programa".
2. Napišite program koji sadrži naredbu **`0.upcase`** tako da se ispiše "Greška" i nakon toga izvrši standardna obrada greške (što znači da će se prekinuti izvođenje programa). Stavite na kraj programa ispis niza znakova "Ovo se neće izvršiti" da biste provjerili je li se rad programa stvarno prekinuo.

7.5. Pitanja za ponavljanje

1. Koja klasa služi za upravljanje greškama?
2. Čemu služi metoda **raise**?
3. Koja je razlika između metoda **raise** i **rescue**?

U ovom je poglavlju obrađeno:

- upravljanje greškama
- metode **rescue** i **ensure**
- metode **catch** i **throw**.

8. Moduli

8.1. Korištenje više datoteka

Programski kôd moguće je podijeliti u više datoteka. Za uključivanje koda iz dodatnih datoteka služimo se ključnom riječju **require**.

Na primjer:

```
require 'datoteka1'
require 'datoteka2'
```

Također, interpreter neće prilikom pokretanja učitati sve klase koje su postoje u programskom jeziku Ruby nego samo osnovne. Stoga za uključivanje dodatnih klasa također trebamo rabiti naredbu **require**.

Na primjer, naredba

```
require 'pathname'
```

učitat će klasu *Pathname* i omogućiti njezino korištenje.

8.2. Definiranje modula

Modul je izolirani kôd koji se može rabiti u više aplikacija. Moduli omogućavaju grupiranje metoda, klasa i konstanti u jednu cjelinu.

Kreirat ćemo modul koji sadrži funkciju za preračunavanje brzine zadane u metrima u sekundi u kilometre na sat:

```
module Brzina
  def self.konverzija metara_u_sekundi
    metara_u_sekundi * 3.6
  end
end
```

Sada možemo pozvati novu funkciju:

```
>> Brzina.konverzija 10
=> 36.0
```

8.3. Jedinstvenost modula

Kako bi se spriječili konflikti prilikom imenovanja modula, uobičajeno je da se modulima daje jedinstveni prefiks (eng. *namespace*). Na primjer, ukoliko je prethodni modul nastao u Sveučilišnom računskom centru, možemo ga napisati ovako:

```
module Srce
  module Brzina
    def self.konverzija metara_u_sekundi
      metara_u_sekundi * 3.6
    end
  end
end
```

Tako definirani modul *Brzina* pozvat ćemo na sljedeći način:

```
Srce::Brzina.konverzija 10
=> 36.0
```

Ugnježđujući modul *Brzina* u nadređeni modul specifičnog imena *Srce* smanjili smo vjerojatnost konflikta s imenima drugih modula.

Ovako definirani moduli najčešće se nalaze u zasebnoj datoteci, stoga prije njihova korištenja moramo učitati datoteku u kojoj se nalaze koristeći se ključnom riječju **require**.

8.4. *Mixin*

Važno

Dvije su važne funkcionalnosti koje omogućavaju moduli:

- izbjegavanje konflikata s imenima (*namespace*)
- uključivanje modula u definiciju klasa (*mixin*).

Moduli se mogu uključivati u definicije klasa. Time metode definirane unutar modula postaju dostupne unutar klasa.

Ova se mogućnost zove *mixin* (*mixed-in modules*).

Ponovo ćemo definirati modul *Brzina* i uključiti ga u klasu *Automobil* rabeći ključnu riječ **include**:

```
module Brzina
  def self.konverzija metara_u_sekundi
    metara_u_sekundi * 3.6
  end
end

class Automobil
  include Brzina
  attr_accessor :proizvodjac, :model,
       :godina_proizvodnje, :snaga_motora,
       :prijedjeni_kilometri
end
```

Sada se možemo služiti metodom *konverzija* izravno unutar objekta klase *Automobil*:

```
>> a = Automobil.new
=> #<Automobil:0xb7d09de8>
>> a.konverzija 10
=> 36.0
```

