VISOKA KONKURENTNOST NA JVM-u

Zlatko Sirotić, univ.spec.inf.

Istra informatički inženjering d.o.o.

Pula

Uvod

Riječ "konkurentnost" u naslovu ima **trostruko značenje**:

- **1. Velika konkurencija između jezika koji mogu raditi na JVM-u**, uz Javu:
  Jython, Jruby, Groovy, Scala, Clojure, Ceylon (u razvoju)
- **2. Podrška programskog jezika konkurentnom programiranju** sve je važnija danas, u eri višejezgrenih mikroprocesorskih čipova
- **3. Konkurencija između imperativnih (i to uglavnom objektnih) i funkcijskih jezika;** u zadnje vrijeme sve su popularniji funkcijski jezici (functional languages), kod kojih su neke varijante konkurentnog programiranja lakše izvedive

Teme

- 1. Jezici, automati, gramatike
- 2. Neke klasifikacije programskih jezika
- 3. Neki trendovi u razvoju hardvera koji utječu na konkurentno programiranje
- 4. Konkurentno programiranje u Javi
- 5. Scala kao objektno-orijentirani programski jezik
- 6. Scala kao funkcijski jezik
- 7. Scala i konkurentno programiranje

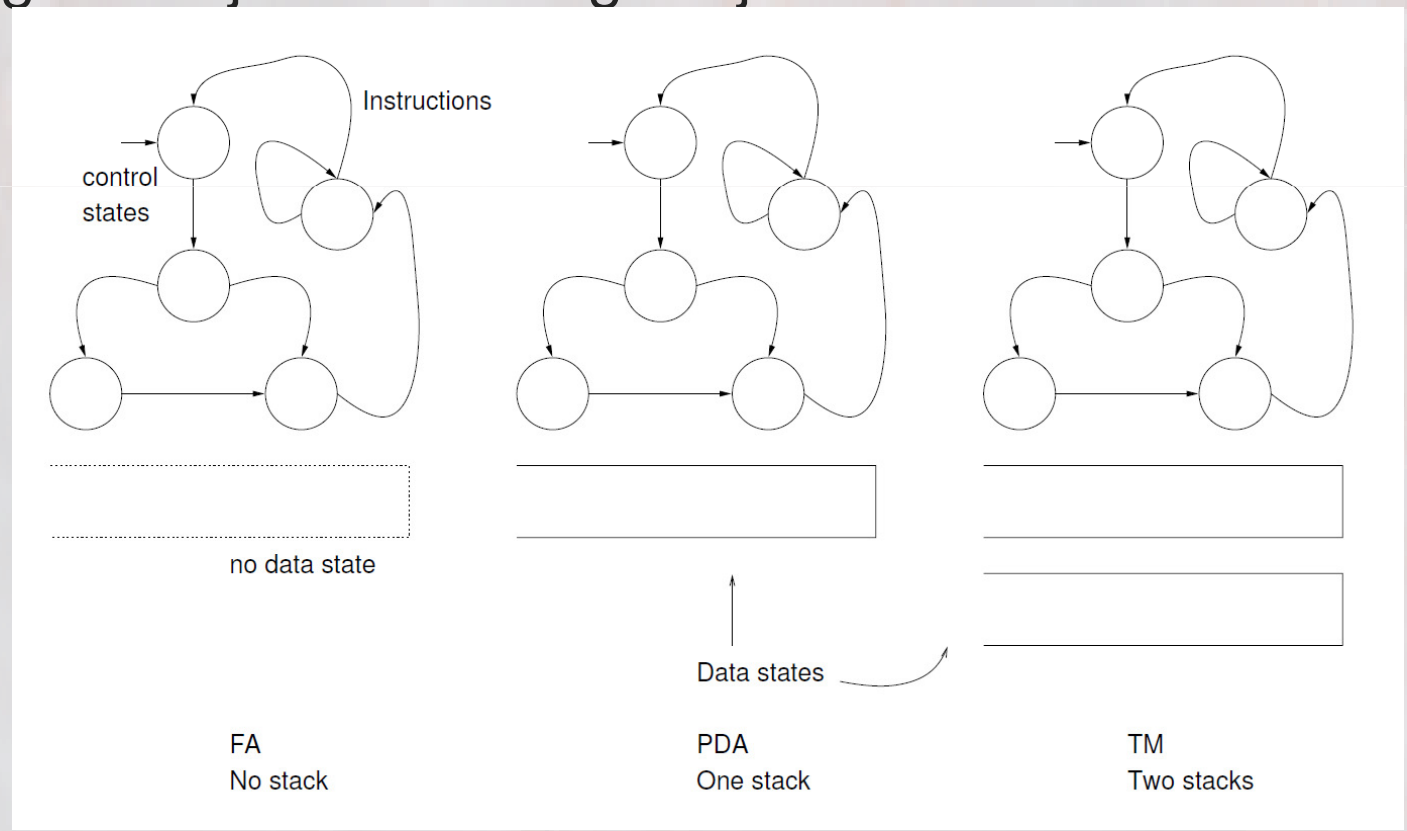
1. Hijerarhija jezika - Noam Chomsky

- Noam Chomsky još je 50-ih godina napravio poznatu klasifikaciju jezika (ljudskih i formalnih), koja je postala važna u računarstvu, naročito u konstrukciji jezičnih procesora i teoriji 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.			

1. Konačni automat (FA), potisni automat (PDA) i Turingov stroj (TM)

- Prvo je 30-ih godina otkriven najjači stroj, Turingov stroj, a konačni automat i potisni automat tek 50-ih godina.
- Turingov stroj ima dva stoga ili jednu traku:



1. Osnovni elementi formalnih 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

1. Primjer regularne, kontekstno neovisne i kontekstno ovisne gramatike

Finite state	Context free	Context sensitive
$S \rightarrow 0S$	$S \rightarrow 0S1$	$S \rightarrow 0AS2$
$S \rightarrow A$	$S \rightarrow \varepsilon$	$S \rightarrow 012$
$A \rightarrow 1A$	$L = 0^n1^n$	$A0 \rightarrow 0A$
$A \rightarrow \varepsilon$		$A1 \rightarrow 11$
$L = 0^m1^n$		$L = 0^n1^n2^n$

1. Uobičajene faze rada kompajlera

Uobičajeno, kompajler radi ove faze:

- **Leksička analiza programa**; koristi se 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; često se koriste ad hoc pristupi; nažalost, u području kontekstno ovisne gramatike ne postoje formalizmi koji bi po jednostavnosti i praktičnosti odgovarali BNF notaciji

1. Uobičajene faze rada kompajlera - nastavak

- **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)

2. Neki kriteriji za klasifikaciju programskih jezika

- **Primjena** (application): neki programski jezici imaju opću namjenu (general-purpose); drugi služe za određeno specijalno područje primjene
- **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
- **Provjerljivost** (verifiability): neki jezici su dizajnirani tako da kompajler (i ostali alati) nađe što više potencijalnih grešaka u programu prije izvršavanja; drugi jezici su fleksibilniji, ali se kod njih neke greške nalaze tek kod izvršavanja, npr. greške kod provjere tipova kod dinamičkih jezika

2. Neki kriteriji za klasifikaciju programskih jezika - nastavak

- **Razina apstrakcije** (abstraction level)
- **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), npr. funkcijski jezici i logički jezici
- **Arhitekturni stil** (architectural style): definira glavne kriterije za podjelu sustava u programske module – najčešće procedure (proceduralni stil) ili klase (objektni stil)

2. Osam modela računanja

- Da li se koristi konkurentnost (concurrency) - C
- Da li se koristi eksplicitno stanje (state) - S
- Da li se koristi lijena evaluacija (lazy evaluation) - L

