JAVA PARALELIZACIJA II - STREAMS

Zlatko Sirotić, univ.spec.inf.
ISTRA TECH d.o.o.
Pula

Neki autorovi radovi zadnjih godina

- HrOUG 2015a: Povratak u Prolog
- HrOUG 2015b: Kada Oracle naredba nije serijabilna
- CASE 2015: Višestruko nasljeđivanje – san ili Java 8?
- JavaCro 2015: Java paralelizacija
- HrOUG 2014: Nasljeđivanje je dobro, naročito višestruko - Eiffel, C++, Scala, Java 8
- CASE 2014: Trebaju li nam distribuirane baze u vrijeme oblaka?
- JavaCro 2014: Da li postoji samo jedna "ispravna" arhitektura web poslovnih aplikacija
- HrOUG 2013: Transakcije i Oracle - baza, Forms, ADF
- CASE 2013: Što poslije Pascala? Pa … Scala!

Teme

- Korištenje Java 5/6 Executora i Java 7 ForkJoin frameworka, na jednom jednostavnom primjeru paralelnog programa (podsjećanje na materijal od prošle godine)
- Da li je višestruko nasljeđivanje (klasa) zaista loše? Kako to rade Eiffel, C++, Scala - ukratko
- Java 8 - najvažnije nove mogućnosti: lambda izrazi, default metode, Streams API
- Rješenje paralelizacije (na istom jednostavnom primjeru od prošle godine) pomoću Java 8 Streams API-a
- Potencijalni problemi s Java ForkJoin frameworkom, pa time i sa Streams API-em (kod paralelnog rada)

Java 5/6 Executor

- U Javi 5 su kroz paket java.util.concurrent uvedeni **executori** (tj. sučelja Executor, ExecutorService, Callable, Future, klase Executors, ThreadPoolExecutor, FutureTask i dr.)
- Executori, za razliku od direktnog rada s klasom Thread, pomažu da se programeri koncentriraju na kreiranje **zadataka** koji će se poslati executoru na izvršavanje, a optimizaciju izvršavanja rade executori, **koji koriste thread pool**.
- Executorima se daje zadatak koji je instanca klase (najčešće anonimne) koja implementira sučelje **Runnable** ili **Callable**.
- **Zadatak je: naći koliko ima prim (prostih) brojeva među prvih N (npr. 10 000 000) prirodnih brojeva (N zadajemo).**
- Naravno, cilj nam je izvršiti zadatak u što kraćem vremenu. Zbog toga želimo zadatak riješiti kroz paralelno programiranje, tako da maksimalno koristimo procesorske resurse.

Java 5/6 Executor

- **Pitanje je kako podijeliti zadatak na podzadatke.**
- **Također, pitanje je koliko Java dretvi kreirati?**
  Možda onoliko koliko ima podzadataka?
  To najčešće ne bi bilo dobro!
- Za (približno) određivanje broja Java dretvi može se koristiti jednostavna formula:
  broj_Java_dretvi =
  broj_HW_dretvi / (1 – koeficijent_blokiranja)
  Koeficijent blokiranja (blocking coefficient) je broj između 0 i 1 i nije ga lako odrediti. Računski intenzivni problemi imaju koeficijent koji se približava nuli, pa je tada preporučeni broj Java dretvi jednak ili nešto malo veći od broja HW dretvi.
- No, sada treba odrediti kako podijeliti problem, tj. koliko ćemo imati podzadataka. Svaki podzadatak radit će konkurentno, pa ih svakako treba biti barem onoliko koliko ima Java dretvi.

Java 5/6 Executor

- U općenitom slučaju može se primijeniti relativno jednostavna tehnika: broj podzadataka treba biti dovoljno velik da se iskoriste postojeće Java dretve, tj. **ne smije se desiti da neke Java dretve ostanu dugo neiskorištene.**
- Dakle, broj podzadataka svakako mora biti veći od broja Java 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 podzadataka dobije značajno povećanje performansi, a s daljnjim povećanjem, povećanje performansi je sve manje (performanse se mogu i smanjiti).
- Slijedi program, koji ima ove ulazne parametre:
  - **number**: gornja granica do koje se traže prim brojevi;
  - **poolSize**: broj Java dretvi;
  - **numberOfParts**: broj podzadataka.

