Upoznavanje sa objektima

„Prirodu parcelišemo i razvrstavamo u koncepte kojima pripisujemo značenja, uglavnom zato što se držimo dogovora koji važi u našoj jezičkoj zajednici i kodifikovan je u obrascima našeg jezika ... uopšte ne možemo govoriti ukoliko se ne pridržavamo organizacije i klasifikacije podataka koje taj dogovor propisuje.“ Benjamin Lee Whorf (1897–1941)

RAČUNARSKA REVOLUCIJA JE NASTALA U MAŠINI. NAŠI PROGRAMSKI JEZICI STOGA TEŽE DA izgledaju kao ta mašina.

Ali računari nisu toliko mašine koliko pojačala uma („točkovi za um“, kako Steve Jobs voli da kaže) i drugi način izražavanja. Zbog toga ti alati sve manje liče na mašine, a sve više na delove našeg uma, i na druge oblike izražavanja, kakvi su pisanje, slikanje, vajanje, animiranje i snimanje filmova. Objektno orijentisano programiranje (OOP) deo je ovog pomeranja ka korišćenju računara kao sredstva izražavanja.

Ovo poglavlje vas uvodi u osnovne koncepte OOP-a, uključivši i pregled metoda razvoja, uz pretpostavku da imate iskustva s programiranjem, iako to ne mora biti na jeziku C. Ako mislite da treba da se još pripremate za programiranje pre nego što se uhvatite u koštac sa ovom knjigom, proučite multimedijski seminar *Thinking in C*, koji možete preuzeti na adresi *www.MindView.net.*

Ovo poglavlje je i podloga i dodatni materijal. Mnogi se ne osećaju prijatno u svetu objektno orijentisanog programiranja ako pre toga ne razumeju celinu. Stoga je ovde dat bogat pregled koncepata OOP-a. Drugi ne mogu da shvate glavne principe dok prvo ne upoznaju barem neke mehanizme. Ako pripadate toj grupi i željni ste da otkrijete specifičnosti jezika, slobodno preskočite ovo poglavlje – u ovom trenutku to vas neće sprečiti da pišete programe ili naučite jezik. Međutim, poželećete da se vratite na ovaj deo knjige da biste dopunili svoje znanje i shvatili zašto su objekti važni i kako da ih koristite pri pisanju programa.

Razvoj apstrakcije

Svi programski jezici obezbeđuju apstrakciju. Moglo bi se raspravljati o tome da li je složenost problema koje ste u stanju da rešite direktno povezana s vrstom i kvalitetom apstrakcije. Pod „vrstom“ mislim na to „šta apstrahujete“. Mašinski jezik je mala apstrakcija mašine na kojoj se programi izvršavaju. Mnogi takozvani „proceduralni“ jezici koji su sledili mašinski (kao FORTRAN, BASIC i C) bili su apstrakcija mašinskog jezika. Ovi jezici su veliki napredak u odnosu na mašinski jezik, ali njihova primarna apstrakcija ipak zahteva od vas da razmišljate iz ugla strukture računara umesto iz ugla problema koji rešavate. Programer mora da uspostavi vezu između modela mašine (u „prostoru rešenja“ koji predstavlja mesto gde realizujete rešenje problema, na primer, u računaru) i modela problema koji se rešava (u „prostoru problema“ koji predstavlja mesto gde problem postoji, recimo poslovanje). Napor koji iziskuje ovo preslikavanje i činjenica da je ono nebitno za programski jezik proizvode programe koji se teško pišu i čije je održavanje skupo, a kao sporedni efekat nastaje celokupna industrija „programskih metoda“.

Alternativa modelovanju mašine je modelovanje problema koji pokušavate da rešite. Rani jezici kao LISP i APL odražavali su pojedine predstave o svetu („Svi problemi se na kraju svode na liste" ili „Svi problemi su algoritamske prirode"). Prolog prebacuje sve probleme u korake odlučivanja. Stvoreni su jezici zasnovani na ograničenom programiranju i programiranju isključivo manipulacijom grafičkim simbolima (što se pokazalo kao previše restriktivno). Svaki ovaj pristup može biti dobro rešenje za određenu klasu problema kojoj su namenjeni, ali kada istupite iz tog domena, oni postaju nezgrapni.

Objektno orijentisani pristup ide korak dalje, obezbeđujući alate pomoću kojih programer predstavlja elemente u prostoru problema. Ovo predstavljanje u principu ne ograničava programera na jednu vrstu problema. Elemente u prostoru problema i njihovo predstavljanje u prostoru rešenja nazivamo „objekti". (Trebaće vam i drugi objekti koji nemaju svoj par u prostoru problema.) Ideja je da se programu dozvoli da se prilagodi nerazumljivom jeziku problema tako što će se dodati novi tipovi objekata, te kada čitate kôd koji opisuje rešenje, u isto vreme čitate i reči koje izražavaju problem. Ovo je mnogo fleksibilnija i snažnija apstrakcija od prethodne.[1] Stoga OOP dozvoljava da opišete problem iz ugla problema, umesto iz ugla računara na kome će se to rešenje izvršavati. Još uvek postoji povratna veza ka računaru: svaki objekat izgleda posve kao mali računar – on ima unutrašnje stanje i operacije koje možete zahtevati da izvrši. Međutim, ovo i nije tako loša analogija sa objektima u stvarnom svetu – svi imaju svoje karakteristike i osobeno se ponašaju.

Alan Kay je naveo pet osnovnih obeležja Smalltalka, prvog uspešnog objektno orijentisanog jezika, i jednog od jezika na kome je Java zasnovana. Ta obeležja predstavljaju čist i neiskvaren pristup objektno orijentisanom programiranju.

1. **Sve je objekat.** Posmatrajte objekat kao poboljšanu promenljivu; on čuva podatke, ali možete i da mu „postavite zahteve" koje ispunjava vršeći operacije nad tim podacima. Teoretski, možete uzeti bilo koju idejnu komponentu problema koji rešavate (pse, zgrade, usluge itd.) i predstaviti je kao objekat u svom programu.
2. **Program je skup objekata koji jedni drugima porukama saopštavaju šta da rade.** Da biste uputili zahtev objektu, vi „šaljete poruku" tom objektu. Konkretnije, možete zamisliti da je poruka zahtev da se pozove metoda koja pripada određenom objektu.
3. **Svaki objekat ima svoj memorijski prostor koji se sastoji od drugih objekata.** Drugačije rečeno, vi stvarate novu vrstu objekta praveći paket koji sadrži neke postojeće objekte. Stoga možete da usložnjavate program koji će biti skriven iza jednostavnih objekata.
4. **Svaki objekat ima tip.** Stručno rečeno, svaki objekat je *instanca (primerak)* neke *klase*, pri čemu su „klasa" i „tip" sinonimi. Najvažnija odlika klase glasi: „Koje poruke joj možete poslati?"

[1] Neki autori programskih jezika su smatrali da objektno orijentisano programiranje nije dovoljno da omogući lako rešavanje svih programskih problema, pa su podržavaliu kombinovanje različitih pristupa kroz *multistandardne* programske jezike. Pogledajte *Multiparadigm Programming in Leda*, Timothy Budd (Addison-Wesley 1995).

5. **Svi objekti određenog tipa mogu da primaju iste poruke.** Ovo je, u stvari, višeznačna izjava, kao što ćete kasnije videti. Kako je objekat tipa „krug" istovremeno objekat tipa „oblik", krug će zasigurno moći da prima poruke za oblik. To znači da možete da napišete kôd koji komunicira sa oblicima i automatski podržava i sve drugo što potpada pod opis oblika. Ova *zamenljivost* je jedna od najmoćnijih osobina OOP-a.

Booch daje još kraću definiciju objekta:

Objekat ima stanje, ponašanje i identitet.

To znači da objekat može imati interne podatke (koji definišu njegovo stanje) i metode (koje definišu njegovo ponašanje), i da je svaki objekat jedinstven (razlikuje se od svih drugih objekata) – konkretno, svaki objekat ima jedinstvenu adresu u memoriji.[2]

Objekat ima interfejs

Aristotel je verovatno prvi počeo da pažljivo proučava podelu na *tipove* – govorio je o „klasi riba i klasi ptica". Ideja da svi objekti, premda jedinstveni, istovremeno pripadaju klasi objekata sa zajedničkim karakteristikama i ponašanjem, direktno je iskorišćena u prvom objektno orijentisanom jeziku, Simula-67. Njegova osnovna rezervisana reč **class** uvodi novi tip u program.

Simula, kao što joj ime govori, projektovana je za razvoj simulacija poput klasičnog problema „bankarskog blagajnika". U tom problemu imate više blagajnika, klijenata, računa, transakcija i novčanih jedinica – puno „objekata". Objekti koji su identični po svemu osim po stanju tokom izvršenja programa, grupisani su u „klase objekata" i odatle potiče rezervisana reč **class**. Stvaranje apstraktnih tipova podataka (klasa) osnovna je ideja u objektno orijentisanom programiranju. Apstraktni tipovi podataka rade gotovo isto kao ugrađeni tipovi: možete stvarati promenljive datog tipa (koje se nazivaju *objekti* ili *instance* u terminologiji objektno orijentisanog programiranja) i raditi s tim promenljivama (što se naziva *slanje poruka* ili *zahteva*: vi pošaljete poruku a objekat sâm odredi šta će s njom da uradi). Članovi (elementi) svake klase imaju neke zajedničke osobine: svaki račun ima saldo, svaki blagajnik može da primi depozit itd. Istovremeno, svaki član ima vlastito stanje: svaki račun ima drugačiji saldo, svaki blagajnik ima ime. Stoga blagajnici, klijenti, računi, transakcije itd., pojedinačno mogu biti predstavljeni jedinstvenim entitetom u računarskom programu. Ti entiteti su objekti, a svaki objekat pripada određenoj klasi koja definiše njegove karakteristike i ponašanje.

Dakle, iako mi u objektno orijentisanom programiranju stvaramo nove tipove podataka, svi objektno orijentisani programski jezici koriste rezervisanu reč „class". Kada vidite reč „tip", pomislite na „klasu" i obrnuto.[3]

---

[2] Ovo je zapravo previše usko, pošto objekti mogu postojati u raznim računarima i adresnim prostorima, a mogu biti snimljeni i na disk. U tim slučajevima, identitet objekta se mora utvrditi na neki drugi način, a ne pomoću njegove adrese u memoriji.

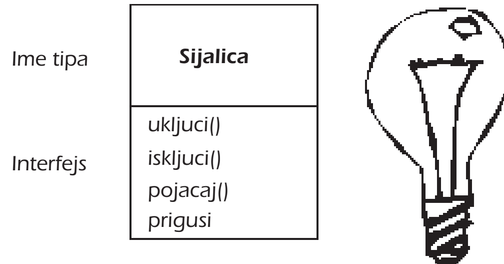
[3] Neki ljudi prave razliku i tvrde da tip određuje interfejs, dok je klasa posebna realizacija tog interfejsa.

Pošto klasa opisuje skup objekata koji imaju identične karakteristike (elementi s podacima) i jednako se ponašaju (funkcionalnost), klasa je zaista tip podatka, jer i broj u formatu pokretnog zareza, na primer, takođe ima skup karakteristika i ponašanja. Razlika je u tome što programer definiše klasu koja odgovara problemu, umesto da bude prinuđen da koristi postojeći tip koji predstavlja memorijsku jedinicu unutar računara. Programski jezik proširujete dodavanjem novih tipova podatka, specifičnih za vaše potrebe. Programski sistem prihvata nove klase, o njima vodi računa i proverava tipove, kao što radi i sa ugrađenim tipovima.

Objektno orijentisan pristup ne ograničava se samo na pravljenje simulacija. Bez obzira na to da li se slažete sa stavom da je svaki program simulacija sistema koji projektujete, upotrebom tehnika, OOP-a veliki skup problema lako se može da svesti na jednostavna rešenja.

