VISOKA KONKURENTNOST NA JVM-u

Zlatko Sirotić
Istra informatički inženjering d.o.o., Pula
e-mail: zlatko.sirotic@iii.hr

SAŽETAK

Već duže vrijeme na Java virtualnom stroju (JVM) možemo izvršavati i programe koji nisu izvorno napisani u Javi, već u nekom drugom jeziku koji se može prevesti u Java bytecode. Takvi jezici, kao npr. Jython (JVM verzija jezika Phyton), JRuby (JVM verzija jezika Ruby), Groovy, Scala, Clojure, Ceylon (u razvoju), predstavljaju konkurenciju Javi na JVM-u. S druge strane, danas, u eri višejezgrenih mikroprocesorskih čipova, jako je važna podrška programskog jezika konkurentnom programiranju. Java od svojih početaka ima podršku konkurentnom programiranju, ali ta podrška nije bez mana. Od verzije 5 Java je dobila značajna poboljšanja na području konkurentnog programiranja. No, noviji programski jezici Scala i Clojure podržavaju konkurentno programiranje na još lakši način. U radu se daje usporedba konkurentnog programiranja u Javi u odnosu na Scalu.

For quite a long time now, the Java virtual machine (JVM) has been able to execute the programs that were not originally written in Java, but in a different language which can be translated into Java bytecode. Such languages as, for example, Jython (the JVM version of the Phyton language), JRuby (the JVM version of the Ruby language), Groovy, Scala, Clojure, and Ceylon (in development), are Java's competitors on the JVM. On the other hand, today, in the era of multi-core microprocessor chips, it is very important that a computer language should support concurrent programming. Ever since its beginnings, Java has been supporting concurrent programming, but this support is not without flaws. Since its version 5, Java has undergone significant improvements in the area of concurrent programming. However, the more recent programming languages Scala and Clojure support the concurrent programming in an even easier way. This paper provides a comparison of concurrent programming in Java in relation to Scala.

UVOD

Riječ "konkurentnost" u naslovu ima trostruku ulogu. Prije svega, željeli smo podsjetiti da danas postoji više jezika koji mogu raditi na JVM-u, kao što su Jython (JVM verzija jezika Phyton), JRuby (JVM verzija jezika Ruby), Groovy, Scala, Clojure, Ceylon (u razvoju) i drugi. S druge strane, danas, u eri višejezgrenih mikroprocesorskih čipova, jako je važna podrška programskog jezika konkurentnom programiranju, pa će se prikazati neke osobine i mane konkurentnog programiranja u Javi, te osnove konkurentnog programiranja u jeziku Scala. S treće strane, upravo zbog potrebe za "boljim konkurentnim programiranjem", u zadnje vrijeme sve su popularniji funkcijski jezici (functional languages), kod kojih su neke varijante konkurentnog programiranja lakše izvedive, pa se javlja konkurentnost između imperativnih (i to uglavnom objektnih) i funkcijskih jezika.

Funkcijski jezici nisu novost. Prvi funkcijski jezik Lisp nastao je 1958., svega godinu dana nakon prvog višeg programskog jezika Fortran. Funkcijski jezici (Lisp i njegove izvedenice) su, unatoč puno slabijoj zastupljenosti u odnosu na imperativne jezike, uvijek imali manju, ali jaku zajednicu koja ih je podržavala. Ta je zajednica bila (i jeste) naročito snažna u SAD-u, posebno za potrebe programiranja aplikacija za umjetnu inteligenciju, dok se u Evropi i Japanu za te namjene puno više koristi logički programski jezik Prolog i njegove izvedenice. I na JVM-u postoji više funkcijskih programskih jezika, a jedan od njih je Clojure, koji je isto tako nasljednik jezika Lisp.

U zadnjih (otprilike) desetak godina sve se više govori i o tome da bi programiranje trebalo biti mulitiparadigmatsko, tj. da bismo trebali koristiti onu jezičnu paradigmu koja je najprirodnija za rješavanje određenog problema, a ne koristiti "jedan alat za sve probleme". Ovdje postoje barem dva pristupa – jedan je da koristimo više programskih jezika (npr. jedan objektni, jedan funkcijski, jedan logički ...), a drugi pristup kaže da je puno jednostavnije i bolje imati jedan programski jezik koji podržava različite paradigme (po mogućnosti sve, kao što je jezik Oz kojeg opisuju autori u [20]). Programski jezik Scala je prije svega objektni programski jezik, ali tako nadograđen da ima i značajne funkcijske osobine, pa se može reći da je to i funkcijski programski jezik.

Rad je podijeljen u sedam točaka. U 1. točki podsjećamo se na neke teoretske aspekte programskih jezika (jezici, automati, gramatike), a u 2. točki prikazuju se neke klasifikacije programskih jezika. U 3. točki prikazuju se neki trendovi u razvoju hardvera (povezano s tim, i sistemskog softvera), koji utječu na konkurentno programiranje. U 4. točki podsjećamo se na konkurentno programiranje u jeziku Java, te probleme koji se javljaju kod toga. Sljedeće tri točke posvećene su programskom jeziku Scala, u kojima se ukratko obrađuju njegove objektne osobine, funkcijske osobine, te konkurentno programiranje. Važno je istaknuti da mi (autor ovog rada) nismo eksperti za Scalu, nego se tek učimo (a Scala je bogat programski jezik), ali mišljenja smo da je Scala vrlo perspektivan programski jezik za programiranje na JVM-u.

Programskom jeziku Clojure nismo se posvetili u ovom radu. Razloga za to postoje barem tri. Prvo, Clojure poznajemo vrlo površno. Drugo, za razliku od Scale, koja je i objektni i funkcijski jezik, Clojure je samo funkcijski jezik, tj. ne podržava multiparadigmatsko programiranje. Treće, nije nevažno da držimo kako su statički programski jezici (tj. programski jezici sa statičkom provjerom tipova, kod kojih se provjere tipova rade kod kompajliranja) sigurniji za profesionalno programiranje od dinamičkih programskih jezika (kod kojih se greška javlja tek kod izvođenja programa). Scala (kao i Fortran, Algol, Pascal, C/C++, Eiffel, PL/SQL, funkcijski jezik Haskell, Java, C# ...) su statički programski jezici, dok je Clojure dinamički (kao i Lisp, izvorni Prolog, Smalltalk, Phyton, Ruby, Groovy, Ceylon ...). Za statičke jezike se često govori da "sputavaju programera", ali za Scalu mnogi kažu da ona pruža (statičku) sigurnost i fleksibilnost dinamičkog jezika. Vrijeme će pokazati što je bolje. Za sada programski jezik Scala ima značajan uspon, dok za Clojure to vrijedi u daleko manjoj mjeri.

1. PROGRAMSKI JEZICI, AUTOMATI, GRAMATIKE

Kako kaže biolog i matematičar M.Nowak u [13], (ljudski) jezik je najvažnija invencija prirode nakon što su se prije oko 600 milijuna godina pojavili prvi razvijeni višestanični organizmi. (Prije toga važna je bila pojava prokariota, prije oko 3500 milijuna godina, i eukariota, prije oko 1500 milijuna godina. Eukarioti su organizmi, odnosno stanice, kod kojih je nasljedni materijal smješten u jezgri obavijenoj posebnom jezgrinom membranom: životinje, čovjek, biljke, gljive i protisti. Prokarioti nemaju jezgru i ostale složene stanične strukture: bakterije i modrozelene alge.) Kao što se ne zna približno vrijeme pojave života, ne zna se niti kada se pojavio jezik, ali neki drže da bi to moglo biti prije oko milijun godina.

Slavni američki lingvist i filozof (ali i neumorni politički aktivist) Noam Chomsky još je 50-ih godina prošlog stoljeća napravio poznatu klasifikaciju jezika (ljudskih i formalnih). Chomskyjeva je hijerarhija jezika postala važna u računarstvu, naročito u konstrukciji jezičnih procesora i teoriji automata. Ta je klasifikacija dala vezu između određenih jezika, gramatika koje ih opisuju i generiraju nizove znakova u tim jezicima, te (apstraktnih) automata koji prihvaćaju rečenice u tim jezicima. Sljedeća slika (1.1) prikazuje Chomskyjevu hijerarhiju jezika, gramatika i automata:

Teorija automata: formalni jezici i formalne gramatike			
Chomskyjeva hijerarhija	Gramatike	Jezici	Minimalni automat
Tip 0	Neograničenih produkcija	Rekurzivno prebrojiv	Turingov stroj
n/a	(nema uobičajenog imena)	Rekurzivni	Odlučitelj
Tip 1	Kontekstno ovisna	Kontekstno ovisni	Linearno ograničen
n/a	Indeksirana	Indeksirani	Ugniježđenog stoga
Tip 2	Kontekstno neovisna	Kontekstno neovisni	Nedeterministički potisni
n/a	Deterministička kontekstno neovisna	Deterministički kontekstno neovisni	Deterministički potisni
Tip 3	Regularna	Regularni	Konačni
Svaka kategorija jezika ili gramatika je pravi podskup nadređene kategorije.			

Slika 1.1. Chomskyjeva hijerarhija jezika; Izvor: Wikipedija, pod Noam Chomsky

Tip 0 je najširi skup, a svaki sljedeći je (pravi) podskup gornjeg skupa. Treba napomenuti da je izvorna Chomskyjeva hijerarhija uključivala samo četiri tipa (0, 1, 2 i 3, kako ih je Chomsky nazvao).

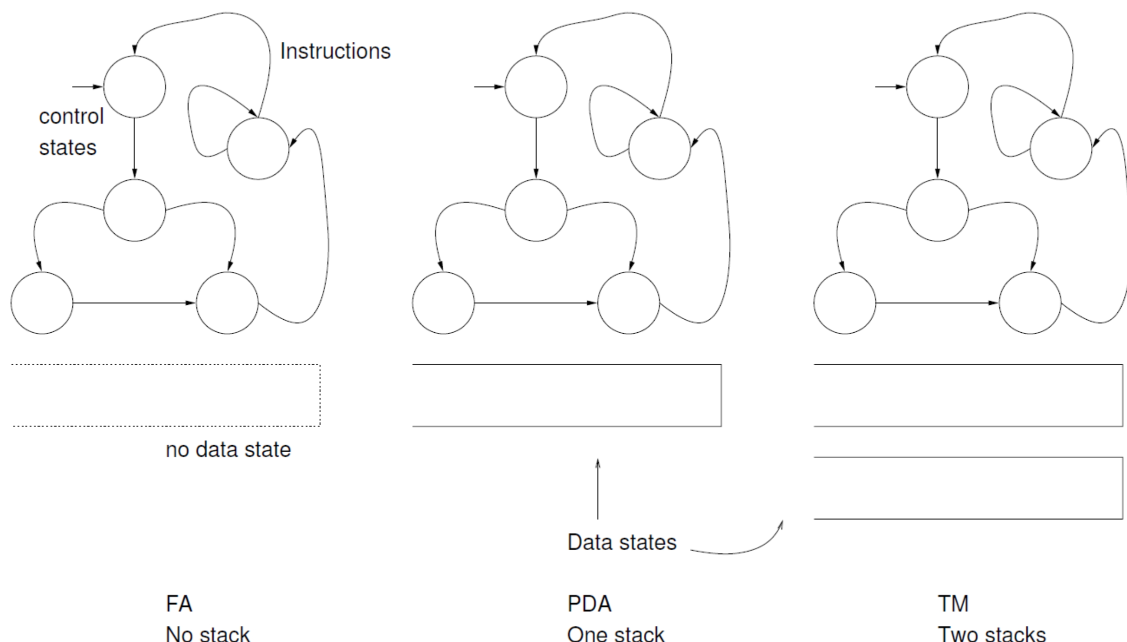
Također, važno je napomenuti da postoji i širi skup od tipa 0, a to je skup svih mogućih jezika nad određenom abecedom (zapravo, teoretski nije važno koja je abeceda, jer se svaka može svesti na binarnu abecedu, koja ima samo znakove 0 i 1). Međutim, taj širi skup jezika (koji je, uzgred, neprebrojivo beskonačan, kao i skup realnih brojeva, dok je skup rekurzivno prebrojivih jezika beskonačan, ali prebrojiv) nema odgovarajuće gramatike i automata. Budući da se prepoznavanje i generiranje jezika može interpretirati kao rješavanje problema, postoje i problemi (i to neprebrojivo beskonačno njih) za koje ne postoje algoritmi (tj. za jezike automati i gramatike). To su tzv. neizračunljivi (non-computable, undecidable) jezici i problemi ([4], [10], [17]).

Usput, poznato je da je Kurt Gödel 1931. dokazao teorem o nekompletnosti aritmetike. Pojednostavljeno, teorem izriče: "Ne postoji konačan skup aksioma aritmetike kojim bi se mogle dokazati sve istinite aritmetičke tvrdnje". Kako kaže autor u [13], govoreći o tom Gödelovom teoremu, zaista je ironija sudbine da matematika, najprecizniji od svih jezika, nema gramatike.

Odlučiv (decidable) jezik je onaj kod kojeg Turingov stroj (najjači mogući stroj) uvijek stane i odluči o prihvaćanju ili neprihvaćanju niza znakova (tog jezika). Rekurzivni jezici su uvijek odlučivi. No, rekurzivno prebrojivi jezici (tip 0) koji nisu rekurzivni, nemaju Turingov stroj koji uvijek stane. Kod njih, ako niz pripada jeziku, Turingov stroj uvijek stane, ali ako niz ne pripada jeziku, moguće je da Turingov stroj ne stane. Rekurzivno prebrojivi jezici koji nisu rekurzivni, jesu izračunljivi, ali nisu odlučivi ([17]). U [4] ih se naziva polu-odlučivim (semi-decidable), a koriste se i izrazi parcijalno odlučivi (partially decidable) ili Turing-prihvatljivi (Turing-acceptable).

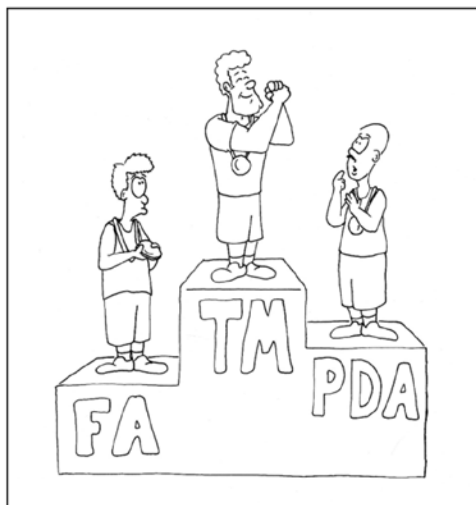
Jedan od najpoznatijih neizračunljivih problema je tzv. Halting problem ([4], [10], [12], [17]). On se može izraziti na ovaj način: "Ako je dat bilo koji računalni programa i bilo koji ulaz, nađimo (opći) algoritam koji će odrediti da li će program nad ulazom završiti, ili će raditi beskonačno." Alan Turing (poznat i po ogromnoj ulozi u kriptoanalizi šifarskog sustava njemačkog stroja Enigma u 2. svjetskom ratu) je 1936. dokazao da takav opći algoritam ne postoji, tj. da je taj problem općenito neizračunljiv. Ključni dio dokaza bila je definicija računala i programa, što je postalo poznato kao Turingov stroj.

Kako kaže autor u [4], igra sudbine je da je prvo otkriven najjači stroj, Turingov stroj, dok su slabiji strojevi, tj. konačni automat (finite automaton, FA) i potisni automat (push-down automaton, PDA) otkriveni tek kasnije. Za razliku od Turingovog stroja, koji ima jednu (potencijalno) beskonačnu traku ili polutraku, potisni automat nema traku, već jedan stog (stack), a konačni automat nema niti stog. Zapravo, umjesto trake, Turingov stroj može imati dva stoga. Turingov stroj sa više od dva stoga, ili sa više od jedne trake, teoretski je po snazi jednak stroju sa jednom trakom ili dva stoga (no, razlike se mogu javiti u trajanju izračuna ili potrebnoj memoriji, tj. vremenskoj ili memorijskoj kompleksnosti algoritma koji se na njima izvodi). Sljedeća slika (1.2) prikazuje glavne elemente sva tri stroja:



Slika 1.2. Konačni automat (FA), potisni automat (PDA) i Turingov stroj (TM); Izvor: [4]

Konačni automat pripada tipu 3, a potisni automat tipu 2 po Chomskyjevoj hijerarhiji. Turingov stroj obuhvaća i tip 1 i tip 0, jer je linearno ograničeni automat (koji odgovara tipu 1, tj. prihvaća kontekstno ovisne jezike) zapravo Turingov stroj sa ograničenjem na dužinu trake (pa je po tome bliži stvarnim računalima od Turingovog stroja sa potencijalno beskonačnom trakom). Sljedeća slika (1.3) prikazuje odnos snaga ta tri automata na šaljiv način:



Slika 1.3. Konačni automat (FA), potisni automat (PDA) i Turingov stroj (TM); Izvor: [4]

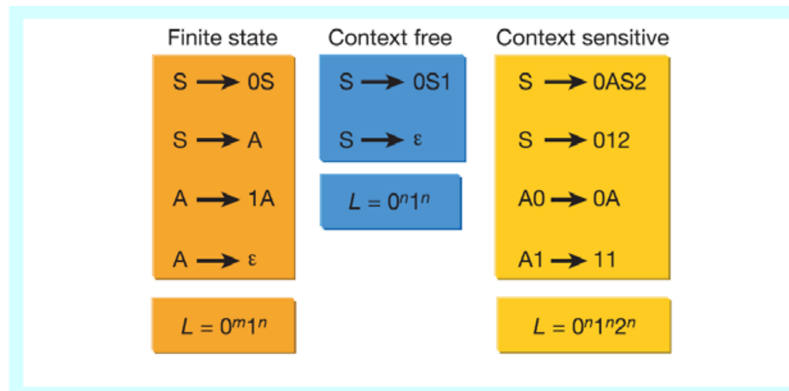
Poznata je Church-Turingova hipoteza (inače, Turing je kod Alonza Churcha radio doktorat), koja se ne može matematički dokazati, već se smatra intuitivno prihvatljivom. Ona (pojednostavljeno, te u današnjoj terminologiji) kaže da sve što se efektivno može izračunati, može izračunati pomoću Turingovog stroja, pa su sva uobičajena računala međusobno ekvivalentna u terminima teoretske računske moći, tj. nije moguće izgraditi uređaj za računanje koji će biti moćniji od najjednostavnijeg računala, Turingovog stroja. Zapravo se prvotno ta teza zvala Churchova teza, a on ju je izrazio u terminima vlastitog Lambda računa (lambda calculus). Kasnije se (50-ih godina) formalno pokazalo da su ne samo Turingov stroj i lambda račun ekvivalentni, već postoje i neki drugi sustavi koji su njima ekvivalentni (Gödelove rekurzivne funkcije, Postov sustav, Markovljevi algoritmi i dr.). No, bez obzira na teoretsku ekvivalentnost, Turingovi strojevi su po svom ponašanju bliži imperativnoj programskoj paradigmi, dok je lambda račun teoretska osnova za funkcijske programske jezike.

Sljedeća slika (1.4) prikazuje odnos između abecede (na slici, znakovi abecede su 0 i 1), rečenica (konačnih nizova znakova), jezika (konačnog ili beskonačnog skupa rečenica) i gramatike (konačnog skupa pravila za definiranje ili generiranje jezika):

An alphabet is a set of symbols:
{0,1}
Sentences are strings of symbols:
0,1,00,01,10,1,...
A language is a set of sentences:
L = {000,0100,0010,..}
A grammar is a finite list of rules defining a language.
S ⟶ 0A
B ⟶ 1B
A ⟶ 1A
B ⟶ 0F
A ⟶ 0B
F ⟶ ε

Slika 1.4. Osnovni elementi formalnih jezika; Izvor: [13]

Sljedeća slika (1.5) prikazuje primjer regularne, kontekstno neovisne i kontekstno ovisne gramatike, te tri jezika koja su opisana (generiraju se) sa te tri gramatike:



Slika 1.5. Primjer regularne, kontekstno neovisne i kontekstno ovisne gramatike; Izvor: [13]

Gramatika konačnih stanja ima pravila u obliku: jedan neterminalni simbol (lijevo od strelice) se zamjenjuje sa (desno od strelice) jednim terminalnim simbolom (to je ovdje znak abecede, tj. 0 ili 1, ali i prazan znak ε), iza kojega može slijediti neterminalni simbol. Na slici se generira regularni jezik u kojem se valjana rečenica sastoji od niza nula, iza kojih slijedi niz jedinica.

Kontekstno neovisna gramatika ima pravila u obliku: jedan neterminalni simbol (lijevo) se zamjenjuje sa bilo kojim nizom terminala i neterminala. Na slici se generiraju rečenice koje na početku imaju niz nula, iza kojeg slijedi niz sa istim brojem jedinica.

Kontekstno ovisna gramatika ima pravila u obliku aAb -> acb, gdje su a, b, c nizovi terminala i/ili neterminala, dok je A neterminal. Dok a i b mogu biti prazni, c ne smije biti prazan. Važna restrikcija je i ta da kompletni niz na desnoj strani mora biti najmanje dugačak kao kompletni niz na lijevoj strani. Na slici se generiraju rečenice koje na početku imaju niz nula, iza kojeg slijedi niz sa istim brojem jedinica, a na kraju je niz sa istim brojem dvojki (u ovom slučaju abeceda ima tri znaka: 0, 1 i 2).

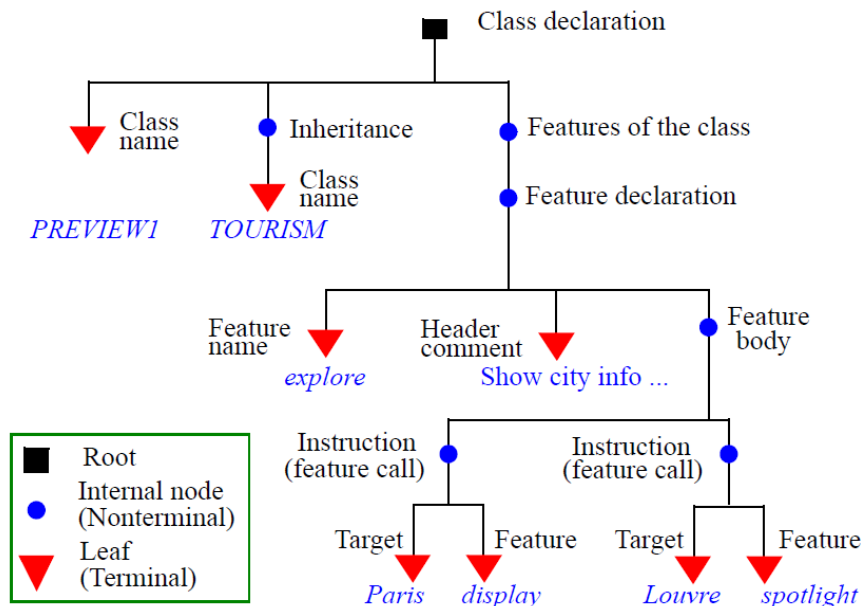
Vidljivo je da se i pravila gramatike pišu u nekom jeziku, koji je drugačiji od jezika koji opisuje i generira ta gramatika. Riječ je o metajeziku, jeziku koji služi za opis drugih jezika. Primjer metajezika je BNF notacija (Backus-Naur Form), nazvana po Johnu Backusu (voditelju tima koji je stvorio programski jezik Fortran) i Peteru Nauru, koji su kreirali tu notaciju tokom dizajniranja programskog jezika Algol 60 ([12]). Postoje različite varijacije BNF gramatike, koje su prilagođenije za opis određenog programskog jezika (Niklaus Wirth je prvi napravio grafičku varijantu, za prikaz svog programskog jezika Pascal). BNF gramatika sastoji se od tri konačna skupa: skupa delimitera, koji pripadaju ciljnom programskom jeziku (npr. ključne riječi kao if, ili specijalni znakovi kao točka), skupa konstrukata (constructs) koji reprezentiraju strukture jezika (npr. konstrukt Conditional, koji reprezentira uvjetne instrukcije jezika) i skupa produkcija (productions). Slijedi jedan primjer produkcije za konstrukt Conditional. Delimiteri ciljnog jezika su boldani, a neboldane riječi pripadaju metajeziku, bilo kao imena konstrukata (označena sa italic), bilo kao simboli metajezika (u ovom slučaju tri simbola: ::=, [,]):

Conditional ::= **if** *Then_part_list* [*Else_part*] **end**

BNF notacija koristi se ([12]) za:
- razumijevanje sintakse postojećeg jezika, posebno (ali ne nužno) programskog jezika;
- definiranje sintakse jezika koji se želi dizajnirati;
- pisanje sintaksnog analizatora, parsera (parser).