PrimesExecutor


```
public class PrimesExecutor {
public static void main(final String[] args) {
  if (args.length != 3) {
    System.out.println("Usage: number poolSize numberOfParts");
    return; }
  final long number = Long.parseLong(args[0]);
  final int poolSize = Integer.parseInt(args[1]);
  final long numberOfParts = Long.parseLong(args[2]);
  final long numberOfPrimes;
  final long startTime = System.nanoTime();
  if (number <= 1) numberOfPrimes = 0;
  else {
    PrimesExecutor task = new PrimesExecutor();
    numberOfPrimes =
      task.countPrimes(number, poolSize, numberOfParts); }
  final long endTime = System.nanoTime();
  System.out.printf("Number of primes under %d is %d\n",
    number, numberOfPrimes);
  System.out.println("Time (seconds) taken is " +
    (endTime - startTime) / 1.0e9);
} // main
```

PrimesExecutor


```
private long countPrimes(final long number,
  final int poolSize, final long numberOfParts) {

  long count = 0;
  try {
    final List<Callable<Long>> partitions = new ArrayList<>();
    final long chunksPerPartition = number / numberOfParts;

    for(long i = 0; i < numberOfParts; i++) {
      final long lower = (i * chunksPerPartition) + 1;
      final long upper = lower + chunksPerPartition - 1;
      // (pod)zadatak je instanca anonimne klase tipa Callable
      partitions.add(new Callable<Long>() {
        public Long call() {
          return countPrimesInRange(lower, upper);
        }
      });
    }
...
```

PrimesExecutor


```
...
    final ExecutorService executorPool =
      Executors.newFixedThreadPool(poolSize);

    final List<Future<Long>> resultFromParts =
      executorPool.invokeAll
        (partitions, 10000, TimeUnit.SECONDS);

    executorPool.shutdown();

    for(final Future<Long> result : resultFromParts) {
      count += result.get();
    }
  } catch(Exception ex) {
    throw new RuntimeException(ex);
  }

  return count;
} // countPrimes
```

PrimesExecutor

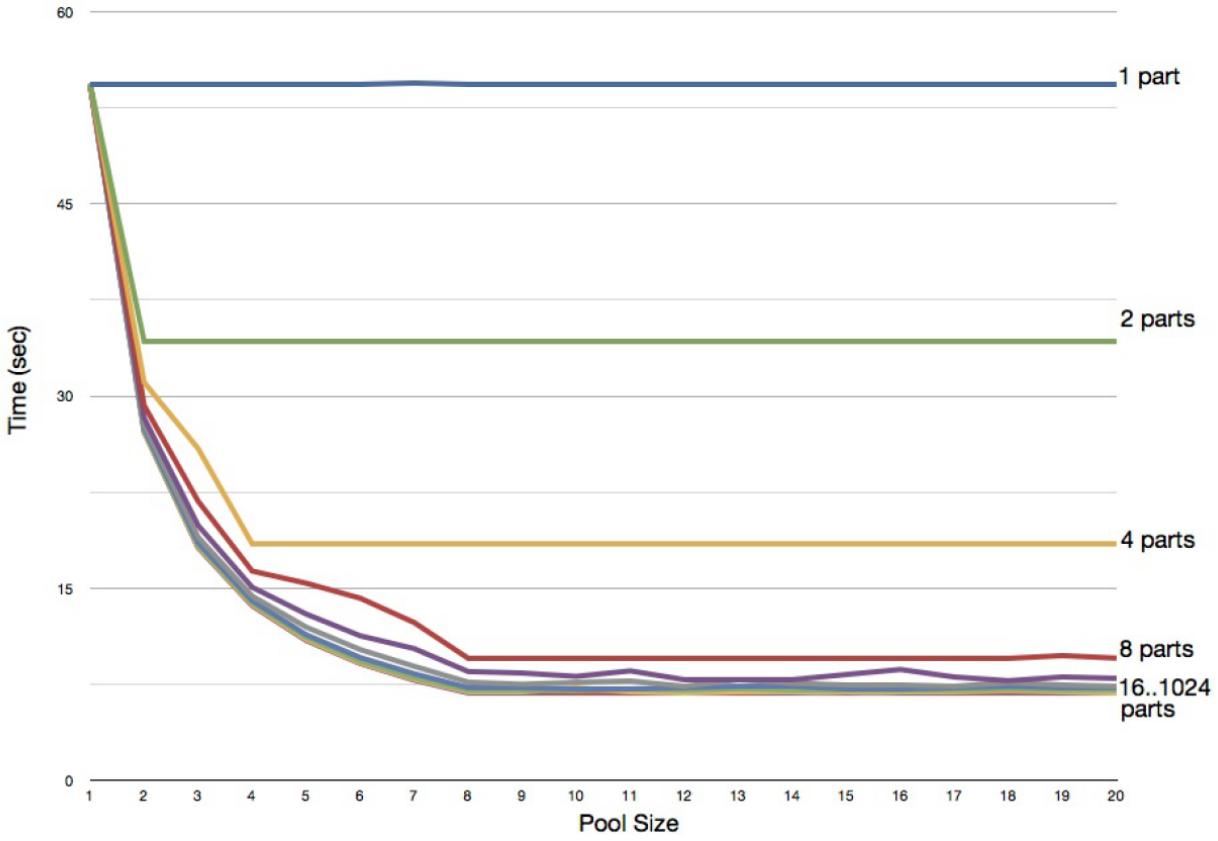

```
private static long countPrimesInRange(long lower, long upper){
  long total = 0;
  // Provjeravaju se i parni brojevi, zbog usporedbe sa Streams
  for(long i = lower; i <= upper; i++) {
    if (isPrime(i)) total++;
  }
  // 1 nije prim, ali 2 je prim (a za 2 isPrime daje false),
  // pa je ukupan broj točan.
  return total;
}

private static boolean isPrime(final long number) {
  if (number % 2 == 0) return false;
  for(long i = 3; i * i <= number; i = i + 2) {
    if (number % i == 0) return false;
  }
  return true;
}

} // class PrimesExecutor
```