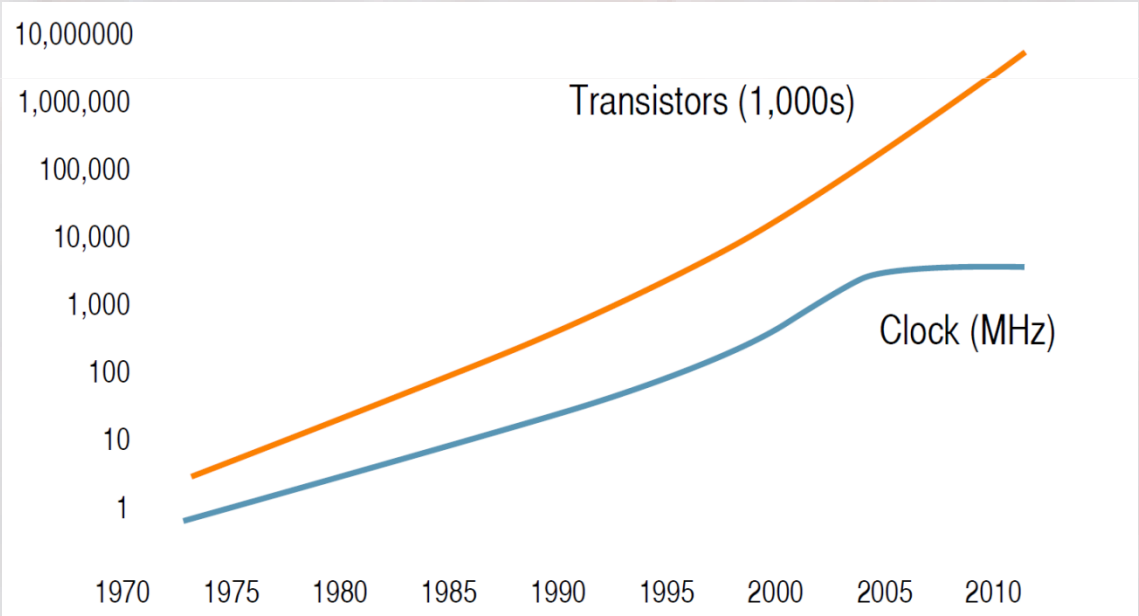
C	L	S	Description
			Declarative model (chapters 2 &3, Mercury, Prolog).
		×	Stateful model (chapters 6 & 7, Scheme, Standard ML, Pascal).
	×		Lazy declarative model (Haskell).
	×	×	Lazy stateful model.
×			Eager concurrent model (chapter 4, dataflow).
×		×	Stateful concurrent model (chapters 5 & 8,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

2. Različiti pristupi konkurentnom programiranju

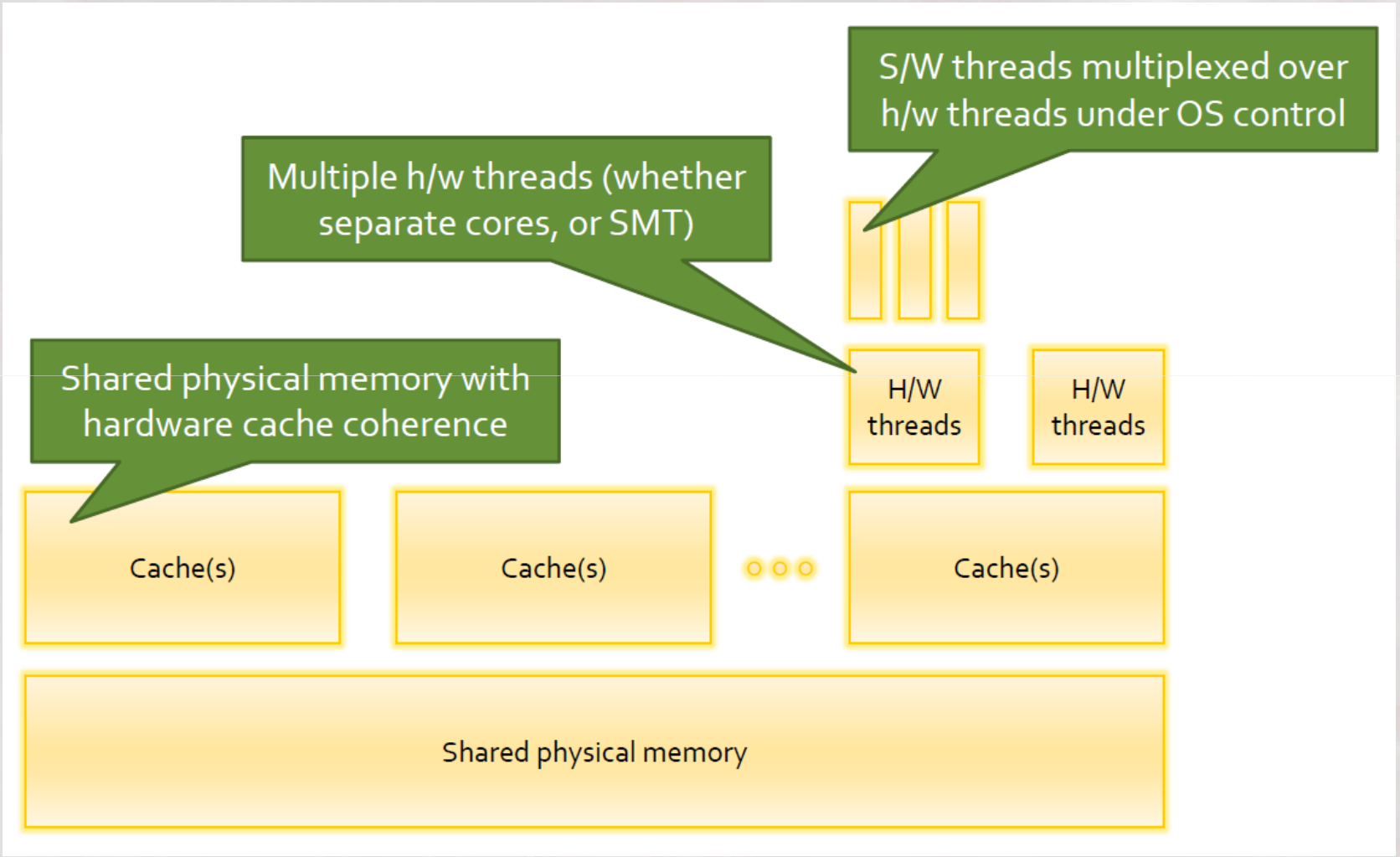
Model	Approaches
Sequential (declarative or stateful)	Sequential programming Order–determining concurrency Coroutining Lazy evaluation
Declarative concurrent	Data–driven concurrency Demand–driven concurrency
Stateful concurrent	Use the model directly Message–passing concurrency Shared–state concurrency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

3. Mooreov zakon vrijedi i dalje, ali malo drugačije

- Mooreov zakon: "Broj tranzistora na mikroprocesorskom čipu udvostručuje se otprilike svake dvije godine"
- No, radni takt procesora praktički je prestao rasti oko 2004. / 2005. godine – povećava se broj jezgri procesora