Kada se klasa ustanovi, možete da napravite koliko god želite objekata te klase, a zatim da radite s tim objektima kao da su elementi u problema koji pokušavate da rešite. Zaista, jedan od izazova objektno orijentisanog programiranja jeste ostvarivanje jednoznačnog preslikavanja između elemenata u prostoru problema i objekata u prostoru rešenja.

Kako da naterate objekat da uradi koristan posao? Mora postojati način da postavite zahtev objektu da nešto uradi, na primer, da završi transakciju, iscrta nešto na ekranu ili uključi prekidač. A svaki objekat može da zadovolji samo izvesne zahteve. Zahtevi koje možete da postavite objektu definisani su preko njegovog *interfejsa*, koji je određen tipom. Prost primer bi mogla biti sijalica:




```
Sijalica sj = new Sijalica();
sj.ukljuci();
```

Interfejs određuje koje zahteve možete da postavite određenom objektu. Naravno, negde mora postojati kôd koji će zadovoljiti taj zahtev. Taj kôd, zajedno sa skrivenim podacima, čini *implementaciju* (realizaciju ili primenu interfejsa). S tačke gledišta proceduralnog programiranja to i nije tako komplikovano. Tip uz svaki mogući zahtev ima pridruženu metodu i kada napravite određeni zahtev, poziva se odgovarajuća metoda. Za ovaj proces obično se kaže da smo „poslali poruku“ (postavili zahtev) objektu, a objekat određuje šta će s porukom da uradi (izvrši kôd).

U našem primeru, ime tipa/klase je **Sijalica**, ime posebnog objekta tipa **Sijalica** je **sj**, a zahtevi koje možemo postaviti objektu **Sijalica** su uključi se, isključi se, pojačaj svetlo ili priguši svetlo. Objekat tipa **Sijalica** pravite kada definišete *referencu* (**sj**) na dati objekat i

pozovete **new**, čime zahtevate novi objekat tog tipa. Da biste poslali poruku objektu, navedite ime objekta i povežite ga s porukom – zahtevom, razdvajajući ih tačkom. S tačke gledišta korisnika unapred definisane klase, to je skoro sve što vam treba da biste programirali sa objektima.

Gornja slika je napravljena u skladu s formatom koji koristi *unifikovani jezik za modelovanje* (engl. *Unified Modeling Language, UML*). Svaka klasa je predstavljena pravougaonikom, ime tipa je u gornjem delu, *podaci članovi* koje treba opisati nalaze se u srednjem delu, a u donjem delu pravougaonika su *metode* (funkcije koje pripadaju tom objektu, koje primaju poruke koje šaljete tom objektu). Često se u UML dijagramima prikazuju samo ime klase i javne metode, a srednji deo ne – tako je i na prethodnoj slici. Ako vas zanima samo ime klase, nema potrebe da prikazujete ni donji deo.

Objekat pruža usluge

Dok pokušavate da razvijete ili shvatite strukturu nekog programa, bilo bi dobro da objekte smatrate „davaocima usluga". I vaš program će pružati usluge korisniku, i to pomoću usluga koje pružaju drugi objekti. Vaš cilj je da napravite (ili još bolje, pronađete u bibliotekama koda) skup objekata koji pružaju idealne usluge koje rešavaju vaš problem.

To biste mogli postići ako se zapitate: „Da ih mogu čarobnim štapićem izvući iz šešira, koji objekti bi odmah rešili moj problem?" Na primer, pretpostavimo da pišete program za knjigovodstvo. Mogli biste zamisliti određene objekte koji sadrže unapred definisane slike ekrana za unos podataka, drugi skup objekata koji obavljaju knjigovodstvene proračune, i objekat koji štampa čekove i račune na svim mogućim vrstama štampača. Neki od tih objekata možda već postoje? Kako bi izgledali oni koji ne postoje? Koje usluge bi *ti* objekti davali i koji bi im objekti bili potrebni za izvršavanje zadataka? Ako tako budete radili, doći ćete do tačke kada možete reći: „Ovaj objekat izgleda dovoljno jednostavno da se može napisati" ili „Mora da postoji ovakav objekat". To je racionalan način razlaganja problema na skup objekata.

Kada se objekat smatra davaocem usluga, stiče se još jedna prednost: time se povećava usklađenost objekta. *Velika usklađenost* je jedan od temeljnih kvaliteta projektovanja softvera: to znači da su razni aspekti softverske komponente (kao što je objekat, iako isto važi i za metodu ili biblioteku objekata) međusobno „dobro uklopljeni". Prilikom projektovanja objekata, programeri često trpaju previše funkcionalnosti u jedan objekat. Kada se radi o pomenutom modulu za štampanje čekova, možete odlučiti da vam treba objekat koji zna sve o formatiranju i štampanju. Verovatno će vas iskustvo naučiti da je to previše za jedan objekat i da vam treba tri ili više objekata. Jedan objekat bi mogao biti katalog svih mogućih izgleda čekova, kojem se mogu slati upiti radi podataka o tome kako odštampati određeni ček. Jedan objekat ili skup objekata može biti opšti (generički) interfejs za štampanje, koji zna sve o raznim vrstama štampača (ali ništa o knjigovodstvu – taj je kandidat za kupovinu, umesto da ga sami pišete). A treći objekat može koristiti usluge prethodna dva da bi obavio posao. Tako bi svaki objekat imao usklađen skup usluga koje pruža. U dobrom objektno orijentisanom dizajnu, svaki objekat dobro radi jedan posao, ali ne pokušava da radi više poslova. Ne samo da se tako mogu pronaći objekti koje treba kupiti (recimo, objekat interfejsa štampača), nego se proizvode i novi objekti koji se mogu ponovo upotrebljavati na drugim mestima (katalog izgleda čekova).

Jako ćete pojednostaviti sebi život ako objekte budete smatrali davaocima usluga. To će koristiti ne samo vama tokom procesa projektovanja, nego i drugima koji budu pokušavali da razumeju vaš kôd ili upotrebe neki od vaših objekata. Ako budu mogli da utvrde vrednost objekta na osnovu usluge koje on pruža, biće im mnogo lakše da ga uklope u svoje projekte.

Skrivena realizacija

Vrlo je korisno podeliti polje delatnosti na *autore klasa* (one koji prave nove tipove podataka) i *programere klijente*[4] (korisnike klasa koji te tipove upotrebljavaju u svojim aplikacijama). Cilj programera klijenta je da sakupi klase u „kutiju sa alatom", koju će koristiti za brzi razvoj aplikacija. Cilj autora klasa je da napravi klasu koja otkriva samo ono što je neophodno programeru klijentu, a sve ostalo drži sakriveno. Zašto? Programer klijent ne može da koristi sakrivene delove, što znači da autor klase može da promeni skriveni deo kad god hoće, ne razmišljajući da li će se to odraziti na ostale. Skriveni deo obično predstavlja osetljivu unutrašnjost objekta koju lako može da ošteti neobazriv ili neobavešten programer klijent, pa sakrivanje realizacije smanjuje mogućnost da se pojave greške u programima.

U svakom odnosu je važno da sve uključene strane poštuju granice. Kada pravite biblioteku, vi stvarate odnos sa klijentom koji je takođe programer, ali koji sastavlja aplikaciju koristeći vašu biblioteku, da bi, možda, napravio veću biblioteku.

Ako bi svi članovi klase bili dostupni svakome, onda bi programer klijent mogao da uradi bilo šta s klasom i ne bi postojao način da se nametnu pravila. Iako biste vi voleli da programer klijent ne radi direktno s nekim članicama vaše klase, bez kontrole pristupa ne bi postojao način da to sprečite. Sve bi bilo izloženo javnosti.

Stoga je prvi razlog za uvođenje kontrole pristupa onemogućavanje programera klijenta da pristupa delovima koje ne sme da dira, a koji su neophodni za interni rad s tipom podataka. Ti delovi nisu deo interfejsa koji je potreban korisnicima za rešavanje njihovih problema. Ovo je u isto vreme usluga programerima klijentima, jer lako mogu da razluče šta je za njih važno, a o čemu ne treba da misle.

Kontrolu pristupa treba uvesti i da bi se dozvolilo projektantu biblioteke da promeni način na koji klasa interno radi, a da ne mora da brine kako će se to odraziti na programere klijente. Na primer, možete realizovati neku klasu na jednostavan način, a kasnije otkriti da je treba ponovo napisati kako bi radila brže. Ako su interfejs i realizacija jasno razdvojeni i zaštićeni, ovo možete lako da izvedete.

Java koristi tri rezervisane reči da postavi granice unutar klase: **public**, **private** i **protected**. Ovi *specifikatori pristupa* određuju ko može da koristi definicije koje slede iza njih. **public** (javni) znači da je naredni element dostupan svakome. Rezervisana reč **private** (privatni) znači da tom elementu ne može da pristupi niko sem autora klase, unutar metoda tog tipa. **private** predstavlja zid između autora i programera klijenta. Onaj ko pokuša da pristupi članici označenoj kao **private**, izazvaće grešku prilikom prevođenja.

---

[4] Zahvaljujem svom prijatelju Scottu Meyersu na ovom terminu.

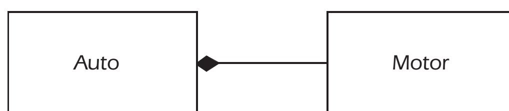
Rezervisana reč **protected** (zaštićeni) deluje kao **private**, s tom razlikom što klase naslednice imaju pristup **zaštićenim** članicama, ali ne i **privatnim** članicama. O nasleđivanju ćemo govoriti uskoro.

Java ima i „podrazumevani" pristup koji se koristi ako ne zadate nijedan od navedenih specifikatora. Ovo se obično naziva *paketski pristup*, jer klase mogu da pristupe članicama drugih klasa u istom paketu, ali izvan tog paketa te iste članice vide se kao **privatne**.

Ponovno korišćenje realizacije

Kada se klasa napravi i testira, ona bi (u idealnom slučaju) trebalo da predstavlja korisnu jedinicu koda. Ispostavlja se da ponovno korišćenje ni izbliza nije tako lako ostvarljivo kako se mnogi nadaju; potrebno je iskustvo i pronicljivost da bi se napisao višekratno upotrebljiv objekat. Ali kada ga napišete, on moli da bude ponovo iskorišćen. Ponovno korišćenje koda je jedna od najvećih prednosti objektno orijentisanih programskih jezika.

Najjednostavniji način da ponovo iskoristite klasu jeste da direktno koristite objekat te klase; međutim, objekat te klase možete da stavite i u novu klasu. Ovo se naziva „pravljenje objekta člana" (engl. *member object*). Vaša nova klasa može biti sastavljena od ma koliko drugih objekata, ma kojeg tipa i u bilo kojoj kombinaciji koja vam je potrebna da biste postigli željenu funkcionalnost svoje nove klase. Novu klasu sastavljate od postojećih klasa, što se naziva *kompozicija* (ako se dešava dinamički, onda obično *agregacija*). Kompozicija se često poredi s relacijom „ima", na primer „auto ima motor".



(Na ovom UML dijagramu kompoziciju označava popunjen romb, koji kazuje da postoji jedan auto. Kad označavam spajanje, obično ću koristiti jednostavniji oblik: samo liniju, bez romba.)[5]

Kompozicija je vrlo fleksibilna. Objekti članovi vaše nove klase obično su privatni, što ih čini nedostupnim programerima klijentima koji koriste klasu. Ovo omogućava da izmenite te članove ne remeteći postojeći klijentski kôd. Možete menjati objekte članove i u vreme izvršavanja da biste dinamički menjali ponašanje svog programa. Nasleđivanje koje malo dalje opisujemo, nema ovu fleksibilnost pošto prevodilac mora da ugradi ograničenja za prevođenje u klase stvorene nasleđivanjem.