Java 5/6 Executor

- Pronalaženje prim brojeva na µP sa 8 HW dretvi (4 jezgre * 2) - prikaz efekta mijenjanja broja Java dretvi (PoolSize, os x) i broja podzadataka (različite krivulje):



Java 7 ForkJoin framework

- "Divide and Conquer" ("Divide et impera", "Podijeli pa vladaj" ili "Zavadi pa vladaj" – stara rimska poslovica).
- Implementaciju ExecutorService sučelja u slučaju ForkJoin frameworka radi klasa **ForkJoinPool**.
- Tipično se ForkJoinPool instanci šalje samo jedan zadatak (task), a onda ForkJoinPool instanca i zadatak zajedno primjenjuju tehniku Divide and Conquer.
- Broj Java dretvi u poolu se može zadati eksplicitno ili implicitno:
  -- **broj Java dretvi se zadaje eksplicitno** (u ovom slučaju 8)
  ForkJoinPool fjPool = new ForkJoinPool(8);

  -- **ili implicitno** (na računalu sa 8 HW dretvi, bit će ih 8)
  -- framework koristi metodu **Runtime.availableProcessors()**
  ForkJoinPool fjPool = new ForkJoinPool();

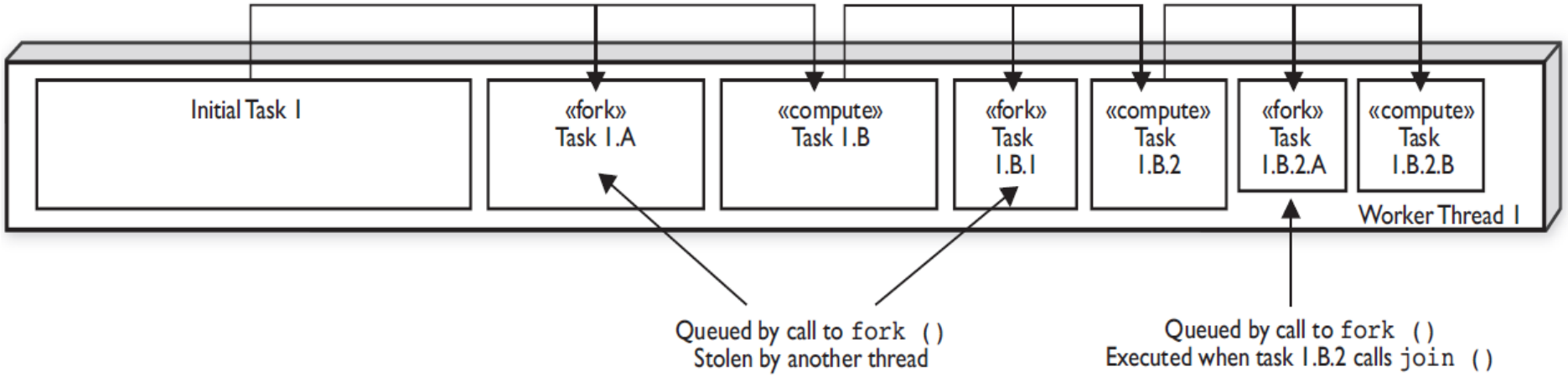
Java 7 ForkJoin framework

- Zadatak (task) treba biti instanca podklase apstraktne klase **ForkJoinTask**, preciznije podklasa apstraktne podklase klase ForkJoinTask, a najčešće su to **RecursiveTask** (u primjeru) ili **RecursiveAction**.
- ForkJoinTask ima puno metoda, ali najvažnije su: **compute(), fork(), join()**
- **Pseudo kod za compute:**

```
if (podzadatakJeDovoljnoMali()) {
  rijesiPodzadatak();
} else {
  MojForkJoinTask lijevaPolovica = ...
  MojForkJoinTask desnaPolovica = ...
  lijevaPolovica.fork();  -- stavi u queue
  desnaPolovica.compute();-- radi(rekurzivno)
  lijevaPolovica.join();  -- čekaj završetak
}
```