3. Sistemski model računala



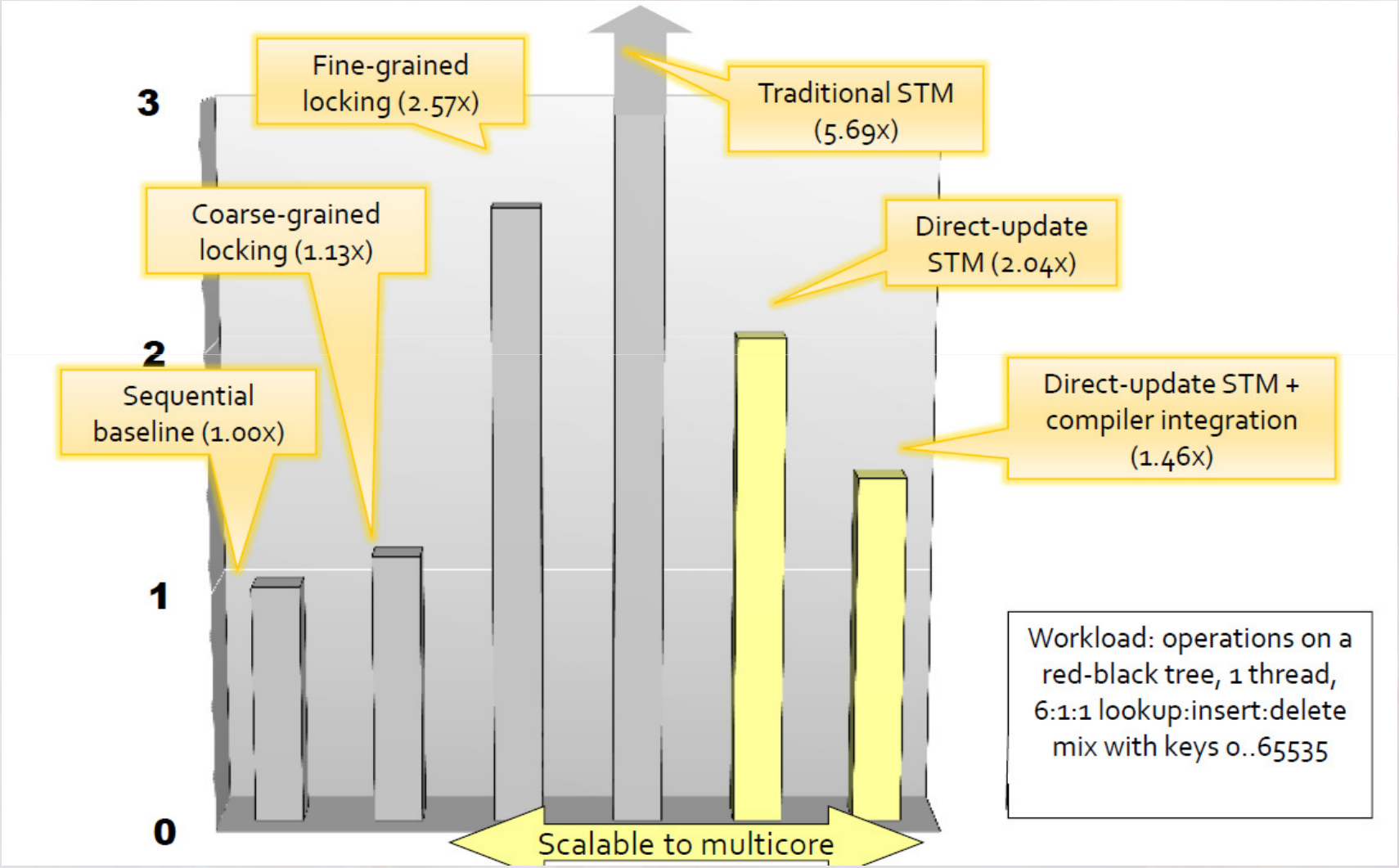
3. Uobičajeni načini sinkronizacije konkurentnih procesa

- **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)

3. Poboljšanja kod današnjih realizacija STM-a

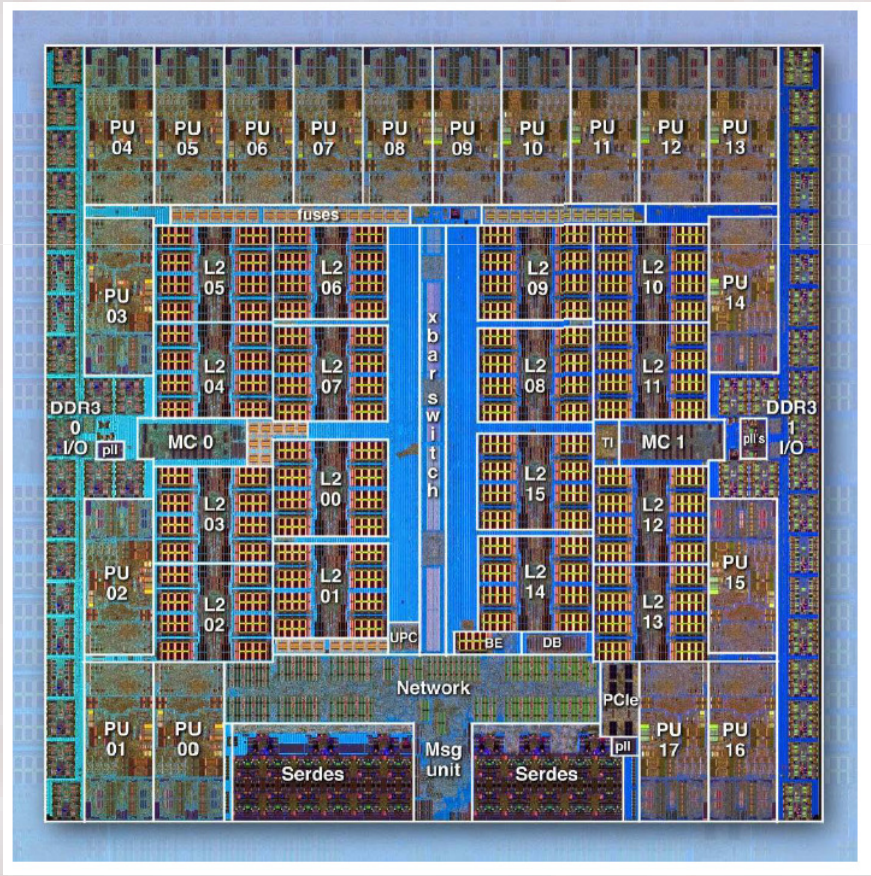


3. Procesor IBM BlueGene/Q ima hardversku transakcijsku memoriju

- Ima 18 jezgri – jedna je namijenjena za OS, jedna je rezerva
- Superračunalo Sequoia

ima 100 000 procesora,

tj.1 800 000 jezgri !



3. Intel mikroprocesori Haswell arhitekture imat će HTM

- IBM BlueGene/Q mikroprocesor je ipak prvenstveno namijenjen za izradu superračunala, pa je za "obično" programiranje značajnija **najava Intel mikroprocesora Haswell arhitekture koji će isto podržavati HTM** (prvi bi trebao izaći na tržište 2013. godine).
- Time će se **HTM uvesti u masovnu primjenu** i značajno će se olakšati pisanje konkurentnih programa.
- 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.**

3. HTM poezija

Međ' programerskim pukom

zavladala silna euforija

- uskoro stiže nam trkom

hardverska transakcijska memorija.

4. Java i konkurentno programiranje

- **Svaka Java aplikacija koristi dretve.** Kada se starta JVM, on kreira posebne dretve, npr. za GC (garbage collection), uz main dretvu.
- Kada koristimo Java AWT ili **Swing** framework, oni kreiraju posebnu dretvu za upravljanje GUI-em.
- Kada koristimo **servlete ili RMI**, oni kreiraju pričuvu (pool) dretvi. Zato, kada koristimo te frameworke, moramo biti upoznati sa konkurentnošću u Javi.
- Svaki takav framework uvodi u našu aplikaciju konkurentnost na implicitan način, te moramo znati napraviti da **mješavina našeg koda i frameworkovog koda bude sigurna u višedretvenom radu.**

4. Problemi kod nesinkronizacije – race condition

❖ Problemi nastaju kada se dvije dretve upliću jedna drugoj u posao, npr. tako da modificiraju isti objekt. To može stvoriti netočne rezultate i naziva se **race condition**:

```
class Counter {
   private volatile int value = 0;
   public int getValue() {return value;}
   public void setValue(int someValue) {
      value = someValue;
   }
   public void increment() {value++}
}
```

4. Sinkronizacija međusobnim isključivanjem (mutual exclusion)

- Pretpostavimo da neka metoda u drugoj klasi radi:

  x.setValue(0); x.increment(); int i = x.getValue();
- **Koju vrijednost ima varijabla i na kraju ovih naredbi?** Za jednodretveni program, odgovor je 1.
- U konkurentnom radu brojač može biti modificiran od drugih dretvi, tako da rezultat ovisi o ispreplitanju naredbi ove dretve sa naredbama neke druge dretve.
- Taj se problem rješava pomoću sinkronizacije koja se zove **međusobno isključivanje** (mutual exclusion).
- **Svaki objekt u Javi ima lokot (lock)**. Nasljeđuje se automatski od superklase Object. Lokot istovremeno **može držati (zaključati) samo jedna dretva**.

4. Zaključavanje lokota – naredba synchronized

- Objekt koji će služiti kao lokot može se kreirati ovako:

  Object lock = new Object();
- Dretva koja traži lokot to radi pomoću naredbe **synchronized**, koja označava početak **synchronized bloka**:

  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 zaključanim sve do kraja tog bloka**.