8.5. Vježba: Moduli

1. Kreirajte datoteku *automobil_common.rb* koja sadrži definiciju klase *Automobil* s metodom *pokreni_motor* koja ispisuje „Pokrećem motor“. Zatim iz glavne datoteke *automobil.rb* pozovite datoteku *automobil_common.rb*, kreirajte novi objekt klase *Automobil* i pozovite metodu *pokreni_motor*.
2. Kreirajte datoteku *motorna_vozila.rb*. Ona neka sadrži modul *MotornaVozila*, koji ima jednu metodu pod nazivom *pokreni_motor* i ispisuje tekst "Pokrećem motor". Nakon toga uključite modul *MotornaVozila* u klasu *Automobil* i pozovite metodu *pokreni_motor* na novom objektu klase *Automobil* (neka sve bude u istoj datoteci *motorna_vozila.rb*).

8.6. Pitanja za ponavljanje

1. Kako možemo podijeliti kod u više datoteka?
2. Što je to modul?
3. Kako možemo postići jedinstvenost imena modula?

U ovom je poglavlju obrađeno:

- korištenje više datoteka
- definiranje modula
- uvrštavanje modula u klasu.

9. Korištenje postojećega koda

9.1. Korištenje programa gem

Programski jezik Ruby sadrži osnovni skup dodatnih funkcija pod nazivom *standard library*.

Proširenja (ekstenzije) napisane u programskom jeziku Ruby zovu se gem, a tako se zove i program za upravljanje ekstenzijama.

Trenutačno instalirana proširenja možemo vidjeti naredbom:

```
$ gem list
```

Ukoliko nam je potrebna pomoć u radu s programom gem, koristit ćemo se naredbom:

```
$ gem help commands
```

Program gem možemo ograničiti tako da radi isključivo s lokalnim ili isključivo s udaljenim repozitorijem. Lokalni repozitorij sadrži proširenja koja su instalirana na lokalnom računalu, a udaljeni repozitorij čini skup svih proširenja koja su dostupna putem Interneta a nalaze se na serveru **rubygems.org**.

Za pristup udaljenom repozitoriju služi opcija **--remote.** Na primjer, ukoliko želimo ispisati sve inačice proširenja *cmdparse*, rabit ćemo naredbu:

```
$ gem list cmdparse --remote
```

Ako želimo instalirati gem pod nazivom "Chronic" to možemo napraviti na sljedeći način

```
$ gem install chronic
Successfully installed chronic-0.2.3
1 gem installed
Installing ri documentation for chronic-0.2.3...
Installing RDoc documentation for chronic-
0.2.3...
```

Moguće je istovremeno rabiti više inačica istog proširenja.

Također je moguće instalirati određenu inačicu koristeći se opcijom **--version**.

9.2. Korištenje tuđega koda u aplikaciji

Nakon što je gem pravilno instaliran, možemo ga rabiti unutar datoteke ili interaktivne ljuske na sljedeći način:

```
>> require 'chronic'
=> true
```

Ako pokušavamo učitati proširenje koje je već učitano, kao rezultat učitavanja dobit ćemo vrijednost **false**:

```
>> require 'rubygems'
=> false
```

(Proširenje *rubygems* automatski se učitava prilikom pokretanja interaktivne ljuske.)

Ukoliko želimo instalirati neku konkretnu inačicu nekog proširenja, nju ćemo navesti nakon naziva proširenja:

```
$ gem 'cmdparse', "= 2.0.2"
```

Ukoliko ne navedeno konkretnu inačicu koju želimo instalirati, instalirat će se najnovija inačica koja se nalazi u repozitoriju.

Gem *chronic* ima metodu *parse* kojom možemo pretvarati vremenske oznake koje rabimo u svakodnevnom govoru u konkretne datume:

```
>> Chronic.parse "7 hours ago"
=> Sun Mar 30 09:40:06 0200 2008
```

9.3. Vježba: Rad s programom gem

1. Instalirajte gem pod nazivom *chronic*.
2. Kreirajte datoteku *chronic_test.rb* u kojoj ćete se koristiti ekstenzijom *chronic* za ispisivanje vremena i datuma od prije točno tri mjeseca.
3. Ispišite trenutačno instalirane ekstenzije.
4. Ispišite sve verzije proširenja *chronic*.
5. Ispišite sva proširenja u kojima se spominje riječ *graph*.
6. Instalirajte inačicu 1.0.3 proširenja *cmdparse*.
7. Instalirajte najnoviju inačicu proširenja *cmdparse*.
8. Uklonite inačicu 1.0.3 proširenja *cmdparse*.