Treća primjena, parsiranje, korisna je kod pisanja kompajlera (i sličnih alata). Jedna od prvih zadaća kompajlera je rekonstrukcija strukture programskog teksta, u formi apstraktnog sintaksnog stabla. Parseru treba formalna definicija sintakse dana BNF gramatikom. Slijedi jedan primjer programskog odsječka (u jeziku Eiffel), iza kojeg slijedi slika odgovarajućeg apstraktnog sintaksnog stabla:

```
class PREVIEW1 inherit TOURISM
feature
  explore -- Show city info including a monument.
  do
    Paris.display;  Louvre.spotlight
  end
end
```

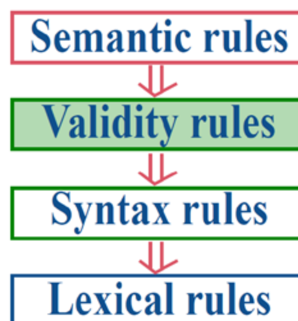


Slika 1.6. Primjer za apstraktno sintaksno stablo; Izvor: [12]

No, parsiranje nije prva faza koju radi tipični kompajler. Uobičajeno, kompajler radi ove faze ([12]):

- leksička analiza programa: programski tekst pretvara se u nizove tokena, koji predstavljaju identifikatore, ključne riječi i specijalne simbole jezika; za to se koristi regularna gramatika i konačni automat;
- parsiranje: kreira se apstraktno sintaksno stablo; koristi se kontekstno neovisna gramatika i potisni automat, tj. koristi se BNF notacija;
- provjera validnosti (validity checking): kod statičkih jezika uključuje provjeru tipova (type checking), te ostale verifikacije; ovdje se ne koristi BNF notacija, već formalizmi za opis kontekstno ovisnih aspekata, npr. gramatika atributa (attribute grammar), ili ad hoc pristupi; nažalost, u području kontekstno ovisne gramatike ne postoje formalizmi koji bi po jednostavnosti i praktičnosti odgovarali BNF notaciji (koja se koristi samo za kontekstno neovisnu gramatiku);
- semantička analiza: uključuje procesiranje rezultata faze parsiranja, koristeći i obogaćujući apstraktno sintaksno stablo;
- generiranje koda;
- optimizacija: npr. relokacija naredbi, relokacija registara, eliminiranje nepotrebnog koda i dr. (napomenimo da je ova faza vrlo delikatna, naročito uzevši u obzir konkurentno programiranje).

Kako kaže autor u [12], kod jezika sa statičkom provjerom tipova korisno je razlikovati validnost (validity) od korektnosti (correctnes) programa. Validan program zadovoljava pravila za tipove i ostala statička pravila konzistentnosti. Korektan program je onaj koji se izvršava u skladu sa željenim ponašanjem. Validnost je preduvjet korektnosti. Sljedeća slika prikazuje razine opisa programskog jezika (svaka razina temelji se na razini ispod nje):



Slika 1.7. Razine opisa programskog jezika; Izvor: [12]

2. KLASIFIKACIJE PROGRAMSKIH JEZIKA I MODELA PROGRAMIRANJA

Osim rijetkih izuzetaka, svi današnji programski jezici su Turing ekvivalentni, tj. njihova izražajna moć jednaka je Turingovom stroju, koji je (iako konceptualno jednostavan) najmoćniji stroj za rješavanje algoritamskih problema. Primjer jezika koji nije Turing ekvivalentan je XML (eXtensible Markup Language), koji je služi samo za opis struktura podataka. XML je neka vrsta proširenja HTML-a (koji, za razliku od XML-a, ima specifičnu namjenu). XML je stvoren sa idejom da bude opći jezik, jednostavno čitljiv i ljudima i računalnim programima. Danas postoje i kritička mišljenja da XML nije dobar niti za ljude, niti za računala, te da njegovu upotrebu treba barem značajno smanjiti, ako ne i ukinuti.

No, iako je većina programskih jezika Turing ekvivalentna, postoje velike razlike u lakoći (ili teškoći) programiranja određenih problema u određenim programskim jezicima, pa postoje ne samo brojni programski jezici, nego i brojni stilovi programskih jezika. Autor u [12] navodi sljedeće kriterije za klasifikaciju programskih jezika:

- primjena (application): neki programski jezici imaju opću namjenu (general-purpose); drugi služe za određeno specijalno područje primjene – za njih se obično koristi termin jezik za specifičnu domenu (domain-specific language, DSL); naravno, ta klasifikacija nije uvijek čvrsta, jer je npr. Fortran prvo bio zamišljen kao jezik za matematičke izračune (FORmula TRANslation), da bi poslije služio kao jezik opće namjene; slično je Java bila prvo zamišljena kao jezik za programiranje specijalnih uređaja, pa kao jezik za pisanje Internet apleta, da bi postala jezik opće namjene;

- obujam programa (program scope): neki jezici su zamišljeni za široki obujam, koji uključuje veliki programski kod, velik broj programera, dug period razvoja; na drugom su kraju jezici za brz razvoj, sa relativno malo koda i malim brojem programera – obično se kaže da su to jezici za prototipni razvoj, a tu najčešće spadaju skriptni jezici; naravno, često ne postoji garancija da će mali program ostati mali;

- provjerljivost (verifiability): neki jezici su dizajnirani tako da kompajler (i ostali alati) nađe što više potencijalnih grešaka u programu prije izvršavanja, ali su takvi jezici obično manje fleksibilni; drugi jezici su fleksibilniji, ali se kod njih neke greške nalaze tek kod izvršavanja (npr. greške kod provjere tipova - kod dinamičkih jezika);

- razina apstrakcije (abstraction level): neki jezici se oslanjaju na direktnu upotrebu koncepata na razini hardvera (strojne instrukcije); drugi su apstraktniji;

- uloga u životnom ciklusu (lifecycle role): neki jezici su dizajnirani samo za implementaciju, drugi samo za specifikaciju, treći pokrivaju oba područja (npr. Eiffel);

- imperativnost nasuprot deskriptivnosti (imperativeness vs descriptiveness): imperativni jezici sastoje se od naredbi koje mijenjaju programsko stanje (varijable); drugi jezici su deskriptivni (često se kaže deklarativni) u matematičkom smislu, ostavljajući programskom sustavu da sam bira način kako da dođe do rezultata, bez da se navode precizni programski koraci;

- arhitekturni stil (architectural style): definira glavne kriterije za podjelu sustava u programske module; postoje dva glavna arhitekturna stila; po jednome se moduli organiziraju na temelju softverskih funkcija, a po drugome se moduli organiziraju na temelju tipova objekata sa kojima sustav radi; odgovarajući programski stilovi zovu se proceduralni (funkcijski znači nešto drugo, što će biti navedeno u nastavku), odnosno objektno-orijentirani stil (npr. jezik C je proceduralni, a C++ je objektno-orijentirani, iako nije čisti OOPL).

Naravno, moguće su različite kombinacije ovih kriterija. Npr. programski jezik Java je jezik opće namjene, namijenjen za široki obujam, provjerljiv (ima statičku provjeru tipova), relativno visoke razine apstrakcije, implementacijski (za specifikaciju se često koristi UML), imperativni, objektno orijentirani.

Kako autor navodi u [12], danas se najviše koriste imperativni, a usto i objektno-orijentirani jezici, kao što su Java, C++, Objective C, C#, sve redom jezici na vrhu liste korištenosti (na vrhu je još C, koji jeste imperativan, ali nije objektno-orijentiran). No, nisu svi zadovoljni sa imperativnim programiranjem, kod kojeg program mijenja stanje (stanje se može smatrati apstraktnijom verzijom pojma memorija računala) i time stvara popratne efekte (side effects). Zagovornici tzv. funkcijskog programiranja i jezika (functional programming and languages) žele takav način programiranja koji je što bliži matematici, u kojoj se ništa ne mijenja i ne stvaraju popratne efekte.

Osnovni koncept funkcijskih jezika je funkcija, koja odgovara matematičkom pojmu funkcije, kod koje se rezultat dobiva iz ulaznih argumenata, bez popratnih efekata. Time se postižu jednostavniji programi, o kojima je lakše rezonirati na temelju standardnih matematičkih tehnika. Može se reći da su funkcijski jezici ne-imperativni ili aplikativni, dok su suprotni termini deskriptivni i deklarativni, iako ti termini ne moraju uvijek značiti isto što i funkcijski (npr. za programski jezik SQL se često kaže da je deklarativan, a on nije funkcijski, nego relacijski jezik).

Prvi funkcijski jezik bio je Lisp (skraćenica od LISt Processing), čija je specifikacija napravljena još 1958. godine. On i njegove izvedenice naročito se koriste u SAD-u, posebno za potrebe programiranja aplikacija za umjetnu inteligenciju, dok se u Evropi i Japanu za te namjene puno više koristi logički programski jezik Prolog i njegove izvedenice. Lisp se obično nalazi unutar liste 20-tak najtraženijih programskih jezika (zanimljivo, i PL/SQL je među njima). Lisp je jezik sa dinamičkom provjerom tipova, a njegov najkorišteniji "potomak", Haskell, ima statičku provjeru tipova. Haskell se obično nalazi unutar liste 50-tak najtraženijih jezika, zajedno sa jezikom Prolog (tu se još uvijek nalaze i stari programski jezici Fortran i Cobol, a danas je tu i programski jezik Scala). Programski jezik Clojure također je Lispov "potomak", te ima dinamičku provjeru tipova.

Može se postaviti pitanje zašto funkcijski jezici nisu zastupljeniji, s obzirom na prednosti koje imaju. Autor u [12] navodi sljedeće razloge:

- performanse: u mnogim slučajevima, imperativni stil ima bolje performanse od funkcijskog stila;

- skalabilnost: za gradnju velikih softverskih sustava, notacije klase (koje postoje u objektno-orijentiranim jezicima) je puno efektivnija od notacije funkcije; no, danas su mnogi funkcijski jezici dodali neke objektno-orijentirane konstrukte; zapravo, vrijedi i obrnuto – mnogi objektno-orijentirani jezici dodali su neke funkcijske osobine (npr. funkcije višeg reda uveli su Eiffel i C#, dok se to očekuje u Javi 8; objektno-orijentirani programski jezik Scala tu je otišao najdalje – može se reći da Scala je i objektno-orijentiran i funkcijski jezik);

- neimanje stanja (statelessness): dok čisto aplikativni stil može značajno pojednostaviti algoritme nad kompleksnim strukturama podataka, neki aspekti računanja traže eksplicitno stanje (statefull); zato su neki funkcijski jezici (posebno Haskell) dodali neke specijalne mehanizme za promjenu stanja, ali mješavina više nije jednostavna kao što je osnovni funkcijski model.

U [12] autor, iako je u svoj objektno-orijentirani jezik Eiffel uspješno ugradio mnoge funkcijske osobine, ipak drži da će i u bliskoj budućnosti značajno prevladavati objektno-orijentirani jezici.

Za razliku od toga, autori u [20] drže da će se sve više koristiti multiparadigmatsko programiranje, tj. da će se koristiti ona jezična paradigma koja je najprirodnija za rješavanje određenog problema, a ne "jedan alat za sve probleme". Autori daju vrlo bogatu klasifikaciju modela programiranja, te zagovaraju pristup učenju programskih jezika na temelju jezgrenih jezika (kernel languages approach). Oni navode da u principu postoje tri pristupa učenju programskih jezika:

- učenje programiranja kao zanata (craft) na temelju određenog programskog jezika, koji pripada određenoj paradigmi; prednost takvog pristupa je da se dobro nauči određeni jezik i paradigma, ali na taj način programer naučiti razmišljati o "samo jednom alatu za sve probleme";

- učenje programiranja kao grane matematike; kod ovog pristupa, često se koriste jezici koji su toliko restriktivni ili toliko fundamentalni da nisu za praktičnu upotrebu;

- treći pristup je onaj koji zagovaraju autori; to je pristup na temelju koncepata, koji se elaboriraju od najjednostavnijih prema složenijima; ovaj pristup ima barem dvije moguće varijante; jedna je da se koristi više programskih jezika (npr. jedan objektni, jedan funkcijski, jedan logički ...) za učenje pojedinog koncepta, što nije baš jednostavno za učenje; druga varijanta se temelji na jezgrenim jezicima, koji su konceptualni jezici, neovisni od konkretnog programskog jezika; zapravo, najbolja podvarijanta druge varijante je kada se neki (ili još bolje – svi) konceptualni jezici mogu prikazati u sklopu jednog jedinog konkretnog programskog jezika; autori za to imaju konkretan programski jezik Oz (u čijem su razvoju i sami značajno sudjelovali), koji pokriva sve danas poznate programske paradigme; no, kako i autori kažu, Oz je dinamički jezik, što se neće sviđati baš svima. (Npr. nama su prihvatljiviji statički jezici, iako u praksi radimo i sa dinamičkim programiranjem, npr. sa dinamičkim SQL-om i dinamičkim PL/SQL-om, ali to radimo samo kad nemamo druge alternative.)

Autori u [20] definiraju osam modela računanja (computation models), na temelju tri svojstva:
- da li se koristi konkurentnost (concurrency); na sljedećoj slici označeno sa C;
- da li se koristi eksplicitno stanje (state), na slici označeno sa S;
- da li se koristi lijena (spora, pasivna) evaluacija (lazy evaluation; laziness), na slici označeno sa L; kod lijene evaluacije, računanje (evaluacija) se radi tek kad je stvarno potrebno, a ne "onda kad program dođe na taj dio", što je uobičajeno i zove se pohlepna evaluacija (eager evaluation).

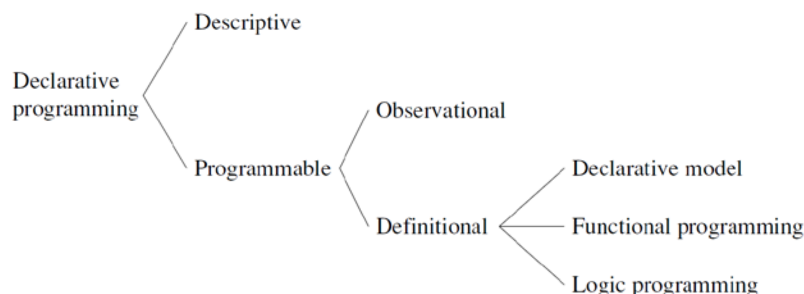
Slika (2.1) prikazuje tih osam modela računanja:

C	L	S	Description
			Declarative model (chapters 2 &3, Mercury, Prolog).
		×	Stateful model (chapters 6 & 7, Scheme, Standard ML, Pascal).
	×		Lazy declarative model (Haskell).
	×	×	Lazy stateful model.
×			Eager concurrent model (chapter 4, dataflow).
×		×	Stateful concurrent model (chapters 5 & 8, Erlang, Java, FCP).
×	×		Lazy concurrent model (chapter 4, demand-driven dataflow).
×	×	×	Stateful concurrent model with laziness (Oz).

Slika 2.1. Osam modela računanja; Izvor: [20]

Kako se vidi, postoje četiri deklarativna modela (svi oni koji nisu označeni kao statefull) i četiri nedeklarativna (statefull) modela. Napomenimo da dataflow (u Eager concurrent model) označava sljedeće ponašanje: imamo varijablu kojoj još nije pridružena vrijednost, pa (neka) operacija kojoj je ta varijabla potrebna jednostavno počeka da neka druga dretva (thread) pridruži vrijednost toj varijabli, kako bi (prva operacija) mogla nastaviti. Takva se varijabla uobičajeno naziva dataflow varijabla.

Autori daju i ovakvu klasifikaciju deklarativnog programiranja (slika 2.2):



Slika 2.2. Klasifikacija deklarativnog programiranja; Izvor: [20]

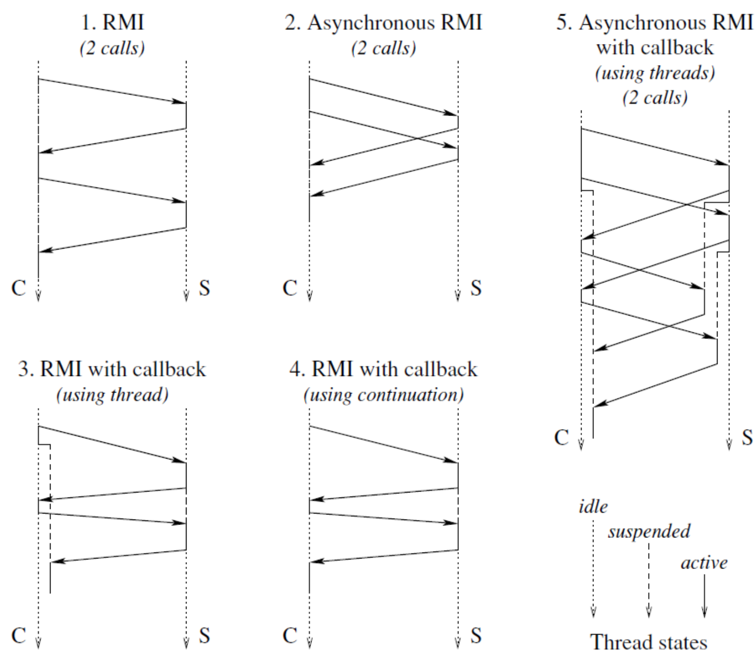
Primijetimo da se po ovoj klasifikaciji može zaključiti da logičko programiranje potpada pod deklarativno programiranje. No, autori na drugom mjestu kažu da se logičko programiranje, osim kroz deklarativno programiranje, može realizirati i kroz relacijsko programiranje i kroz programiranje na temelju ograničenja (constraint programming), koja (dva) nisu uključena u navedenih osam modela računanja!

Što se tiče konkurentnog programiranja, autori u [20] prikazuju ove pristupe (slika 2.2):

Model	Approaches
Sequential (declarative or stateful)	Sequential programming Order–determining concurrency Coroutining Lazy evaluation
Declarative concurrent	Data–driven concurrency Demand–driven concurrency
Stateful concurrent	Use the model directly Message–passing concurrency Shared–state concurrency
Nondeterministic concurrent	Stream objects with merge

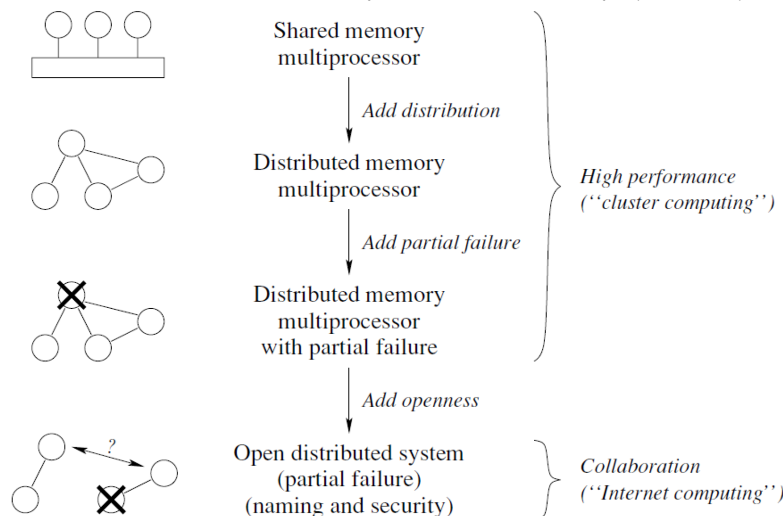
Slika 2.3. Različiti pristupi konkurentnom programiranju; Izvor: [20]

Na prethodnoj slici (2.3), kod statefull concurrent programming modela navedena su i ova dva pristupa: shared-state concurrency (dobro poznat, npr. kao kod Jave) i message-passing concurrency (kojega je realizirao npr. jezik Erlang, a u novije vrijeme i Scala). To je programski stil kod kojega se program sastoji od nezavisnih entiteta, aktora (actors), koji si šalju poruke asinkrono, bez čekanja na odgovor. Model aktora kreirao je 70-ih Carl Hewitt. Konkurentnost na temelju slanja poruka danas se jako zagovara, naročito kao temelj za tzv. multiagentne sustave. Konkurentnost na temelju slanja poruka je prirodni stil za distribuirane sustave i omogućava visoku raspoloživost. Sljedeća slika (2.4) prikazuje dijagrame poruka kod nekih jednostavnih protokola (RMI označava Remote Method Invocation):



Slika 2.4. Dijagrami poruka kod jednostavnih protokola; Izvor: [20]

Što se tiče distribuiranih sustava, autori u [20] daju ovakvu klasifikaciju (slika 2.5):

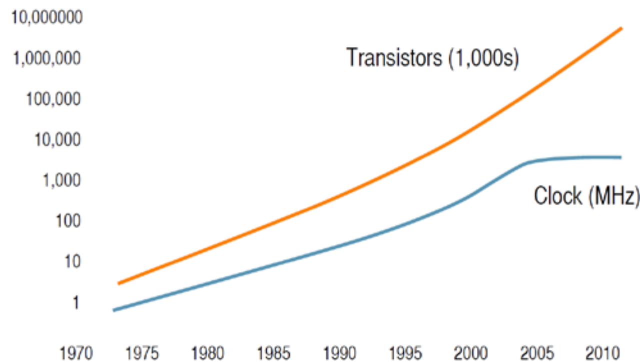


Slika 2.5. Jednostavna taksonomija distribuiranih sustava; Izvor: [20]

Autori u [20] prikazuju i relacijsko programiranje, za koje kažu da je puno fleksibilnije od funkcijskog programiranja. Prvo, kod relacijskog programiranja rezultat poziva može dati nula, jedan ili više odgovora. Drugo, kod relacijskog programiranja, kod svakog poziva procedure može biti različito što je ulaz (input), a što izlaz (output) procedure. Ta fleksibilnost čini relacijske jezike vrlo pogodnima za baze podataka i parsere, a naročito za deduktivne baze podataka i parsiranje ambicioznih gramatika. Kako je već prije rečeno, logičko programiranje može se ostvarivati kroz deklarativno, relacijsko, te programiranje na temelju ograničenja (constraint programming). Ovo zadnje je danas "najmoćnije" programiranje, a sastoji se od skupa tehnika za rješavanje problema zadovoljavanja uvjeta (constraint satisfaction problems).