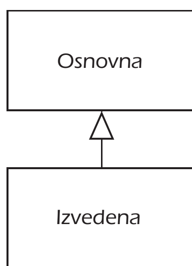
Pošto je nasleđivanje veoma bitno, u objektno orijentisanom programiranju često se veoma naglašava, i nov programer može pomisliti da nasleđivanje treba koristiti svuda. Kao rezultat može se dobiti veoma nezgrapan i veoma složen program. Umesto toga, prvo treba videti da li pri pravljenju novih klasa može da se iskoristi kompozicija, pošto je ona jednostavnija i fleksibilnija. Ako primenite ovaj pristup, program će biti čistiji. Kada steknete nešto iskustva, biće vam očigledno kada treba da koristite nasleđivanje.

[5] Ovo je u principu dovoljno detaljno za većinu dijagrama i ne treba da precizirate da li koristite agregaciju ili kompoziciju

Nasleđivanje

Sam po sebi, koncept objekta je veoma koristan. On omogućava da podatke i funkcionalnost grupišete po *konceptu*, tako da možete predstaviti odgovarajuću ideju u prostoru problema, umesto da budete prinuđeni da koristite izraze računara na kome radite. Kada se primeni rezervisana reč **class**, te ideje su izražene kao osnovne jedinice u ovom programskom jeziku.

Bila bi šteta da se namučite i napravite neku klasu a da onda budete primorani da pravite potpuno novu klasu koja ima sličnu funkcionalnost. Bilo bi jednostavnije da uzmemo postojeću klasu, da je kloniramo, a zatim dopunjujemo i menjamo kloniranu klasu. Ovo se zapravo postiže *nasleđivanjem* (engl. *inheritance*), osim kada se originalna klasa (koja se naziva *osnovna* klasa ili *natklasa* ili *klasa roditelj*) izmeni, a te izmene se odraze i na „klonu" (koji se naziva *izvedena* ili *nasleđena* klasa ili *potklasa* ili klasa *naslednik*; engl. *derived class).*



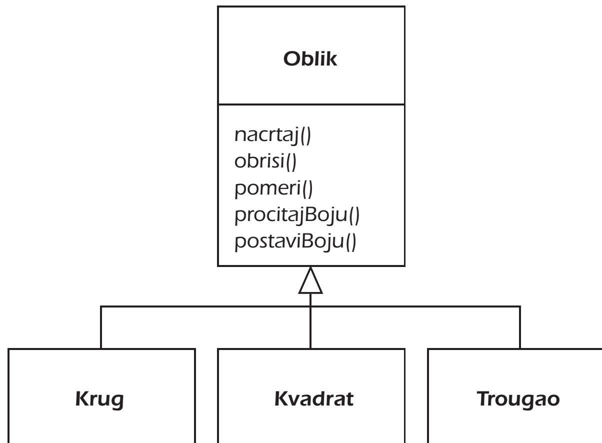
(Strelica u ovom UML dijagramu polazi od izvedene klase ka osnovnoj klasi. Kao što ćete videti, može da postoji više izvedenih klasa.)

Tip ne označava samo ograničenja skupa objekata; on ima odnose i s drugim tipovima. Dva tipa mogu imati neke zajedničke karakteristike i jednako se ponašati, ali pri tom jedan tip može imati još neke dodatne karakteristike i može obrađivati više poruka (ili ih drugačije obrađivati). Kod nasleđivanja, ova sličnost između tipova se izražava preko koncepta osnovnih i izvedenih tipova. Osnovni tip sadrži sve karakteristike i ponašanja koja su zajednička za tipove izvedene iz njega. Praveći osnovni tip, izražavate suštinu svojih ideja o nekim objektima u sistemu. Iz osnovnog tipa izvodite ostale tipove i pokazujete različite načine realizacije ove suštine.

Na primer, mašina za recikliranje raspoređuje komadiće otpada. Osnovni tip je „otpad", a svaki pojedini otpadak ima težinu, vrednost itd. i može biti isečen, istopljen ili rastavljen. Iz osnovnog tipa izvodimo posebne vrste otpada s dodatnim karakteristikama (boca ima boju), ili ponašanjima (aluminijumska konzerva može da se zdrobi, čelične konzerve privlači magnet). Ponašanja mogu biti različita (vrednost papira zavisi od njegove vrste i stanja). Koristeći nasleđivanje možete izgraditi hijerarhiju tipova. Ta hijerarhija opisuje problem koji pokušavate da rešite preko tipova koji se u problemu pojavljuju.

Drugi primer je klasičan „oblik" koji može da se koristi u CAD sistemu (engl. *Computer-Aided Design*) ili u nekoj kompjuterskoj igri. Osnovni tip je „oblik" a svaki oblik ima veličinu, boju, poziciju itd. Svaki oblik može da se iscrta, obriše, pomera, oboji itd. Odavde izvodimo (nasleđujemo) pojedine vrste oblika – krug, kvadrat, trougao i druge – od

kojih svaki može imati dodatne karakteristike i ponašanje. Neki oblici, na primer, mogu da se okreću. Neka ponašanja mogu biti drugačija, recimo kada želite da izračunate površinu oblika. Hijerarhija tipova objedinjuje sličnosti i razlike između oblika.

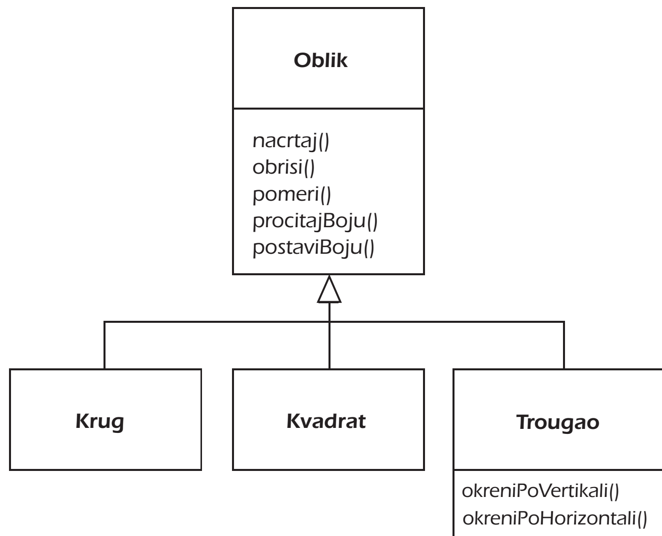


Predstavljanje rešenja i problema u istom obliku veoma je korisno jer vam ne treba veliki broj međumodela da biste od opisa problema došli do opisa rešenja. Kod objekata, hijerarhija tipova je osnovni model tako da sa opisa sistema u realnom svetu direktno prelazite na opis sistema kodom. Jedna od teškoća sa objektno orijentisanim programiranjem jeste ta da je previše jednostavno stići od početka od kraja. Um treniran da traži složena rešenja često spočetka zbunjuje ova jednostavnost.

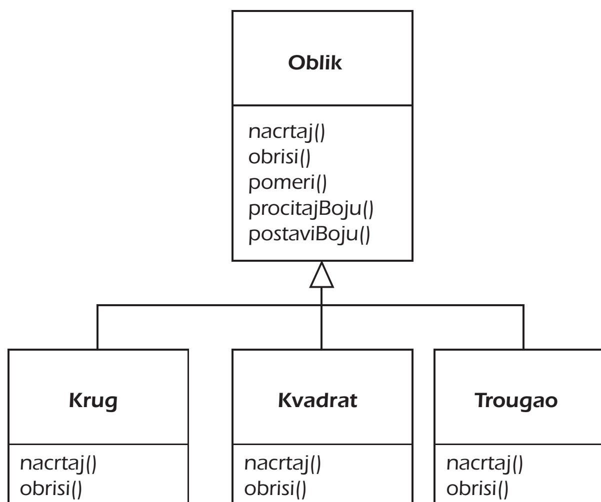
Kada iskoristite nasleđivanje iz postojećeg tipa, stvarate novi tip. Taj novi tip ne samo da sadrži sve članice postojećeg tipa (iako su **privatne** članice sakrivene i nedostupne), već, što je mnogo važnije, kopira interfejs osnovne klase. Znači, sve poruke koje možete da pošaljete objektima osnovne klase, takođe možete poslati i objektima izvedene klase. Pošto se tip klase određuje na osnovu poruka koje joj možemo poslati, to znači da je izvedena klasa *istog tipa kao* i *osnovna klasa.* U prethodnom primeru „krug je oblik". Ekvivalencija tipova dobijena nasleđivanjem je osnovni korak na putu ka razumevanju smisla objektno orijentisanog programiranja.

Pošto i osnovna klasa i izvedena klasa imaju isti osnovni interfejs, mora da postoji realizacija koja ide uz taj interfejs. Znači, mora postojati kôd koji se izvršava kada objekat primi određenu poruku. Ako nasledite klasu i ništa više ne uradite, metode interfejsa osnovne klase će preći i u izvedenu klasu. To znači da objekti izvedene klase imaju i isti tip i isto ponašanje, što i nije previše korisno.

Postoje dva načina da napravite razliku između svoje nove izvedene klase i osnovne klase. Prvi je jasan: izvedenoj klasi dodate potpuno nove metode. Te nove metode nisu deo interfejsa osnovne klase, što znači da osnovna klasa nije radila sve ono što ste hteli, pa ste joj zato dodali još metoda. Ovaj jednostavan način korišćenja nasleđivanja ponekad je savršeno rešenje problema. Međutim, pažljivo proučite da li su i vašoj osnovnoj klasi potrebne te dodatne metode. Ovaj proces otkrivanja i iteracije programa redovna je pojava u objektno orijentisanom programiranju.



Iako nasleđivanje ponekad može da nagovesti (posebno u Javi, gde rezervisana reč koja označava nasleđivanje glasi **extends** – proširuje) kako ćete dodavati nove metode interfejsu, to nije uvek istina. Drugi i važniji način da napravite razliku u novoj klasi jeste da *promenite* ponašanje postojeće metode osnovne klase. To se naziva *redefinisanje* metode (engl. *overriding*).

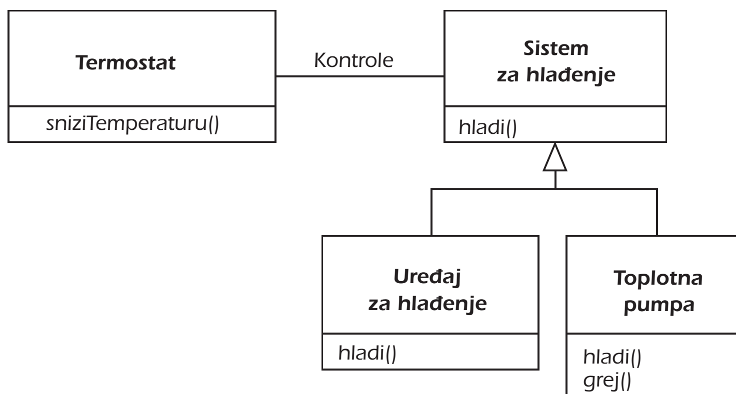


Da biste redefinisali metodu, napravite novu definiciju te metode u izvedenoj klasi. Vi kažete: „Koristim ovde istu metodu interfejsa, ali želim da ona u mom novom tipu radi nešto drugo."

Relacije JE i JE-KAO

Može se postaviti jedno pitanje o nasleđivanju: „Da li nasleđivanje treba da redefiniše *samo* metode osnovne klase (i da ne dodaje nove metode koje ne postoje u osnovnoj klasi)?" To bi značilo da je izvedena klasa *potpuno istog* tipa kao osnovna klasa, jer ima istovetan interfejs. Kao rezultat, objekat izvedene klase može se svesti na tip objekta osnovne klase. Ovo je tzv. *čista supstitucija*, ili *princip supstitucije*. Unekoliko, ovo je idealan način primene nasleđivanja. U ovom slučaju, mi se često pozivamo na relaciju između osnovne i izvedene klase kao na relaciju *je*, jer možemo reći „krug *je* oblik". Pokušaj da utvrdimo da li možemo da uspostavimo relaciju *je* između klasa, a da to ima smisla, predstavlja svojevrstan test nasleđivanja.