4. Sinkronizacija na temelju uvjeta (condition synchronization)

- Zaštita pristupa djeljivim varijablama nije jedini razlog zašto dretve moraju biti međusobno sinkronizirane.
- Često puta 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 čest u praksi (u različitim varijantama).

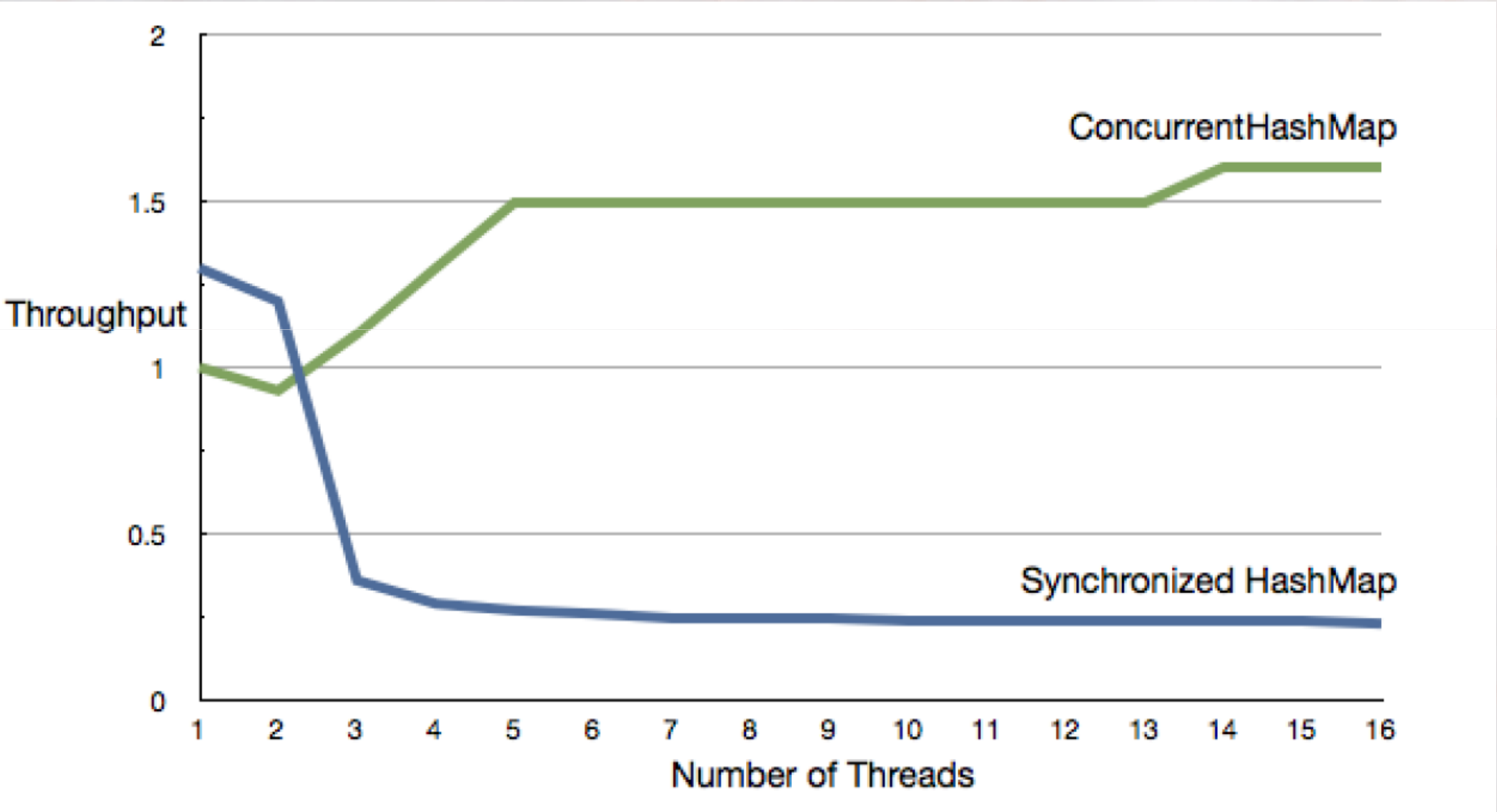
4. Sinkronizacija na temelju uvjeta - primjer

```
public void consume() throws … {
   int value;
   synchronized(buffer) {
     while (buffer.size() == 0) {buffer.wait();}
     value = buffer.get();}
}
public void produce() {
   int value = random.produceValue();
   synchronized(buffer) {
     buffer.put(value);
     buffer.notify();}
}
```

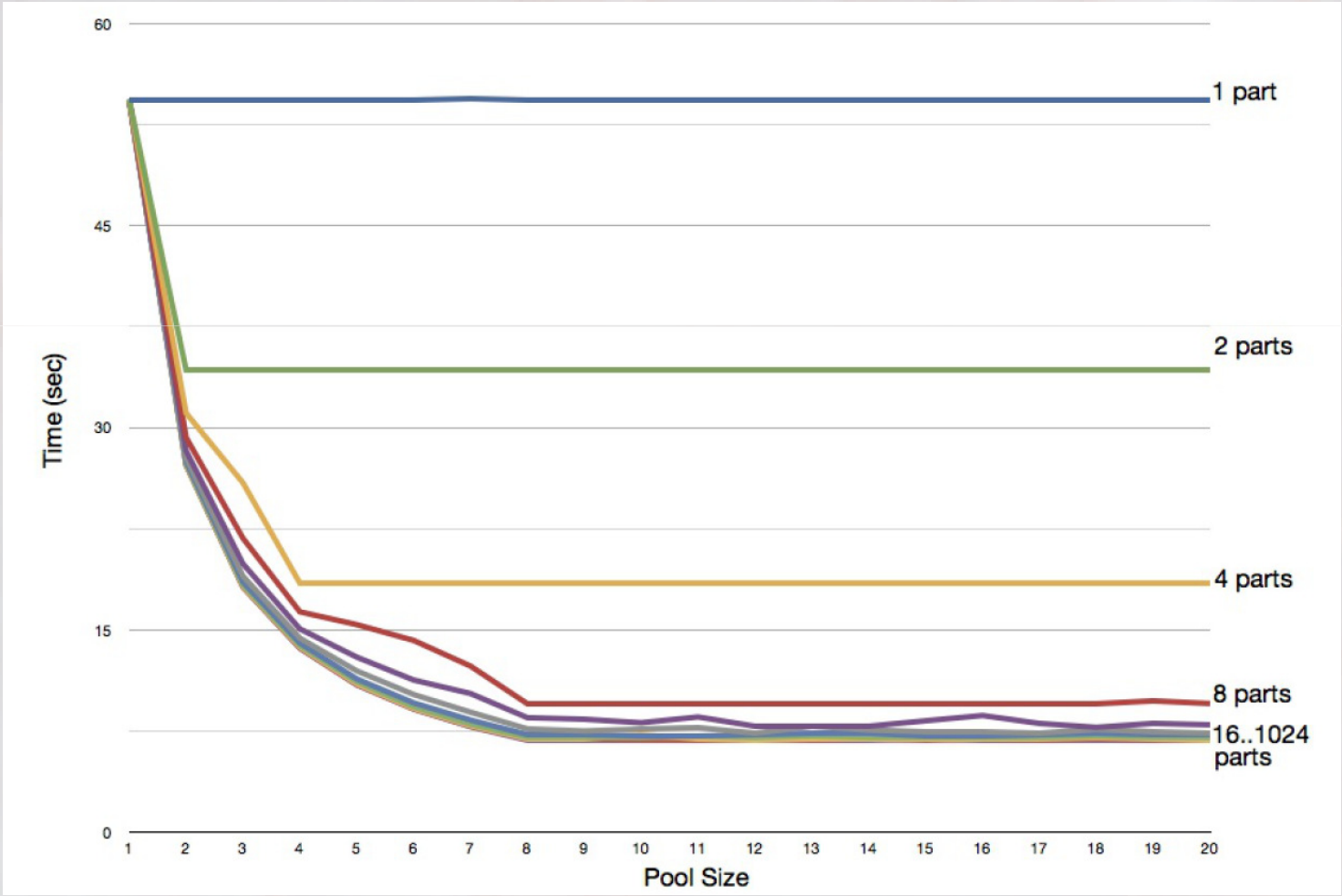
4. Poboljšanja konkurentnosti u Javi 5, 6, 7

- U Javi verzije 5 kroz novi paket **java.util.concurrent** uvedene su novosti:
  - Locks (ReentrantLock, ReadWriteLock...)
  - Conditions
  - Atomic variables
  - Executors (thread pools, scheduling)
  - Futures
  - Concurrent Collections
  - Synchronizers (Semaphores, Barriers...)
  - System enhancements.
- U Javi 6 riješeni su neki bugovi i poboljšane performanse, a u Javi 7 uveden je **Fork/Join Framework.**

4. Usporedba propusnosti ConcurrentHashMap i sinkronizirane kolekcije (8 - jezgreni procesor)



4. Pronalaženje prim brojeva na 8 - jezgrenom procesoru - analiza efekta mijenjanja broja dretvi i broja dijelova programa



5. Povijest Scale

- Programski jezik Scala kreirao je **Martin Odersky**, profesor na Ecole Polytechnique Fédérale de Lausanne (EPFL).
- Krajem 80-ih doktorirao je 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. 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).