3. NEKI TRENDOVI U RAZVOJU HARDVERA (I SISTEMSKOG SOFTVERA) KOJI UTJEČU NA KONKURENTNO PROGRAMIRANJE

Mooreov zakon vrijedi i dalje (naravno, to nije zakon niti u matematičkom, niti u fizikalnom smislu, to je statistička prognoza), a on kaže da se broj tranzistora na mikroprocesorskom čipu udvostručuje otprilike svake dvije godine. No, vrijeme jednostavnog povećanja brzine programa je prošlo. Radni takt procesora praktički je prestao rasti oko 2004. / 2005. godine, kako prikazuje sljedeća slika (3.1):



Slika 3.1. Povećanje broja tranzistora i radnog takta procesora od 1973. do 2010.

Razlog za to je prije svega veliko povećanje potrošnje struje procesora na velikim brzinama. Zbog toga su se proizvođači okrenuli drugačijem načinu povećanja performansi procesora. Umjesto povećanja brzine, povećali su broj CPU-a na jednom mikroprocesorskom čipu, tj. počeli su proizvoditi višejezgrene procesore. No, dok smo kod jednojezgrenih procesora povećanjem takta procesora dobili linearno povećanje brzine programa, kod višejezgrenih procesora program najčešće moramo pisati drugačije da bismo iskoristili raspoložive jezgre, tj. moramo preći na konkurentno programiranje. Razlog za to objašnjava Amdahlov zakon [9]. Ako je u nekom programu proporcija onih dijelova programa koji se mogu paralelno izvršavati jednaka *p* (što znači da je proporcija dijelova koji se ne mogu paralelno izvršavati jednaka *1 – p*), onda se povećanjem broja CPU-a može dobiti ovo povećanje:

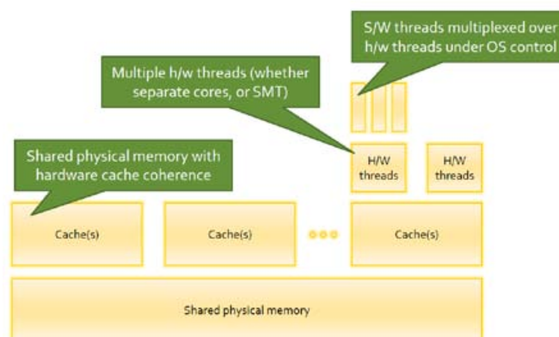
Sequential fraction
Parallel fraction

$$speedup = \frac{1}{1 - p + \frac{p}{n}}$$

Number of processors

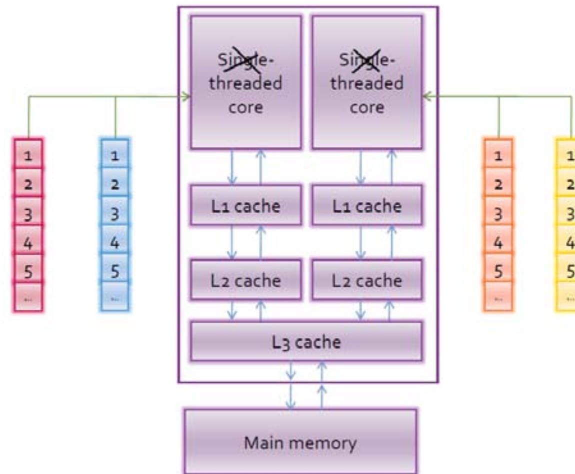
Npr. ako imamo 10 procesora i ako je moguće paralelizirati 90% programa, onda je maksimalno povećanje brzine 5,26 puta, što je skoro dva puta manje od broja procesora. Ako je moguće paralelizirati 99% programa, onda je povećanje 9,17 puta. Nažalost, konkurentne programe nije lako pisati (barem ne imperativne, za razliku od deklarativnih). Međutim, (neki) današnji procesori mogu znatno pomoći u pisanju konkurentnih programa, a posebno se očekuje da to budu oni procesori (zasad rijetki) koji podržavaju tzv. hardversku transakcijsku memoriju (HTM).

Autor u [8] daje sljedeći prikaz (slika 3.2) sistemskog modela tipičnog računala:



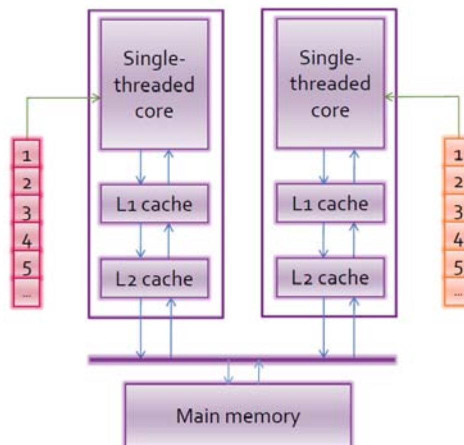
Slika 3.2. Sistemski model računala; Izvor: [8]

Pritom možemo imati računalo sa jednim procesorom, koji ima više jezgri, kao na slici 3.3:



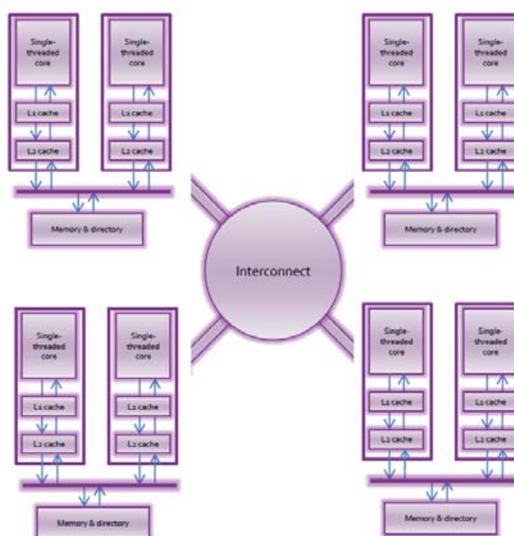
Slika 3.3. Računalo sa jednim procesorom, više jezgri (2) i više HW dretvi (4); Izvor: [8]

... ili računalo sa više procesora (sa jednom ili više jezgri) u SMP arhitekturi (Symmetric Multi-Processors), kao na slici 3.4 (procesori na slici imaju samo jednu jezgru i jednu HW dretvu):



Slika 3.4. SMP multiprocesor; Izvor: [8]

... ili računalo sa više procesora (sa jednom ili više jezgri) u NUMA arhitekturi (Non-Uniform Memory Access), kao na slici 3.5:



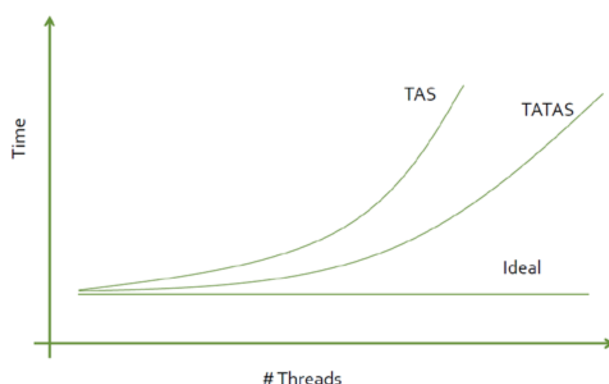
Slika 3.5. NUMA multiprocesor; Izvor: [8]

Autor u [8] navodi da su danas uobičajena tri načina sinkronizacije konkurentnih procesa (ili dretvi):
- lokoti, tj. blokirajuća sinkronizacija;
- neblokirajući sinkronizacijski mehanizmi, bez lokota (lock-free);
- softverska, hardverska i hibridna transakcijska memorija (STM, HTM, hibridna TM).

Pritom se koriste sljedeće vrste lokota:
- test-and-set (TAS) lokoti;
- test-and-test-and-set (TATAS) lokoti;
- lokoti temeljeni na redovima (queue-based locks);
- hijerarhijski lokoti;
- lokoti tipa čitatelj-pisac (reader-writer locks).

TAS lokoti su uobičajeni lokoti. Operacija TAS bila je osnovna operacija za sinkronizaciju u mikroprocesorima 90-ih godina. No, praktički svaki mikroprocesor dizajniran poslije 2000. godine podržava jaču operaciju compare-and-swap (CAS; zove se i compareAndSet, npr. u Java terminologiji), ili njoj ekvivalentnu (uzgred, CAS, kao i CASD, Compare-and-Swap-Double, nisu novost – bile su dio IBM 370 arhitekture od 1970).

TATAS lokoti imaju bolje performanse od TAS lokota, iako na početku rade dodatnu radnju provjere zaključanosti. No, ta dodatna radnja je bez čekanja (ako je lokot zaključan, operacija se odmah prekida), pa TATAS lokot ukupno radi brže, kao što pokazuje sljedeća slika (3.6).



Slika 3.6. Usporedba performansi TAS i TATAS lokota; Izvor: [8]

Što se tiče lokota čitatelj-pisac (reader-writer locks), oni uobičajeno dozvoljavaju da istovremeno radi više čitatelja, ili samo jedan pisac, ali ne čitatelji i pisac istovremeno. No, varijanta Read-Copy-Update (RCU) omogućava da istovremeno radi više čitatelja i jedan pisac.

Kako navodi autor u [2], RUC lokoti se intenzivno koriste u Linux kernelu od 2002. godine (autor radi na Linux kernelu). U istom radu, autor navodi i svoje neslaganje sa raširenim mišljenjem da je neblokirajuća sinkronizacija općenito bolja od lokota. On navodi da je multiprocesorsko programiranje bilo dobro poznato i prije ere višejezgrenih procesora, te da višejezgreni procesori ne stvaraju neke posebne dodatne poteškoće. Tvrdi da zapravo nisu višejezgreni procesori izazvali potrebu za konkurentnim programiranjem, već da je upravo zrelost konkurentnog programiranja potaknula proizvođače mikroprocesora da počnu raditi višejezgrene procesore. On se ne slaže sa tvrdnjom "nitko stvarno ne zna kako organizirati i održavati veliki sustav temeljen na zaključavanju" (koju npr. navode i autori u [9]). Također, mišljenja je da bi uglavnom trebalo izbjegavati neblokirajuće mehanizme za sinkronizaciju, jer je (po njegovom mišljenju) livelock puno teži za otkrivanje i popravljanje od deadlocka, a upravo neblokirajući mehanizmi povećavaju opasnost od livelocka.

Većina se ipak ne slaže sa prethodno navedenim mišljenjima i drži da je neblokirajuća sinkronizacija u načelu bolja od lokota. Neblokirajuća sinkronizacija temelji se na već navedenoj operaciji CAS (ili ekvivalentnoj). Kako se navodi u [9], strojevi (računala) koja imaju operacije kao što je compareAndSet znače na području asinkronog računanja (asynchronous computation) ono što Turingovi strojevi znače na području sekvencijalnog računanja (sequential computation): svaki konkurentni objekt koji se može potencijalno implementirati, može se implementirati na način bez čekanja (wait-free manner) na takvim strojevima. Po autorima u [9]: "compareAndSet() is the king of all wild things".

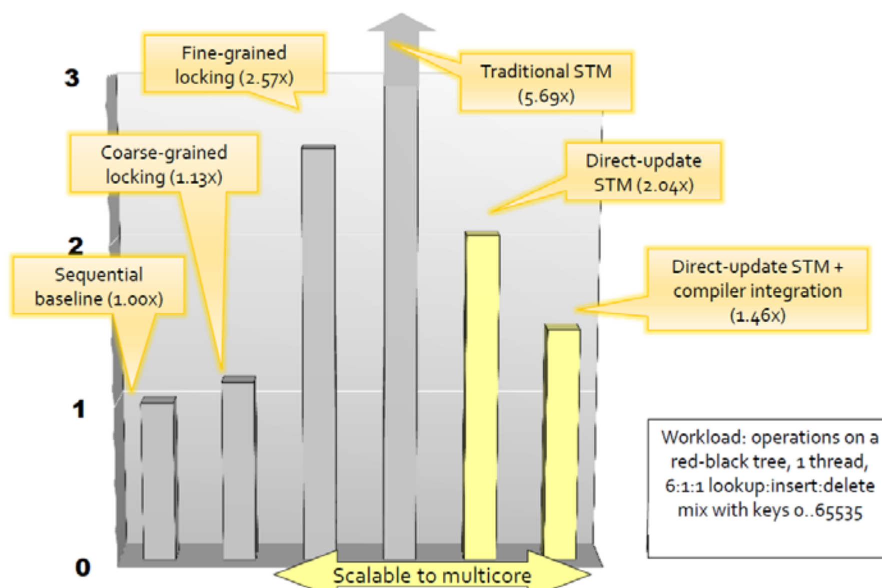
No autori u [9] navode i mane CAS operacije: algoritme koji koriste CAS (ili ekvivalentne operacije) vrlo je teško smisliti i često su vrlo neintuitivni. Osnovna teškoća sa svim današnjim sinkronizacijskim operacijama (pa i CAS) je da one rade na samo jednoj riječi memorije, što tjera na korištenje kompleksnih i neprirodnih algoritama.

Zapravo, postoji operacija CASD (Compare-and-Swap-Double), ali ona ažurira dvije susjedne riječi, a još nije realizirana operacija DCAS koja bi ažurirale dvije memorijske riječi na nezavisnim lokacijama. Pogotovo nije realizirana nekakva operacija multiCompareAndSet(). No, čak i da postoji, niti ona ne bi u potpunosti riješila problem sinkronizacije.

Problem sa svim dosadašnjim sinkronizacijskim mehanizmima i operacijama (i onima koje postoje i navedenim nepostojećim operacijama), bez obzira da li rade ili ne rade zaključavanje, je da se ne mogu lagano komponirati, što ima veliki negativan utjecaj na modularnost konkurentnih programa. Zato je izmišljena transakcijska memorija (TM), a njena realizacija može biti softverska (STM), hardverska (HTM) ili hibridna. Ispravno implementirane, transakcije nemaju problem deadlocka ili livelocka.

Postoje različite softverske implementacije transakcijske memorije. Jednu implementaciju za Javu daju autori u [9] (inače, jedan autor je prvi dao ideju hardverske transakcijske memorije, a drugi autor je (su)inventor softverske transakcijske memorije). Programski jezik Clojure podržava STM na razini jezika, a Scala podržava STM na razini biblioteke. Java za sada ne podržava STM niti na razini jezika, niti na razini biblioteka.

Autor u [7] navodi kako su se u zadnjih pet godina performanse STM sustava značajno poboljšale, posebno onih koji su integrirani sa optimizirajućim kompajlerima. Sljedeća slika (3.7) prikazuje razvoj STM-a (žuto su označene današnje performanse STM-a; vidi se da čak nadmašuju fino-zrnate lokote):



Slika 3.7. Poboljšanja kod današnjih realizacija STM-a (žuto); Izvor: [8]

Autor u [7] navodi da postoje različite varijante STM-a. Neke se temelje na lokotima, a neke na neblokirajućim mehanizmima. Lokoti mogu biti pesimistični ili optimistični. Autor navodi da su prve varijante STM-a uglavnom bile temeljene na neblokirajućim mehanizmima, ali se i kod STM-a pokazalo da je teško pisati neblokirajuće algoritme koji osiguravaju atomarnost više operacija. Dalje, koriste se dvije različite tehnike za upravljanje verzijama podataka u memoriji, koje sliče dobro poznatim rješenjima kod baza podataka. Ponekad se koristi undo-log, koji omogućava da se već promijenjeni podaci vrate na prethodnu vrijednost (ukoliko se pokaže da transakciju treba abortirati). U drugim se varijantama koristi redo-log mehanizam, kod kojeg se promjene pišu samo u redo-log, a onda se upisuju u memoriju tek kada treba potvrditi transakciju (naravno, i undo-log i redo-log su također memorijske strukture).

Unatoč poboljšanjima STM-a, autor navodi da još nije sigurno može li STM biti koristan u praksi, bez podrške od strane hardvera. U svakom slučaju, navodi da STM ima neke prednosti pred HTM-om:

- softver je fleksibilniji od hardvera, pa STM omogućava širi raspon sofisticiranih algoritama;
- softver je lakši za modifikaciju i poboljšavanje;
- STM se lakše integrira sa postojećim sustavima i mogućnostima programskih jezika (kao što je npr. garbage collection);
- STM ima manje ograničenja za razvoj u odnosu na HTM, koji je ograničen postojećim hardverskim rješenjima (kao što su npr. cachevi).

S druge strane, autor navodi i prednosti HTM-a nad STM-om, koje se u načelu mogu svesti na (značajno) bolje performanse i dobru podršku za sistemske jezike. No, autor naglašava da se HTM mora vrlo rigorozno verificirati i validirati, i da se mora paziti da povećanje kompleksnosti procesora (zbog uvođenja HTM-a) ne naškodi performansama.

I u [9] je dat prikaz hardverske transakcijske memorije (HTM). Prikazano je kako se standardna hardverska arhitektura može prilagoditi za podršku kratkotrajnih i malih (po veličini) transakcija. Temeljna ideja za podršku HTM-a je u tome da današnji mikroprocesori podržavaju protokole usklađivanja priručne memorije (cache-coherence protocols), pa time već podržavaju većinu toga što je potrebno za realizaciju HTM-a. Oni već detektiraju i rješavaju sinkronizacijske konflikte između dretvi pisaca (writers), kao i između dretvi čitatelja (readers) i dretvi pisaca, te stavljaju u međuspremnik (buffer) neke probne izmjene (umjesto da ih direktno upisuju u glavnu memoriju). Treba dodati u hardver samo još neke detalje.

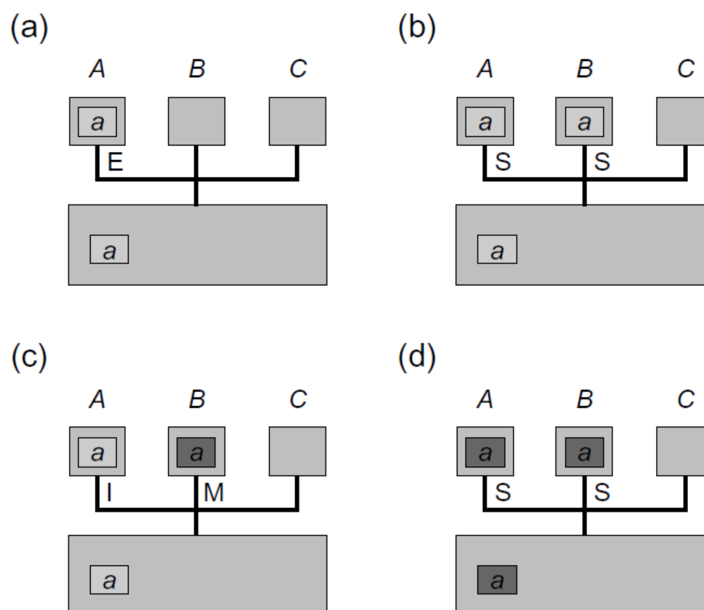
Svaki procesor ima svoju priručnu memoriju. Priručna memorija sastoji se od grupa memorijskih riječi. Grupa se naziva linija (line) i ima oznaku (tag), koja pokazuje stanje linije. Koristi se tzv. MESI protokol, u kojem je svaka linija u jednom od sljedeća četiri stanja (MESI = početna slova tih stanja):

- **M**odified: linija je modificirana i eventualno treba biti upisana natrag u (glavnu) memoriju; nijedan drugi procesor nema tu liniju u svom međuspremniku;
- **E**xclusive: linija nije modificirana, i nijedan drugi procesor ju nema u međuspremniku;
- **S**hared: linija nije modificirana, ali drugi procesori ju mogu imati u međuspremniku;
- **I**nvalid: linija ne sadrži suvisle podatke.

MESI protokol detektira sinkronizacijske konflikte između individualnih čitanja i pisanja, te osigurava da se procesori usklade oko stanja (djeljive) memorije. Kada procesor čita ili piše u memoriju, emitira (broadcasts) zahtjev na sabirnicu (bus), a ostali procesori to slušaju (ponekad se to zove snooping). Pojednostavljeno se dešava sljedeće:

- kada procesor daje zahtjev za učitavanje linije u ekskluzivnom modu, ostali procesori invalidiraju svoju kopiju; svaki procesor sa modificiranom kopijom mora upisati svoju kopiju u memoriju, prije nego se nastavi učitavanje;
- kada procesor daje zahtjev za učitavanje linije u djeljivom modu, svi procesori koji imaju ekskluzivnu kopiju moraju joj promijeniti stanje u Shared; ostatak je kao prije;
- ako priručna memorija postane puna, mora se izbaciti neka linija; ako je ta linija Shared ili Exclusive, jednostavno se zanemari; ako je Modified, mora se upisati u memoriju.

Slika 3.8 prikazuje jedan slijed događaja. Prvo je (a) procesor A učitao (neku) liniju u ekskluzivnom modu. Onda je (b) procesor B isto učitao tu liniju, pa je ona sada u djeljivom modu. Zatim je (c) procesor B mijenjao tu liniju, pa je kod njega ona u statusu Modified, a kod procesora A je u statusu Invalid. Na kraju je (d) procesor B upisao tu liniju u memoriju, procesor A je osvježio svoju liniju, i oba procesora opet imaju status linije Shared:



Slika 3.8. MESI protokol; Izvor: [9]

Kako navode autori u [9], za podršku HTM-a treba zadržati MESI protokol kakav jeste, samo treba dodati jedan transakcijski bit u svaku oznaku linije priručne memorije (cache line's tag). Uobičajeno je taj bit postavljen na 0. Kada se u priručnoj memoriji mijenja vrijednost kao posljedica transakcije, bit se postavlja na 1. Treba samo osigurati da se modificirane transakcijske linije ne upisuju natrag u memoriju i da invalidacija linije abortira transakciju. Mana ove sheme je zajednička mana skoro svih HTM shema.

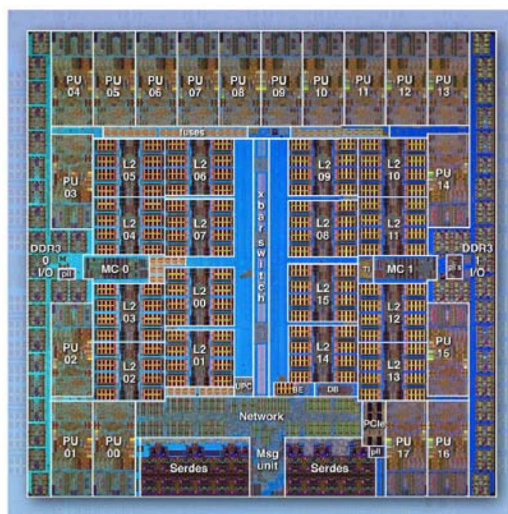
Prvo, veličina transakcije ograničena je veličinom priručne memorije (zato npr. IBM BlueGene/Q mikroprocesor ima čak 32 MB L2 memorije, koja se koristi za transakcije). Drugo, većina operacijskih sustava briše priručnu memoriju kada se dretva pasivizira, tako da trajanje transakcije može biti limitirano dužinom "vremenskog kvanta" operacijskog sustava. Po tome bi slijedilo da je HTM najbolji za kratkotrajne i male transakcije, a ostale transakcije trebale bi koristiti STM, ili kombinaciju HTM-a i STM-a.

Unatoč brojnim otvorenim pitanjima vezanima za HTM i teškoćama kod razvoja procesora sa HTM-om, danas postoje barem dva mikroprocesora koja podržavaju HTM. Vega procesor firme Azul Systems ima HTM već više od pet godina, a prošle godine (kolovoz 2011.) mu se pridružio i BlueGene/Q firme IBM. No, otkazan je razvoj procesora Oracle (prije Sun) Rock, koji je isto trebao podržavati HTM.