Java 7 ForkJoin framework

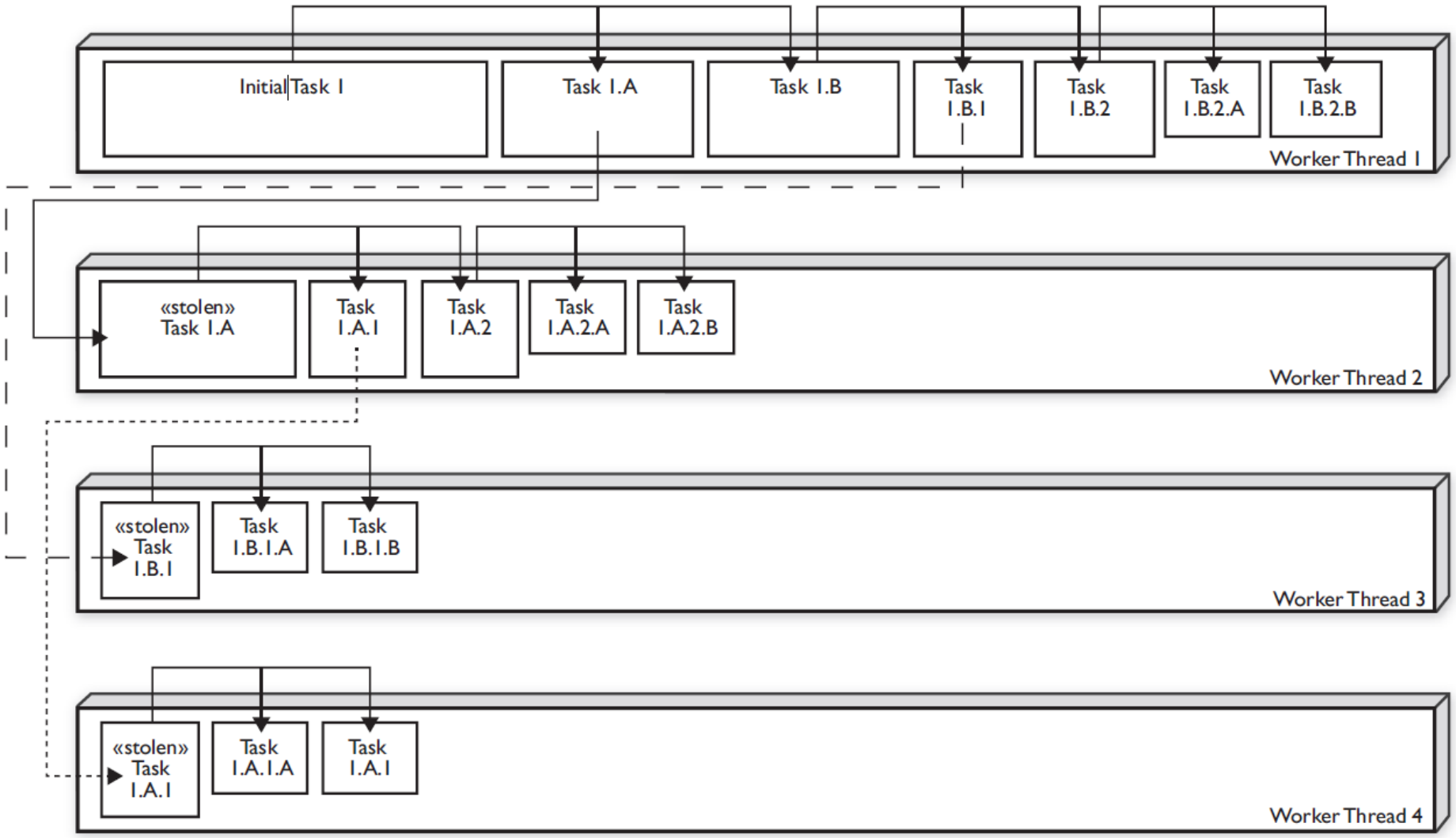
- Za razliku od većine ExecutorService implementacija, kod ForkJoin frameworka **svaka Java dretva u ForkJoinPool-u ima svoj red (queue)** podzadataka na kojima radi.
- Metoda fork() stavlja ForkJoinTask u red tekuće Java dretve.
- Inicijalno je zauzeta samo jedna Java dretva - kada joj pošaljemo (cijeli) zadatak. Dretva tada počinje dijeliti zadatak u dva podzadatka, pa prvi podzadatak (lijevi) stavi u red, a drugi podzadatak (desni) pokuša izvršiti (i tako rekurzivno):