5. Povijest Scale - nastavak

- 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.
- Odersky je 2002. počeo raditi novi jezik Scala. Tako je 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, testira se 2.10).
- Od tada, Scala se sve više koristi u praksi. Došla je 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 iz jezika Ruby u Scalu.

5. Osnovne osobine Scale

- **Scala je čisti objektno-orijentirani jezik** (sa statičkom provjerom stipova). Osim toga, na temelju objektno-orijentiranih mogućnosti izgrađene su i brojne funkcijske mogućnosti, **tako da je Scala i funkcijski jezik (ali nije čisti)**.
- Sa funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za **konkurentno programiranje**.
- **Scala je izvrstan jezik i za pisanje DSL-ova** (Domain-Specific Language), jezika za specifičnu problemsku domenu.
- 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**.

5. Mišljenja drugih kreatora programskih jezika o Scali

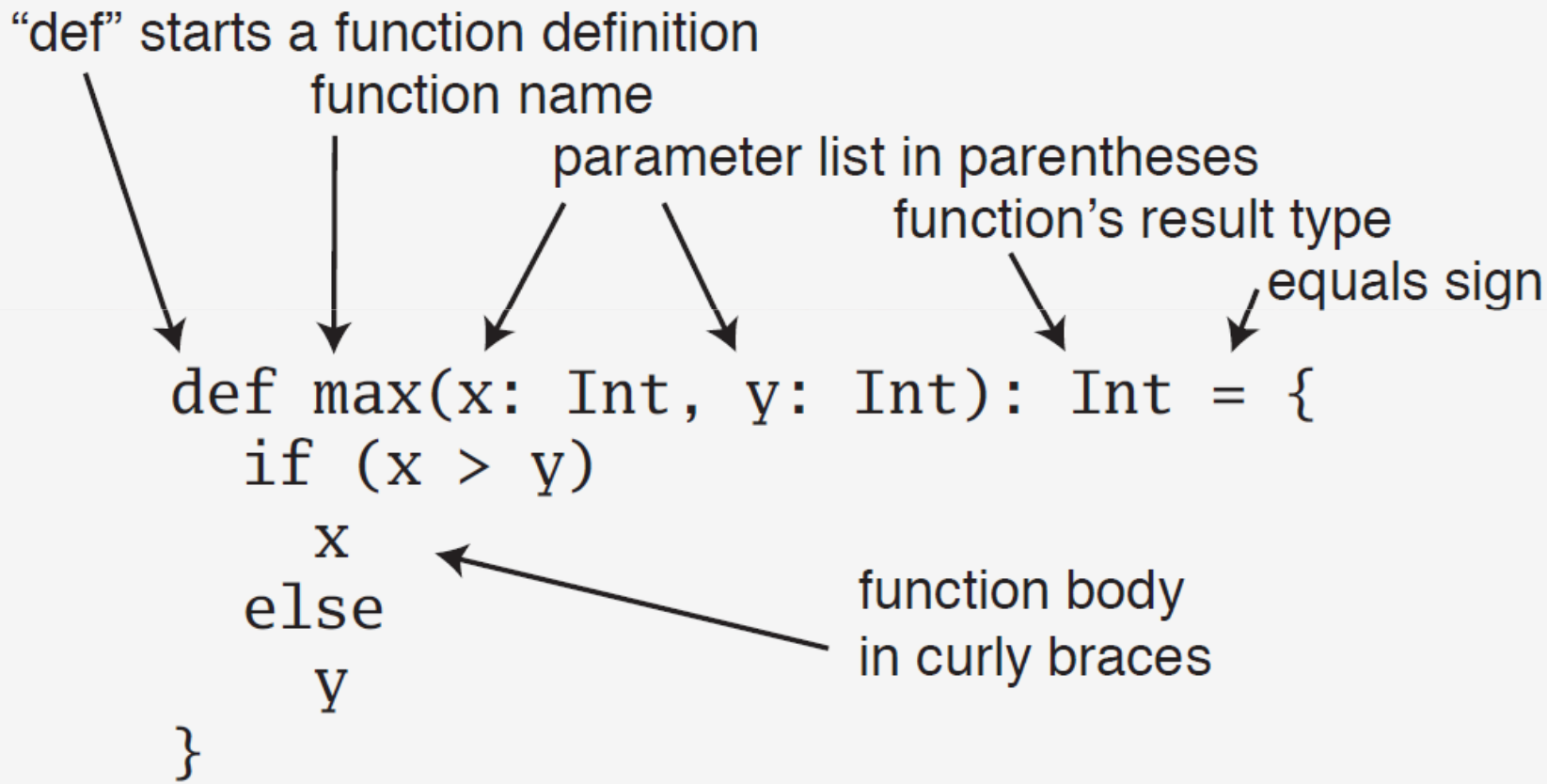
- "If I were to pick a language to use today other than Java, **it would be Scala**."
  - **James Gosling, creator of Java**