Procesor Vega 3 prvenstveno je ciljan za tržište Java servera. Ima 54 jezgre, a moguće je spojiti 16 procesora (koristi se UMA arhitektura), čime se dobije sustav sa 864 jezgre. Osim Vega procesora (verzije 1, 2 i 3), Azul Systems je prilagodio Sun JVM tako da podržava heap veličine i preko 500 GB, sa maksimalnim pauzama za GC (Garbage Collector) od samo 10-20 ms, a to zahvaljujući i podršci procesora Vega za GC. Prije par mjeseci Azul Systems najavio je svoju inicijativu da podržava Open Source Community sa besplatnim Zing JVM-om.

Zanimljivo je napomenuti da IBM BlueGene/Q ima 18 jezgri, od kojih je 16 namijenjeno za rad aplikacijskih programa, jedna je namijenjena za operacijski sustav, a jedna je jezgra pričuvna (spare) i može zamijeniti bilo koju od preostalih 17 jezgara koja se pokvari. Koristi se (i) kod IBM superračunala Sequoia (završeno ove godine, za Lawrence Livermore National Labs), koje se sastojati od 100 000 čipova BlueGene/Q. Stručnjaci IBM-a predviđaju da će se kod navedenog superračunala svaka 3 tjedna pokvariti (u prosjeku) jedna jezgra, i zato je pričuvna jezgra vrlo značajna kod tog superračunala. Naglasimo da IBM BlueGene/Q ima jedno značajno ograničenje kod HTM-a. Naime, njegova realizacija HTM-a ograničena je na jedan mikroprocesorski čip, tj. na procesorske jezgre unutar jednog čipa. Kod spajanja više mikroprocesorskih čipova, transakcijska memorija između njih ne radi. Istina, to ne stvara veliki problem kod IBM superračunala Sequoia, koje je projektirano za specifične namjene.

Sljedeća slika (3.9) prikazuje procesor IBM BlueGene/Q:



Slika 3.9. BlueGene/Q procesor; Izvor: [6]

Intel u svojim najavama mikroprocesora Haswell arhitekture (prvi bi trebao izaći na tržište 2013. godine) navodi da će on isto podržavati transakcijsku memoriju. Za razliku od mikroprocesora IBM BlueGene/Q, Intelovi mikroprocesori će izgleda podržavati transakcijsku memoriju i između mikroprocesorskih čipova, a ne samo između jezgri jednog čipa

Intelovi budući mikroprocesori zanimljivi su i po tome što će imati dva načina podržavanja transakcijske memorije. Jedan način, tzv. Hardware Lock Elision, omogućit će da se programi koji koriste zaključavanje vrlo lako prerade tako da rade sa transakcijskom memorijom. Suština je u tome da će procesor "lagati" dretve da su dobile lokot, a da stvarno nikakvog zaključavanja neće biti. Novonapisani programi moći će koristiti napredniji način Restricted Transactional Memory, tj. moći će koristiti eksplicitne naredbe XBEGIN, XEND ili XABORT za pokretanje transakcije, normalno završavanje ili abortiranje. Kod pokretanja nove transakcije (sa XBEGIN) dretva će moći navesti i fallback rutinu, koja će se automatski izvršiti ako transakcija ne uspije.

4. JAVA I KONKURENTNO PROGRAMIRANJE

U ovoj točki prvo ćemo ukratko ponoviti neke poznate mehanizme za podršku konkurentnom programiranju, koje Java ima od svojih početaka, te ukratko spomenuti par relativno novih mogućnosti, uvedenih od Jave 5 dalje. Zatim ćemo pogledati primjere grešaka iz prakse iz [3], te neke preporuke na području konkurentnog programiranje iz [19] (ta je knjiga preporučljiva kao nastavak na [5]).

Svaki Java program ima barem jednu dretvu, onu koja izvršava metodu main(). Kada želimo napraviti svoju dretvu, imamo u Javi dva načina. Jedan način je da napravimo klasu koja nasljeđuje klasu Thread i da nadjačamo (override) metodu run(). Problem je u tome što u Javi (za sada) postoji samo jednostruko nasljeđivanje ponašanja, tj. klasa (a ne samo sučelja). Bez višestrukog nasljeđivanja klasa, ako naša klasa "potroši" jednostruko nasljeđivanje za klasu Thread, ne može naslijediti od neke druge klase. Zbog toga se češće koristi drugi način kreiranja dretve. Prvo se napiše "pomoćna" klasa koja nasljeđuje sučelje (interface) Runnable, pri čemu mora implementirati metodu run(). Onda se objekt te "pomoćne" klase šalje konstruktoru koji kreira objekt klase Thread (tj. dretvu) u kodu "glavne" klase.

Na http://www.infoq.com/articles/neal-gafter-on-java je diskusija od 14.09.2011 (tada je već bila izašla Java 7) sa znanstvenikom Nealom Gafterom, koji sada radi za Microsoft Research, ali je prije radio za Sun i bio primarni dizajner i implementator Java SE 4 i Java SE 5 proširenja. On navodi da su dretve u Javi skupe, i da je to razlog zašto se koristi thread pool. A razlog zašto su dretve skupe je taj što je u Javi stog (stack) unaprijed alocirane veličine. Dretve dijele istu gomilu (heap), ali svaka dretva ima svoj stog. Budući da su stogovi prealocirane veličine, a ne želi se da dretva pukne zbog prekoračenja kapaciteta stoga, onda stog mora biti kreiran sa dovoljnom veličinom, što znači da nije moguće kreirati veliki broj dretvi (npr. stotine tisuća, pa ni desetke tisuća), jer ne bi bilo dovoljno memorije. Kako kaže Gafter, sve bi bilo puno bolje kad bi se mogao imati segmentirani stog, koji bi inicijalno zauzimao manje memorije i mogao se proširivati po potrebi. Tada bi se moglo kreirati puno više dretvi i konkurentno programiranje bi bilo jednostavnije. Gafter ističe: ako bi morao glasati o nečemu što će poboljšati Javu, glasao bi za segmentirane stogove.

Kada se dvije dretve (ili više njih) upliću jedna drugoj u posao, npr. tako da modificiraju isti objekt, nastaje problem - to može stvoriti netočne rezultate i naziva se race condition. Taj se problem rješava pomoću sinkronizacije, koja se zove međusobno isključivanje (mutual exclusion), za što Java ima jednostavno rješenje - svaki objekt u Javi ima lokot (lock), koji se automatski nasljeđuje od superklase Object. Lokot istovremeno može držati (zaključati) samo jedna dretva.

Objekt koji će služiti kao lokot može se kreirati npr. ovako:

```
Object lock = new Object();
```

Dretva koja će tražiti lokot (tj. zaključati lokot) to radi pomoću naredbe synchronized, koja označava početak tzv. synchronized bloka. Inače, inače synchronized i volatile su jedine Java ključne riječi vezane za konkurentnost, a ostalo su metode klase Object (wait, wait(msec), notify, notifyAll) ili metode klasa specijaliziranih za konkurentnost (iz paketa java.util.concurrent):

```
synchronized(lock) {
  // critical section
}
```

Kada dretva dođe do početka tog bloka, pokuša zaključati lokot objekta koji je naveden kao argument naredbe synchronized. Ako je lokot zaključan od neke druge dretve, polazna dretva čeka dok on ne postane otključan. Nakon toga ga polazna dretva drži zaključanog sve do kraja tog bloka.

Osim bloka, i cijela metoda (funkcija / procedura) može imati synchronized na početku:

```
synchronized type method(args) {
  // body
}
```

što je, zapravo, isto kao i ovo:

```
type method(args) {
  synchronized(this) {
    // body
  }
}
```

Prethodno je slično konceptu monitora, ali dizajneri Jave su napravili određena odstupanja od tog koncepta. Java objekt se razlikuje od monitora u tri važne stvari, kompromitirajući time sigurnost dretve:

- atributi ne moraju biti privatni;
- metode ne moraju biti sinkronizirane;
- unutarnji lokot je pristupačan klijentima.

Zbog toga je jedan od inventora monitora, Per Brinch Hansen, 1999. godine izjavio [1]: "Zaprepašćuje me da je ovaj nesigurni Java paralelizam shvaćen tako ozbiljno od programske zajednice, četvrt stoljeća nakon invencije monitora i programskog jezika Concurrent Pascal."

Zaštita pristupa djeljivim varijablama nije jedini razlog zašto dretve moraju biti međusobno sinkronizirane. Često treba odgoditi izvođenje metode (ili dijela metode) u nekoj dretvi, dok se ne zadovolji određeni uvjet (a taj uvjet nije samo otključavanje određenog lokota). To se zove sinkronizacija na temelju uvjeta (condition synchronization), koja se u Javi implementira pomoću naredbi wait / notifyAll / notify, koje se pozivaju nad sinkroniziranim objektima.

Jedan primjer problema koji traži sinkronizaciju na temelju uvjeta je tzv. problem proizvođač-potrošač (producer-consumer problem), koji je,u različitim varijantama, čest u praksi. Može se apstraktno opisati na ovaj način ([5]):

- Proizvođač: u svakoj iteraciji (beskonačne) petlje, proizvodi podatke koje potrošač konzumira;
- Potrošač: u svakoj iteraciji (beskonačne) petlje, troši podatke koje je proizveo proizvođač.

Proizvođači i potrošači komuniciraju preko djeljivog međuspremnika (buffer) koji implementira red (queue), za koji smatramo (u ovom primjeru) da je neograničen, pa će nam biti važno samo da li je prazan. Prvo pretpostavimo da imamo klasu Buffer (koja implementira neograničeni red) i da imamo jedan objekt te klase:

```
Buffer buffer = new Buffer();
```

Proizvođači dodaju podatke na kraj reda, koristeći metodu (reda) void put(int item), a potrošači skidaju podatak sa reda koristeći metodu int get(). Broj podataka koji se nalaze u redu može se dobiti pomoću metode int size(). Dretva potrošač treba čekati dok red bude neprazan i tek tada pročitati podatak iz njega. Čekanje se u Javi može napraviti pomoću metode wait(), koja se može pozvati samo nad objektom koji je prethodno zaključan, tj. samo unutar synchronized bloka. Wait() tada blokira tekuću dretvu i otpušta lokot koji je dretva držala. Slijedi primjer iz [5]:

```
public void consume() throws InterruptedException {
  int value;
  synchronized(buffer) {
    while (buffer.size() == 0) {buffer.wait();}
    value = buffer.get();
  }
}
```

Sada lokot može preuzeti (zaključati) neka druga dretva, koja može mijenjati stanje navedene kondicije. Da obavijesti prvu dretvu o promjeni kondicije, druga dretva poziva notify(), što odblokira prvu dretvu, koja čeka na taj lokot. No, druga dretva ne otključa lokot odmah, već tek na kraju svog synchronized bloka (unutar kojega je i pozvala notify()). Primjer u kojem dretva proizvođač obavještava potrošača o promjeni kondicije:

```
public void produce() {
  int value = random.produceValue();
  synchronized(buffer) {
    buffer.put(value);
    buffer.notify();
  }
}
```

Proizvođač je proizveo neki slučajni broj, zaključao red, spremio podatak u njega i pozvao notify(), čime je odblokirao jednu (bilo koju!) dretvu koja je čekala na taj lokot. Na kraju synchronized bloka (što je u ovom slučaju bilo odmah iza) je i otključao taj lokot. No, odblokirana dretva ne može biti sigurna da je taj uvjet valjan i kada ona dođe na red, jer je u međuvremenu neka treća dretva mogla potrošiti podatak i red je možda opet prazan. Stoga je dretva potrošač morala ispitivati uvjet u while petlji. Važno je i to da notify() uvijek odblokira samo jednu (bilo koju!) dretvu koja čeka na određeni lokot. Stoga je u praksi puno sigurnije pozivati notifyAll(), koja odblokira sve dretve koje čekaju na određeni lokot.

U Javi verzije 5, koja se pojavila 2004. godine, uvedeno je dosta novina na drugim područjima (npr. generičke klase, bolje kolekcije i dr.), ali i na području konkurentnog programiranja. JSR 166, koji se odnosi na konkurentnost, temeljen je uglavnom na paketu edu.oswego.cs.dl.util.concurrent, kojega je napravio Doug Lea. Kroz novi paket java.util.concurrent uvedene su sljedeće nove mogućnosti:

- Executors (thread pools, scheduling);
- Futures;
- Concurrent Collections;
- Locks (*ReentrantLock*, *ReadWriteLock...*);
- Conditions;
- Synchronizers (Semaphores, Barriers...);
- Atomic variables;
- System enhancements.

U Java verziji 6, koja se pojavila godinu i pol nakon verzije 5, nije se pojavilo ništa revolucionarno, ali su se na području konkurentnog programiranja (kao i na drugim područjima) "iznutra" poboljšale biblioteke, bilo da su se riješili bugovi ili poboljšale performanse. U Java verziji 7, koja je izašla u ljeto prošle godine (2011.) u području konkurentnog programiranja novost je Fork/Join framework.

Prikazat ćemo samo jedno od tih poboljšanja, konkurentne kolekcije (collections), na temelju [19]. Kolekcije su u Javi originalno postojale od početka, ali su bile kreirane za siguran rad sa dretvama (thread safety), a ne za visoke performanse. Kasnije kolekcije, kao ArrayList, donijele su brzinu na račun sigurnosti. Opet na uštrb performansi, a u korist sigurnosti, kasnije su uvedeni sinkronizacijski omotači (synchronized wrapper). Dakle, moralo se birati između sigurnosti i performansi. Onda su u Javi 5 uvedene (kroz java.util.concurrent) konkurentne kolekcije, kao što je npr. ConcurrentHashMap, tako da se sada može dobiti i sigurnost i dobre performanse.

Ako mijenjamo sinkroniziranu kolekciju i istovremeno iteriramo kroz nju, dobit ćemo ConcurrentModificationException. Drugim riječima, kada iteriramo kroz sinkroniziranu kolekciju, zaključamo ekskluzivni lokot. Za razliku od toga, konkurentne kolekcije dozvoljavaju modifikaciju dok iteriramo kroz kolekciju. Slijedi primjer ponašanja ConcurrentHashMap u odnosu na (dvije) starije vrste mapa (map). U sljedećem kodu iz [19], iteriramo kroz mapu (nekih) rezultata u odvojenim dretvama. Upravo u sredini iteracije, dodajemo novi ključ u mapu, što je označeno crvenom bojom:

```
public class AccessingMap {
  private static void useMap(final Map<String, Integer> scores)
    throws InterruptedException {
    scores.put("Fred", 10);
    scores.put("Sara", 12);
    try {
      for(final String key : scores.keySet()) {
        System.out.println(key + " score " + scores.get(key));
        scores.put("Joe", 14);
      }
    } catch(Exception ex) {
      System.out.println("Failed: " + ex);
    }
    System.out.println("Number of elements in the map: " +
      scores.keySet().size());
  }
```

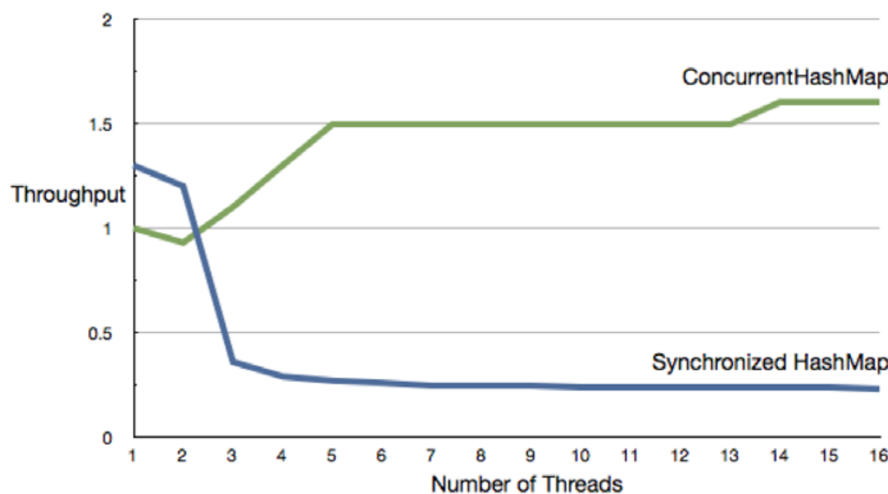
Ako koristimo HashMap ili njen sinkronizirani omotač, desit će se iznimka (exception) ConcurrentModificationException. Ako koristimo ConcurrentHashMap, iznimka se neće desiti. Sljedeći kod koristi tri vrste mapa:

```
public static void main(final String[] args)
    throws InterruptedException {
    System.out.println("Using Plain vanilla HashMap");
    useMap(new HashMap<String, Integer>());
    System.out.println("Using Synchronized HashMap");
    useMap(Collections.synchronizedMap(new HashMap<String, Integer>()));
    System.out.println("Using Concurrent HashMap");
    useMap(new ConcurrentHashMap<String, Integer>());
  }
}
```

Ovo su rezultati izvršavanja navedenog koda:

Using Plain vanilla HashMap
Sara score 12
Failed: java.util.ConcurrentModificationException
Number of elements in the map: 3
Using Synchronized HashMap
Sara score 12
Failed: java.util.ConcurrentModificationException
Number of elements in the map: 3
Using Concurrent HashMap
Sara score 12
Fred score 10
Number of elements in the map: 3

Osim što dozvoljavaju prepletanje čitanja i pisanja, konkurentne kolekcije imaju i bolju propusnost (throughput) u odnosu na njihove sinkronizirane verzije, jer ne koriste ekskluzivne lokote. Sljedeća slika (4.1) prikazuje razlike u propusnosti između sinkronizirane kolekcije i ConcurrentHashMap kolekcije. U svakoj dretvi, autor je generirao slučajni ključ, te radio insert u oko 80% vremena. Ako je ključ već postojao, uklanjao ga je u oko 20% vremena. To je rađeno na sustavu koji je imao procesor sa osam jezgri i zato se vidi da se propusnost nakon osam dretvi (zapravo, i nešto prije) uglavnom ne povećava:



Slika 4.1. Usporedba propusnosti ConcurrentHashMap i sinkronizirane kolekcije; Izvor: [19]

Pogledat ćemo neke primjere grešaka u konkurentnom programiranju, iz materijala [3] (prikazat ćemo tri primjera od šest navedenih u [3]). Kako kažu autori, studiranjem obrazaca (pattern) konkurentnih grešaka, povećava se naše poznavanje konkurentnog programiranja, te se uči prepoznavanje programskih idioma koji rade, ili ne rade dobro. Prvi primjer je iz HTTP servera Jetty. Tu su se koristile ne-atomarne (non-atomic) operacije na volatilnim (tj. promjenjivim) poljima bez zaključavanja:

```
// Jetty 7.1.0, org.eclipse.jetty.io.nio, SelectorManager.java, line 105
private volatile int _set;
...
public void register(SocketChannel channel, Object att) {
  int s = _set++;
  ...
}
...
public void addChange(Object point) {
  synchronized (_changes) {...}
}
```

Iako je varijabla _set označena kao volatilna (volatile), to nije dovoljno, jer je operacija _set++ ne-atomarna, tj. ona se ne izvršava kao jedinstvena, nedjeljiva operacija, već se u osnovi sastoji od tri diskretne operacije: čitaj-mijenjaj-piši.

Budući da se operacija register simultano poziva iz različitih dretvi, trebala je biti zaštićena sa lokotom (kao npr. metoda addChange, koja ima synchronized). Suština prethodnog problema je u tome da volatile ne osigurava atomarnost varijable, već samo njenu vidljivost. Općenita pouka je da volatilna varijabla koja nije zaštićeno sa lokotom može uzrokovati race condition, ako se ne-atomarna operacija, u kojoj se ta varijabla mijenja, poziva iz različitih dretvi. Dakle, u takvim slučajevima treba koristiti sinkronizirane blokove, ili klase lokota, ili atomarne klase iz java.util.concurrent, a ne ključnu riječ volatile.

Sljedeći primjer iz [3] prikazuje tzv. curenje lokota (lock leak). U Javi 5, kroz java.util.concurrent, uvedeni su eksplicitni lokoti (lokoti klasa ReentrantLock i ReentrantReadWriteLock). Oni imaju prednosti u odnosu na "staru" synchronized varijantu, ali omogućavaju i da se napravi greška – ako se otključavanje lokota ne stavi u exception dio, a desi se iznimka (exception). Ovo je pogrešan kod, jer se kod greške u metodi accessResource preskače otključavanje lokota i skače na obradu greške:

```
private final Lock lock = new ReentrantLock();
public void lockLeak() {
   lock.lock();
   try {
     accessResource();  // access the shared resource
     lock.unlock();
   } catch (Exception e) {}
}
```

Ispravno je da se otključavanje stavi u finally dio, koji se izvršava i kad se desi greška:

```
private final Lock lock = new ReentrantLock();
public void lockLeak() {
   lock.lock();
   try {
     accessResource(); // access the shared resource
   } catch (Exception e) {}
   finally {
     lock.unlock();
   }
}
```

Sljedeći primjer iz [3] prikazuje nešto što ne uzrokuje grešku, ali može uzrokovati loše performanse. U primjeru je sinkronizacija izvršena korektno (neće doći do greške), ali unutar sinkroniziranog bloka se nalazi više programskog koda nego što je nužno potrebno:

```
public class Operator {
  private int generation = 0; //shared variable
  private float totalAmount = 0; //shared variable
  private final Object lock = new Object();

  public void workOn(List<Operand> operands) {
    synchronized (lock) {
      int curGeneration = generation; //requires synch
      float amountForThisWork = 0;
      for (Operand o : operands) {
        o.setGeneration(curGeneration);
        amountForThisWork += o.amount;
      }
      totalAmount += amountForThisWork; //requires synch
      generation++; //requires synch
    }
  }
}
```

Bolje performanse mogu se postići ako se sinkronizirani blok podijeli u dva, i u svaki stavi samo ono što je potrebno, tako da izvan sinkronizacije (tj. kada ništa nije zaključano) u ovom slučaju bude for petlja. Naravno, uvijek treba vrlo pažljivo analizirati da li nešto smije biti izvan sinkronizacijskog bloka i da li se jedan sinkronizacijski blok smije podijeliti u dva (ili više).

Slijedi primjer (mijenjana je samo metoda workOn) koji ima bolje performanse (na računalu koje ima više procesora ili jezgri):

```
public void workOn(List<Operand> operands) {
  int curGeneration;
  float amountForThisWork = 0;
  synchronized (lock) {
    curGeneration += generation++;
  }
  for (Operand o : operands) {
    o.setGeneration(curGeneration);
    amountForThisWork += o.amount;
  }
  synchronized (lock) {
    totalAmount += amountForThisWork;
  }
}
```

Slijede neke preporuke iz [19] vezane za konkurentno programiranje, u kojima autor pokazuje kako podijeliti problem na manje dijelove, kako odrediti koliko dretvi (misli se na softverske, a ne na hardverske dretve) treba kreirati i koliko veliko ubrzanje očekivati (u odnosu na sekvencijalni rad). Za kompleksne probleme, treba kreirati barem toliko dretvi koliko ima procesorskih jezgri na raspolaganju (ili koliko ima hardverskih dretvi, ako ih procesor podržava). Broj jezgri se lako nađe sa:

Runtime.getRuntime().availableProcessors();

Ako je problem računski vrlo intenzivan, tj. uglavnom ovisi o procesoru, a ne o IO (Input/Output) operacijama, onda broj dretvi treba postaviti tako da bude jednak broju jezgri. No, ako je problem IO intenzivan, onda se može kreirati više dretvi nego što ima procesorskih jezgri (ili hardverskih dretvi). Za određivanje broja dretvi može se koristiti jednostavna formula:
broj_dretvi = broj_raspoloživih_jezgri / (1 – koeficijent_blokiranja)

gdje je koeficijent blokiranja (blocking coefficient) između 0 i 1. Naravno, koeficijent blokiranja nije lako odrediti. No, računski intenzivni problemi imaju koeficijent koji se približava nuli, tako da je preporučeni broj dretvi jednak ili nešto malo veći od broja raspoloživih jezgri. IO intenzivni problemi mogu imati koeficijent koji se i približava jedinici. Ako je koeficijent blokiranja npr. 50%, onda je po ovoj formuli preporučeni broj dretvi duplo veći od broja raspoloživih jezgri.