Java 7 ForkJoin framework

- Ključna značajka ForkJoin frameworka je **Work Stealing** (krađa posla). **Java dretve kradu posao (podzadatak) drugoj Java dretvi iz njenog reda podzadataka, i stavljaju ga u svoj red** (nešto što većina nas ne bi nikad napravila ☺).
- Vrlo je važan redoslijed stavljanja podzadataka u red. **Podzadaci koji se prvi stavljaju u red trebaju predstavljati veći dio posla.** Npr. na početku imamo jedan zadatak, koji pokriva 100% posla. Njega dijelimo u dva podzadatka, svaki po (otprilike) 50% posla. Prvi stavljamo u red, a drugi obrađujemo, pri čemu drugi opet dijelimo u polovice, itd.
- One Java dretve koje nemaju posla (tj. njihov red je prazan), kradu posao drugim Java dretvama, i to tako da uzmu posao (podzadatak) **koji je najstariji u redu, a to je istovremeno i najveći posao** iz reda druge Java dretve.
- Sljedeća slika pokazuje ta događanja, s 4 Java dretve.

Java 7 ForkJoin framework



Java 7 ForkJoin framework

- Dijeljenje na podzadatke se, za razliku od primjera sa običnim executorima, radi implicitno – ne zadaje se broj podzadataka, **već se određuje kada je podzadatak dovoljno mali**.
- Nažalost, ne postoji opća metoda kojom se ispravno određuje veličina tog malog podzadatka – **to se radi eksperimentalno**.
- Podzadatak na kojem se poziva metoda join() može tada biti već gotov, jer ga je možda ukrala druga Java dretva, i već napravila. Ili može biti ukraden, ali ga druga dretva upravo radi, pa tada treba čekati da završi. Treći je slučaj da podzadatak nije ukraden i tada ga radi tekuća dretva.
- Poziv join() metode u compute() metodi treba biti jedna od zadnjih radnji, iza poziva fork() metode i rekurzivnog poziva compute() metode. Budući da je to jako važno, **postoji metoda invokeAll(a2, a1); koja zamjenjuje niz a1.fork(); a2.compute(); a1.join();**

PrimesForkJoin

```
public class PrimesForkJoin extends RecursiveTask<Long> {

private static int threshold;
private long start;
private long end;
private PrimesForkJoin(final long theStart, final long theEnd)
  { start = theStart; end = theEnd; }

public static void main(final String[] args) {
  if (args.length < 1 || args.length > 3) {
    System.out.println("Usage: number poolSize threshold
      OR  number poolSize  OR  number");
    return;
  }
  final long number = Long.parseLong(args[0]);
  final long numberOfPrimes;
  final long startTime = System.nanoTime();
...
```

PrimesForkJoin

```
… if (number <= 1) numberOfPrimes = 0;
  else {
    PrimesForkJoin task = new PrimesForkJoin(1, number);
    ForkJoinPool fjPool;
    if (args.length == 2 || args.length == 3) {
      fjPool = new ForkJoinPool(Integer.parseInt(args[1]));
    } else {
      fjPool = new ForkJoinPool();
    }
    if (args.length == 3){threshold=Integer.parseInt(args[2]);}
    else {threshold = 100;}
    numberOfPrimes = fjPool.invoke(task);
  }
  final long endTime = System.nanoTime();
  System.out.printf("Number of primes under %d is %d\n",
    number, numberOfPrimes);
  System.out.println("Time (seconds) taken is " +
    (endTime - startTime) / 1.0e9);
} // main
```

PrimesForkJoin

```
protected Long compute() {
  if (end - start <= threshold) {
    return countPrimesInRange(start, end);
  } else {
    long halfWay = ((end - start) / 2) + start;
    PrimesForkJoin t1 = new PrimesForkJoin(start, halfWay);
    PrimesForkJoin t2 = new PrimesForkJoin(halfWay + 1, end);
    t1.fork();
    long count2 = t2.compute();
    long count1 = t1.join();
    return count1 + count2;
  }
}

private static long countPrimesInRange ... // kao prije
private static boolean isPrime ... // kao prije

} // class PrimesForkJoin
```