- "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.**

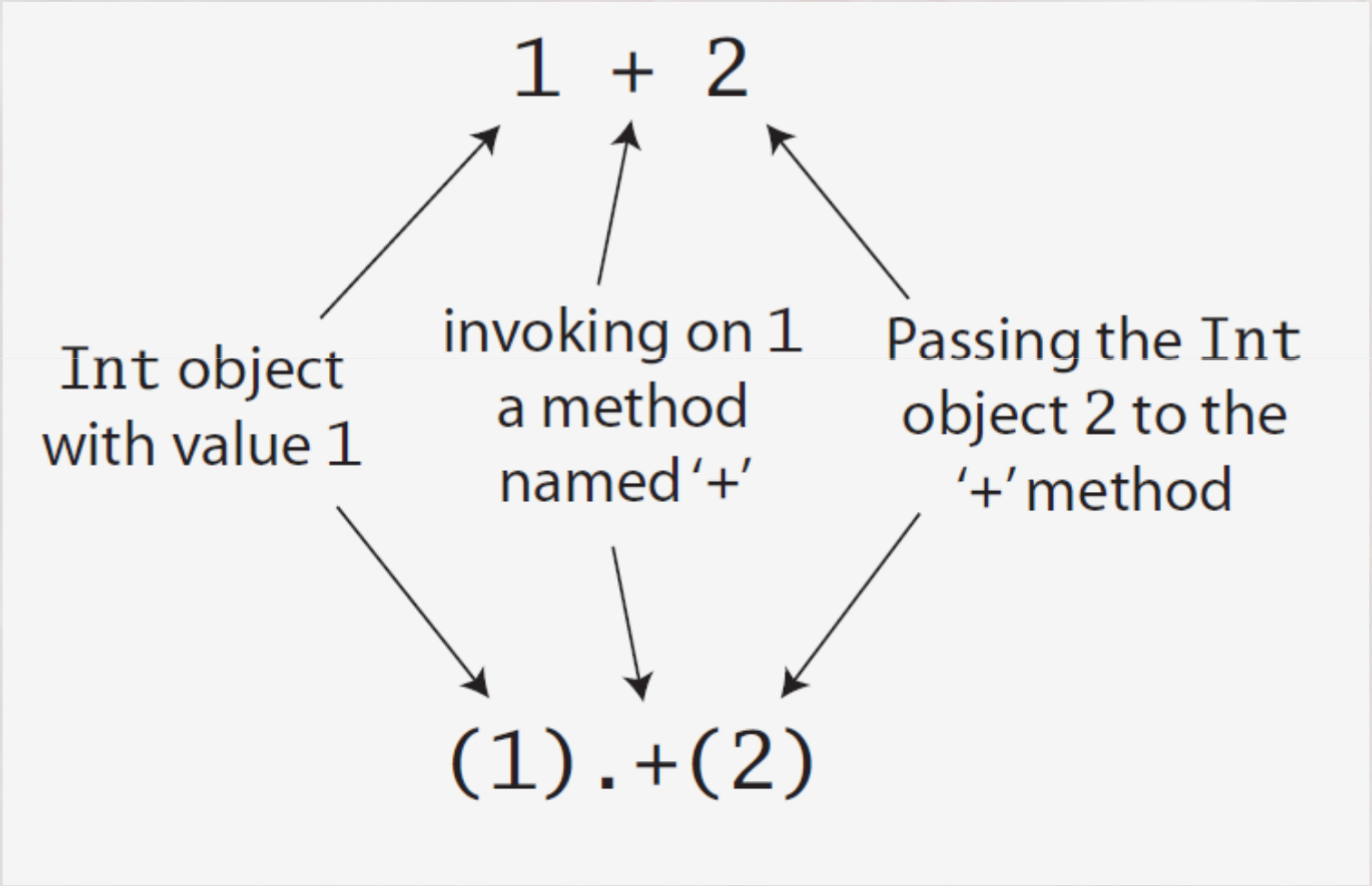
5. Različite osobine Scale

- Ne mora se pisati točka-zarez.
- Tip funkcije nije nužno uvijek navesti, jer ga Scala kompajler može izvesti iz drugih podataka. To je tzv. izvođenje tipova ili **zaključivanje o tipovima** (type inference).
- Funkcije se u Scali mogu **gnijezditi**, tj. mogu se raditi lokalne funkcije. To isto vrijedi i za pakete, i ostale elemente jezika.
- Kod nasljeđivanja i nadjačavanja metoda, mora se koristiti riječ **override.**
- 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**.

5. Definicija funkcije u Scali



5. U Scali su sve operacije metode neke klase, pa i 1 + 2

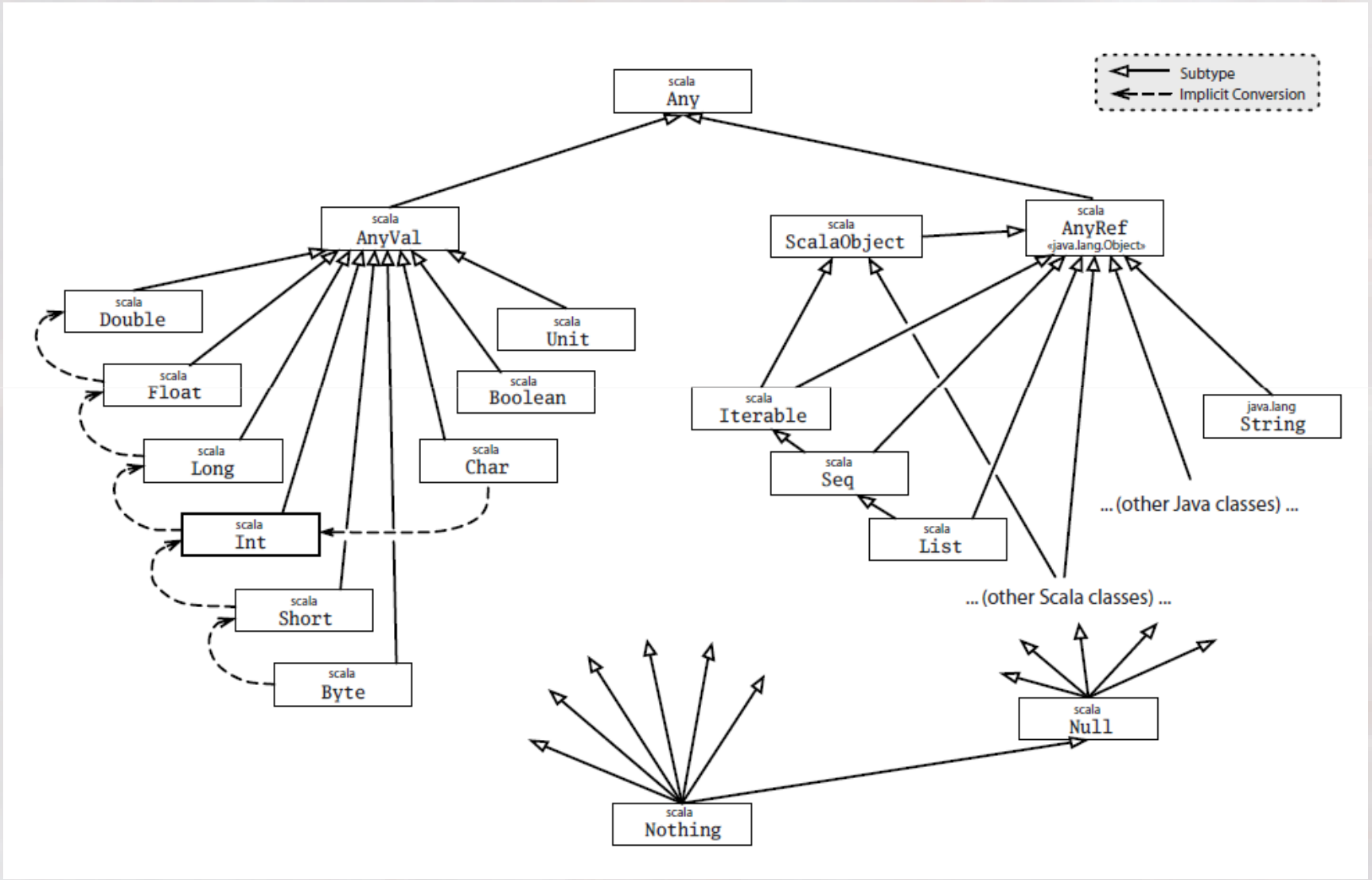


5. Operatori su u Scali metode

- Tehnički, Scala nema operatore, već metode imenovane specijalnim simbolima (kao simbol +). **Opterećenje operatora** (operator overloading), koju Java nema, a C++, Eiffel, C# imaju, rješava kao preopterećenje funkcija:

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

5. Hijerarhija Scala klasa



5. Višestruko nasljeđivanje

- 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 ga je uveo u **Eiffel od početka (1986), dok je u C++ uvedeno naknadno** (1989.), pa je vjerojatno zato izvedeno neoptimalno.
- 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).
- Tokom vremena se shvatilo da je višestruko nasljeđivanje korisno, pa je u Scali uveden programski konstrukt koji omogućava skoro isti efekt - **trait**.

5. 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**.
- Iako traitovi sliče na višestruko nasljeđivanje, 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.