Ponekad morate da dodate nove elemente u interfejs izvedenog tipa i time proširite postojeći interfejs. Novi tip i dalje može biti sveden na osnovni tip, ali supstitucija nije savršena, jer nove metode nisu dostupne iz osnovnog tipa. Ovo se može opisati kao relacija *je-kao* (moj termin). Novi tip ima interfejs starog tipa ali sadrži i druge metode, tako da se ne može reći kako je potpuno isti. Uzmimo za primer uređaj za klimatizaciju. Pretpostavimo da su po kući razvedene kontrole za hlađenje, odnosno, kuća ima interfejs koji omogućava da kontrolišete hlađenje. Zamislite da se uređaj za klimatizaciju pokvari i da ga zamenite toplotnom pumpom koja može i da greje i da hladi. Toplotna pumpa *je kao* uređaj za hlađenje, ali ona može i više. Pošto je kontrolni sistem vaše kuće projektovan samo da kontroliše hlađenje, on je ograničen na komunikaciju s delom za hlađenje novog objekta. Interfejs novog objekta je proširen, a postojeći sistem ne poznaje ništa drugo osim originalnog interfejsa.



Čim proučite ovaj plan, postaće jasno da osnovna klasa „sistem za hlađenje" nije dovoljno opšta i da treba da se preimenuje u „sistem za kontrolu temperature", kako bi mogla da sadrži i grejanje – nakon čega bi princip supstitucije važio. Ovaj dijagram prikazuje šta u stvarnom svetu može da se desi tokom projektovanja.

Kada upoznate princip supstitucije, lako možete pomisliti da je taj pristup (čista supstitucija) jedini način da se nešto uradi, i zaista *jeste* dobro da program napravite na taj način. Ali shvatićete da u interfejs izvedene klase ponekad morate da dodate nove metode. Nakon što ispitate problem, trebalo bi da bude prilično očigledno o kojem se od ta dva slučaja radi.

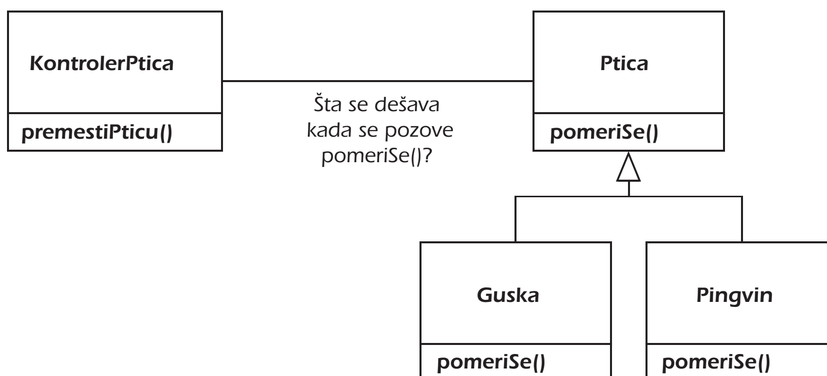
Virtuelizacija objekata preko polimorfizma

Ukoliko imate posla s hijerarhijom tipova, često objekat ne tretirate kao specifičan tip, već kao osnovni tip. To vam omogućava da pišete kôd koji ne zavisi od specifičnog tipa. U prethodnom primeru sa oblicima, metode manipulišu generičkim tipovima (oblicima) bez obzira na to da li se radi o krugovima, kvadratima, trouglovima ili čak oblicima koji još nisu definisani. Svi oblici mogu biti iscrtani, obrisani i pomerani, tako da te metode samo pošalju poruku objektu tipa oblik i ne brinu kako će objekat izaći na kraj s tom porukom.

Takav kôd se ne menja ni nakon dodavanja novih tipova, a dodavanjem novih tipova najčešće se proširuju objektno orijentisani programi kada treba da se savlada neka nova situacija. Na primer, možete da izvedete novi podtip oblika, pod imenom petougao, bez menjanja metoda koje rade samo s generičkim oblicima. Mogućnost da se program lako proširi izvođenjem novih podtipova jeste jedan od osnovnih načina da se kapsuliraju promene. Time se znatno unapređuje program i istovremeno smanjuju troškovi održavanja softvera.

Problem nastaje pri pokušaju da se objekti izvedenog tipa tretiraju kao generički osnovni tipovi (krugovi kao oblici, bicikli kao vozila, kormorani kao ptice itd.). Ako metoda naredi generičkom obliku da se iscrta ili generičkom vozilu da smota volan ili generičkoj ptici da se pomeri, prevodilac prilikom prevođenja ne može tačno da zna koji deo koda će biti izvršen. To i jeste poenta – kada je poruka poslata, programer ne *želi* da zna koji deo koda će biti izvršen. Metoda za crtanje može podjednako da se primeni na krug ili na kvadrat ili na trougao, a objekat će izvršiti odgovarajući kôd u zavisnosti od svog specifičnog tipa.

Ako ne morate znati koji će deo koda biti izvršen, dodajte nov podtip; kôd koji će taj novi tip izvršavati može biti drugačiji a da pri tom ne morate ništa da menjate u metodi koja ga poziva. Znači, prevodilac ne zna tačno koji će deo koda biti izvršen. Šta onda radi? Na primer, u sledećem dijagramu objekat **KontrolerPtica** radi samo s generičkim objektima tipa **Ptica** i ne zna kog su oni tipa. Ovo je pogodno iz perspektive **KontroleraPtica** jer ne treba pisati poseban kôd koji bi odredio s kojim se tačno tipom **Ptice** radi, ili kakvo je ponašanje te određene **Ptice**. Kako se događa da se, kada se pozove metoda **pomeriSe**() uz ignorisanje specifičnog tipa **Ptice**, primeni pravilno ponašanje (**Guska** hoda, leti ili pliva, a **Pingvin** hoda ili pliva)?



Odgovor leži u osnovnoj začkoljici objektno orijentisanog programiranja: prevodilac ne može da pozove funkciju na tradicionalan način. Prevodilac koji nije objektno orijentisan izaziva *rano vezivanje* (engl. *early binding*). Ovaj termin možda niste ranije čuli, jer o pozivanju funkcija niste razmišljali ni na koji drugi način. To znači da prevodilac generiše poziv funkcije određenog imena, a izvršni sistem kasnije razreši ovaj poziv ubacivanjem apsolutne adrese koda koji treba da se izvrši. U OOP-u, program ne može da odredi adresu koda sve do trenutka izvršavanja, tako da je neophodna neka druga šema kada se poruka šalje generičkom objektu.

Da bi rešili ovaj problem, objektno orijentisani jezici koriste koncept *kasnog vezivanja* (engl. *late binding*). Kada pošaljete poruku objektu, kôd koji se poziva ne biva određen sve do trenutka izvršavanja. Prevodilac ipak proverava da li telo specifične metode postoji te koji su tipovi argumenata i povratne vrednosti, ali ne zna koji će kôd izvršiti.

Da bi razrešila kasno povezivanje, umesto apsolutnog poziva Java koristi specijalan delić koda koji adresu tela metode izračunava pomoću informacija ugrađenih u konkretan objekat. (Ovaj proces je detaljno opisan u poglavlju o polimorfizmu.) Tako svaki objekat može da se ponaša različito, u skladu sa sadržajem svakog pojedinog specijalnog delića koda. Kada pošaljete poruku objektu, on sam odlučuje šta će s njom da uradi.

U nekim jezicima morate izričito navesti da određenu metodu treba kasno povezati (u jeziku C++ to se radi pomoću rezervisane reči **virtual**). U tim jezicima, metode se podrazumevano *ne* povezuju dinamički. U Javi se dinamičko povezivanje podrazumeva, tako da ne morate dodavati posebne rezervisane reči da biste dobili polimorfizam.

Posmatrajmo primer oblika. Dijagram porodice klasa (koje su zasnovane na istom uniformnom interfejsu) dat je ranije u ovom poglavlju. Da bismo prikazali polimorfizam, napisaćemo naredbe koje ignorišu specifične detalje tipova i obraćaju se samo osnovnoj klasi. Te naredbe *nisu povezane* sa informacijama specifičnim za pojedine tipove, pa se stoga jednostavnije pišu i lakše razumevaju. Ako pomoću nasleđivanja dodamo nov tip – **Sestougaonik**, na primer – te naredbe radiće jednako dobro s novim tipom **Oblika** kao što su radile sa već postojećim tipovima. Znači, program je *proširiv.*

Ako napišete metodu u Javi (što ćete uskoro naučiti):

```
void radiNesto(Oblik oblik) {
oblik.obrisiSe();
// ...
oblik.iscrtajSe();
}
```

ta metoda se obraća bilo kom **Obliku**, pa ne zavisi od specifičnog tipa objekta koji se iscrtava i briše. Ako u nekom drugom delu programa pozovemo funkciju **radiNesto**():

```
Krug krug = new Krug();
Trougao trougao = new Trougao();
Linija linija = new Linija();
radiNesto(krug);
radiNesto(trougao);
radiNesto(linija);
```

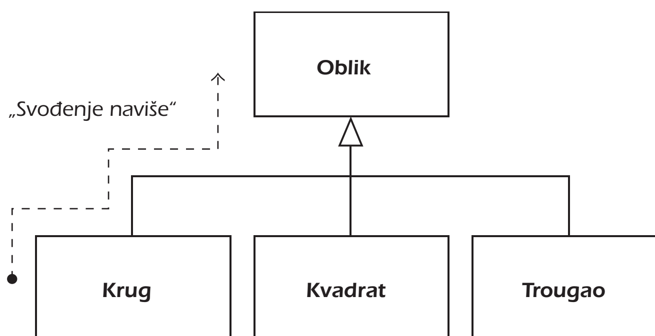
pozivi metode **radiNesto**() automatski rade ispravno, bez obzira na tačan tip objekta.

Ovo je prilično zadivljujući trik. Posmatrajte red:

```
radiNesto(krug);
```

Ovde je **Krug** prosleđen metodi koja očekuje **Oblik**. Pošto **Krug** *jeste* **Oblik**, funkcija **radiNesto**() može tako da ga tretira. Odnosno, bilo koju poruku koju metoda **radiNesto**() može da pošalje **Obliku**, **Krug** može da prihvati. Zato je gornji poziv potpuno siguran i logičan.

Proces pri kome izvedeni tip tretiramo kao osnovni tip, nazivamo *svođenje naviše* (engl. *upcasting*). Ime *cast* je iskorišćeno zato što označava ubacivanje u kalup (engl. *casting into a mold* – ubacivanje u kalup), a *up* potiče od načina na koji se obično organizuje dijagram nasleđivanja, sa osnovnim tipom na vrhu i izvedenim klasama koje se lepezasto šire naniže. Prema tome, konverzija u osnovni tip je penjanje uz dijagram nasleđivanja: svođenje naviše ili *upcasting*.



U objektno orijentisanom programu uvek postoji svođenje naviše, jer pri tom ne morate znati tačan tip s kojim radite. Pogledajte funkciju **radiNesto**():

```
oblik.obrisiSe();
// ...
oblik.iscrtajSe();
```

Obratite pažnju na to da se nigde ne govori: „Ako si **Krug**, radi ovo, ako si **Kvadrat** radi ono itd." Tako napisan kôd koji proverava sve moguće tipove koje **Oblik** može da pokrije prljav je i morate da ga menjate svaki put kada dodate novu vrstu **Oblika**. U našem slučaju, kažemo: „Ti si oblik, ja znam da možeš da se iscrtaš i obrišeš, to i uradi, i sam vodi računa o detaljima".