9.4. Pitanja za ponavljanje

1. Što je to gem?
2. Može li se istovremeno instalirati više inačica istog proširenja?

U ovom je poglavlju obrađeno:

- korištenje programa gem
- korištenje tuđega koda u aplikaciji.

10. Testiranje koda

10.1. Test::Unit

Testovi su mehanizam u programskom jeziku Ruby kojima se olakšava otkrivanje programskih grešaka.

Pogledajmo kako se pišu i rabe testovi. Prvo ćemo napisati kôd koji želimo testirati i spremiti ga u datoteku *automobil.rb*:

```
class Automobil
  attr_reader :motor, :rucna_kocnica
  def parkiraj
    @motor = false
    @rucna_kocnica = true
  end
end
```

Nakon što je automobil parkiran, logično je da motor bude ugašen i da ručna kočnica bude uključena. Sada ćemo napisati test koji to provjerava i snimiti ga u datoteku *automobil_test.rb*:

```
require 'automobil'
require 'test/unit'

class AutomobilTest < Test::Unit::TestCase
  def test_parkiranje
    a = Automobil.new
    a.parkiraj
    assert a.motor == false
    assert a.rucna_kocnica == true
  end
end
```

U prvom redu programa nalazi se naredba kojom se učitava datoteka *automobil.rb* u kojoj se nalazi klasa koju želimo testirati. Nakon toga se učitava klasa za testiranje.

Sam test je također klasa koja se može imati bilo koje ime, a u ovom slučaju smo izabrali ime *AutomobilTest*.

AutomobilTest klasa je koja svoje metode nasljeđuje od klase *TestCase*. *TestCase* klasa je koja se nalazi unutar modula *Unit*, a modul *Unit* nalazi se unutar modula *Test*.

Testnu klasu definiramo tako da definiramo metode, a svaka metoda sadrži jednu ili više pretpostavki koje moraju biti ispunjene. Ime metode koja testira kôd mora početi s "test_", a pretpostavke počinju sa **assert**.

Unutar testa *test_parkiranje* prvo kreiramo novi objekt klase *Automobil* i spremamo ga u varijablu *a*. Nakon toga izvršavamo metodu *parkiraj* i testiramo pretpostavke. Prva je pretpostavka da je motor ugašen, tj. da se u varijabli **@motor** nalazi vrijednost **false**. Također provjeravamo je li uključena ručna kočnica.

Test izvodimo na sljedeći način:

```
$ ruby auto_test.rb
Loaded suite auto_test
Started
.
Finished in 0.001726 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
```

10.2. Vježba: Testiranje koda

1. Kreirajte klasu *Automobil* i konstruktor čiji je argument boja automobila. Objekt smije biti kreiran samo ako je zadana boja jedna od mogućih koje su definirane metodom *boje* koja je metoda klase *Automobil*.
2. Kreirajte testnu datoteku sa sljedećim testovima:
    - Treba doći do greške ako boja nije među specificiranima.
    - Ako je boja ispravna, treba biti kreiran objekt klase *Automobil*.
    - Kreirani objekt treba imati boju koja je specificirana pri kreiranju objekta.

10.3. Pitanja za ponavljanje

1. Kojom metodom se služimo prilikom testiranja koda?
2. Koji modul trebamo učitati na početku koda za testiranje?

U ovom je poglavlju obrađeno:

- testiranje koda
- klasa Test::Unit.

11. Radionica

Napravite aplikaciju koja će raditi kao kalkulator koji može zbrajati, oduzimati, izračunavati ostatak dijeljenja dva broja. Aplikacija se sastoji od sljedećih dijelova:

- glavne aplikacije pod nazivom *calc.rb* koja prima argumente
- modula pod nazivom *Srce::Izracun* koji provodi zadane izračune i nalazi se u posebnoj datoteci
- testova koji testiraju modul i aplikaciju.