No, sada treba odrediti i kako podijeliti problem, tj. odrediti koje će dijelove imati program. Svaki dio radit će konkurentno, pa svakako treba imati barem onoliko dijelova koliko ima dretvi. Na prvi pogled izgleda logično da bi broj dijelova trebao biti upravo jednak broju dretvi. Ali, time bi se ignorirala priroda problema koji treba rješavati. Naime, u tom slučaju bi dijelovi programa trebali biti savršeno dobro izbalansirani. Autor u [19] navodi primjer (računski intenzivnog) programa za računanje prim (prostih) brojeva od 1 do nekog velikog broja (npr. 10 milijuna). Ako se broj dijelova naivno odredi tako da se raspon brojeva podijeli sa brojem dretvi, zanemaruje se važna činjenica da je lakše naći prim brojeve u donjim dijelovima raspona brojeva, nego u gornjim. Mogle bi se primijeniti različite tehnike, npr. da se zadatak balansira tako da se miješaju dijelovi iz donjih i gornjih raspona brojeva, ali bi sve te tehnike činile rješenje problema kompleksnijim, a nije sigurno da bi stvarno značajno doprinijele povećanju performansi. Zato autor navodi da je u općenitom slučaju bolje primijeniti relativno jednostavnu tehniku: treba osigurati da je broj dijelova programa (particija) dovoljno velik da iskoristi postojeće dretve, tj. da se ne desi da neke dretve ostanu neiskorištene. To možemo osigurati na taj način da broj dijelova bude veći od broja dretvi. Naravno, nije lako odrediti (a ponekad niti moguće) koliki točno treba biti taj broj – najčešće treba eksperimentirati. Uglavnom se pokazuje da se kod početnog povećanja broja dijelova (u odnosu na broj dretvi) dobije značajno povećanje performansi, a sa daljnjim povećanjem broja dijelova povećanje performansi je sve manje (ponekad se performanse mogu i smanjiti).

Slijedi primjer iz [19] u kojem se problem nalaženja prim brojeva (od jedan do nekog velikog broja) rješava na opisani način. Program omogućava da se broj dijelova programa definira kao parametar. Program koristi mogućnosti Jave 5, ne samo generičke klase (genericse), već i konkurentne mogućnosti iz paketa java.util.concurrent.

Ulazni parametri koji se zadaju programu (kroz naredbeni redak za pokretanje programa) su:
- number: gornja granica do koje se traže prim brojevi (npr. 10 milijuna);
- poolsize: broj dretvi (koji se može unaprijed odrediti na temelju prije navedene formule);
- numberOfParts: broj dijelova programa (particija).

```
public class ConcurrentPrimeFinder extends AbstractPrimeFinder {
  private final int poolSize;
  private final int numberOfParts;
  public ConcurrentPrimeFinder(final int thePoolSize,
    final int theNumberOfParts) {
    poolSize = thePoolSize;
    numberOfParts = theNumberOfParts;
  }
  public int countPrimes(final int number) {
    int count = 0;
    try {
      final List<Callable<Integer>> partitions =
        new ArrayList<Callable<Integer>>();
      final int chunksPerPartition = number / numberOfParts;
      for(int i = 0; i < numberOfParts; i++) {
        final int lower = (i * chunksPerPartition) + 1;
        final int upper =
          (i == numberOfParts - 1) ? number
            : lower + chunksPerPartition - 1;
        partitions.add(new Callable<Integer>() {
          public Integer call() {
            return countPrimesInRange(lower, upper);
          }
        });
      }
      final ExecutorService executorPool =
        Executors.newFixedThreadPool(poolSize);
      final List<Future<Integer>> resultFromParts =
        executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);
      executorPool.shutdown();
      for(final Future<Integer> result : resultFromParts)
        count += result.get();
    } catch(Exception ex) { throw new RuntimeException(ex); }
    return count;
  }
  public static void main(final String[] args) {
    if (args.length < 3)
      System.out.println("Usage: number poolsize numberOfParts");
    else
      new ConcurrentPrimeFinder(
          Integer.parseInt(args[1]), Integer.parseInt(args[2]))
        .timeAndCompute(Integer.parseInt(args[0]));
  }
}
```

U metodi countPrimes, unutar try bloka, raspon brojeva unutar kojeg se traže prim brojevi (od 1 do željenog maksimalnog broja) dijeli se na određeni broj particija. Svaka particija onda predstavlja zadatak za zasebnu instancu Callable servisa. Prethodna klasa ConcurrentPrimeFinder nasljeđuje (apstraktnu) klasu AbstractPrimeFinder (koja je prikazana u nastavku) i koristi njene neapstraktne metode:

```
public abstract class AbstractPrimeFinder {
  public boolean isPrime(final int number) {
    if (number <= 1) return false;
    for(int i = 2; i <= Math.sqrt(number); i++)
      if (number % i == 0) return false;
    return true;
  }
  public int countPrimesInRange(final int lower, final int upper) {
    int total = 0;
    for(int i = lower; i <= upper; i++)
      if (isPrime(i)) total++;
    return total;
  }
```

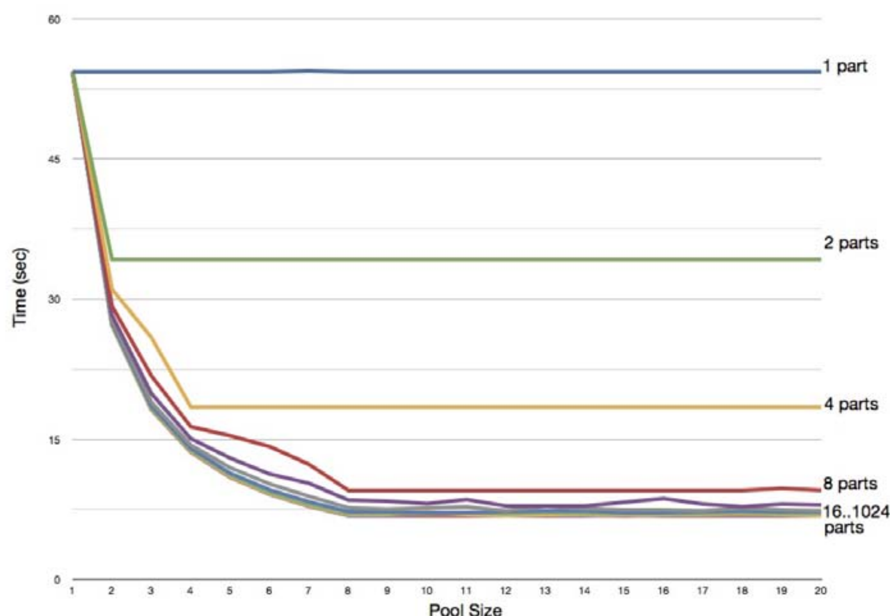
```
  public void timeAndCompute(final int number) {
    final long start = System.nanoTime();

    final long numberOfPrimes = countPrimes(number);

    final long end = System.nanoTime();

    System.out.printf("Number of primes under %d is %d\n",
      number, numberOfPrimes);
    System.out.println("Time (seconds) taken is " + (end - start)/1.0e9);
  }
  public abstract int countPrimes(final int number);
}
```

Sljedeća slika (4.2) pokazuje kako se mijenja vrijeme izvršavanja programa u zavisnosti od zadanog broja (softverskih) dretvi (pool size) i broja particija programa (parts). Autor ([19]) je stavljao gornju granicu za traženje prim brojeva na 10 milijuna, startao je Java program u klijentskom modu, a raspolagao je sa računalom koje ima 16 hardverskih dretvi (8 jezgri x 2 hardverske dretve po jezgri). Zato se vidi da se vrijeme značajno poboljšavalo povećavanjem broja dretvi na 8, a nešto manje na 16. Može se primijetiti da je u ovom slučaju optimalan broj particija bio vrlo blizu broju dretvi, tj. da nije bilo potrebno stavljati broj particija značajno veći od broja dretvi:



Slika 4.2. Pronalaženje prim brojeva na 8 jezgrenom procesoru (16 hardverskih dretvi) - analiza efekta mijenjanja broja dretvi (pool size) i broja dijelova programa (parts); Izvor: [19]

U primjeru se vidi da autor dosta koristi varijable označene sa final. Razlog je taj što se povećanje konkurentnosti programa u načelu može povećati smanjivanjem tzv. djeljive mutabilnosti (shared mutability), kod koje različite (softverske) dretve mijenjaju istu varijablu, što znači da se to mijenjanje mora raditi na sinkronizirani način. Druga krajnost djeljivoj mutabilnosti je čista imutabilnost (pure immutability), kod koje se eliminira promjenjivost varijable. Možda izgleda malo nelogično govoriti o nepromjenjivoj varijabli – zar nije to, zapravo, konstanta? No, postoji značajna razlika između konstante i nepromjenjive varijable. Konstanta je unaprijed definirana, a nepromjenjiva varijabla se jednom postavi na željenu vrijednost i više se ne mijenja. U Javi se taj efekt postiže upravo sa final varijablama (u Scali postoji čišća notacija od Java final notacije: imutabilne varijable označavaju se sa val, mutabilne sa var).

Iako treba težiti ka čistoj imutabilnosti, nju nije uvijek moguće postići, ili je to teško. No, autor u [19] navodi kako se i sam iznenadio koliko toga je moguće postići čistom imutabilnošću. Problem je što je većina nas naučena na mutabilan način programiranja, za razliku od onih koji koriste funkcijske jezike.

Srednja struja je tzv. izolirana mutabilnost (isolated mutability), kod koje je samo jednoj softverskoj dretvi dano pravo mijenjanja varijable, a sve ostale dretve tu varijablu mogu čitati, ali ne i mijenjati. Često se takav stil konkurentnog programiranje zove programiranje pomoću aktora (actor), što je dosta korišteni stil kod jezika Scala (a takav stil je u široku praktičnu upotrebu uveden sa jezikom Erlang).

5. SCALA KAO OBJEKTNI PROGRAMSKI JEZIK

5.1 Nastanak jezika Scala

Programski jezik Scala kreirao je Martin Odersky, profesor na Ecole Polytechnique Fédérale de Lausanne (EPFL). Kako je naveo u seriji od četiri intervjua [14], profesor Odersky je još kao student početkom 80-ih, zajedno sa jednim kolegom, napisao jedan od prvih kompajlera za jezik Modula-2. Kompajler je kupila firma Borland, koja ih je pozvala da rade kod njih. Odersky se odlučio za akademsku karijeru i krajem 80-ih doktorirao na ETH Zürich kod profesora Niklausa Wirtha (kreatora Pascala i Module-2). Nakon toga naročito se bavio istraživanjima u području funkcijskih jezika, zajedno sa kolegom Philom Wadlerom (jednim od dva glavna kreatora funkcijskog jezika Haskell). Kada je izašla Java, Odersky i Wadler su 1996. napravili jezik Pizza nad JVM-om, koji je uključivao tri mogućnosti iz funkcijskog programiranja: generičke klase (generics), funkcije višeg reda (higher-order functions) i podudaranje (sparivanje) uzorka (pattern matching). To je bila jedna od prvih implementacija ne-Java jezika nad JVM-om. Na temelju projekta Pizza, napravili su 1997./98. Generic Java (GJ), koji je uveden u Javu 5 (malo ga je nadopunio Gilad Bracha, sa wildcardsima). Dok je za primjenu GJ-a Sun čekao skoro 6 godina, odmah su preuzeli Java kompajler koji je Odersky napravio za GJ. Taj se kompajler koristi od Jave 1.3 dalje.

Nakon rada sa GJ-om, Odersky je postao profesor na EPFL i mogao se posvetiti razvoju novog jezika, koji bi bio vezan za JVM i Java biblioteke, ali koji bi bio bolji od Jave. Prvo je kreirao jezik Funnel. Zaključio je da je taj jezik previše akademski, pa je 2002. počeo raditi novi jezik Scala. Scala je tako nazvana kako bi se naglasila njena skalabilnost. Prva javna verzija izašla je 2003., a relativno značajni redizajn napravljen je 2006 (trenutačna verzija je 2.9, a u testiranju je 2.10). Od tada, Scala se sve više koristi u praksi, tako da je došla među prvih 50 najkorištenijih jezika, sa tendencijom da se probije među prvih 20. Zanimanje za Scalu naročito se povećalo kada je Twitter prebacio glavne dijelove svojih programa (koji rade procesiranje poruka) iz jezika Ruby u Scalu.

Odersky je naglasio da nije želio da Scala bude 100% kompatibilna sa Javom pod svaku cijenu. Npr. odustao je od toga da Scala nizovi (arrays) budu kovarijantni (covariant) kao u Javi. Inače, Odersky naglašava da je odluka Java kreatora da (u nedostatku generičkih klasa, na početku razvoja Jave) nizovi budu kovarijantni, bila jedna od najvećih grešaka kod razvoja Jave. Kreatori Jave su brzo toga postali svjesni, ali bilo je već kasno. Scala je kreirana kao čisti objektno-orijentirani jezik (kao i Eiffel, dok Java nije čisti OOPL), i u nju su uvedene neke objektno-orijentirane mogućnosti koje Java nema. Osim toga, na temelju objektno-orijentiranih mogućnosti izgrađene su i brojne funkcijske mogućnosti, tako da je Scala i funkcijski jezik (ali nije čisti). Objektno-orijentirani pristup dobar je kada se skup klasa prirodno proširuje. Funkcijski pristup pogodniji je kada je skup struktura relativno fiksan, ali se žele uvesti nove operacije nad postojećim strukturama, gdje je najbolji pristup podudaranje uzorka (pattern matching). Sa funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za konkurentno programiranje.

Odersky je u intervjuima spominjao i neke relativno manje važne odluke, vezane za sintaksu. Npr., odlučio je zadržati C / C++ / Java / C# vitičaste zagrade umjesto npr. riječi BEGIN / END ili sličnih, jer mu se činilo da su vitičaste zagrade sasvim dobre. No, želio je da se pridruživanje varijabli označava kao u jezicima Pascal / Modula-2 / Ada ... , tj. pomoću := (a da se znak = ostavi za označavanje jednakosti, kao u matematici), a ne kao u jezicima Fortran (on je "krivac" za takav stil) / C / C++ / Java / C# ... (gdje se onda za označavanje jednakosti koristi ==). Ipak je odustao, nakon žestokih negativnih reakcija programera naviklih na Java stil. Za razliku od toga, nije odustao od sintakse da se ime varijable piše prije tipa varijable (kao npr. u Pascalu: brojcanaVarijabla: int), a ne obrnuto (kao npr. u Javi: int brojcanaVarijabla), između ostalog i zato što bi inače bilo vrlo otežano parsiranje Scala koda.

Scala je izvrstan jezik i za pisanje DSL-ova (Domain-Specific Language), programskih jezika za specifičnu problemsku domenu. Profesor Odersky je krajem 2010. od European Research Councila (ERC) dobio Advanced Investigator Grant (oko 2,5 milijuna eura za projekt u trajanju od 5 godina, 2011.-2016.) za svoj projekt vezan za DSL-ove za paralelno programiranje pisane u Scali. Istu potporu dobio je krajem 2011. (za razdoblje 2012.-2017.) profesor sa ETH Zürich, Bertrand Meyer, kreator jezika Eiffel, za svoj projekt "Concurrency Made Easy", koji se temelji na SCOOP metodi (SCOOP metoda detaljno je opisana u [11]). Zanimljivo je da su i Eiffel (kreiran 1986., skoro deset godina prije Jave) i Scala vrlo rigorozno matematički specificirani, za razliku od Jave. Eiffel je i ISO standardiziran, što Java i Scala za sada nisu. Scala podržava statičku provjeru tipova, kao i C++, Eiffel, Haskell, Java, C# ... , za razliku od npr. jezika Python, Ruby, Groovy, Clojure ... , kod kojih se greška u tipovima pokaže tek kod izvođenja programa.

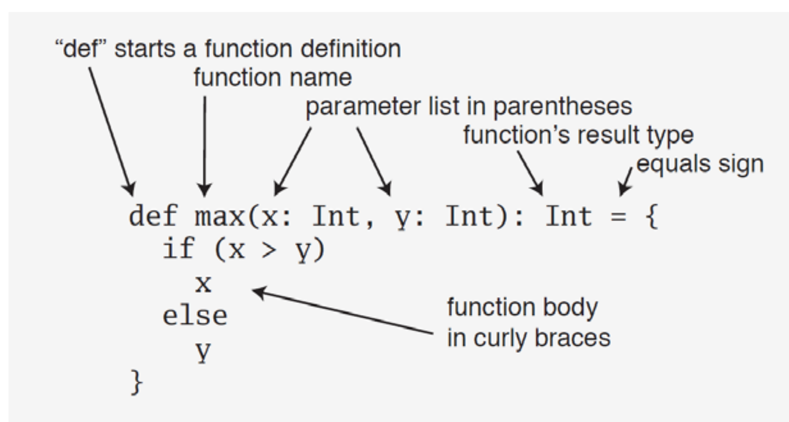
Značajno je da i kreatori drugih ("konkurentnih") jezika imaju vrlo visoko mišljenje o Scali:
"If I were to pick a language to use today other than Java, it would be Scala."
- James Gosling, creator of Java

"Scala, it must be stated, is the current heir apparent to the Java throne. No other language on the JVM seems as capable of being a "replacement for Java" as Scala, and the momentum behind Scala is now unquestionable. While Scala is not a dynamic language, it has many of the characteristics of popular dynamic languages, through its rich and flexible type system, its sparse and clean syntax, and its marriage of functional and object paradigms."
- Charles Nutter, creator of JRuby

"I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy."
- James Strachan, creator of Groovy.

5.2 Neke osobine

U Scali se definicija funkcije (ili procedure) radi na malo drugačiji način nego u Javi. Ne samo da se koristi ključna riječ def, već se (kao i kod varijabli) prvo navodi ime funkcije, a tip funkcije slijedi na kraju, iza parametara funkcije. Tip funkcije nije nužno uvijek navesti, jer ga Scala kompajler može (kao i tipove drugih elemenata programskog koda) izvesti iz drugih podataka. To je tzv. izvođenje tipova ili zaključivanje o tipovima (type inference), što statički jezik Scala čini fleksibilnim kao da je jezik dinamički. Iza (neobaveznog) tipa funkcije, slijedi znak jednakosti (ako je riječ o funkciji). U funkciji se ne mora pisati riječ return, jer se podrazumijeva da zadnja izvršena operacija vraća vrijednost funkcije. Sljedeća slika 5.1 prikazuje osnovnu formu Scala funkcije:



Slika 5.1. Osnovna forma definicije funkcije u Scali; Izvor: [15]

Funkcije se u Scali mogu gnijezditi, tj. mogu se raditi lokalne funkcije (kao što ima i PL/SQL). Slijedi primjer iz [15], koji za zadanu datoteku ispisuje retke veće od zadane širine:

```
def processFile(filename: String, width: Int) {
  def processLine(filename: String, width: Int, line: String) {
    if (line.length > width)
      println(filename +": "+ line)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines()) {
    processLine(filename, width, line)
  }
}
```

Primjećuje se da se kod definicije varijable source koristi ključna riječ val (kao value), za razliku od ključne riječi var (kao variable). Obje ključne riječi označavaju varijablu, ali varijabla označena sa val je imutabilna, tj. nakon jednokratnog pridruživanja vrijednosti više se ne može mijenjati - kao Java final. Također, primjećuje se da u Scali nije nužna upotreba separatora točka-zarez (osim u nekim vrlo specijalnim slučajevima).

Funkcije u Scali, uz uobičajene pozicijske parametre, mogu imati i imenovane parametre (named parameters). Na taj način omogućeno je da se kod poziva funkcije parametri (netko kaže da su kod poziva to argumenti, a ne parametri), navedu drugačijim redom u odnosu na redoslijed parametara u definiciji funkcije (to ima i PL/SQL). Također, Scala ima default parametre (kao i PL/SQL), koji su posebno korisni u kombinaciji sa imenovanim parametrima. Npr., kod poziva neke funkcije koja ima 5 parametara, ali su prvi i četvrti default, mogli bismo u pozivu navesti eksplicitno svih 5 parametara, ili samo drugi, treći i peti parametar (koji nemaju default vrijednosti) kao imenovane parametre, ali možemo navesti i default parametre, ako im želimo staviti ne-default vrijednost. U Scali poziv metode može imati varijabilan broj argumenata (varargs), ali to vrijedi za zadnji parametar metode (kao i od Jave 5).

Kao i Java, tako i Scala ima try – catch - finally blok za obradu iznimki. Za razliku od Jave, Scala nema tzv. kontrolirane iznimke (checked exceptions), tj. ne traži da se iznimka ili obradi, ili deklarira u throws klauzuli. Također, Scala try – catch – finally blok uvijek vraća vrijednost.

Kao i Java (i većina drugih jezika), Scala dopušta rekurziju. No, u ne-funkcijskim jezicima često se izbjegava rekurzija, zato što ona (uobičajeno) kod svakog poziva funkcije povećava stog (stack), čime se troši memorija, usporava rad, a program može i puknuti ako se stog prepuni. Zato se obično sljedeći rekurzivni kod iz [15]:

```
def approximate(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else approximate(improve(guess))
```

zamjenjuje sa nerekurzivnom varijantom, koja koristi petlju (u ovom primjeru while):

```
def approximateLoop(initialGuess: Double): Double = {
  var guess = initialGuess
  while (!isGoodEnough(guess))
    guess = improve(guess)
  guess
}
```

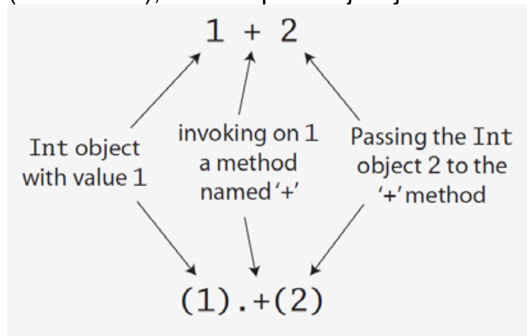
Funkcijski jezici obilato koriste rekurziju, jer oni imaju optimizaciju kojom mogu izbjeći povećavanje stoga kod tzv. repne rekurzije (tail recursion), tj. onda kada je zadnja operacija u kodu ("na repu" koda) poziv funkcije same. To može i Scala (uz neka ograničenja), tako da su u Scali dvije prethodne varijante praktički istovjetne. No, ako funkcija nije repno rekurzivna (tail-recursive), onda nema optimizacije. Npr. sljedeća funkcija nije repno rekurzivna, jer na kraju iza rekurzivnog poziva postoji još i dodatno zbrajanje (označeno crvenom bojom):

```
def boom(x: Int): Int =
  if (x == 0) throw new Exception("boom!")
  else boom(x - 1) + 1
```

Korištenje optimizacije kod repne rekurzije u Scali je ipak ograničeno, jer JVM skup instrukcija vrlo otežava implementaciju naprednih formi optimizacije rekurzivnih poziva. Npr., u slučaju dvije međusobno pozivajuće funkcije, gdje je repna rekurzija indirektna, Scala optimizator ne prepoznaje repnu rekurziju.