Važno je da se naredbe u funkciji **radiNesto**() izvršavaju na pravi način. Sam poziv metode **iscrtajSe**() za **Krug** prouzrokuje izvršavanje drugačijeg koda nego kada pozivamo metodu **iscrtajSe**() za **Kvadrat** ili **Liniju**; ali kada poruku **iscrtajSe**() pošaljemo nepoznatom **Obliku**, dobija se ispravno ponašanje, zasnovano na stvarnom tipu **Oblika**. Ovo je dobra osobina, kao što je ranije pomenuto, jer kada prevodilac prevodi metodu **radiNesto**(), on ne zna tip s kojim radi. Stoga bismo obično očekivali da on pozove verziju metoda **obrisiSe**() i **iscrtajSe**() za osnovnu klasu **Oblik**, a ne za određeni **Krug**, **Kvadrat**

ili **Liniju**. Zbog polimorfizma se sve ipak ispravno odvija. Prevodilac i sistem za izvršavanje vode računa o svim detaljima. Zasad je dovoljno da znate da polimorfizam funkcioniše i kako da pišete programe koristeći taj pristup. Kada pošaljete poruku objektu, objekat će uraditi šta treba, čak i kada treba svoditi naviše.

Hijerarhija s jedinstvenim korenom

Jedno od pitanja u OOP-u koje je postalo veoma značajno od uvođenja C++-a glasi: da li sve klase treba da budu izvedene iz jedinstvene osnovne klase. U Javi (i gotovo svim ostalim OOP jezicima) odgovor je potvrdan. Ova osnovna klasa se naziva **Object**. Ispostavlja se da postoje brojne prednosti hijerarhije s jedinstvenim korenom.

Svi objekti u hijerarhiji s jedinstvenim korenom imaju zajednički interfejs i, prema tome, svi su istog osnovnog tipa. Postoji i druga mogućnost (koju nudi C++), da svi objekti ne budu istog osnovnog tipa. Po vertikalnoj kompatibilnosti ovo više odgovara modelu jezika C i može se smatrati manje restriktivnim; ali kada želite da se bavite objektno orijentisanim programiranjem u celini, morate napraviti svoju hijerarhiju da biste obezbedili istu fleksibilnost koja je već ugrađena u druge OOP jezike. Takođe, u svakoj novoj biblioteci klasa koju nabavite, biće korišćen neki drugi nekompatibilni interfejs. Treba uložiti napor (i verovatno višestruko nasleđivanje) da biste ugradili taj novi interfejs u svoj program. Da li je dodatna „fleksibilnost“ C++-a vredna toga? Ako vam treba – zato što ste puno uložili u C – prilično je vredna. Ukoliko počinjete od nule, drugi jezici, poput Jave, često mogu biti mnogo produktivniji.

Svi objekti u hijerarhiji s jedinstvenim korenom (kakvu obezbeđuje Java) zasigurno imaju određenu zajedničku funkcionalnost – znate da možete da izvršite određene osnovne operacije nad svakim objektom u svom sistemu. Hijerarhija s jedinstvenim korenom, uz stvaranje dinamičkih objekata, veoma pojednostavljuje prosleđivanja argumenata.

Hijerarhija sa jedinstvenim korenom olakšava realizaciju *sakupljača smeća*, što je jedna od temeljnih prednosti Jave nad jezikom C++. Pošto je pri izvršavanju u svim objektima garantovano postojanje informacija o tipu, nikada nećete naići na objekat čiji tip ne možete da utvrdite. To je naročito važno kod operacija na sistemskom nivou, kao što je obrada izuzetaka, i obezbeđuje veću fleksibilnost pri programiranju.

Kontejneri

Po pravilu, ne možete znati koliko objekata će vam biti potrebno za rešavanje nekog zadatka, niti koliko dugo oni treba da postoje u memoriji. Ne znate ni kako da uskladištite te objekte. Ako pre izvršavanja programa ne znate ni broj objekata ni njihov vek trajanja, kako da odredite količinu memorijskog prostora za njihovo skladištenje?

Rešenje većine problema u objektno orijentisanom programiranju ponekad deluje neozbiljno: stvoriti još jedan tip objekta. Novi tip objekta koji rešava pomenuti problem čuva reference na druge objekte. Naravno, to isto možete da uradite i pomoću *nizova*, koji postoje u većini jezika. Ali taj novi objekat, obično pod imenom *kontejner* (naziva se i *kolekcija*, ali pošto Javine biblioteke koriste taj termin u drugom smislu, ova knjiga će se

držati imena „kontejner“), širiće se po potrebi da bi primio sve što u njega stavite. Stoga ne morate da znate koliko ćete objekata čuvati u kontejneru. Samo napravite kontejnerski objekat i prepustite njemu da se stara o detaljima.

Srećom, dobar OOP jezik sadrži skup kontejnera kao deo paketa. U jeziku C++ on je deo standardne C++ biblioteke i često se naziva standardna biblioteka šablona (engl. *Standard Template Library, STL*). Smalltalk ima prilično dobar skup kontejnera. I Java ima mnogo kontejnera u svojoj standardnoj biblioteci. U nekim bibliotekama jedan ili dva generička kontejnera dovoljni su za sve potrebe, dok u drugim bibliotekama (na primer u Javi) postoje različiti tipovi kontejnera za različite potrebe: više različitih klasa spiskova (tipova **List**a) za čuvanje sekvenci, tipova **Map**a (koje neki nazivaju *asocijativni nizovi*) za pridruživanje jednih objekata drugima, tipova skupova (**Set**) za čuvanje po jednog primerka raznih tipova objekata i druge komponente među kojima su redovi čekanja, stabla, stekovi itd.

Sa stanovišta programa, za vas je važan kontejner s kojim možete da radite i rešite problem. Ako kontejner jednog tipa zadovoljava vaše potrebe, nema razloga da uvodite druge. Izbor treba da vam bude ponuđen iz dva razloga. Prvo, kontejneri obezbeđuju različite tipove interfejsa i načina rada. Stek ima interfejs i način rada različit od reda čekanja, koji se razlikuje od skupa ili liste. Za rešenje vašeg problema jedan od njih je obično bolji od ostalih. Drugo, različiti kontejneri obavljaju iste operacije s različitom efikasnošću. Na primer, postoje dve osnovne vrste lista: **ArrayList** i **LinkedList**. I jedna i druga su jednostavne sekvence koje mogu imati identične interfejse i, spolja gledano, načine rada. No, postoji bitna razlika u trajanju i zahtevima određenih operacija nad njima. Primera radi, nasumično pristupanje elementima kontejnera **ArrayList** je operacija s konstantnim vremenom izvršavanja; bez obzira na to koji element izaberemo, potrebno je isto vreme. Međutim, pomeranje kroz listu **LinkedList** do nasumično izabranog elementa je komplikovanje i treba više vremena da se pristupi elementima što su oni dalje od početka liste. S druge strane, ako želite da ubacite element usred sekvence, to je mnogo jednostavnije i brže u listi **LinkedList**, nego u **ArrayList**. Ove i druge operacije nisu jednako efikasne, što zavisi od strukture koja se nalazi u osnovi pojedine sekvence. U fazi pisanja programa, možete krenuti od kontejnera **LinkedList** a kada budete popravljali performanse, prebacite se na kontejner **ArrayList.** Zahvaljujući apstrakciji preko interfejsa **List**, prebacivanje iz jedne strukture u drugu imaće minimalan uticaj na vaš kôd.

Parametrizovani (generički) tipovi

Pre Jave SE5, kontejneri su mogli da sadrže samo jedini univerzalni tip elemenata koji postoji u Javi: **Object**. Hijerarhija s jedinstvenim korenom znači da je sve tipa **Object**, tako da kontejner koji može da čuva **Object**, može da čuva bilo šta.[6] Ovo pojednostavljuje ponovno korišćenje kontejnera.

Da biste koristili takav kontejner, dodajte u njega reference na objekte, a kasnije ih tražite nazad. Međutim, pošto kontejner čuva samo tip **Object**, kada dodate referencu na svoj

6 Ne može da čuva proste tipove, ali *automatsko pakovanje* (engl. *autoboxing*) koje je donela Java SE5 gotovo da potpuno uklanja to ograničenje. O tome će još biti reči u ovoj knjizi.

objekat u kontejner, ona se svodi naviše na **Object** i gubi identitet. Kada je izvadite iz kontejnera, dobijate referencu na **Object**, a ne referencu na tip koji ste stavili unutra. Kako onda da tu referencu vratite u nešto što ima konkretan tip objekta koji ste stavili u kontejner?

Ovde se ponovo koristi eksplicitna konverzija (pretvaranje) tipova, ali ovog puta se ne penjete uz hijerarhiju nasleđivanja do opštijeg tipa, već se spuštate niz hijerarhiju do određenijeg tipa. Ovaj način eksplicitne konverzije naziva se *svođenje naniže* (engl. *downcasting*). Pri svođenju naviše, na primer, znate da je **Krug** tipa **Oblik**, pa možete slobodno da izvršite svođenje naviše; međutim, neki **Object** nije obavezno **Krug** ili **Oblik**, pa svođenje naniže nije baš sigurno, osim ako znamo s čim imamo posla.

Ovo i nije previše opasno: ako izvršite pogrešno svođenje naniže, javlja se greška pri izvršavanju – *izuzetak* (engl. *exception*), o kome uskoro govorimo. Kada iz kontejnera uzimate reference, morate obezbediti neki način da zapamtite na šta se te reference odnose kako biste mogli da izvršite pravilno svođenje naniže.

Svođenje naniže i provere pri izvršavanju zahtevaju dodatno vreme za izvršavanje programa i dodatni napor za programera. Zar ne bi bilo logično da nekako napravimo kontejner tako da on zna koje tipove čuva, čime se isključuje potreba za svođenjem naniže kao i moguće greške. Ovo se rešava pomoću *parametrizovanih tipova*, a to su klase koje prevodilac automatski može da prilagodi tako da rade sa određenim tipom. Na primer, parametrizovani kontejner prevodilac bi mogao da prilagodi tako da prihvata i vraća samo klasu **Oblik**.

Jedna od velikih promena koje je donela Java SE5 jesu parametrizovani tipovi, koje u Javi nazivamo *generički tipovi*. Prepoznaćete ih po uglastim zagradama unutar kojih se navode; recimo, ovako se može napraviti **ArrayList** koji sadrži **Oblik**:

```
ArrayList<Oblik> oblici = new ArrayList<Oblik> ( );
```

Da bi se generički tipovi dobro iskoristili, izmenjene su i mnoge standardne komponente bibilioteke. Kao što ćete videti, generički tipovi su uticali na dobar deo koda u ovoj knjizi.

Pravljenje objekata i njihov životni vek

Jedan od najvažnijih činilaca jeste način na koji se objekti stvaraju i uništavaju. Da bi mogao da postoji, svakom objektu su neophodni neki resursi, prvenstveno memorija. Kada objekat više nije potreban, on mora da se počisti kako bi se resursi oslobodili i učinili dostupnim za ponovno korišćenje. U jednostavnim programskim situacijama pitanje kako očistiti objekat ne čini se previše komplikovanim: napravite objekat, koristite ga koliko vam je potreban, a onda neka se uništi. Međutim, moguće su i složenije situacije.

Pretpostavimo, na primer, da pišete program za upravljanje vazdušnim saobraćajem na nekom aerodromu. (Isti model bi mogao da se primeni i na rukovanje sanducima u skladištu, ili na sistem iznajmljivanja video kaseta, ili na smeštaj za kućne ljubimce.) Na prvi pogled, ovo izgleda jednostavno: napravite kontejner za čuvanje aviona, a zatim napravite novi avion i stavite ga u kontejner svaki put kada avion uđe u zonu kontrole vazdušnog saobraćaja. Da biste očistili sistem kada avion napusti zonu, počistite iz memorije odgovarajući objekat avion.