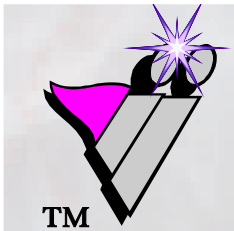
5. Primjeri korištenja traitova

❖ Primjer za prikaz **linearizacije**:

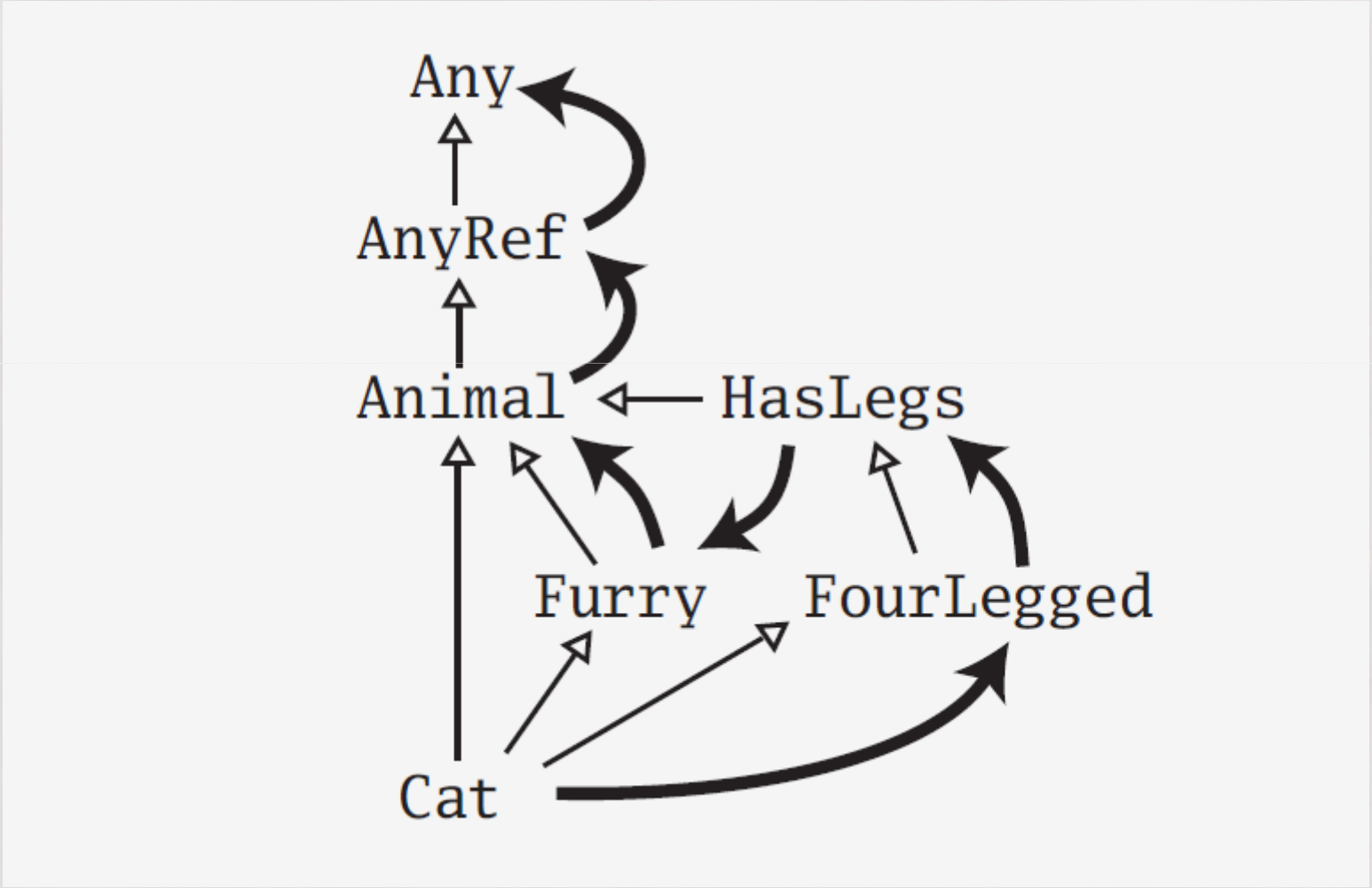
```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal
    with Furry with FourLegged
```

❖ Trait se može koristiti i selektivno na razini objekta. Npr., pretpostavimo da klasa Macka ne nasljeđuje trait Programer. **Instanca (objekt) ipak može naslijediti taj trait**:

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



5. Hijerarhija nasljeđivanja i linearizacije klase Cat



5. Generičke klase

- U Scali možemo **generički tip (npr. T) ograničiti gornjom ili donjom granicom**. Npr., ako je gornja granica je tip Pet, ograničavanje se 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.
- Slično je ograničavanje na donju granicu:

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

5. Generičke klase - nastavak

- U prethodna dva primjera ograničavali smo (pomoću gornje ili donje granice) generički tip u definiciji metode. No, može se koristiti i tzv. **anotacija varijance** (variance annotation).

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

- Sa +T označili smo da taj generički tip ima **kovarijantno (covariant) ponašanje**, odnosno da je MyList[Int] podtip od MyList[Any], u skladu s tim što je Int podtip od Any (zato se naziva kovarijantno ponašanje).
- Suprotno od kovarijantnog ponašanja je **kontravarijantno (contravariant) ponašanje**, a označava se sa –T.

6. Funkcijsko programiranje

- **Prvi funkcijski jezik Lisp** (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).
- U zadnjih nekoliko godina se **povećao interes za funkcijske jezike**. Oni 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**. Bez obzira na količinu iskustva u imperativnom programiranju, suočavanje sa funkcijskim programiranjem predstavlja izazov, jer traži promjenu načina razmišljanja.

6. Kako prepoznati imperativni stil programiranja

- Ako kod sadrži barem jednu **var varijablu (mutabilnu)**, onda je to vjerojatno imperativni stil, a ako sadrži samo **val varijable (imutabilne)**, 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. Slijedi primjer postepene transformacije 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
  }
}
```

6. Postepena transformacija imperativnog programa u funkcijski


```
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.

6. Čisti funkcijski kod nema popratni efekt

- Više funkcijski pristup je da se napravi metoda koja formatira primljene argumente, ali ih ne štampa:

```
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. 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.

6. Osobine funkcijskih jezika

- 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)

6. Osobine funkcijskih jezika - dodatak

- ❖ Neki dodaju još npr:
- Funkcije kao vrijednosti, "građani prvog reda" ("first-class value")
- Anonimne funkcije
- Currying
- Sakupljanje smeća (garbage collection)
- ❖ Funkcije su u Scali vrijednosti. Kao i svaka druga vrijednost, mogu biti pridružene nekoj varijabli, poslane kao parametri nekoj drugoj funkciji, ili vraćene kao rezultat funkcije.
- ❖ Budući da su u Scali funkcije vrijednosti, a istovremeno su u Scali sve vrijednosti objekti, **slijedi da su u Scali sve funkcije objekti**.

6. Funkcije višeg reda

- Funkcije koje kao parametar ili povratnu vrijednost imaju neku drugu funkciju, zovu se funkcije višeg reda (ovdje je to sum):

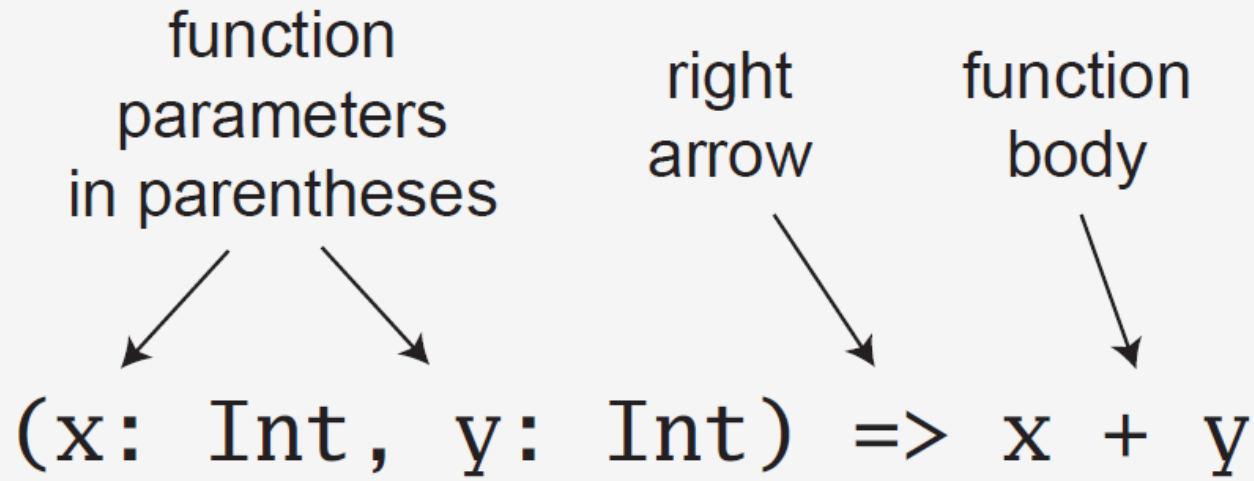
```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