Primjer rada aplikacije je sljedeći:

```
$ ruby calc.rb 3 + 4
7
$ ruby calc.rb 3 - 4
-1
$ ruby calc.rb 7 ostatak 4
3
```

Dodatak: Rješenja vježbi

1.5. Vježba: Instalacija programskog jezika Ruby

1. Prijavite se na sustav koristeći se podacima koje ćete dobiti od predavača.
2. Instalirajte Ruby koristeći naredbu **yum**.

```
$ yum install ruby
```

3. Instalirajte sustav upravljanja proširenjima Ruby gems.

```
$ yum install rubygems
```

4. Instalirajte interaktivnu ljusku.

```
$ yum install ruby-irb
```

5. Pokrenite interaktivnu ljuski i upišite `puts "Ovo je irb."`. Izađite iz interaktivne ljuske naredbom `exit`.

```
$ irb
>> puts "ovo je irb"
quit
```

6. Kreirajte datoteku koja sadrži programsku liniju `puts "Ovo je datoteka."`. Pohranite je pod imenom *prvi_program.rb*. Izvršite je.

   Kreirajte datoteku *moja_datoteka.rb* sa zadanim sadržajem u editoru. Nakon toga pokrenite naredbu:

   ```
   $ruby moja_datoteka.rb
   ```

7. Koristeći se opcijom `-e` programa `ruby` izvršite kôd `puts "Ovo je komandna linija."`.

```
$ ruby -e 'puts "ovo je komandna linija"'
```

8. Nađite u *on-line* dokumentaciji opis metode **upcase** koja pripada klasi *String.*

   Otvorite *web*-stranicu: http://www.ruby-doc.org/core.

   Izaberite **upcase** iz popisa metoda koji se nalazi u gornjem lijevom kutu ili pritisnite *crtl-f* za pretraživanje stranice i utipkajte **upcase**.

2.9. Vježba: Tipovi podataka, operatori, kontrolne strukture, ulaz/izlaz, sistemske komande

1. Napišite kod koji će podijeliti 15 sa 7 tako da rezultat bude decimalni broj.

```
15.0 / 7
```

2. Zapišite decimalni broj 10 u heksadecimalnom obliku.

```
0xa
```

3. Zapišite broj milijarda na najkraći način.

```
1e9
```

4. Zapišite niz znakova koji će u sebi imati izračun umnoška brojeva 456 i 32.

```
"Umnožak je #{456 * 32}"
```

5. Definirajte varijablu x koja je niz koji se sastoji od niza znakova "Ruby" i od broja 2. Nakon toga na kraj tog niza dodajte broj 100.

```
x = [ "Ruby", 2 ]
x << 100
```

6. Izračunajte treću potenciju broja 4 i provjerite je li rezultat veći od 50.

```
puts "istina" if 4 ** 3 > 50
```

7. Napišite kontrolnu strukturu koja će ispisati niz znakova "istina" ako je vrijednost varijable **b** djeljiva sa 7 i manja od 100.

```
puts "istina" if b % 7 == 0 and b < 100
```

8. Ispišite niz znakova "Vjezba" bez da se na kraju ispisa ispiše znak za novi red.

```
print "Vjezba"
```

9. Ispišite datum iz pozivajući odgovarajuću naredbu operacijskog sustava.

```
system "date"
```

3.5. Vježba: Izrazi

1. Pridružite varijablama **a**, **b**, **d** i **f** vrijednost 100 koristeći se samo jednom naredbom:

```
a = b = d = f = 100
```

2. Rabeći samo jednu naredbu pridružite vrijednost 3 varijabli **a**, dodajte tom izrazu 5 i rezultat pohranite u varijablu **b**.

```
b = 5 + a = 3
```

3. U varijablu **a** pohranite vrijednost 2 a u varijablu **b** pohranite niz znakova "pero" koristeći se samo jednom naredbom.

```
a,b = 2, "pero"
```

4. Pohranite vrijednost varijable **b** u varijablu **a**, vrijednost varijable **c** u varijablu **b** i vrijednost varijable **a** u varijablu **c** rabeći samo jednu naredbu.

```
a,b,c = b,c,a
```