Možda imate i neki drugi sistem za zapisivanje podataka o avionima o kojima ne treba neposredno voditi računa. Možda je to evidencija o planovima leta malih aviona koji napuštaju aerodrom. Znači, trebao bi vam drugi kontejner za male avione i kad god napravite objekat aviona, ako je to mali avion, stavili biste ga takođe i u ovaj drugi kontejner. Zatim bi neki pozadinski proces obavljao operacije nad objektima iz tog kontejnera kada računar ne radi ništa drugo.

Sada je problem nešto teži: kako uopšte možete znati kada da uništite objekte? Kada završite sa objektom, nekom drugom delu sistema možda je još uvek potreban. Isti problem se može pojaviti i postati vrlo složen u mnogim drugim situacijama i u programskim sistemima (kao što je C++) u kojima morate izričito da obrišete objekat kada s njim završite.

Gde se nalaze podaci iz objekta i kako se kontroliše trajanje objekta? C++ zauzima stav da je najvažnija efikasnost, pa programeru prepušta izbor. Da bi se postigla maksimalna brzina izvršavanja, čuvanje i trajanje mogu biti određeni prilikom pisanja programa, tako što se objekti stavljaju na stek (oni se ponekad nazivaju i *automatske* ili *vidljive* – engl. *scoped* – promenljive) ili u statičku oblast za čuvanje. Ta mesta imaju visok prioritet i prostor se u njima brzo zauzima i oslobađa, pa je kontrola nad njima veoma značajna u nekim situacijama. Međutim, time žrtvujete fleksibilnost jer morate da znate tačan broj, trajanje i tip objekata prilikom pisanja programa. Ako pokušavate da rešavate opštiji problem, na primer projektovanje pomoću računara (CAD), upravljanje skladištem ili kontrolu vazdušnog saobraćaja, ova metoda je previše restriktivna.

Drugi pristup je dinamičko stvaranje objekata u dinamičkoj oblasti memorije (engl. *heap)*. Pri ovom pristupu, sve dok ne počne izvršavanje, ne znate koliko vam objekata treba, koliko će trajati, niti kog su tipa. Sve se to određuje kada program već radi (ovakav način pravljenja objekata zove se dinamički). Ako vam zatreba novi objekat, napravite ga u dinamičkoj memoriji, u trenutku kada vam je zatrebao. Pošto se oblašću za čuvanje upravlja dinamički, prilikom izvršavanja, zauzimanje memorijskog prostora traje znatno duže od odvajanja prostora na steku. Odvajanje prostora na steku često se postiže samo asemblerskom naredbom da se pokazivač steka pomeri naniže i da se vrati nazad. Vreme za koje se odvoji prostor u dinamičkoj memoriji, zavisi od dizajna mehanizma za skladištenje.

Dinamički pristup se zasniva na najčešće opravdanoj pretpostavci da su objekti složeni, tako da dodatni režijski troškovi za nalaženje prostora i njegovo oslobađanje neće bitno uticati na stvaranje objekata. Veća fleksibilnost dinamičkog pristupa je neophodna za rešavanje opštijih programskih problema.

Java isključivo koristi drugi pristup.[7] Svaki put kada želite da stvorite objekat, koristite rezervisanu reč **new** da biste napravili dinamički primerak tog objekta.

Nameće se još i pitanje trajanja objekata. U jezicima koji dozvoljavaju da se objekti stvaraju na steku, prevodilac određuje koliko dugo objekti traju i može automatski da ih uništi. Međutim, kada objekat stvorite dinamički, prevodilac nema informacije o njegovom životnom veku. U jeziku kao što je C++ morate programski da odredite kada ćete da uništite objekat, što može dovesti do „curenja" memorije ako se to ne uradi ispravno (to je čest problem u programima pisanim na C++-u). Java obezbeđuje tzv. *sakupljač smeća* (engl. *garbage collector*) koji automatski otkriva koji se objekat više ne upotrebljava i uništava ga.

---

[7] Prosti tipovi, o kojima ćete kasnije saznati više, predstavljaju specijalan slučaj.

Sakupljanje smeća je veoma korisno, jer smanjuje broj stavki na koje morate da mislite i količinu koda koji morate da napišete. Još je bitnije što sakupljanje smeća obezbeđuje mnogo viši nivo zaštite od podmuklih problema sa „curenjem" memorije (koje je mnoge projekte pisane na jeziku C++ oborilo na kolena).

U Javi, sakupljač smeća vodi računa o problemu oslobađanja memorije (iako u to ne spadaju drugi aspekti čišćenja objekta). Sakupljač smeća „zna" kada se objekat više ne koristi i automatski oslobađa memoriju koju je zauzimao taj objekat. To, uz činjenicu da su svi objekti izvedeni iz jedne osnovne klase **Object**, kao i da postoji samo jedan način za pravljenje objekata – dinamički – čini proces programiranja u Javi mnogo jednostavnijim od programiranja na jeziku C++. Mnogo je manje odluka koje treba da donesete i prepreka koje morate da prevaziđete.

Obrada izuzetaka: postupanje s greškama

Još od prvobitnih programskih jezika, obrada grešaka je bila jedna od najtežih oblasti. Pošto je teško napraviti dobru šemu za obradu grešaka, u mnogim jezicima zanemarena je ta oblast i to pitanje, a problem se prepušta projektantima biblioteka. Projektanti biblioteka su iznašli polovične mere primerene mnogim situacijama, ali koje se lako mogu zaobići, obično ignorisanjem. Osnovni problem sa svim šemama za obradu grešaka je to što se oslanjaju na obazrivost programera, i na ugovorene konvencije koje nisu nametnute jezikom. Ako programer nije obazriv – što se često dešava kad žuri – te šeme lako može ispustiti iz vida.

Obrada izuzetaka ugrađuje obradu grešaka direktno u programski jezik i ponekad čak i u operativni sistem. Izuzetak je objekat „bačen" (engl. *thrown*) s mesta greške, koji odgovarajući upravljač izuzecima koji obrađuje određeni tip grešaka, može da „uhvati" (engl. *catch*). Kao da je obrada izuzetaka drugi, paralelni put izvršavanja, kojim može da se krene kada stvari pođu naopako. Zato što koristi poseban put za izvršavanje, obrada izuzetaka ne mora da se meša s vašim kodom koji se normalno izvršava. Pošto ne morate stalno da proveravate da li je bilo grešaka, pisanje koda će najčešće biti jednostavnije. Sem toga, bačen izuzetak se razlikuje od vrednosti greške koja je vraćena iz metode i od indikatora stanja koji ukazuje na grešku, po tome što oni mogu da se ignorišu. Izuzetak se ne može ignorisati i garantuje se da će biti obrađen u nekom trenutku. Konačno, izuzeci obezbeđuju da se pouzdano izvučete iz loše situacije. Umesto da samo izađete iz programa, često ste u prilici da sredite stvari i nastavite sa izvršavanjem, čime dobijate mnogo robusnije programe.

Javina obrada izuzetaka se izdvaja od drugih programskih jezika jer je ugrađena od početka, pa ste prinuđeni da je koristite. To je jedini prihvatljivi način prijavljivanja grešaka. Ako ne napišete kôd tako da pravilno obrađuje izuzetke, javiće se greška pri prevođenju. Ova garantovana doslednost katkada znatno pojednostavljuje obradu grešaka.

Treba napomenuti da obrada izuzetaka nije samo objektno orijentisana mogućnost, iako su u objektno orijentisanim jezicima izuzeci obično predstavljeni objektima. Obrada izuzetaka je postojala i pre objektno orijentisanih jezika.

Paralelni rad

Osnovni koncept u računarskom programiranju jeste obrada više od jednog zadatka istovremeno. Mnogi programski poslovi zahtevaju da program može da zaustavi rad, pozabavi se nekim drugim problemom a zatim se vrati glavnom procesu. Problemu se pristupalo na više načina. Na početku, programeri koji su poznavali računar do najsitnijih pojedinosti, pisali su prekidne servisne rutine, a suspenzija glavnog procesa je inicirana preko hardverskog prekida. Iako je sve to dobro radilo, bilo je teško i neprenosivo, jer je prebacivanje programa na novi tip računara bilo sporo i skupo.

Ponekad su za obradu zadataka koji se moraju odmah izvršiti prekidi rada neophodni, ali postoji velika klasa poslova u kojoj problem pokušavamo da izdelimo na više delova koji se izvršavaju zasebno i paralelno, da bi ceo program brže reagovao. Delovi koji se zasebno izvršavaju unutar programa nazivaju se niti (engl. *thread*), a ceo koncept *paralelni rad* (engl. *concurrency*). Tipičan primer paralelnog rada je korisničko okruženje. Zbog podele programa na niti, korisnik može da pritisne dugme i da dobije brz odziv, umesto da bude prinuđen da čeka dok program ne završi trenutni zadatak.

Nitima se obično raspodeljuje vreme jednog (i jedinog) procesora. Međutim, ako operativni sistem podržava višeprocesorski rad, svaka nit može biti dodeljena različitim procesorima, i tada one zaista mogu da rade paralelno. Jedna od korisnih osobina paralelnog rada na nivou jezika jeste sledeća: programer ne mora da brine postoji li jedan ili više procesora. Program je logički podeljen na niti i ako računar ima više od jednog procesora, program će se brže izvršavati, bez ikakvih posebnih prilagođavanja.

Nakon svega ovoga, paralelni rad zvuči prilično jednostavno. Postoji ipak jedna začkoljica: deljeni resursi. Ako se simultano izvršava više niti koje očekuju da pristupe istom resursu, pojaviće se problem. Na primer, dva procesa ne mogu istovremeno da šalju podatke istom štampaču. Da bi se rešio problem, resursi koji mogu biti deljeni, kao što je štampač, moraju biti zaključani dok se koriste. Znači, nit zaključa resurs, završi zadatak, a zatim otključa resurs, posle čega neko drugi može da ga koristi.

Paralelni rad je ugrađen u Javu, a Java SE5 ga dodatno podržava svojom bibliotekom.

Java i Internet

Ako je Java zapravo samo još jedan računarski programski jezik, možete se zapitati zašto je toliko važna i zašto se predstavlja kao revolucionarni korak u računarskom programiranju. Odgovor nije odmah očigledan, ukoliko se posmatra iz tradicionalne programerske perspektive. Iako je Java veoma korisna za rešavanje uobičajenih zasebnih programskih problema, još je važnije to što pomoću nje možete rešiti i programske probleme koji se tiču Weba.

Šta je Web?

Isprva Web može da deluje pomalo tajanstveno, uz celu priču o „krstarenju“, „prisustvu“ i „ličnim prezentacijama“. Korisno je malo se udaljiti i sagledati šta Web zaista jeste, ali za to morate da razumete sisteme klijent/server, još jedan aspekt računarske obrade koji je pun zbunjujućih tema.

Klijent/server obrada

Osnovna ideja sistema klijent/server jeste postojanje centralnog skladišta informacija – neke vrste podataka, obično u bazi podataka – koje hoćete da šaljete, po zahtevu, grupi ljudi ili računara. U konceptu klijent/server ključno je to što je skladište informacija centralizovano tako da informacije mogu da se menjaju i da se te promene prenose svim korisnicima informacija. Skladište informacija, softver koji šalje informacije i računar (računari) gde se softver i informacije nalaze, nazivaju se *server*. Softver koji se nalazi na korisničkom računaru, sarađuje sa serverom, preuzima informacije, obrađuje ih i prikazuje na udaljenom računaru, naziva se *klijent*.