- Sada definiramo dvije funkcije i koristimo ih (za punjenje vrijednosti val varijabli) kao 1. parametar funkcije sum:

```
def square(x: Int): Int = x * x
def powerOfTwo(x: Int): Int =
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)
val sumSquares = sum(square, 1, 5) // 1+4+9…=55
val sumPowersOfTwo = sum(powerOfTwo, 1, 5) //62
```

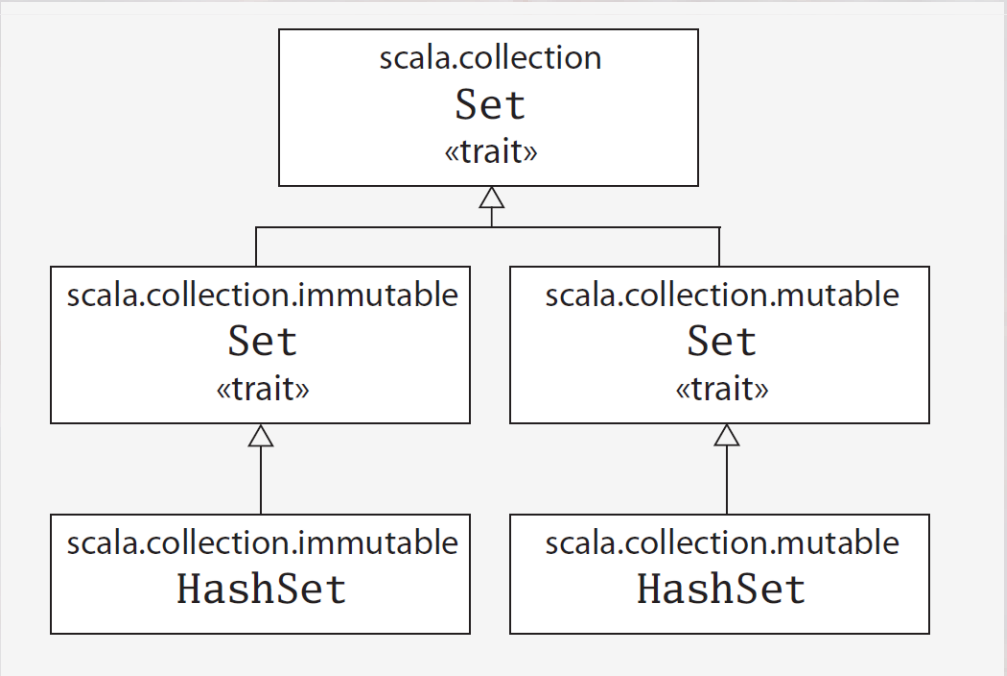
6. Funkcijski literal - anonimna funkcija je također funkcijski literal

- Često kao parametre koristimo kratke funkcije, i to jednokratno. Tada, umjesto da definiramo eksplicitne funkcije, možemo direktno kao aktualne parametre koristiti tzv. **anonimne funkcije**, koje spadaju u **funkcijske literale**:



6. Kolekcije u Scali - većinom postoje u dvije varijante, imutabilna i mutabilna

- 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).



6. Podudaranje (sparivanje) uzorka (pattern matching)

❖ Vrlo važna mogućnost u Scali, iako spada u "sintaksni šećer". Naročito se koristi kod aktora (actors), ali i za parsiranje i sl.

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

6. Lijena evaluacija (lazy evaluation)

- Inicijalizacija vrijednosti odgađa se do trenutka kada se (lijena) vrijednost prvi put koristi:

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

7. Scala i konkurentno programiranje

- Scala ima tri "stila" konkurentnog programiranja:
- uobičajeni **imperativan stil (kao Java)**
- stil temeljen na **aktorima** (actors)
- stil kod kojeg se koristi **softverska transakcijska memorija**.
- Kao i u Javi, i u Scali se svaka instanca klase AnyRef (= Java klasa Object) može koristiti kao monitor. Synchronized je Java ključna riječ, ostalo su metode Java klase Object:

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

7. Neimperativni stil – konkurentnost temeljena na slanju poruka

- Slijedi kratak prikaz neimperativnog stila, pomoću **poštanskih pretinaca (mailboxes) i aktora (actors)**. Sve se 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.

7. Poštanski pretinci

- Omogućavaju slanje i primanje poruka, a poruka je bilo koji objekt. 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. Poruke se redom preuzimaju iz pretinca pomoću **receive** (ili receiveWithin) metode, kojoj se šalje **procesor poruka f** (parcijalna funkcija). Tipično se funkcija f implementira pomoću podudaranja uzorka (pattern matching).

7. Jednostavan aktor – nasljeđuje klasu Thread i trait MailBox

- **Receive metoda** blokira dok se u pretincu ne pojavi odgovarajuća poruka, tj. **"beskonačno" čeka na poruku, a metoda receiveWithin čeka zadano vrijeme**. Zatim se poruka (ne mora biti zadnja) vadi iz pretinca i blokirana dretva se restarta primjenjujući procesor f na poruku.
- Slijedi prikaz vrlo pojednostavljene implementacije aktora:

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

7. Ne-lagani i lagani aktori

- U standardnoj Scala biblioteci aktora imamo dvije varijante aktora. U jednoj svaki ne-lagani aktor utor dobiva svoju dretvu. No **dretve su u JVM-u skupe**, zato jer svaka dretva ima svoj stog (stack), a on je prealocirane veličine.
- **Lagani aktori dijele dretve iz pričuve dretvi** (thread poll). Umjesto metoda receive i receiveWithin, u toj varijanti postoje **ekvivalentne metode react i reactWithin**.
- Dok se u prvoj varijanti može kreirati tisuće aktora, u drugoj varijanti (aktori dijele dretve iz pričuve) **može se bez problema kreirati na stotine tisuća aktora**. 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.

7. Primjer ne-laganog aktora sa receive metodom

```
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 // vraća sumu pozivatelju
}
accumulator ! 1; // pa se šalje 7, 8,"quit"
receive {case result=>println("Tot." + result)}
// Tot. 16
```

7. Softverska transakcijska memorija

- U prethodnim primjerima rad sa aktorima bio je temeljen na standardnoj Scala biblioteci. **Puno bogatiji rad sa aktorima ima framework Akka** (pisan u Scali).
- Ovdje neće biti prikazan rad sa Akka aktorima, već **rad sa Akka softverskom transakcijskom memorijom**, Akka STM. Može se istovremeno koristiti oboje, i aktori i STM.
- Scala transakcije se u Akka STM-u definiraju jednostavno. Transakcija se stavlja unutar bloka koji počinje rječju **atomic**:

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

7. Akka STM transakcije mogu se gnijezditi jedna unutar druge

```
class Account(val initialBalance: Int) { …
  def deposit(amount: Int)  = { atomic { … } }
  def withdraw(amount: Int) = { atomic { … } }
}
object AccountService {
  def transfer(from: Account, to: Account,
    amount: Int) = {
    atomic {to.deposit(amount);
            from.withdraw(amount)}
  }
  def transferAndPrintBalance …; def main …
}
```

7. Preporuke za konkurentno programiranje u Scali

- STM ima puno dobrih strana, npr:
- **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**.
- STM nije bez mana. Pogodan je za konkurentno čitanje, ali sa relativno malo mijenjanja. Kada se dvije transakcije sukobe oko mijenjanja istog objekta ili podataka, **samo jedna od njih će uspjeti, a druga će se automatski poništiti**.
- U slučaju puno kolizija kod pisanja, STM nije dobra solucija. **Tada je bolje koristiti aktore**.

Zaključak

- U zadnjih 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".
- **Scala je čisti objektno-orijentirani jezik** (sa statičkom provjerom tipova), sa nekim objektno-orijentiranim mogućnostima 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)**.
- Sa funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za **konkurentno programiranje**.

Zaključak - nastavak

- **Scala podržava tri "stila" konkurentnog programiranja**: prvi je uobičajeni imperativan stil, sličan onome u Javi, drugi stil temelji se na aktorima , a treći je korištenje softverske transakcijske memorije.
- **Scala je izvrstan jezik i za pisanje DSL-ova** (Domain-Specific Language), jezika za specifičnu problemsku domenu.
- 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 napuštanja Jave, jer se **Java i Scala programski kod mogu jako dobro upotpunjavati**.