5.3 Klase

Scala je čisti objektno-orijentirani jezik (ali nije čisti funkcijski jezik). Svaka vrijednost je objekt, a svaka operacija je poziv metode (neke klase), kao što prikazuje sljedeća slika 5.2:



Slika 5.2. U Scali su sve operacije metode (neke klase); Izvor: [15]

Dakle, obično zbrajanje brojeva 1 + 2 predstavlja poziv metode + nad objektom (1), a metodi se šalje parametar (2). Treba naglasiti da se pritom ništa ne gubi na performansama, jer Scala optimizator napravi kod koji je isto tako optimalan kao i Java kod koji koristi primitivne tipove (npr. int) koji nisu klase. Ponekad se može i obrnuto, tj. metoda se može pisati u "operatorskom" obliku. Npr., uz uobičajeni stil:

nekiObjekt.nekaMetoda(nekiParametar)

u Scali se poziv metode može pisati bez točke, čak i bez zagrada (ako metoda ima 0 ili 1 parametar):

nekiObjekt nekaMetoda nekiParametar

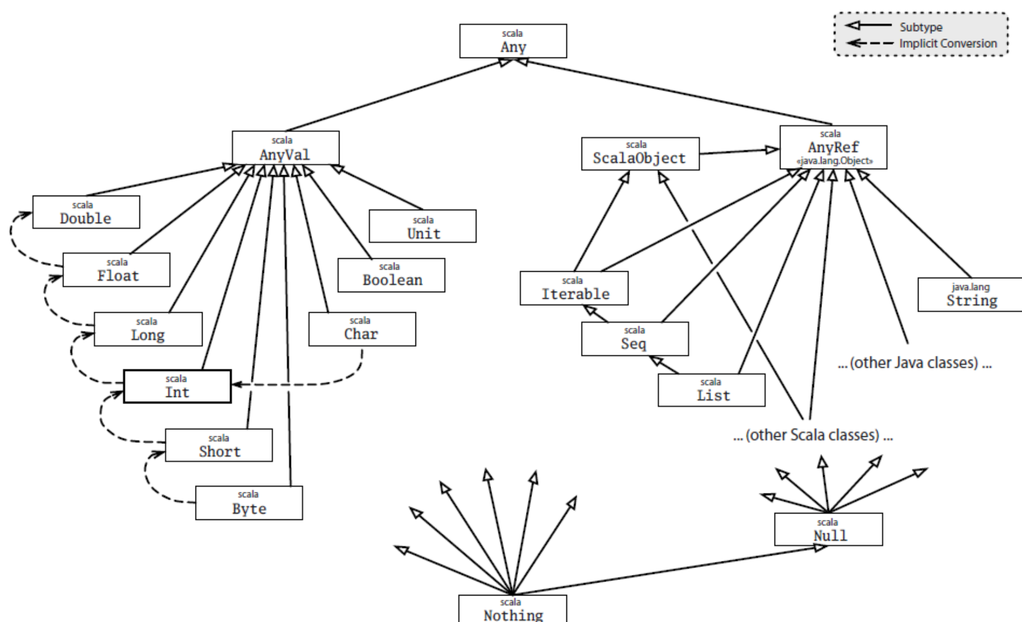
Gledano tehnički, Scala nema operatore, već metode imenovane specijalnim simbolima (kao simbol +), pa se mogućnost opterećenja operatora (operator overloading), koju Java nema, a npr. C++, Eiffel, C# imaju, rješava kao uobičajeno preopterećenje funkcija, kao u ovom primjeru iz [18], gdje je uvedena metoda + koja "glumi" operator za zbrajanje kompleksnih brojeva:

```
class Complex(val real: Int, val imaginary: Int) {
  def +(operand: Complex): Complex = {
    new Complex(real + operand.real, imaginary + operand.imaginary)
  }
  override def toString(): String = {
    real + (if (imaginary < 0) "" else "+") + imaginary + "i"
  }
}

val c1 = new Complex(1, 2)
val c2 = new Complex(2, -3)
val sum = c1 + c2
println("(" + c1 + ") + (" + c2 + ") = " + sum)
```

Prethodni primjer prikazuje i to da se u Scali koristi ključna riječ override za označavanje nadjačavanja (naslijeđene) metode toString specifičnom varijantom klase Complex. Takvo rješenje ima i Eiffel, dok se u Javi može, ali ne mora, navesti odgovarajuća anotacija (uvedeno u Javi 5). Obavezno eksplicitno označavanje je sigurnije, jer se u Javi greškom može desiti da smo zapravo uveli novu metodu, sa drugačijom signaturom od one u nadklasi, umjesto da nadjačamo metodu nadklase.

Slika 5.3 prikazuje hijerarhiju klasa u Scali. Vidi se da sve klase nasljeđuju klasu Any, a nju nasljeđuju klasa AnyVal (koja je nadklasa za "primitivne" tipove) i klasa AnyRef, koja je zapravo Java klasa Object. Klasa Null je podklasa svih AnyRef podklasa, a klasa Nothing je podklasa svih klasa:



Slika 5.3. Hijerarhija Scala klasa; Izvor: [15]

Sljedeći primjer iz [18] prikazuje nasljeđivanje klasa. Klasa Car nasljeđuje klasu Vehicle i dodaje novi atribut fuelLevel, te nadjačava metodu toString:

```
class Vehicle(val id: Int, val year: Int) {
  override def toString(): String = "ID: " + id + " Year: " + year
}

class Car(override val id: Int, override val year: Int,
  var fuelLevel: Int) extends Vehicle(id, year) {
  override def toString(): String =
    super.toString() + " Fuel Level: " + fuelLevel
}

val car = new Car(1, 2009, 100)
println(car)
```

Scala nema statička polja i statičke metode (kao ni Eiffel), jer to nije u skladu sa objektno-orijentiranim pristupom. No, zato Scala ima singleton klase (klase koje imaju samo jednu instancu), koje se označavaju pomoću ključne riječi object, umjesto riječi class. Sljedeći primjer iz [18] prikazuje klasu Marker, čiji se objekti kreiraju pomoću singleton klase MarkerFactory:

```
class Marker(val color: String) {
  println("Creating " + this)
  override def toString(): String = "marker color " + color
}

object MarkerFactory {
  private val markers = Map(
    "red" -> new Marker("red"),
    "blue" -> new Marker("blue"),
    "green" -> new Marker("green")
  )
  def getMarker(color: String) =
    if (markers.contains(color)) markers(color) else null
}

println(MarkerFactory getMarker "blue") //...
```

U prethodnom primjeru objekt MarkerFactory je samostalan (stand-alone) objekt. Međutim, Scala omogućava i da se napravi objekt koji ima isto ime kao klasa, tzv. drugarski objekt (companion object). Time se omogućava da klasa i njen drugarski objekt mogu međusobno pristupati tuđim privatnim poljima i metodama, čime se postižu iste mogućnosti kao kod statičkih polja i metoda, ali na "čišći" način. Sljedeći primjer iz [18] prikazuje klasu Marker i njen drugarski objekt:

```
class Marker private (val color: String) {
  override def toString(): String = "marker color " + color
}

object Marker {
  private val markers = Map(
    "red" -> new Marker("red"),
    "blue" -> new Marker("blue"),
    "green" -> new Marker("green")
  )
  def primaryColors = "red, green, blue"
  def apply(color: String) =
    if (markers.contains(color)) markers(color) else null
}

println("Primary colors are : " + Marker.primaryColors)
println(Marker("blue"))
println(Marker("red")) //...
```

Scala ima n-torke (tuples) i višestruko pridruživanje (multiple assignments). Primjer iz [18]:

```
def getPersonInfo(primaryKey: Int) = {
  // Assume primaryKey is used to fetch a person's info...
  // Here response is hard-coded
  ("Venkat", "Subramaniam", "venkats@agiledeveloper.com")
}
val (firstName, lastName, emailAddress) = getPersonInfo(1)
println("First Name is " + firstName) // ...
```

Scala može imati pakete unutar paketa. Default u Scali je public vidljivost, ali Scala omogućava puno finiju granularnost vidljivosti (u paketu i klasi) u odnosu na Javu. Postoji i mogućnost da se vidljivost privatnih polja / metoda ograniči samo na određenu instancu klase (kao što imaju i Eiffel i Ruby).

5.4 Scala trait

Scala nema pravo višestruko nasljeđivanje (multiple inheritance) kao što imaju C++ i Eiffel. Često se kaže da je višestruko nasljeđivanje komplicirano, vjerojatno na temelju iskustva sa C++. Bertrand Meyer je uveo višestruko nasljeđivanje u Eiffel od početka (1986), dok je u C++ ono uvedeno naknadno (1989.), pa je vjerojatno zato izvedeno neoptimalno. Javlja se tzv. dijamantni problem (diamond problem), kad neka klasa nasljeđuje (barem) dvije klase koje pak nasljeđuju istu klasu-pretka (to liči na skicu dijamanta, otud ime problema), pa se ne može izbjeći dvostruko nasljeđivanje istoimenih metoda i polja. Eiffel je elegantno riješio taj problem, kao i općenitiji problem istih imena metoda i polja u klasama od kojih se nasljeđuje.

Kod dizajniranja Jave, odabralo se samo jednostruko nasljeđivanje klasa, a višestruko nasljeđivanje imaju samo Java sučelja (interface), koji su zapravo potpuno apstraktne klase (bez konkretnih metoda, tj. bez programskog koda). Budući da se ipak tokom vremena shvatilo da je višestruko nasljeđivanje korisno, u Scali je uveden programski konstrukt koji omogućava skoro isti efekt kao višestruko nasljeđivanje - trait. Može izgledati da je trait nešto kao Java sučelje sa konkretnim metodama. No, trait može imati skoro sve što ima i klasa, npr. može imati i polja, a ne samo metode. Trait se, zapravo, kompajlira u Java sučelje i pripadajuće pomoćne klase koje sadrže implementaciju metoda i atributa.

U Scali, klasa može naslijediti samo jednu (direktnu) nadklasu, ali zato može naslijediti više traitova. Može se reći da klasa koja nasljeđuje drugu klasu i (barem jedan) trait, predstavlja "miksanu" klasu (dok pojam mixin označava trait ili klasu od koje miksana klasa nasljeđuje metode, a ne strukturu). Sljedeći primjer iz [15] prikazuje klasu Frog, koja nasljeđuje (označava se sa extends) klasu Animal (u ovom slučaju je to apstraktna klasa) i nadopunjava se sa dva traita (označava se sa with; kada bi se nasljeđivao samo jedan trait, i nijedna klasa, mogla bi se koristiti riječ extends, umjesto with). Vidi se i da klasa može nadjačati (override) metodu koju je naslijedila iz traita, kao što može onu koju je naslijedila iz klase:

```
class Animal

trait Philosophical {
  def philosophize() { println("I consume memory, therefore I am!") }
}

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
  override def philosophize() {
    println("It ain't easy being "+ toString +"!")
  }
}
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green
scala> phrog.philosophize()
It ain't easy being green!
```

Jedna od dvije najvažnije upotrebe traita je automatsko dodavanje metoda klasi. Kaže se da traitovi obogaćuju tanko sučelje (thin interface) klase (ne misli se na Java sučelje), pretvarajući ga u bogato sučelje (rich interface).

Postoje dvije važne razlike između klase i traita. Prvo, trait ne može imati parametre klase (class parameters – ne misli se na generičke parametre), tj. parametre koji se šalju primarnom konstruktoru:

```
class Point(x: Int, y: Int)
trait NoPoint(x: Int, y: Int) // Does not compile
```

Druga razlika je da su kod klasa pozivi metode super statički vezani (statically bound), a kod traita su dinamički vezani, što omogućava da se traitovi koriste za nadograđujuću modifikaciju (stackable modifications) klasa, tj. da se metode klase dinamički nadograđuju metodama iz traita. U sljedećem primjeru iz [15] kreira se apstraktna klasa IntQueue, koju nasljeđuju jedna konkretna klasa BasicIntQueue i dva traita. Trait Incrementing povećava element reda za jedan, a trait Filtering uzima samo one elemente koji su veći ili jednaki nuli nuli:

```
import scala.collection.mutable.ArrayBuffer
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) { super.put(x + 1) }
}
trait Filtering extends IntQueue {
  abstract override def put(x: Int) { if (x >= 0) super.put(x) }
}

scala> val queue = (new BasicIntQueue with Incrementing with Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0); queue.put(1)
scala> queue.get()
res15: Int = 1

scala> queue.get()
res16: Int = 2
```

Budući da je prvi element (-1) manji od nule, filtriranje ga je odbacilo, ali je zadržalo elemente 0 i 1, koji su nakon inkrementiranja dali vrijednost 1 i 2. Te su vrijednosti smještene u red i onda preuzete iz reda. Dakle, redoslijed miksanja je važan. Precizna pravila slijede, ali ukratko se može reći da se prvo pozivaju oni traitovi koji su desnije. Ako oni koriste metodu super, onda pozivaju metodu koja se nalazi u traitu slijeva itd. U konkretnom slučaju prvo se pozvala metoda iz (desnog) traita, tj. napravilo se filtriranje, a onda je ona pozvala metodu iz lijevog traita, tj. napravilo se inkrementiranje elementa.

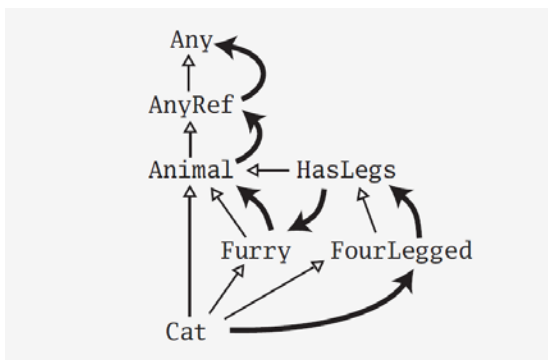
Trait se može koristiti i selektivno na razini objekta, bez da se veže za (cijelu) klasu. Npr., pretpostavimo da klasa Macka ne nasljeđuje trait Programer. Instanca ipak može naslijediti taj trait:

```
val jakoPametnaMacka = new Macka("Mica maca") with Programer
```

Kako kažu autori u [15], iako traitovi sliče na višestruko nasljeđivanje u nekim drugim jezicima, razlikuju se u barem jednoj važnoj stvari – interpretaciji metode super. Kod višestrukog nasljeđivanja metoda koja se poziva sa super može se odrediti tamo gdje se poziv nalazi. Kod traitova se to, međutim, određuje metodom koja se zove linearizacija (linearization) klase i traitova koji su miksani s tom klasom. Linearizaciju slikovito prikazuje ovaj primjer iz [15]:

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Slika 5.4 prikazuje na tom primjeru hijerarhiju klasa i traitova, te linearizaciju. Nasljeđivanje se prikazuje pomoću tradicionalne UML notacije: strelice sa bijelim, trokutastim vrhovima označavaju nasljeđivanje (strelice pokazuju prema nadtipu). Strelice sa crnim vrhovima označavaju linearizaciju. Te strelice kreću se u smjeru u kojem se dešavaju pozivi super metoda:



Slika 5.4. Hijerarhija nasljeđivanja i linearizacije klase Cat; Izvor: [15]

Linearizacija svake klase ili traita zasebno, prikazana je na sljedećoj slici 5.5:

Type	Linearization
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any
FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

Slika 5.5. Linearizacija tipova u hijerarhiji klase Cat; Izvor: [15]

Autori u [15] kažu – kad god implementiramo višestruko iskoristivu kolekciju ponašanja (a reusable collection of behavior), moramo odlučiti da li ćemo koristiti trait ili apstraktnu klasu. Iako nema čvrstih pravila, daju ove smjernice:

- ako se ponašanje neće ponovno koristiti, napravimo konkretnu klase;
- ako će se ponašanje koristiti u više nepovezanih klasa, napravimo trait, jer se samo on može miksati u različite dijelove hijerarhije klasa;
- ako želimo nasljeđivati iz Java koda, napravimo apstraktnu klasu; naime, budući da u Javi ne postoji nešto analogno traitovima, vrlo je teško u Javi naslijediti trait; nasuprot tome, naslijediti u Javi Scala klasu jednostavno je, kao da je to Java klasa.

5.5 Generičke klase i tipovi

Prije govora o generičkim klasama i tipovima, podsjetimo se da u Scali ponekad možemo izostaviti eksplicitno navođenje tipa (npr. kod varijable ili povratne vrijednosti funkcije), jer Scala kompajler često može iz programskog konteksta zaključiti kojeg tipa bi trebao biti neki element. Kako je navedeno u 5.2, to je tzv. zaključivanje o tipovima (type inference). Slijedi primjer iz [18]:

```
var year: Int = 2009 // eksplicitno naveden tip Int
var anotherYear = 2009 // Scala zaključuje da je tip Int
var greet = "Hello there" // ... String
var builder = new StringBuilder("hello") // ... StringBuilder
println(builder.getClass())
```

Generičke klase imaju parametre koji predstavljaju tipove, tzv. parametre-tipove (type parameters; u [11] i [12] ih se naziva generičkim parametrima). Kad ne bi postojale generičke klase, mogli bismo reći da su tip i klasa jedno te isto (ako zanemarimo Java primitivne tipove, koji nisu klase). No, generička klasa nije "gotov" tip dok se u njenim parametrima-tipovima generički tipovi ne zamijene konkretnim tipovima. Generičke klase mogu se nasljeđivati.

U sljedećem primjeru iz [16] prikazuje se (apstraktna) generička klasa Stack, koju nasljeđuje (konkretna) generička klasa EmptyStack, pri čemu obje imaju isti generički tip A. Klasa EmptyStack koristi se tako da se generički tip A zamijeni sa konkretnim tipom Int:

```
abstract class Stack[A] {
  def push(x: A): Stack[A] = new NonEmptyStack[A](x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
class EmptyStack[A] extends Stack[A] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[A] extends Stack[A] {...}

val x = new EmptyStack[Int]
val y = x.push(1).push(2)
println(y.pop.top)
```

Osim klasa, i metode mogu imati parametre-tipove, kao u sljedećem primjeru iz [16], gdje se funkcija koristi tako da se generički tip A zamijeni sa konkretnim tipom String:

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}

val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

U sljedećim primjerima iz [18] vidjet ćemo kako se generički tipovi u metodama ograničavaju sa gornjom ili donjom granicom (upper / lower bound). Pretpostavimo da imamo klasu Dog koja nasljeđuje klasu Pet i metodu workWithPets( ) koja prihvaća niz objekata tipa Pet:

```
class Pet(val name: String) {
  override def toString() = name
}

class Dog(override val name: String) extends Pet(name)

def workWithPets(pets: Array[Pet]) {}
```

Ako sada kreiramo niz objekata klase Dog i pošaljemo ga prethodnoj metodi, dobit ćemo grešku:

```
val dogs = Array(new Dog("Rover"), new Dog("Comet"))
workWithPets(dogs) // Compilation ERROR
```

U Scali ne možemo poslati niz objekata tipa Dog metodi koja prihvaća niz objekata tipa Pets (Java bi to kod kompajliranja dozvolila, ali bi puklo kod izvođenja). Ono što trebamo napraviti je da definiramo generički tip (npr. T) i ograničimo ga gornjom granicom. U ovom slučaju gornja granica je tip Pet, pa se ograničavanje označava sa [T <: Pet] (uzgred, u Javi bi se to označilo sa <T extends Pet>, a u Eiffelu sa [T -> Pets]), što znači da konkretan tip mora biti (nepravi) podtip od tipa Pet:

```
def playWithPets[T <: Pet](pets: Array[T]) =
  println("Playing with pets: " + pets.mkString(", " ))

val dogs = Array(new Dog("Rover"), new Dog("Comet"))
playWithPets(dogs) // rezultat je: Playing with pets: Rover, Comet
```

Ako bismo prethodnoj metodi pokušali poslati niz objekata koji ne pripadaju podtipu od Pet, opet bismo dobili grešku kod kompajliranja. U sljedećem primjeru vidjet ćemo ograničavanje generičkog tipa na donju granicu. U metodi copyPets(), koja kopira izvorišni niz Array[S] u destinacijski niz Array[D], destinacijski generički tip D je morao biti označen kao nadtip izvorišnog generičkog tipa S, sa D >: S:

```
def copyPets[S, D >: S](fromPets: Array[S], toPets: Array[D]) = { //...}
val pets = new Array[Pet](10)
copyPets(dogs, pets)
```

U prethodna dva primjera ograničavali smo (pomoću gornje ili donje granice) generički tip u definiciji metode koja koristi kolekciju. Ako bismo bili autori kolekcije, slično ograničavanje bismo mogli raditi nad generičkim tipom u definiciji kolekcije (tj. klase). No, mogli bismo koristiti i tzv. anotaciju varijance (variance annotation). To se radi tako da se u definiciji klase generički tip označi sa prefiksima + ili -, npr. +T ili –T umjesto T, kao u sljedećem primjeru:

```
class MyList[+T] //...
var list1 = new MyList[Int]
var list2: MyList[Any] = null
list2 = list1 // OK
```

Stavljajući prefiks + generičkom tipu T, označili smo da taj generički tip ima kovarijantno (covariant) ponašanje, odnosno rekli smo Scali da je npr. MyList[Int] podtip od MyList[Any], u skladu s tim što je Int podtip od Any (zato se naziva kovarijantno ponašanje). Zato smo mogli pridružiti varijablu tipa MyList[Int] varijabli tipa MyList[Any].

Suprotno od kovarijantnog ponašanja je kontravarijantno (contravariant) ponašanje, kada se generički parametar T označi sa –T, kao u sljedećem primjeru, gdje je MyList[Any] podtip od MyList[Int], što je suprotno ("kontra") tome što je Int podtip od Any (zato se naziva kontravarijantno ponašanje):

```
class MyQueue[-T] //...
var list1 = new MyQueue[Any]
var list2: MyQueue[Int] = null
list2 = list1 // OK
```

Podrazumijevano (default) ponašanje u Scali je invarijantno (nonvariant) ponašanje. Vidjeli smo prije da su nizovi u Scali invarijantni, pa smo kod njih morali ograničavati generičke tipove sa gornjom ili donjom granicom, da bismo postigli željeno ponašanje.

Primijetimo još nešto vezano za nasljeđivanje. Za razliku od Eiffela, a slično kao Java, Scala nema mogućnost da tip parametra u metodi podklase (koja je nadjačala metodu iz nadklase) bude kovarijantan u odnosu na tip parametra u nadjačanoj metodi nadklase. No, zahvaljujući riječi override, Scala kompajler takav pokušaj prepoznaje kao grešku, dok bismo kod Jave imali (neželjeno) preopterećenje (overloading) metode. U primjeru, parametar metode eat() ne može u podklasi promijeniti tip iz Food u childFood:

```
class Food(val name: String) {
  override def toString() = "Food: " + name
}
class childFood(override val name: String, var childMinAge: Int)
  extends Food(name) {
  override def toString() = "child " + name
}
class Parent(val name: String) {
  def eat (pFood: Food) {
    println ("In parent procedure: " + name + " eats " + pFood)
  }
}
class Child(override val name: String, var childMinAge: Int)
  extends Parent(name) {
  override def eat (pChildFood: childFood) {
    println ("In child procedure: " + name + " eats " + pChildFood)
  }
}
<console>:11: error: method eat overrides nothing
         override def eat (pChildFood: childFood) {
```

6. SCALA KAO FUNKCIJSKI PROGRAMSKI JEZIK

Kako je već navedeno u uvodu i u 2. točki, prvi funkcijski jezik Lisp (skraćenica od LISt Processing), nastao je još davne 1958. godine. Funkcijski jezici se od tada neprekidno koriste. Ipak, većina programera koristi se imperativnim programskim jezicima (objektnim ili neobjektnim). No, u zadnjih nekoliko godina se povećao interes za funkcijske jezike. Funkcijski jezici obećavaju bolju modularnost, a modularni programi sastoje se od komponenti koje se mogu razumjeti i koristiti nezavisno od cjeline, pa se lakše spajaju u cjelinu.