4.7. Vježba: Klase, metode, objekti, atributi i varijable

1. Kreirajte klasu *Automobil.*

```
class Automobil
end
```

2. Kreirajte klasu *Automobil* koja sadrži metodu *ukljuci_motor* koja ispisuje "Uključujem motor".

```
class Automobil
def ukljuci_motor
puts "Uključujem motor"
end
```

3. Kreirajte novi objekt klase *Automobil* i spremite ga u varijablu **a**.

```
a = Automobil.new
```

4. Pokrenite metodu *ukljuci_motor* u novom objektu koji ste upravo kreirali

```
a.ukljuci_motor
```

5. Provjerite i zabilježite kojoj klasi pripadaju sljedeći objekti:

```
"Niz znakova"
102
102.233
10**100
[]
{}
```

```
>> "Niz znakova".class
=> String
>> 102.class
=> Fixnum
```

(I tako dalje …)

5.8. Vježba: Rad s datotekama

1. Napišite kôd koji će kreirati datoteku */tmp/moja_datoteka* i u nju upisati tekst "Moja datoteka".

```
f = File.open("/tmp/moja_datoteka", "w")
f.puts "Moja datoteka"
f.close
```

2. Učitajte cijelu datoteku */tmp/moja_datoteka* kao niz znakova u varijablu ***a***.

```
a = File.read("/tmp/moja_datoteka")
```

3. Provjerite koji je trenutačni direktorij, preselite se u direktorij ***/*** te ponovno provjerite koji je trenutačni direktorij.

```
Dir.pwd
Dir.chdir("/")
Dir.pwd
```

4. Napišite program koji će ispisati sadržaj konfiguracijske datoteke *config.txt*. Program mora raditi iz bilo kojeg direktorija.

```
puts File.read(File.join
        (File.dirname(__FILE__),"config.txt"))
```

5. Izbrišite datoteku */tmp/moja_datoteka*.

```
File.delete("/tmp/moja_datoteka")
```

6.5. Vježba: Klase i nasljeđivanje

1. Definirajte metodu *broj_kotaca* unutar klase *Automobil* koja vraća broj 4. Provjerite ispravnost koda koji ste napisali na nekom primjeru.

```
class Automobil
  def self.broj_kotaca
    4
  end
end

>> Automobil.broj_kotaca
```

2. Definirajte klasu *Terenac* koja nasljeđuje metode iz klase *Automobil* te dodajte joj metodu *pogon_na_4_kotaca* (koja ne radi ništa). Zatim kreirajte novi objekt klase *Terenac* u varibli a i pokrenite metodu *pogon_na_4_kotaca*.

```
class Terenac < Automobil
  def pogon_na_4_kotaca
  end
end

>> a = Terenac.new
=> #<Terenac:0x4034d5b0>
>> a.pogon_na_4_kotaca
=> nil
```

3. Obradite niz znakova "Ovo je string" tako da ima samo mala slova te nakon toga obrnite redoslijed znakova.

```
"Ovo je string".downcase.reverse.reverse
```

7.4. Vježba: Upravljanje greškama

1. Napišite program koji sadrži naredbu **0.upcase** tako da se ne prekine izvršavanje programa nego se ispiše poruka "Greska". U nastavku program treba ispisati poruku "Kraj programa".

```
begin
 0.upcase
rescue
  puts "Greska"
end
puts "Kraj programa"
```

2. Napišite program koji sadrži naredbu **0.upcase** tako da se ispiše "Greska" i nakon toga izvrši standardna obrada greške (što znači da će se prekinuti izvođenje programa). Stavite na kraj programa ispis niza znakova "Ovo se neće izvrsiti" da biste provjerili je li se rad programa stvarno prekinuo.

```
begin
  0.upcase
rescue
  puts "Greska"
  raise
end
puts "Ovo se nece izvrsiti"
```

8.5. Vježba: Moduli

1. Kreirajte datoteku *automobil_common.rb* koja sadrži definiciju klase *Automobil* s metodom *pokreni_motor* koja ispisuje „Pokrecem motor“. Zatim iz glavne datoteke *automobil.rb* pozovite datoteku *automobil_common.rb*, kreirajte novi objekt klase *Automobil* i pozovite metodu *pokreni_motor*.