Osnovni koncept klijent/server obrade, znači, nije previše složen. Problemi se javljaju jer imate jedan server koji pokušava da opsluži više klijenata u isto vreme. Uglavnom se koristi sistem za upravljanje bazama podataka, tako da programer „uravnotežava“ raspored podataka u tabelama da bi postigao optimalno korišćenje. Sistemi često dozvoljavaju klijentima i da dodaju informacije na server. To znači da morate obezbediti da novi podaci jednog klijenta ne „pregaze“ nove podatke drugog klijenta, ili da se ti podaci ne izgube u procesu dodavanja u bazu. (Ovo se zove obrada transakcija – engl. *transaction processing*). Kada klijentski softver treba izmeniti, on mora da se prevede, očisti od grešaka i instalira na klijentskim računarima, što je u stvarnosti znatno komplikovanije i skuplje nego što mislite. Posebno je problematično podržati više tipova računara i operativnih sistema. Konačno, tu je i veoma bitno pitanje performansi: stotine klijenata mogu da postavljaju zahteve vašem serveru u bilo kom trenutku, pa je i najmanje zakašnjenje problematično. Da bi se kašnjenje svelo na najmanju meru, programeri naporno rade da rasterete procesne zadatke, prebacujući deo obrade na klijentski računar, a ponekad i na druge računare na serverskoj strani, koristeći takozvane *posrednike* (engl. *middleware*). (Posrednici se takođe koriste da bi se olakšalo održavanje.)

Realizacija tako jednostavnog postupka – slanja informacija – toliko je slojevita i složena da ceo problem deluje beznadežno zagonetno. I pored toga je veoma bitna: klijent/server obrada nosi otprilike polovinu svih programskih aktivnosti. Ona je zadužena za sve, počev od uzimanja narudžbina, transakcija s kreditnim karticama, pa do slanja raznih vrsta podataka – berzanskih, naučnih, državnih; šta god vam padne na pamet. U prošlosti smo se suočavali sa individualnim rešenjima individualnih problema; svaki put smo smišljali novo rešenje. Bilo je teško osmisliti ih, teško su se upotrebljavala i korisnik je morao da uči nov interfejs za svako od tih rešenja. Ceo klijent/server problem trebalo bi sveobuhvatno da se reši.

Web kao džinovski server

Web je u suštini jedan džinovski sistem klijent/server, i nešto više od toga, jer svi serveri i svi klijenti postoje zajedno i u isto vreme u samo jednoj mreži. To ne morate da znate jer u datom trenutku brinete samo o povezivanju i interakciji s jednim serverom (iako možda pri tom lutate Mrežom tražeći odgovarajući server).

U početku je to bio jednostavan jednosmerni proces. Vi ste ispostavljali zahtev serveru i on vam je prosleđivao datoteku koju je softver za čitanje (klijent) prevodio i formatirao na vašem lokalnom računaru. Ali, ubrzo su ljudi poželeli da rade više od čistog prosleđivanja

stranica sa servera. Želeli su punu klijent/server kompatibilnost kako bi klijent bio u mogućnosti da vraća podatke mogućnost na server, na primer, da pretražuje baze na serveru, dodaje nove informacije na server, ili da ostavi narudžbinu (što je sve zahtevalo posebne bezbednosne mere). Bili smo svedoci tih promena u razvoju Weba.

Čitač Weba je označio veliki korak napred – uveo je mogućnost da se delići informacija neizmenjeni prikazuju na bilo kom tipu računara. Ti čitači su ipak bili prilično primitivni i ubrzo su se zaglibili u zahtevima koji su im ispostavljani. Oni nisu bili previše interaktivni; zagušivali su server i sam Internet jer ste, svaki put kada je trebalo uraditi nešto što zahteva programiranje, morali da vraćate informacije serveru na obradu. Moglo je da prođe više sekundi ili minuta dok ne otkrijete da ste nešto pogrešno otkucali. Pošto je čitač bio predviđen samo za pregled, nije mogao da izvrši ni najjednostavnije zadatke obrade. (S druge strane, bio je potpuno bezbedan, jer na vašem računaru nije mogao da izvrši nijedan program, dakle ni da mu potencijalno donese grešku ili virus.)

Primenjeno je više pristupa rešenju ovog problema. Za početak, poboljšani su grafički standardi da bi omogućili bolju animaciju i video prikaz u čitačima. Drugi deo problema mogao je biti rešen samo ugrađivanjem u čitač mogućnosti za pokretanje programa na klijentskoj strani. To se naziva *programiranje s klijentske strane* (engl. *client-side programming*).

Programiranje s klijentske strane

Početni dizajn Weba (server–čitač) omogućavao je interaktivne sadržaje, ali je celokupnu interaktivnost obezbeđivao server. Server je pravio statičke stranice za klijentski čitač koji ih je tumačio i prikazivao. Elementarni *jezik za označavanje hiperteksta* (HTML) ima jednostavne mehanizme za prikupljanje podataka: polja za tekst, polja za potvrdu, radiodugmad, liste i padajuće liste, kao i dugme koje može biti programirano samo da obriše podatke u obrascu ili da pošalje (engl. *submit*) podatke iz obrasca nazad serveru. Ovo slanje se obavlja preko interfejsa CGI (engl. *Common Gateway Interface*), koji postoji na svim Web serverima. Tekst koji se nalazi u poslatom paketu kazuje serveru šta s tim paketom da uradi. Najčešća akcija je da se pokrene program koji se nalazi na serveru, u direktorijumu koji se obično zove „cgi-bin“. (Ako budete gledali adresno polje na vrhu svog čitača kada pritisnete neko dugme na Web stranici, ponekad ćete videti „cgi-bin“ negde unutar sveg onog zamešateljstva.) Ti programi mogu biti napisani na skoro svim jezicima. Često se pisalo na Perlu, jer je on napravljen za rad s tekstom, a uz to se interpretira, pa može biti instaliran na bilo koji server, bez obzira na procesor ili operativni sistem. Danas se sve više koristi Python (*www.Python.org*), zato što je jači i jednostavniji.

Mnoge današnje moćne Web stranice izgrađene su isključivo na CGI programima, s kojima možete da uradite gotovo sve. Ipak, održavanje Web stranica izgrađenih na CGI programima može brzo da postane suviše složeno, a postoji i problem s vremenom odziva. Odziv CGI programa zavisi od toga koliko podataka se šalje, kao i od opterećenja servera i Interneta. (Uz to, pokretanje CGI programa ume da bude sporo.) Prvi projektanti Weba nisu predvideli koliko brzo će njegova propusnost da postane premala za nove vrste aplikacija. Na primer, skoro je nemoguće dosledno ostvariti bilo koju vrstu dinamičkog crtanja grafika jer za svaku verziju grafika mora da se napravi GIF slika a zatim prenese sa servera do klijenta. (GIF je akronim od *Graphic Interchange Format*, format za razmenu

grafika.) Pored toga, bez sumnje ste stekli neposredno iskustvo s proverom ispravnosti podataka na ulaznom obrascu. Pritisnete dugme za slanje na stranici, server pokrene CGI program koji otkrije grešku, formatira HTML stranicu obaveštavajući vas o grešci, a zatim vam vrati tu stranicu. Onda vi morate da se vratite na prethodnu stranicu i pokušate ponovo. Ovo ne samo da je sporo već je i zamorno.

Rešenje je programiranje s klijentske strane. Većina stonih računara na kojima rade čitači Weba sposobni su da urade ogroman posao, a s prvobitnim statičkim HTML pristupom samo su stajali i besposleno čekali da server isporuči sledeću stranicu. Programiranje s klijentske strane znači da čitač Weba radi sav posao koji može da obavi, korisnik mnogo brže dobija rezultat a osećaj zajedništva pri radu s Web stranicom je potpuniji.

Rasprave o programiranju s klijentske strane malo se razlikuju od rasprava o programiranju uopšte. Parametri su gotovo isti, ali je platforma drugačija: čitač Weba je sličan ograničenom operativnom sistemu. Na kraju, opet morate da programirate, i zato programiranje s klijentske strane povlači vrtoglav niz problema i rešenja. U daljem tekstu dajemo pregled pitanja i pristupa pri programiranju klijentske strane.

Dodaci

Razvoj dodataka (engl. *plug-ins*) predstavlja jedan od najdužih koraka napred u programiranju s klijentske strane. Programer dodaje novu funkcionalnost čitaču tako što korisnik u potrebnom trenutku preuzme deo koda koji se ugradi na odgovarajuće mesto u čitaču. Taj kôd govori čitaču: „Od sada možeš da obavljaš i ovu novu aktivnost". (Dodatak treba da preuzmete samo jednom.) Preko dodataka su čitačima pridodata brza i moćna proširenja, ali pisanje dodataka nije jednostavan zadatak i nije nešto što biste želeli da radite u okviru izgradnje određene stranice. Vrednost dodataka za programiranje s klijentske strane jeste to što oni omogućavaju stručnjaku da pravi nova proširenja i da ih dodaje čitaču bez odobrenja proizvođača čitača. Stoga, dodaci predstavljaju „zadnja vrata" koja omogućavaju stvaranje novih jezika za programiranje s klijentske strane (iako nisu svi jezici realizovani kao dodaci).

Skript-jezici

Dodaci su prouzrokovali razvoj skript-jezika za čitače. Pomoću skript-jezika ugrađujete izvorni kôd programa za klijentsku stranu direktno u HTML stranicu, a dodatak koji tumači taj jezik automatski se aktivira pri prikazivanju HTML stranice. Skript-jezici se obično prilično lako shvataju i pošto se pišu u delu HTML stranice, učitavaju se veoma brzo, jednim pristupom serveru na kome se nalazi ta stranica. Nedostatak je to što je vaš kôd izložen i svako može da ga vidi (i ukrade). U principu, ipak ne pravite neke sofisticirane stvari pomoću skript-jezika, pa ovaj nedostatak i nije neki problem.

Postoji jedan skript-jezik koji većina čitača Weba podržava i *bez* ikakvog dodatka – to je JavaScript (koji ima tek prolaznu sličnost s Javom, i moraćete ga učiti zasebno). Tako je nazvan samo da bi prigrabio deo Javinog marketinškog uzleta. Nažalost, većina čitača Weba je svoju podršku za JavaScript realizovala na svoj jedinstveni način, drugačije od ostalih čitača, pa čak i od ostalih svojih verzija. Nešto je popravila standardizacija JavaScripta u obliku *ECMAScipta*, ali je raznim čitačima trebalo mnogo vremena da dođu do

tog nivoa (tome je doprineo i Microsoft, gurajući svoj VBScript koji pomalo liči na JavaScript). Da bi program mogao da se izvršava na svim čitačima, po pravilu ga morate pisati koristeći najmanji zajednički imenilac svih postojećih verzija JavaScripta. Programska obrada grešaka, kao i otkrivanje i otklanjanje grešaka u toku pisanja programa, mogu se opisati samo kao mučenje. Dokaz tih poteškoća je činjenica da je na JavaScriptu tek nedavno napisan zaista složen program (to je Googleov GMail), što je zahtevalo veliki trud i stručnost.

Ovim se naglašava da su skript-jezici koji se koriste unutar čitača Weba predviđeni da reše određene vrste problema, prvenstveno da se naprave bogatija i interaktivnija grafička korisnička okruženja. Skript-jezik može da reši i do 80 procenata problema koje srećemo pri programiranju s klijentske strane. Vaši problemi verovatno spadaju u tih 80 procenata, a kako skript-jezici omogućuju lakši i brži rad, trebalo bi da razmislite o njima pre nego što se upustite u mnogo zapetljanija rešenja, kao što je programiranje na Javi.