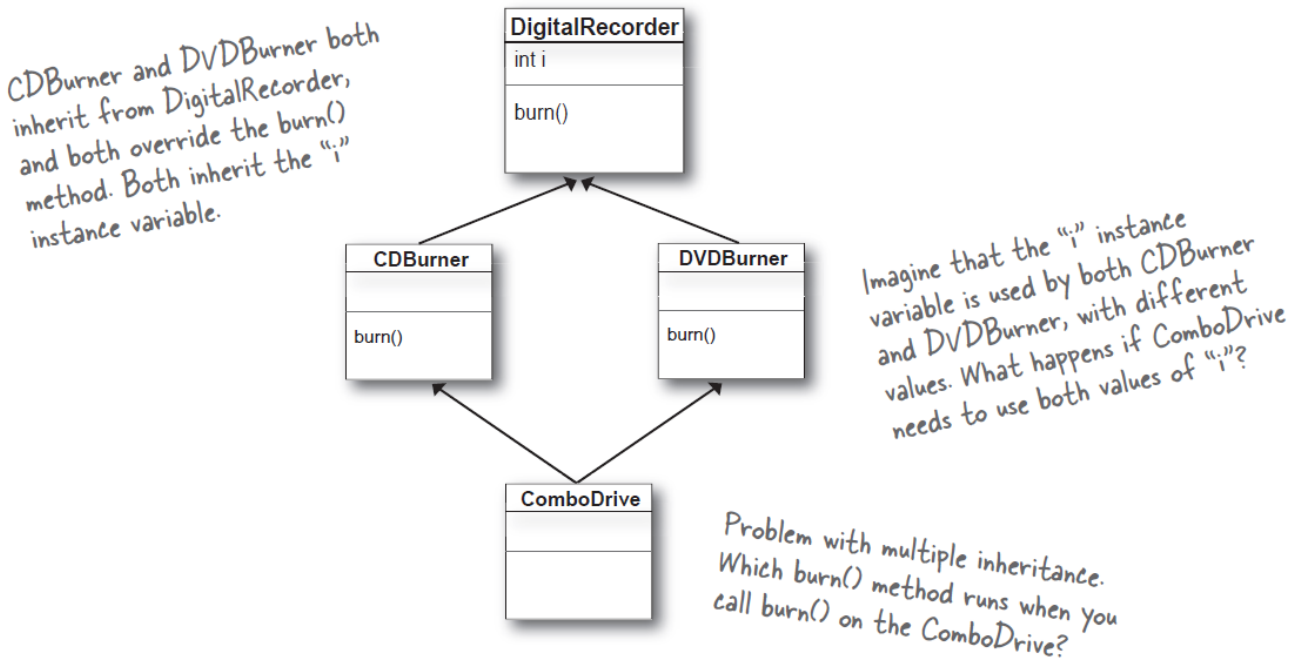
Višestruko nasljeđivanje je loše? ("Head First Java", Sierra K., Bates B., 2005.)

It's called "multiple inheritance" and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

Deadly Diamond of Death



Bertrand Meyer (autor Eiffela) u "Object Oriented Software Construction" (1997.)

- Multiple inheritance is indispensable when you want to state explicitly that a certain class possesses some properties beyond the basic abstraction that it represents. Consider for example a mechanism that makes object structures persistent (storable on long-term storage)

  …

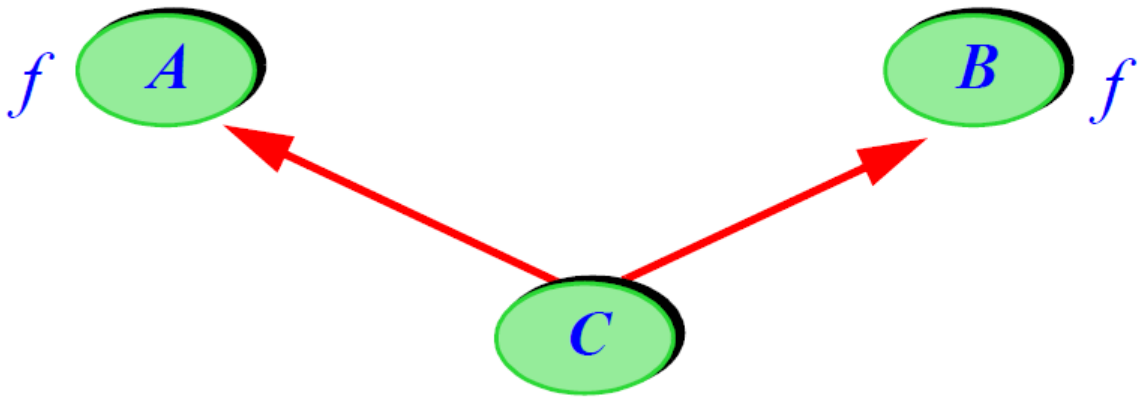
  The discussion of inheritance methodology will define it as inheritance of the structural kind. Without multiple inheritance, there would be no way to specify that a certain abstraction must possess two structural properties — numeric and storable, comparable and hashable.