   Datoteka *automobil_common.rb*:

   ```
   class Automobil
     def pokreni_motor
       puts "Pokrecem motor"
     end
   end
   ```

   Datoteka *automobil.rb*:

   ```
   require 'automobil_common'
   a = Automobil.new
   a.pokreni_motor
   ```

   Pokretanje programa:

   ```
   ruby automobil.rb
   ```

2. Kreirajte datoteku *motorna_vozila.rb*. Ona neka sadrži modul *MotornaVozila*, koji ima jednu metodu pod nazivom *pokreni_motor* i ispisuje tekst "Pokrecem motor". Nakon toga uključite modul *MotornaVozila* u klasu *Automobil* i pozovite metodu *pokreni_motor* na novom objektu klase *Automobil* (neka sve bude u istoj datoteci *motorna_vozila.rb*).

   ```
   module MotornaVozila
     def pokreni_motor
      puts "Pokrecem motor"
     end
   end

   class Automobil
     include MotornaVozila
   end

   a = Automobil.new
   a.pokreni_motor
   ```

9.3. Vježba: Rad s programom gem

1. Instalirajte gem pod nazivom *chronic*.

```
$ gem install chronic
```

2. Kreirajte datoteku *chronic_test.rb* u kojoj ćete se koristiti ekstenzijom *chronic* za ispisivanje vremena i datuma od prije točno tri mjeseca.

```
require 'rubygems'
require 'chronic'
puts Chronic.parse("2 months ago")
```

3. Ispišite trenutačno instalirane ekstenzije.

```
$ gem list
```

4. Ispišite sve verzije proširenja *chronic*.

```
$ gem list chronic
```

5. Ispišite sva proširenja u kojima se spominje riječ *graph*.

```
$ gem list graph
```

6. Instalirajte inačicu 1.0.3 proširenja *cmdparse*.

```
$ gem install cmdparse --version 1.0.3
```

7. Instalirajte najnoviju inačicu proširenja *cmdparse*.

```
$ gem install cmdparse
```

8. Uklonite inačicu 1.0.3 proširenja *cmdparse*.

```
$ gem uninstall cmdparse --version 1.0.3
```

10.2. Vježba: Testiranje koda

1. Kreirajte klasu *Automobil* i konstruktor čiji je argument boja automobila. Objekt smije biti kreiran samo ako je zadana boja jedna od mogućih koje su definirane metodom *boje* koja je metoda klase *Automobil*.

```
class Automobil
  attr_reader :boja
  def initialize boja
    raise "boja nije zadana" unless boja
    raise "Netocna boja" unless
                  Automobil.boje.include? boja
    @boja = boja
  end
  def self.boje
    ["crvena","zuta","zelena"]
  end
end
```

2. Kreirajte testnu datoteku sa sljedećim testovima:
   - Treba doći do greške ako boja nije među specificiranima.
   - Ako je boja ispravna, treba biti kreiran objekt klase *Automobil*.
   - Kreirani objekt treba imati boju koja je specificirana pri kreiranju objekta.

```
require 'auto'
require 'test/unit'
class AutomobilTest < Test::Unit::TestCase
  def test_parkiranje
    # dolazi do greške ako boja nije
    # specificirana
    assert_raise ArgumentError do
      a = Automobil.new
    end

    # ako je boja ispravna, treba biti kreiran
    # objekt klase Automobil
    a = Automobil.new "crvena"
    assert_instance_of Automobil, a

    # Kreirani objekt treba imati boju koja je
    # specificirana pri kreiranju objekta
    assert_equal "crvena", a.boja
  end
end
```

Literatura

1. Dave Thomas, Chad Fowler, Andy Hunt: Programming Ruby: The Pragmatic Programmers' Guide. The Pragmatic Bookshelf, 2006.

2. Peter Cooper: Beginning Ruby: From Novice to Professional. Apress, 2007