Većina nas nije vična funkcijskom programiranju (uključujući autora ovog rada, koji ima neka iskustva sa Prologom, ali to je pak logičko programiranje, i to logičko programiranje u okviru relacijske paradigme, a ne funkcijsko programiranje). Bez obzira na količinu iskustva u imperativnom programiranju, suočavanje sa funkcijskim programiranjem predstavlja izazov, jer traži promjenu načina razmišljanja. Budući da je Scala moćan objektno-orijentirani jezik, koji ima i značajne funkcijske osobine (Scala je jedini statički jezik na JVM-u koji podržava objektno-orijentirano i funkcijsko programiranje), možemo na početku koristiti njegove objektno-orijentirane osobine, te se polako privikavati i na funkcijske. Poslije možemo koristiti onaj stil (imperativni ili funkcijski) koji nam više odgovara za određeni problem.

Kako kažu autori u [15], prvi korak je raspoznavanje razlike (između imperativnog i funkcijskog stila) u programskom kodu. Ako kod sadrži barem jednu var varijablu, onda je to vjerojatno imperativni stil, a ako sadrži samo val varijable, onda je to vjerojatno funkcijski stil. Ako želimo pisati funkcijskim stilom, trebamo pisati bez var varijabli. To nije lako za nas navikle na imperativno programiranje. Sljedeći primjer iz [15] prikazuje postepenu transformaciju imperativnog koda u kod koji je sve više funkcijski:

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i += 1
  }
}

def printArgs(args: Array[String]): Unit = {
  for (arg <- args)
    println(arg)
}

def printArgs(args: Array[String]): Unit = {
  args.foreach(println)
}
```

Refaktorirana metoda printArgs još uvijek nije čisto funkcijska, jer ima popratni efekt (side-efect) – štampa na standardni output stream. Znak da funkcija ima popratni efekt je i tip povratne vrijednosti Unit, što znači da funkcija ne vraća ništa interesantno (praktički je to procedura, a procedura se označava i tako da se ne koristi znak jednakosti prije bloka programskog koda). Više funkcijski pristup je da se napravi metoda koja formatira primljene argumente, ali ih ne štampa, kao ova:

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```

Funkcija formatArgs nema popratni efekt. Zato se ona može lakše testirati. Ako se ipak želi štampati rezultat, ona se može pozvati iz druge metode, koja ima popratni efekt:

```
println(formatArgs(args))
```

Svaki korisni program mora imati neki popratni efekt, inače ne bi mogao slati rezultate vanjskom svijetu. No, cilj je da se popratni efekti izoliraju u manji broj programskih modula. Preporuka za Scala programere bila bi: preferirajmo val varijable, imutabilne objekte, metode bez popratnih efekata, Ali, to ne znači da ćemo potpuno izbaciti var varijable, mutabilne objekte i popratne efekte – treba ih koristiti kada predstavljaju bolji izbor (naravno, izabrati pravu varijantu nije lako).

Često se postavlja pitanje – koji skup osobina mora imati neki programski jezik da bi se mogao nazvati funkcijskim programskim jezikom.

Vjerojatno ne postoji jednoznačan odgovor, ali obično se navode ove osobine kao nužne:
- funkcije višeg reda (higher-order functions);
- leksičko zatvaranje (lexical closure);
- podudaranje (sparivanje) uzorka (pattern matching);
- jednokratno pridruživanje (single assignment);
- lijena evaluacija (lazy evaluation);
- zaključivanje o tipovima (type inference);
- optimizacija repnog poziva (tail call optimization);
- razumijevanje listi (list comprehension): kompaktan i ekspresivan način definiranja listi kao osnovnih podatkovnih struktura funkcijskog programa;
- monadički efekti (monadic effects).

Neki dodaju još npr:
- funkcije kao vrijednosti, "građani prvog reda" ("first-class value");
- anonimne funkcije;
- currying;
- sakupljanje smeća (garbage collection).

Već smo vidjeli što znači jednokratno pridruživanje (single assignment) – to je kada u Scali koristimo val varijablu (ili u Javi final varijablu), kojoj se vrijednost može pridružiti samo jednom. Sakupljanje smeća (garbage collection) poteklo je još od Lispa, ali kasnije ga je preuzeo npr. OOPL Smalltalk, pa Eiffel, pa Java ..., tako da to više nije osobina (samo) funkcijskih jezika. Zaključivanje o tipovima (type inference) smo vidjeli u podtočki 5.5. Optimizacija repnog poziva (tail call optimization) je generalizacija optimizacije repne rekurzije (tail recursion), koju smo spomenuli u podtočki 5.2. U nastavku ćemo ukratko proći kroz većinu preostalih navedenih osobina. Neke od njih su samo "sintaksni šećer" (syntactic sugar), tj. olakšavaju rad programeru, ali ne predstavljaju stvarnu dopunu programskom jeziku i ne traže poseban "napor" od kompajlera. Takve su npr. podudaranje uzorka (pattern matching) ili razumijevanje listi (list comprehension). Neke druge osobine predstavljaju stvarnu dopunu programskom jeziku i postavljaju značajne zahtjeve pred kompajler. Ponekad, kao lijena evaluacija (lazy evaluation), traže čak i podršku od strane run-time sustava.

Funkcije su u Scali vrijednosti, "građani prvog reda" ("first-class value"). Kao i svaka druga vrijednost, mogu biti pridružene nekoj varijabli, poslane kao parametri nekoj drugoj funkciji, ili vraćene kao rezultat neke druge funkcije. Funkcije koje kao parametar ili povratnu vrijednost imaju neku drugu funkciju, zovu se funkcije višeg reda (higher-order functions). Budući da su u Scali funkcije vrijednosti, a istovremeno su u Scali sve vrijednosti objekti, slijedi da su u Scali sve funkcije objekti.

Sljedeći primjer iz [16] prikazuje funkciju višeg reda imena sum, koja kao prvi parametar prima (drugu) funkciju, što se u funkciji sum označava sa f: Int => Int. Time se kaže da ta druga funkcija (koja se koristi kao parametar), mora preslikavati Int u Int (primijetimo i to da je funkcija sum rekurzivna i da ima repnu rekurziju):

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

Sada definiramo tri funkcije koje imaju preslikavanje Int => Int i koristimo ih (za punjenje vrijednosti val varijabli) kao prvi parametar funkcije sum:

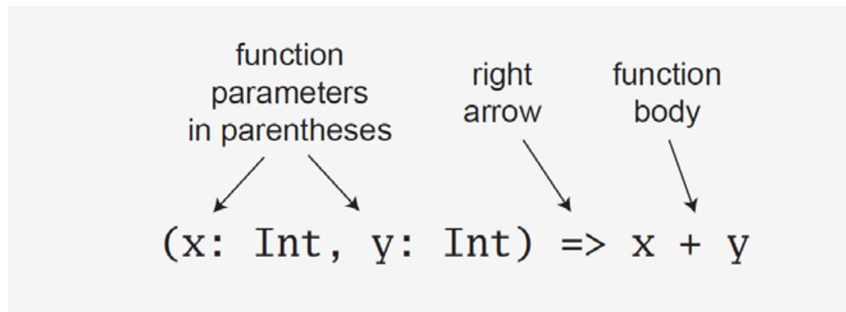
```
def id(x: Int): Int = x
def square(x: Int): Int = x * x
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)

val sumInts = sum(id, 1, 5) // 1+2+3+4+5=15
val sumSquares = sum(square, 1, 5) // 1+4+9+16+25=55
val sumPowersOfTwo = sum(powerOfTwo, 1, 5) // 2+4+8+16+32=62
```

Često kao parametre koristimo kratke funkcije, i to jednokratno. Tada, umjesto da definiramo eksplicitne funkcije, možemo direktno kao aktualne parametre koristiti tzv. anonimne funkcije, kao u sljedećem primjeru:

```
val sumInts2 = sum((x: Int) => x, 1, 5) // = 15
val sumSquares2 = sum((x: Int) => x * x, 1, 5) // = 55
```

Anonimna funkcija je jedna varijanta funkcijskog literala. Sintaksa funkcijskog literala dana je na sljedećoj slici 6.1:



slika 6.1. Sintaksa funkcijskog literala u Scali; Izvor: [15]

U prethodnim primjerima funkcija sum je, uz prvi parametar – funkciju, imala i dva dodatna parametra a i b, koji se ne čine naročito zanimljivima. U nastavku se prikazuje kako se funkcija sa n parametara može pretvoriti u funkciju sa manje od n parametara. Takav stil definicije funkcije naziva se currying, po svom promotoru Haskell B. Curryu, logičaru iz 20. stoljeća (po njemu je ime dobio i programski jezik Haskell), iako su ideju još prije dali Gottlob Frege i Moses Schönfinkel.

Slijedi prikaz modificirane funkcije sum, koja sada ima samo jedan parametar, funkciju f, koja preslikava Int u Int. No, funkcija sum kao povratnu vrijednost više nema Int, već (Int, Int) => Int, jer sadrži unutarnju (rekurzivnu) funkciju sumF, koja ima navedenu signaturu i koju funkcija sum vraća kao povratnu vrijednost:

```
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
  sumF
}
```

Sada možemo definirati funkcije, ili u ovom slučaju varijable (npr. dvije) koje koriste funkciju sum:

```
val sumInts = sum(x => x)
val sumSquares = sum(x => x * x)
```

pa te dvije varijable možemo koristiti tako da im pošaljemo vrijednosti za parametre a i b:

```
sumInts(1, 5) + sumSquares(1, 5) // 15 + 55 = 70
```

Kako se računanje izvršavalo? Scala je pretvorila prethodni izraz u sljedeći:

```
sum (x => x) (1, 5) + sum(x => x * x)(1, 5)
```

Izraz f (args1) (args2) (args3) je ekvivalentan izrazu ((f (args1)) (args2)) (args3), tj. aplikacija funkcije radi se s lijeva na desno.

Zapravo, u Scali se definicija funkcije u "currying stilu" može napisati još jednostavnije, bez unutarnje funkcije. Npr. prethodna verzija funkcije sum može se napisati i ovako:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

No, sada ne možemo definirati varijable (ili funkcije) koje koriste funkciju sum kao što smo to radili prije, već moramo odmah navesti sve parametre. Međutim, parametre možemo navesti, ali ih ostaviti nevezane (unbound), što radimo tako da umjesto konkretnog parametra stavimo znak _, pa na taj način dobijemo parcijalno apliciranu funkciju (partially applied function). Npr. parametri a i b su ovdje nevezani:

```
val sumInts = sum(x => x) // kompajler ne propušta; nedostaju parametri
val sumInts = sum(x => x) (_, _) // ispravno; nevezani parametri a i b
val sumSquares = sum(x => x * x) (_, _)

sumInts(1, 5) + sumSquares(1, 5) // 15 + 55 = 70; isto kao prije
```

Prikažimo kako možemo koristiti blok koda koji ima varijable koje nisu lokalne varijable, a nisu ni povezane sa parametrima tog bloka koda. Prije nego pozovemo taj blok koda, morat ćemo vezati te nevezane varijable. No, moći ćemo ih vezati i sa varijablama izvan lokalnog dosega (scope), tj. moći ćemo ih vezati sa varijablama iz okruženja. To se zove leksičko zatvaranje (lexical closure), ili samo zatvaranje (closure).

U sljedećem primjeru iz [16], funkcija loopThrough() prolazi kroz sve elemente od 1 do zadanog broja, koji je njen prvi parametar. Drugi parametar je blok programskog koda, koji se u funkciji poziva za svaku vrijednost od 1 do zadanog broja:

```
def loopThrough(number: Int)(closure: Int => Unit) {
  for (i <- 1 to number) { closure(i) }
}
```

Definirajmo blok koda koji ćemo poslije poslati prethodnoj metodi i pridružimo ga varijabli addIt:

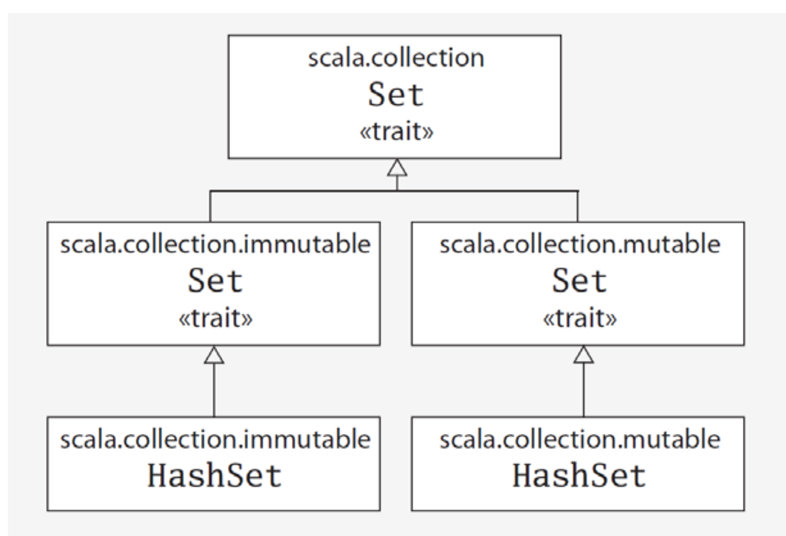
```
var result = 0
val addIt = { value: Int => result += value }
```

Primijetimo da je unutar bloka koda varijabla value vezana za parametar, dok varijabla result nije niti lokalna varijabla, niti vezana za parametar - ona je vezana za varijablu result izvan bloka koda. Sada možemo koristiti blok koda (odnosno varijablu addIt kojoj je on pridružen) u funkciji loopThrougth():

```
loopThrough(5) { addIt }
println("Total of values from 1 to 5 is " + result)
```

Nakon tog poziva, varijabla result (u glavnome kodu) bit će promijenjena (na vrijednost 15).

Funkcijski jezici poznati su po tome da imaju izvrstan način rada sa kolekcijama, naročito sa listama (list). Glavne Scala kolekcije su List, Set i Map. List je uređena kolekcija objekata, Set je neuređena kolekcija objekata, a Map je skup parova (ključ, vrijednost). Set i Map kolekcije imaju dvije varijante - imutabilnu i mutabilnu, što za kolekciju Set prikazuje sljedeća slika 6.2:



Slika 6.2. U Scali većina kolekcija (izuzeci su List, Array) ima mutabilnu i imutabilnu varijantu (na slici je Set); Izvor: [15]

List kolekcija ima samo imutabilnu varijantu, a Array samo mutabilnu. Željena verzija kolekcije bira se iz odgovarajućih paketa scala.collection.mutable ili scala.collection.immutable. Ako želimo modificirati kolekciju tako da su sve operacije unutar jedne dretve, možemo izabrati mutabilnu kolekciju. No, ako kolekciju želimo koristiti između više dretvi ili aktora (actors; bit će prikazani u sljedećoj točki) imutabilne kolekcije su svakako sigurnije. Podrazumijevane (default) kolekcije su imutabilne, tj. treba eksplicitno reći ako želimo koristiti mutabilne kolekcije.

U nastavku prikazujemo samo neke osnovne primjere rada sa Scala Set i List kolekcijama. Mogućnosti rada sa kolekcijama su brojne. Započinjemo sa Set kolekcijama.

```
var colors1 = Set("Blue", "Green", "Red")
var colors2 = colors1
println(colors2)   // prikazuju se sve 3 boje kao u skupu colors1
colors1 += "Black" // skupu colors1 dodaje se crna boja
println(colors1)   // prikazuju se 4 boje
println(colors2)   // prikazuju se 3 boje, skup colors2 nije se promijenio
```

Primijetimo da nismo trebali pisati riječ new za kreiranje skupa. Scala radi sljedeću pretvorbu naredbe:

```
val colors1 = Set("Blue", "Green", "Red") // izvorno
val colors1 = // pretvoreno
  new scala.collection.immutable.Set3[String]("Blue", "Green", "Red")
```

Nad skupovima se može npr. raditi unija ili presjek

```
val unijaBoja = colors1 union colors2
val presjekBoja = colors1 intersect colors2
```

Slijedi par primjera iz [16] sa List kolekcijama (slično se može raditi i sa drugim kolekcijama):

```
val feeds = List("blog.toolshed.com",
 "pragdave.pragprog.com", "dimsumthinking.com/blog")
println("First feed: " + feeds.head) // First feed: blog.toolshed.com
println("Second feed: " + feeds(1))
  // Second feed: pragdave.pragprog.com; prvi element označava se sa (0)!
println(feeds.filter(_ contains "blog").mkString(", "))
  // blog.toolshed.com, dimsumthinking.com/blog
println(feeds.forall(_ contains "com"))  // true
println(feeds.forall(_ contains "dave")) // false
println(feeds.exists(_ contains "dave")) // true
println(feeds.exists(_ contains "bill")) // false

val prijatelji = List("Ana", "Pero", "Marica", "Ivo")
prijatelji foreach
  { prijatelj => println(prijatelj + " je dobar prijatelj") }
// Ana je dobar prijatelj
// Pero je dobar prijatelj ... itd.
```

Podudaranje (sparivanje) uzorka (pattern matching) je vrlo važna mogućnost u Scali, iako spada u "sintaksni šećer". Naročito se koristi kod aktora (actors), ali i za parsiranje i sl. Npr. podudaranje uzorka se u Scali koristi i kod obrade iznimaka (exception), gdje catch dio try - catch - finally bloka radi drugačije nego u Javi (uz to što ne mora imati checked exceptions). Slijedi nekoliko jednostavnih primjera iz [18]. Prvo radimo sa konstantama, te radimo iscrpnu obradu svih mogućnosti:

```
def activity(day: String) {
  day match {
    case "Sunday" => print("Eat, sleep, repeat... " )
    case "Monday" => print("...code for fun..." )
    // ...
    case "Saturday" => print("Hangout with friends... " )
  }
}
```

U sljedećem primjeru radimo sa objektima, ali ne radimo iscrpnu obradu svih mogućnosti. Da se ne bi desila iznimka (exception) MatchError, koristimo višeznačnik (wildcard) _, kojim obuhvaćamo sve mogućnosti koje nisu eksplicitno obrađene:

```
object DayOfWeek extends Enumeration {
  val SUNDAY = Value("Sunday")
  val MONDAY = Value("Monday" ) // ... itd.
}
```

```
def activity(day: DayOfWeek.Value) {
  day match {
    case DayOfWeek.SUNDAY => println("Eat, sleep, repeat..." )
    case DayOfWeek.SATURDAY => println("Hangout with friends" )
    case _ => println("...code for fun..." )
  }
}
```

Osim konstanti i istovrsnih objekata, u case možemo imati i n-torke (tuples) i liste:

```
def processCoordinates(input: Any) {
  input match {
    case (a, b) => printf("Processing (%d, %d)... " , a, b)
    case "done" => println("done" )
    case _ => null
  }
}
processCoordinates((39, -104))
processCoordinates("done" )
```

Često u case imamo i objekte različitih tipova. Ponekad želimo da objekte različitih tipova ili karakteristika obradimo drugačije, pa možemo koristiti provjeru na tip, ali i dodatne if uvjete. Naglasimo da redoslijed obrade ide od gore prema dolje, pa se u sljedećem slučaju prvo obrađuju poruke (msg) tipa Int duže od 1000000, pa ostale poruke tipa Int, pa poruke tipa String:

```
def process(input: Any) {
  input match {
    case (a: Int, b: Int) => print("Processing (int, int)... " )
    case (a: Double, b: Double) =>
      print("Processing (double, double)... " )
    case msg: Int if (msg > 1000000) =>
      println("Processing int > 1000000" )
    case msg: Int => print("Processing int... " )
    case msg: String => println("Processing string... " )
    case _ => printf("Can't handle %s... " , input)
  }
}
```

Lijena evaluacija (lazy evaluation) ili lijene vrijednosti su vrlo važna osobina funkcijskih jezika, te ne predstavljaju samo "sintaksni šećer". Riječ je o tome da se inicijalizacija vrijednosti odgađa do trenutka kada se (lijena) vrijednost prvi put koristi. To može biti značajno i zato što se neka vrijednost možda neće koristiti, a njeno računanje je zahtjevno, pa lijena evaluacija može imati bolje performanse. Lijena evaluacija koristi se i kod generiranja beskonačnih nizova – ona redom generira sljedeći član niza.

U sljedećem primjeru iz [16] se za objekt klase Employee iz baze podataka čita redak koji predstavlja nadređenog radnika, te čitava lista redaka koji predstavljaju tim s kojim radnik radi. Svakako je brže da se baza podataka ne treba čitati dok to nije potrebno. Prva varijanta je bez lijene evaluacije, a druga s njom:

```
// bez lijene evaluacije
class Employee
  (id: Int, name: String, managerId: Int)
{
  val manager: Employee = Db.get(managerId)
  val team: List[Employee] = Db.team(id)
}

// sa lijenom evaluacijom
class Employee
  (id: Int, name: String, managerId: Int)
{
  lazy val manager: Employee = Db.get(managerId)
  lazy val team: List[Employee] = Db.team(id)
}
```

7. SCALA I KONKURENTO PROGRAMIRANJE

U ovoj točki prikazat ćemo ukratko tri "stila" konkurentnog programiranja u Scali. Prvi stil bit će uobičajeni imperativan stil, sličan onome u Javi. Drugi stil temelji se na aktorima (actors). Scala podržava aktore kroz barem dvije biblioteke. Jedna je standardna Scala biblioteka za aktore, a druga je temeljena na Akka frameworku (pisan u Scali). Akka aktori se danas sve više koriste, jer su bogatiji nego standardni Scala aktori (zapravo, moguće je da će u idućoj Scala verziji neke Akka biblioteke postati standardne Scala biblioteke). Akka framework osim aktora ima i puno drugih dodataka. Jedan od njih je i softverska transakcijska memorija, koja će ukratko biti prikazana kao treći stil konkurentnog programiranja.

Kao i u Javi, i u Scali se svaka instanca klase AnyRef (= Java klasa Object) može koristiti kao monitor (uz sve mane koje je naveo Brinch Hansen u [1]), pozivajući ove operacije, koje se ponašaju isto kao što je prikazano za Javu u 4. točki (synchronized je Java ključna riječ, ostalo su metode Java klase Object). U Scali su to metode Scala klase Monitor, koja je primitivna (temeljena na run-time sustavu):

```
def synchronized[A] (e: => A): A
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

Future je vrijednost koja se računa paralelno sa nekom drugim klijentskom dretvom, kako bi se u nekom trenutku iskoristila u tom klijentskom programu. Slijedi tipični primjer korištenja (iz [16]) i definicija future metode iz biblioteke scala.concurrent.ops:

```
import scala.concurrent.ops._
...
val x = future(someLengthyComputation)
anotherLengthyComputation
val y = f(x()) + g(x())

def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  fork { result.set(p) }
  (() => result.get)
}
```

Scala primjer za čitatelje / pisce (više konkurentnih čitatelja, samo jedan pisac):

```
import scala.concurrent._
class ReadersWriters {
  val m = new MailBox
  private case class Writers(n: Int), Readers(n: Int) { m send this }
  Writers(0); Readers(0)
  def startRead = m receive {
    case Writers(n) if n == 0 => m receive {
      case Readers(n) => Writers(0); Readers(n + 1)
    }
  }
  def startWrite = m receive {
    case Writers(n) =>
      Writers(n + 1)
      m receive { case Readers(n) if n == 0 => }
  }
  def endRead = m receive {
    case Readers(n) => Readers(n1)
  }
  def endWrite = m receive {
    case Writers(n) => Writers(n1);
    if (n == 0) Readers(0)
  }
}
```

Slijedi kratak prikaz neimperativnog stila, pomoću poštanskih pretinaca (mailboxes) i aktora (actors). Kako je napomenuto u 2. točki, već neko vrijeme se sve više zagovara konkurentnost temeljena na slanju poruka (message-passing concurrency). Konkurentnost na temelju slanja poruka je prirodni stil za distribuirane sustave i omogućava visoku raspoloživost. To je programski stil kod kojega se program sastoji od nezavisnih entiteta, aktora (actors), koji si šalju poruke asinkrono, bez čekanja na odgovor. Model aktora kreirao je 70-ih Carl Hewitt. Najpoznatiji jezik koji je primijenio taj model (još 80-ih) je Erlang, od kojeg je Scala dosta toga preuzela i nadogradila (za razliku od Scale, Erlang je jezik sa dinamičkom provjerom tipova).