  Selecting one of them as the parent would be like having to choose between your father and your mother.

Eiffel
- kratka povijest

- Eiffel je 1985. godine dizajnirao (a 1986. je napravljen prvi compiler) **Bertrand Meyer**, jedan od autoriteta na području OOPL-a.
- Eiffel je od početka je podržavao **višestruko nasljeđivanje, generičke klase, obradu iznimaka, garbage collection i metodu Design by Contract (DBC).**
- Kasnije su mu dodani **agenti, nasljeđivanje implementacije** (uz nasljeđivanje tipa) i metoda za konkurentno programiranje Simple Concurrent Object-Oriented Programming (**SCOOP**).
- U široj je javnosti daleko manje poznat nego C++ i Java, ali ga mnogi smatraju danas najboljim OOPL jezikom. Eiffel je od 2005. godine ECMA standardiziran, a od 2006. ISO standardiziran.

Eiffel primjer nasljeđivanja dvije klase koje imaju metodu istog imena; koristi se rename



Name clash

```
class C inherit
   A
      rename f as first_f end
   B
feature
   ...
end
```

C++
- kratka povijest

- C (autor je Dennis Ritchie), također Algol-ov potomak, nastao je 1970. kao jezik za sistemsko programiranje operativnog sustava UNIX. U isto vrijeme nastao je i Pascal, isto potomak Algol-a. Za većinu kasnijih programskih jezika možemo reći da (barem po sintaksi) pripadaju C ili Pascal "struji".
- Nadograđujući C sa objektno orijentiranim mogućnostima (uz zadržavanje kompatibilnosti), Bjarne Stroustrup je 1983. godine napravio C++ (1986. godine je objavio knjigu "The C++ Programming Language").
- Tokom vremena je C++ dobivao neke vrlo značajne mogućnosti, **koje na početku nije imao: višestruko nasljeđivanje, generičke klase (predloške), obradu iznimaka** (exceptions) i dr. 1997. godine donesen je ISO standard. U standardu C++11 uvedene su npr. i lambda funkcije. Najnoviji je C++14 (koji donosi manja poboljšanja).

C++ podržava višestruko nasljeđivanje

- Mehanizam višestrukog nasljeđivanja u C++ nije tako fleksibilan kao Eiffel mehanizam.
- Između ostalog, C++ nema preimenovanje (rename) metoda. Ako se od dvije klase naslijede dvije funkcije istog imena, mora se koristiti **scope resolution operator** za rješavanje dvosmislenosti:

```
class A {void f() {…}};
class B {void f() {…}};
class C : public A, public B {…};
void test()
{
    C* p = new C();
    // p->f(); This call would be ambiguous - invalid;
    p->A::f();
    p->B::f();
}
```

Scala
- kratka povijest

- 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).
- Kada je izašla Java, Odersky i Phil 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).
- 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**.

Scala i višestruko nasljeđivanje - Scala 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 klase, razlikuju se u dvije važne stvari. Prvo, trait nema parametre klasa.
- Drugo, kod višestrukog nasljeđivanja klasa, 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).

Primjeri korištenja Scala traitova

- Složeni primjer:

```
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
```

Java 8 - najvažnije nove mogućnosti: lambda, default metode, Streams API

- Na temelju onoga što čitamo i čujemo, mogli bismo zaključiti da je najvažnija nova mogućnost u Javi 8 **lambda izraz** (ili kraće, **lambda**).
- Inače, lambda izraz je (u Javi) naziv za metodu bez imena. U pravilu je ta metoda funkcija, a ne procedura. Zato možemo reći i da **lambda izraz je anonimna funkcija**, koja se može javiti kao parametar (ili povratna vrijednost) druge funkcije (koja je, onda, funkcija višeg reda).
- U Javi 8 pojavile su se i tzv. **default metode** u Java sučeljima (interfaces). One, zapravo, predstavljaju uvođenje **višestrukog nasljeđivanja implementacije** u Javu.
- Međutim, lambda izrazi i default metode su, na neki način, posljedica uvođenja treće važne mogućnosti u Javi 8, a to je **Streams API**, koji nadograđuju dosadašnje Java kolekcije.