Java

Ako skript-jezici mogu da razreše 80 procenata problema pri klijentskom programiranju, šta je s preostalih 20 procenata „zaista teških stvari“? Java je popularno rešenje za tih 20%. Pre svega, to je moćan programski jezik, siguran, međuplatformski i internacionalan. Java se neprekidno širi: dodaju se nove mogućnosti jezika i biblioteke koje mogu elegantno da reše probleme teške i u tradicionalnim programskim jezicima, kao što su istovremeni rad, pristup bazama, mrežno programiranje i distribuirana obrada. Java dopušta programiranje s klijentske strane preko *apleta* i pomoću *Java Web Starta.*

Aplet je mali program koji može da se izvršava samo unutar čitača Weba. Aplet se automatski preuzima kao deo Web stranice (kao što se, na primer, automatski preuzima grafika). Kada se aplet aktivira, on izvršava program. Deo njegove pogodnosti je to što omogućava da automatski distribuirate klijentski softver sa servera tek u trenutku kada je korisniku klijentski softver potreban, a ne ranije. Korisnik bespogovorno dobija najnoviju verziju klijentskog softvera i to bez složene ponovne instalacije. Zbog načina na koji je Java projektovana, programer treba da napravi samo jedan jedini program, a taj program automatski radi na svim računarima koji imaju čitače s podrškom za Javu. (Ovo, bez ikakve brige, obuhvata ogromnu većinu računara.) Pošto je Java potpun programski jezik, možete uraditi sav posao koji je moguć na klijentskoj strani, pre i posle postavljanja zahteva serveru. Na primer, nema potrebe da šaljete obrazac sa zahtevom preko Interneta kako biste otkrili da li ste pogrešili pri upisivanju datuma ili nekog drugog parametra; vaš računar može brzo da iscrtava grafike na osnovu podataka, umesto da čeka da ih server iscrta i vrati sliku. Pored trenutnog poboljšanja brzine i odziva, smanjuju se ukupni mrežni saobraćaj i opterećenje servera i sprečava usporavanje celog Interneta.

Druge mogućnosti

Pošteno rečeno, Java apleti nisu opravdali velika početna očekivanja. Kada se Java tek pojavila, najviše se govorilo upravo o apletima, jer je trebalo da oni najzad omoguće ozbiljno programiranje na klijentskoj strani, povećaju brzinu odziva i smanje protok podataka potreban Internet aplikacijama. Predviđale su se ogromne mogućnosti.

Na Webu se zaista mogu naći i veoma pametni apleti, ali do sveopšteg prelaska na aplete nije došlo. Verovatno je najveći problem bila veličina Javinog izvršnog okruženja (Java Runtime Environment, JRE) od 10 MB, koje je trebalo preuzeti s Weba i instalirati, što je uplašilo prosečnog korisnika. Sudbinu im je možda zapečatila činjenica da je Microsoft odlučio da JRE ne isporučuje kao deo svog Internet Explorera. U svakom slučaju, Java apleti se nisu proširili posvuda.

Bez obzira na to, u nekim situacijama Java aplete i *Java Web Start* aplikacije još uvek vredi imati. Ukoliko upravljate računarima korisnika, recimo unutar preduzeća, bilo bi pametno da distribuciju i ažuriranje klijentskih aplikacija obavljate pomoću ovih tehnologija, jer ćete time uštedeti znatnu količinu vremena, truda i novca, naročito ako morate često da ažurirate njihov softver.

U poglavlju *Grafička korisnička okruženja* upoznaćemo *Flex*, novu Adobeovu tehnologiju koja obećava – u njome se prave Flash apleti. Pošto više od 98 procenata svih čitača Weba ima Flash Player (na Windowsu, Linuxu i Macu), možemo smatrati da je on usvojen standard. Flash Player se instalira i ažurira lako i brzo. Jezik ActionScript je napravljen na osnovu ECMAScripta i stoga je prilično poznat, ali Flex omogućava programiranje bez brige o specifičnostima raznih čitača – zato je daleko privlačniji od JavaScripta. Pri programiranju na klijentskoj strani, ovu mogućnost vredi razmotriti.

.NET i C#

Neko vreme je glavni konkurent Java apleta bio Microsoftov ActiveX, iako samo na Windows računarima. Otad je Microsoft napravio prvog pravog konkurenta celoj Javi u liku platforme .NET i programskog jezika C#. Platforma .NET je približno isto što i Javina virtuelna mašina (*Java Virtual Machine*, JVM) plus Java biblioteke. JVM je softverska platforma na kojoj se izvršavaju Java programi, a C# ima velike sličnosti s Javom. Bez sumnje, radi se o dosad najboljem Microsoftovom proizvodu na području programskih jezika i programskih okruženja. Naravno, Microsoft je imao veliku prednost jer je mogao da uči na tuđim greškama, ali je bogami naučio. Prvi put od svog nastanka, Java je dobila pravu konkurenciju. Zato su projektanti Jave u Sunu dobro pogledali C#, dobro razmislili o tome zašto bi programeri hteli da pređu na C#, i odgovorili: načinili su fundamentalna poboljšanja Jave – Javu SE5.

Trenutno je najveća slabost .NET-a važno pitanje da li će Microsoft dozvoliti njegovo *potpuno* prenošenje na druge platforme. Microsoft tvrdi da se to može napraviti bez problema, i već postoji delimična realizacija .NET-a koja radi na Linuxu (projekat Mono, *www.gomono.com*), ali dok se ne napravi potpuna realizacija iz koje Microsoft nije ništa izbacio, .NET je rizično smatrati rešenjem za sve platforme.

Internet i intranet

Web je najopštije rešenje problema klijent/server, pa je logično da istu tehnologiju upotrebite i da biste rešili podgrupu tog problema, posebno klasični problem klijent/server *unutar* preduzeća. Pri tradicionalnom pristupu klijent/server, postoji problem zbog raznih tipova računara kao i problem zbog teškog instaliranja novog klijentskog softvera. Oba problema dobro su rešena pomoću čitača Weba i programiranja s klijentske strane.

Kada se tehnologija Weba koristi za informacionu mrežu ograničenu na određeno preduzeće, to nazivamo intranet. Intranet obezbeđuje mnogo veću sigurnost nego Internet jer pristup serverima u preduzeću možete fizički da kontrolišete. Izgleda da korisnici, kada jednom shvate osnovni princip rada čitača, mnogo lakše izlaze na kraj s različitim izgledima stranica i apleta, pa brže uče nove sisteme.

Problem s bezbednošću svrstava nas u jednu od grupa koje se obrazuju, čini se, po automatizmu, u svetu klijent/server programiranja. Ako se vaš program izvršava na Internetu, ne znate na kojoj platformi će on raditi i želite da budete sasvim sigurni da ne širite neispravan kôd. Treba vam nešto nezavisno od platforme i bezbedno, kao što su skript-jezik ili Java.

Ako radite na intranetu, pred vas se postavljaju drugačija ograničenja. Nije neuobičajeno da svi računari budu platforme Intel/Windows. Na intranetu ste sami odgovorni za kvalitet svog programa i možete da otklanjate greške kako se koja otkriva. Pored toga, možda već imate dosta nasleđenog koda koji ste koristili za tradicionalniji klijent/server pristup, pri čemu morate fizički da instalirate klijentske programe svaki put kada radite na poboljšanju. Vreme protraćeno na instaliranje poboljšanja pruža razlog više da pređemo na čitače gde su poboljšanja nevidljiva i automatska. (Ovaj problem rešava i Java Web Start.) Ako imate posla s takvim intranetom, najrazboritiji pristup je da krenete najkraćim putem koji vam omogućava da koristite postojeću bazu koda, umesto da ponovo pišete programe na novom jeziku.

Kada se suočite sa ovim zbunjujućim nizom rešenja za programiranje s klijentske strane, najbolji plan napada je analiza isplativosti. Razmotrite ograničenja koja vaš problem nameće i najkraći put do rešenja. Pošto je programiranje s klijentske strane ipak programiranje, uvek je dobro odabrati pristup koji omogućava najbrži razvoj u određenoj situaciji. Ovo je hrabar stav koji vas priprema za neizbežne susrete s problemima u razvoju programa.

Programiranje sa serverske strane

Do sada je pitanje programiranja sa serverske strane bilo zanemareno, a baš tu je po mišljenju mnogih Java imala najveće uspehe. Šta se dešava kada pošaljete zahtev serveru? Najčešće zahtevi glase: „Pošalji mi ovu datoteku". Vaš čitač zatim tumači tu datoteku na odgovarajući način: kao HTML stranicu, sliku, Java aplet, skript itd.

U složenije zahteve serveru obično spadaju i transakcije s bazama podataka. Obično se zahteva kompleksna pretraga baze podataka, koju server zatim formatira u HTML stranicu i šalje kao rezultat. (Naravno, ako je klijent inteligentniji, zato što sadrži kôd na Javi ili skript-jeziku, mogu se slati neobrađeni podaci, a potom formatirati na strani klijenta, što bi bilo brže i manje bi opterećivalo server.) Možda biste poželeli da registrujete svoje ime u bazi podataka kada pristupate nekoj grupi ili ostavljate narudžbinu, što će izazvati promene u toj bazi. Pisanje programa koji obrađuju takve zahteve na serveru naziva se programiranje sa serverske strane (engl. *server-side programming*). Programiranje sa serverske strane obavlja se na Perlu, Pythonu, C++-u ili nekom drugom jeziku za pisanje CGI programa, ali pojavili su se i napredniji sistemi. Među njima su i Web serveri zasnovani na Javi, koji omogućavaju pisanje takozvanih *servleta.* Kompanije koje razvijaju Web

stranice prelaze na Javu zbog tehnologije servleta i iz njih izvedenih JSP strana, najviše zato što se time isključuju problemi pri radu sa čitačima različitih mogućnosti. Programiranje sa serverske strane razmotreno je u knjizi *Thinking in Enterprise Java*, koju možete naći na adresi *www.MindView.net.*

Uprkos tome što se o Javi priča samo u vezi sa Internetom, ona je programski jezik opšte namene pomoću kogae možete rešiti bilo koji tip problema, kao i pomoću drugih jezika. Tu Javina snaga nije samo u prenosivosti, već i u lakoći programiranja, robusnosti, velikoj standardnoj biblioteci i brojnim bibliotekama drugih proizvođača koje se stalno razvijaju.

Sažetak

Znate kako izgleda proceduralni program: definicije podataka i pozivi funkcija. Da biste pronašli svrhu takvog programa, morate malo da se pomučite i pregledate pozive funkcija i koncepte niskog nivoa kako biste stvorili model u glavi. Zbog toga nam treba posredno predstavljanje kada projektujemo proceduralni program: sami za sebe, ti programi mogu da zbunjuju jer su načini izražavanja usmereni više ka računaru nego ka problemu koji rešavamo.

Pošto OOP dodaje mnoge nove ideje onima koje nalazite u proceduralnim jezicima, možda mislite da će Java program biti mnogo komplikovaniji nego ekvivalentni C program. Bićete prijatno iznenađeni: dobro napisan Java program je uglavnom daleko jednostavniji i mnogo razumljiviji od ekvivalentnog C programa. Imaćete definicije objekata koji predstavljaju ideje u vašem prostoru problema (umesto pitanja računarskog prikaza) i poruke poslate tim objektima koje predstavljaju aktivnosti u tom prostoru. Dobra strana u objektno orijentisanom programiranju jeste ta što je uz dobro projektovan program lako razumeti kôd pri čitanju. Obično ima i znatno manje koda, jer ćete mnoge svoje probleme rešiti ponovnim korišćenjem postojećeg koda iz biblioteka.

OOP i Java ne moraju da budu za svakoga. Važno je da procenite svoje potrebe i odlučite da li će ih Java optimalno zadovoljiti ili bi bilo bolje da radite s nekim drugim programskim sistemom (uključujući onaj koji trenutno koristite). Ako znate da će vaše potrebe biti usko specijalizovane u bliskoj budućnosti i ako imate specifična ograničenja koja Java možda ne može da zadovolji, onda ispitajte ostale raspoložive mogućnosti. Konkretno, preporučujem da razmotrite Python; posetite *www.Python.org*. Ako ipak izaberete Javu, barem ćete znati koje ste mogućnosti imali i zašto ste baš nju izabrali.