Poštanski pretinci su fleksibilan konstrukt visoke razine, te služe za procesiranje sinkronizacije i komunikacije. Omogućavaju slanje i primanje poruka, a poruka je bilo koji objekt. Postoji posebna poruka TIMEOUT koja se koristi za signaliziranje time-outa:

```
case object TIMEOUT
```

Poštanski pretinci implementiraju ovu signaturu:

```
class MailBox {
  def send(msg: Any)
  def receive[A](f: PartialFunction[Any, A]): A
  def receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A
}
```

Poruke se dodaju u poštanski pretinac asinkrono, pomoću send metode (pošiljatelj poruke ne čeka nakon što pozove send). Poruke se redom preuzimaju iz pretinca pomoću receive (ili receiveWithin) metode, kojoj se šalje procesor poruka f (parcijalna funkcija) kao parametar. Tipično se ta funkcija f implementira pomoću podudaranja uzorka (pattern matching). Receive metoda blokira dok se u pretincu ne pojavi odgovarajuća poruka (dok metoda receive "beskonačno" čeka na poruku, metoda receiveWithin čeka zadano vrijeme - zato se preporučuje njeno korištenje). Zatim se poruka (koja ne mora biti zadnja) vadi iz pretinca i blokirana dretva se restarta primjenjujući procesor f na poruku.

Jednostavan primjer korištenja pretinca je sljedeći (iz [16]):

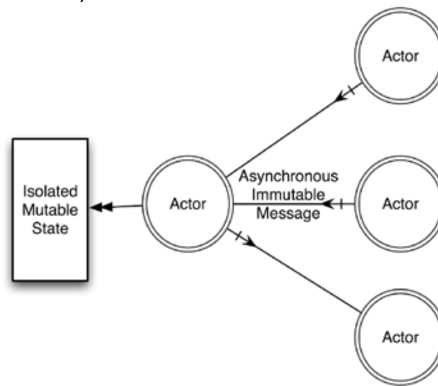
```
class OnePlaceBuffer {
  private val m = new MailBox // An internal mailbox
  private case class Empty, Full(x: Int) // Types of messages we deal with
  m send Empty // Initialization
  def write(x: Int)
    { m receive { case Empty => m send Full(x) } }
  def read: Int =
    m receive { case Full(x) => m send Empty; x }
}
```

Slijedi prikaz vrlo pojednostavljene implementacije aktora (iz [16]). U najjednostavnijoj varijanti, aktor bi se mogao definirati kao miksana kompozicija standardne Java klase Thread i Scala traita MailBox, u kojoj se nadjača run metoda Thread klase, tako da izvršava ponašanje definirano u act metodi. Metoda (neobičnog) imena ! jednostavno poziva send metodu iz klase MailBox:

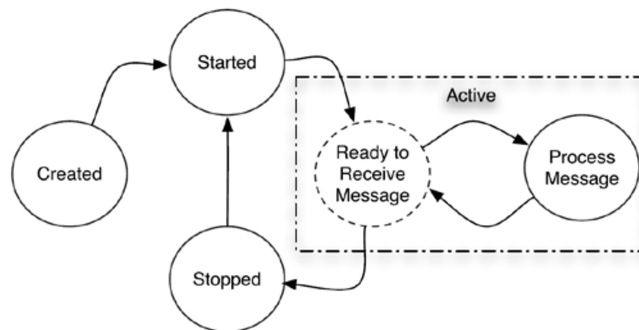
```
abstract class Actor extends Thread with MailBox {
  def act(): Unit
  override def run(): Unit = act()
  def !(msg: Any) = send(msg)
}
```

Prava implementacija aktora je svakako složenija od ove. Prije svega, već u standardnoj Scala biblioteci aktora imamo još jednu varijantu, koja se ne zasniva na tome da svaki aktor bude jedna dretva, već aktori dijele dretve iz pričuve dretvi (thread poll). Umjesto metoda receive i receiveWithin, u toj varijanti postoje ekvivalentne metode react i reactWithin. Razlog postojanja ove druge varijante je taj što su na JVM-u dretve skupe (o čemu je bilo govora na početku 4. točke). Dok se u prvoj varijanti (jedan aktor = 1 dretva) može kreirati tisuće aktora, u drugoj varijanti (aktori dijele dretve iz pričuve) može se bez problema kreirati na stotine tisuća aktora – to su tzv. lagani aktori. No, lagani aktori nemaju samo pozitivne osobine – sa njima je nešto teže raditi. Ako se obrađuju poruke koje nisu jednostavne i kratke, i ako broj aktora koje treba kreirati nije prevelik, onda je bolje koristiti prvu varijantu, tj. ne-lagane aktore i metode receive i receiveWithin.

Sljedeće slike prikazuju kako aktori komuniciraju i kakav je životni ciklus aktora (neovisno da li je implementiran kao lagani ili ne-lagani aktor):



Slika 7.1. Aktori izoliraju mutabilno stanje i komuniciraju slanjem imutabilnih poruka; Izvor: [19]



Slika 7.2. Životni ciklus aktora; Izvor: [19]

Sljedeći primjer iz [18] prikazuje korištenje ne-laganih aktora pomoću receive metode:

```
import scala.actors._
import scala.actors.Actor._
val caller = self
val accumulator = actor {
  var sum = 0
  var continue = true
  while (continue) {
    sum += receive {
      case number: Int => number
      case "quit" =>
        continue = false
        0
    }
  }
  caller ! sum
}
accumulator ! 1
accumulator ! 7
accumulator ! 8
accumulator ! "quit"
receive { case result => println("Total is " + result) }

Total is 16
```

Glavni program šalje aktoru (koji je vezan za val varijablu accumulator) redom poruke 1, 7, 8, "quit", ne čekajući na odgovor, dok ne dođe do (svoje) metode receive, kada čeka odgovor. Aktor se vrti u petlji dok ne dobije poruku "quit", nakon čega šalje rezultat glavnom programu. Aktor u svojoj receive metodi povećava rezultat za dobiveni broj (kad dobije poruku "quit", povećava za nulu).

Druga podvarijanta prikazuje korištenje ne-laganih aktora pomoću receiveWithin metode:

```
val caller = self

val accumulator = actor {
  var sum = 0
  var continue = true
  while (continue) {
    sum += receiveWithin(10000) {
      case number: Int => number
      case TIMEOUT =>
        println("Timed out! Will return result now")
        continue = false
        0
    }
  }
  caller ! sum
}

accumulator ! 1
accumulator ! 7
accumulator ! 8
receiveWithin(20000) { case result => println("Total is " + result) }

Timed out! Will return result now
Total is 16
```

Vidi se da i aktor i glavni program imaju ograničeno čekanje. Aktor čeka maksimalno 10 sekundi, a glavni program 20 sekundi. Aktor ima obradu TIMEOUT iznimke, tako da vraća rezultat nakon isteka njegovog vremena (10 sekundi), pa ovdje glavni program ne šalje aktoru poruku za kraj.

Sljedeći primjer prikazuje varijantu sa laganim aktorima, gdje aktor koristi reactWithin metodu (korištenje metode react nije prikazano). Za razliku od receive metode kod ne-laganih dretvi, react metoda ne vraća nikakav rezultat. Može se pojednostavljeno zamisliti da ona interno izvrši iznimku (exception) i vraća dretvu u pričuvu dretvi. Ako želimo nastaviti sa procesiranjem nakon što se sa react obradi jedna poruka, ta poruka mora pozvati neku drugu poruku, ili samu sebe, kao u ovom primjeru:

```
val caller = self

def accumulate(sum: Int) {
  reactWithin(10000) {
    case number: Int => accumulate(sum + number)
    case TIMEOUT =>
      println("Timed out! Will send result now")
      caller ! sum
  }
  println("This will not be called...")
}

val accumulator = actor { accumulate(0) }
accumulator ! 1
accumulator ! 7
accumulator ! 8
receiveWithin(20000) { case result => println("Total is " + result) }

Timed out! Will send result now
Total is 16
```

Dakle, kod aktora iz standardne Scala biblioteke, lakše je raditi sa ne-laganim aktorima (koji koriste metodu receive), nego sa laganim aktorima (koji koriste metodu react).

U [15] autori daju preporuke o tome kako bi trebalo pisati programe sa aktorima. Prije svega, navode da aktor ne bi smio blokirati, jer ako aktor blokira na nekom zahtjevu, neće uopće niti vidjeti sljedeći zahtjev. U najgorem slučaju može se desiti čak i deadlock (kao što se može desiti kod sinkronizacije pomoću lokota), kad više blokiranih aktora čekaju jedan na drugoga. Aktori bi trebali komunicirati isključivo preko poruka, kako bi omogućili da pisanje višedretvenih programa bude svedeno na pisanje skupa nezavisnih jednodretvenih programa, koji jedan s drugim komuniciraju pomoću asinkronih poruka. Aktori nam omogućavaju da se ne moramo uvijek odreći mutabilnih objekata. Budući da su dobro pisani aktori nezavisni, onda nije važno da li su unutar njih korišteni mutabilni objekti. Međutim, krucijalno je da poruke budu imutabilne.

Za razliku od običnih sinkronih metoda, kod kojih metoda-pozivatelj (druge metode) zna što je radila prije nego je pozvala drugu metodu, aktor-pozivatelj nastavlja sa radom, jer je poziv (drugog) aktora asinkron. Kada dođe odgovor, aktor-pozivatelj "teže interpretira" odgovor drugog aktora. Zbog toga se često u poruke dodaju redundantni podaci.

U prethodnim primjerima rad sa aktorima bio je temeljen na standardnoj Scala biblioteci. Kako je već prije napomenuto, puno bogatiji rad sa aktorima ima framework Akka. Taj framework zajedno sa jezikom Scala i Web frameworkom Play čini tzv. Typesafe Stack (sada u verziji 2.0), koji se može koristiti iz Scale, iz Jave, ili iz bilo kojeg jezika na JVM-u. Kao i Scala jezik, tako su i Akka i Play open-source programi, i mogu se besplatno preuzeti sa stranica www.typesafe.com firme Typesafe. Na stranicama piše (i) sljedeće: "Typesafe was founded in 2011 by the creators of the Scala programming language and Akka middleware, who joined forces to create a modern software platform for the era of multicore hardware and cloud computing workloads."

Ovdje neće biti prikazan rad sa Akka aktorima, već rad sa Akka softverskom transakcijskom memorijom, Akka STM. Napomenimo da se može istovremeno koristiti oboje, i aktori i STM.

Scala transakcije se u Akka STM-u definiraju vrlo jednostavno. Transakcija se stavlja unutar bloka koji počinje riječju atomic (to je različito od transakcija u npr. Oracle bazi podataka, gdje je sve unutar transakcije, jer nova transakcija automatski počinje čim završi stara, tj. nakon COMMIT ili ROLLBACK):

```
atomic {
  //code to run in a transaction....
  /* return */ resultObject
}
```

Akka STM dozvoljava da se transakcije gnijezde, kao u ovom primjeru iz [19]. Prvo se definira klasa Account, sa atomarnim metodama deposit i withdraw:

```
import akka.stm.Ref
import akka.stm.atomic

class Account(val initialBalance: Int) {
  val balance = Ref(initialBalance)
  def getBalance() = balance.get()

  def deposit(amount: Int) = {
    atomic {
      println("Deposit " + amount)
      if(amount > 0)
        balance.swap(balance.get() + amount)
      else
        throw new AccountOperationFailedException()
    }
  }
  def withdraw(amount: Int) = {
    atomic {
      val currentBalance = balance.get()
      if(amount > 0 && currentBalance >= amount)
        balance.swap(currentBalance - amount)
      else
        throw new AccountOperationFailedException()
    }
  }
}
```

Zatim se objekti klase Account koriste u klasi (zapravo singleton klasi, tj. Scala object strukturi) AccountService. Vidi se da se u atomarnoj metodi transfer pozivaju atomarne metode deposit i withdraw, tj. te se transakcije gnijezde unutar transakcije koju čini metoda transfer. Naredbe println služe samo da se lakše prate zbivanja, a naredba sleep unutar metode transfer, kao i kreiranje (drugog) aktora u metodi main, služe da bi se simulirao višekorisnički rad:

```
object AccountService {
  def transfer(from: Account, to: Account, amount: Int) = {
    atomic {
      println("Attempting transfer...")
      to.deposit(amount)
      println("Simulating a delay in transfer...")
      Thread.sleep(5000)
      println("Uncommitted balance after deposit $" + to.getBalance())
      from.withdraw(amount)
    }
  }
  def transferAndPrintBalance(
    from: Account, to: Account, amount: Int) = {
    var result = "Pass"
    try {
      AccountService.transfer(from, to, amount)
    } catch {
      case ex => result = "Fail"
    }
    println("Result of transfer is " + result)
    println("From account has $" + from.getBalance())
    println("To account has $" + to.getBalance())
  }
  def main(args: Array[String]) = {
    val account1 = new Account(2000)
    val account2 = new Account(100)

    actor {
      Thread.sleep(1000)
      account2.deposit(20)
    }

    transferAndPrintBalance(account1, account2, 500)
    println("Making large transfer...")
    transferAndPrintBalance(account1, account2, 5000)
  }
}
```

Kako kaže autor u [19], STM ima puno dobrih strana:

- omogućava maksimalnu konkurentnost, direktno upravljanu sa stvarnim potrebama aplikacije, a ne tehničkim stvarima, kao što su zaključavanje i sl.;

- predstavlja programski model bez lokota (lock free); ne moramo voditi brigu o redoslijedu zaključavanja, a ipak nema deadlocka;

- osigurava da se identiteti mijenjaju samo unutar transakcije.

Uza sve prednosti, STM nije bez mana. STM je pogodan za konkurentno čitanje, ali sa relativno malo konkurentnog mijenjanja. Kada se dvije transakcije sukobe oko mijenjanja istog objekta ili podataka, samo jedna od njih će uspjeti, a druga će se automatski poništiti. Performanse neće biti loše ako su sukobi transakcija koje mijenjaju isti objekt rijetki. Ali, kako se sukobi povećavaju, stvari će u najboljem slučaju biti spore. U najgorem slučaju, desit će se da nijedna transakcija ne napreduje – livelock.

U slučaju puno kolizija kod pisanja, STM nije dobra solucija. Tada je bolje koristiti aktore.

ZAKLJUČAK

U zadnjih (otprilike) desetak godina sve se više govori (i) o tome da bi programiranje trebalo biti multiparadigmatsko, tj. da bismo trebali koristiti onu jezičnu paradigmu koja je najprirodnija za rješavanje određenog problema, a ne koristiti "jedan alat za sve probleme". Ovdje postoje barem dva pristupa – jedan je da koristimo više programskih jezika (npr. jedan objektni, jedan funkcijski, jedan logički ...), a drugi pristup kaže da je puno jednostavnije i bolje imati jedan programski jezik koji podržava različite paradigme (po mogućnosti sve, kao što je jezik Oz kojeg opisuju autori u [20]). Dobar dio ovog rada odnosi se na programski jezik Scala, koji je prije svega objektni programski jezik, ali tako nadograđen da ima i značajne funkcijske osobine.

Kreator Scale, profesor Martin Odersky, naglasio je da nije želio da Scala bude 100% kompatibilna sa Javom pod svaku cijenu. Npr. odustao je od toga da Scala nizovi (arrays) budu kovarijantni (covariant) kao u Javi (jedna od najvećih grešaka kod razvoja Jave). Scala je kreirana kao čisti objektno-orijentirani jezik (kao i Eiffel, dok Java nije čisti OOPL), i u nju su uvedene neke objektno-orijentirane mogućnosti koje Java nema. Osim toga, na temelju objektno-orijentiranih mogućnosti izgrađene su i brojne funkcijske mogućnosti, tako da je Scala i funkcijski jezik (ali nije čisti). Objektno-orijentirani pristup dobar je kada se skup klasa prirodno proširuje. Funkcijski pristup pogodniji je kada je skup struktura relativno fiksan, ali se žele uvesti nove operacije nad postojećim strukturama, gdje je najbolji pristup podudaranje uzorka (pattern matching). Sa funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za konkurentno programiranje.

Naime, osim multiparadigmatičnosti, značajna karakteristika modernog programskog jezika mora biti i dobra podrška za konkurentno programiranje. Mooreov zakon vrijedi i dalje, a on kaže da se broj tranzistora na mikroprocesorskom čipu udvostručuje otprilike svake dvije godine (Gordon Moore je jedan od osnivača tvrtke Intel, a tu je tvrdnju iznio 1965. godine). No, vrijeme jednostavnog povećanja brzine programa je prošlo. Broj tranzistora po jedinici površine se i dalje povećava po Mooreovom zakonu, ali je radni takt procesora praktički je prestao rasti oko 2004. / 2005. godine, te se sada brzina može postići korištenjem modernih višejezgrenih procesora. Procjenjuje se da bi Mooreov zakon zaista mogao prestati vrijediti za otprilike desetak godina, jer minijaturizacija elektroničkih komponenti ima svoje granice, npr. kod 10-nanometarske tehnologije širina tranzistora je svega 30-ak atoma. Naravno, moguće je da će sasvim nove tehnologije (koje su danas u eksperimentalnim fazama), podržati Mooreov zakon i dalje.

Danas postoje ova tri najčešća načina sinkronizacije konkurentnih procesa (ili dretvi):

- lokoti, tj. blokirajuća sinkronizacija;
- neblokirajući sinkronizacijski mehanizmi, bez lokota (lock-free);
- softverska, hardverska i hibridna transakcijska memorija (STM, HTM, hibridna TM).

Scala podržava tri "stila" konkurentnog programiranja. Prvi stil je uobičajeni imperativan stil, sličan onome u Javi. On uključuje i lokote i neblokirajući sinkronizaciju. Drugi stil temelji se na aktorima (actors), koji se do sada nije značajno koristio, osim u jeziku Erlang, od kojega je Scala puno preuzela (i nadogradila). Scala podržava aktore kroz barem dvije biblioteke. Jedna je standardna Scala biblioteka za aktore, a druga je temeljena na Akka frameworku (pisan je u Scali). Akka aktori se danas sve više koriste, jer su bogatiji nego standardni Scala aktori (zapravo, moguće je da će u idućoj Scala verziji neke Akka biblioteke postati standardne Scala biblioteke). Akka framework osim aktora ima i puno drugih dodataka. Jedan od njih je i softverska transakcijska memorija, treći stil konkurentnog programiranja. Moguće je istovremeno koristiti i aktore i STM.

Scala je izvrstan jezik i za pisanje DSL-ova (Domain-Specific Language), programskih jezika za specifičnu problemsku domenu. Profesor Odersky je krajem 2010. od European Research Councila (ERC) dobio Advanced Investigator Grant (oko 2,5 milijuna eura za projekt u trajanju od 5 godina, 2011.-2016.) za svoj projekt vezan za DSL-ove za paralelno programiranje pisane u Scali. Scala je, kao i Eiffel (kreiran 1986., skoro deset godina prije Jave) vrlo rigorozno matematički specificirana, za razliku od Jave. Scala podržava statičku provjeru tipova, kao i C++, Eiffel, Haskell, Java, C# ... , za razliku od npr. jezika Python, Ruby, Groovy, Clojure ... , kod kojih se greška u tipovima pokaže tek kod izvođenja programa.

Držimo da su u konkurenciji na JVM-u velike šanse na strani jezika Scala. Uostalom, to pokazuje i sve veći broj značajnih firmi koje su većim ili manjim dijelom prešle na programiranje u Scali (npr. Twitter, Linkedin, Juniper, Foursquare). No, važno je da se može programirati u Scali bez da se napusti Java, jer se Java i Scala programski kod mogu jako dobro upotpunjavati.

LITERATURA

1. Brinch Hansen, P. (1998): Java insecure Parallelism, članak, Syracuse University, http://brinch-hansen.net/papers/1999b.pdf (rujan 2012.)

2. Cantrill B., Bonwick, J. (2008), Real-world Concurrency, ACM QUEUE September 2008, www.acmqueue.com (rujan 2012.)

3. Da Luo, Z., Nir-Buchbinder, J., Das, R. (2010): Java concurrency bug patterns for multicoresystems - Six lesser known Java concurrency bug patterns, DeveloperWorks, IBM, http://www.ibm.com/developerworks/java/library/j-concurrencybugpatterns/index.html (rujan 2012.)

4. Gopalakrishnan, G. (2006): Computation Engineering - Applied Automata Theory and Logic, Springer

5. Göetz B. i ostali (2006): Java Concurrency in Practice, Addison-Wesley

6. Haring, R. (2011): The Blue Gene/Q Compute Chip, prezentacija, IBM BlueGene Team, ftp://public.dhe.ibm.com/common/ssi/ecm/en/dcw03006usen/DCW03006USEN.PDF (rujan 2012.)

7. Harris, T. (2010): Transactional Memory (2.izdanje), Morgan & Claypool

8. Harris, T. (2011): Multi-Core programming, prezentacija, Microsoft Research, http://www.cl.cam.ac.uk/teaching/1112/R204/slides-tharris.pdf (rujan 2012.)

9. Herlihy, M., Shavit, N. (2012): The Art of Multiprocessor Programming (2.izdanje), Elsevier / Morgan Kaufmann Publishers

10. Martin, J.C. (2010): Introduction to Languages and The Theory of Computation (4.izdanje), McGraw-Hill

11. Meyer, B. (1997): Object-Oriented Software Construction, Prentice Hall

12. Meyer, B. (2009): Touch of Class - Learning to Program Well with Objects and Contracts, Springer

13. Nowak, M.A. (2006): Evolutionary Dynamics, Belknap/Harvard Press, Cambridge Massachusetts

14. Odersky, M. (2009), The Origins of Scala + The Goals of Scala's Design + The Purpose of Scala's Type System + The Point of Pattern Matching in Scala, (4 intervjua), http://www.artima.com/scalazine/articles/origins_of_scala.html (rujan 2012.)

15. Odersky, M., Spoon, L., Venners, B. (2010): Programming in Scala (2.izdanje), Artima Press, Walnut Creek, California

16. Odersky, M. (2011): Scala By Example (draft), Programming Methods Laboratory EPFL, Switzerland

17. Srbljić, S. (2007): Uvod u teoriju računarstva, Element, Zagreb

18. Subramaniam, V. (2008): Programming Scala - Tackle Multicore Complexity on the JVM, The Pragmatic Bookshelf, Dallas, Texas / Raleigh, North Carolina

19. Subramaniam, V. (2011): Programming Concurrency on the JVM - Mastering Synchronization, STM, and Actors, The Pragmatic Bookshelf, Dallas, Texas / Raleigh, North Carolina

20. Van Roy P., Haridi S. (2004): Concepts, Techniques, and Models of Computer Programming, The MIT Press Cambridge, Massachusetts