Lambda račun

- Lambda izrazi (ili lambda funkcije), imaju teoretsko porijeklo u **lambda računu** (lambda calculus), kojega je sredinom 30-ih godina prošlog stoljeća kreirao **Alonzo Church**.
- Poznata je **Church-Turingova hipoteza** (inače, **Alen 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 se izračunati pomoću lambda računa ili **Turingovog stroja**.
- 50-ih godina se formalno dokazalo da postoje i neki drugi sustavi koji su njima ekvivalentni: Gödelove rekurzivne funkcije, Postov sustav, Markovljevi algoritmi i dr.
- 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, od kojih je najstariji **Lisp** (nastao 1959.).

Java 8 lambda izrazi

- Java 8 lambda (izrazi) temelje se na tzv. **funkcijskim sučeljima** (functional interface), koji su postojali od početka.
- Funkcijska sučelja su ona sučelja koja imaju točno jednu (jednu i samo jednu) apstraktnu funkciju. No, od Jave 8 funkcijska sučelja mogu imati i statičke metode i default metode (koje nisu postojale prije Jave 8).
- Npr. kad Java kompajler naiđe na ovakvu naredbu (lambda izraz je desno od znaka jednakosti; ovo je samo jedna od brojnih varijanti pisanja lambda izraza):

```
StringToIntMapper mapper = (String str) -> str.length();
```

kompajler provjerava da li postoji odgovarajuće sučelje StringToIntMapper, koje ima samo jednu apstraktnu funkciju.

Java 8 lambda izrazi

- Pretpostavimo da postoji takvo sučelje:

```
// anotacija @FunctionalInterface uvedena je u Javi 8
@FunctionalInterface
interface StringToIntMapper {
  int map(String str);
}
```

- Kompajler tada napravi nešto što ima jednak efekt kao sljedeća anonimna klasa:

```
StringToIntMapper mapper = new StringToIntMapper() {
  @Override
  public int map(String str) {
    return str.length();
  }
};
```

Java 8 default metode

- Kako je već rečeno, default metode su u Javi 8 trebale prvenstveno zbog uvođenja Stream API-a, iako su default metode korisne i za ostale svrhe (ne samo tvorcima API-a).
- Kod uvođenja Streamsa, bilo je potrebno nadograđivati brojna postojeća Java sučelja, tj. dodavati im nove metode. Međutim, **kad dodajemo nove metode u sučelje, moramo mijenjati sve klase** koje ga (direktno ili indirektno) nasljeđuju, što može značiti izmjenu milijuna redaka programskog koda.
- Default metode su riješile taj problem. **Sučelje sada može imati i implementaciju metode**, a ne samo deklaraciju.
- Java 8 sučelje sa default metodom **nije tako moćno kao Scala trait**. Scala trait može imati i ne-default metode, može imati stanje (atribute), može se komponirati za vrijeme runtimea, može pristupati instanci klase koja ga nasljeđuje. Ništa od toga Java 8 sučelje (sa default metodama) ne može.

Java 8 default metode

```
// 1. ako bismo postojeće sučelje Iterable
// htjeli proširiti sa novom metodom forEach,
// morali bismo mijenjati svaku klasu
// koja (direktno ili indirektno) nasljeđuje to sučelje
public interface Iterable<T> {
  public Iterator<T> iterator();
  public void forEach(Consumer<? super T> consumer);
}

// 2. default metode rješavaju taj problem
public interface Iterable<T> {
  public r<T> iterator();
  public default void forEach(Consumer<? super T> consumer) {
    for (T t : this) {
      consumer.accept(t);
    }
  }
}
```

Java 8 Streams (java.util.stream) - zašto su potrebni?

- Pojava **masivno višejezgrenih procesora** traži bolji i lakši način izrade paralelnih programa od dosadašnjih načina. Jedan (dobar) način je da se koristi funkcijsko programiranje, a naročito paralelne kolekcije.
- Java nema paralelne kolekcije (collections), ali su zato uvedeni Streamsi, kako bi se postojeće kolekcije "zaogrnule" u (paralelne) Streamse i mogle (indirektno) paralelizirati.
- **Kako bi se olakšalo korištenje Streamsa, bilo je potrebno uvesti i lambda izraze i default metode**. Međutim, lambda izrazi i default metode mogu biti korisni i za ostale svrhe, ne samo tvorcima API-a, već i "običnim" programerima.
- Za razliku od dosadašnjih kolekcija, koje sadrže sve svoje elemente u memoriji, Streamse možemo shvatiti kao **"vremenske kolekcije"**, čiji se elementi (kojih teoretski može biti i beskonačan broj) stvaraju po potrebi.
- **Paralelne verzije Streamsa temelje se na ForkJoin.**

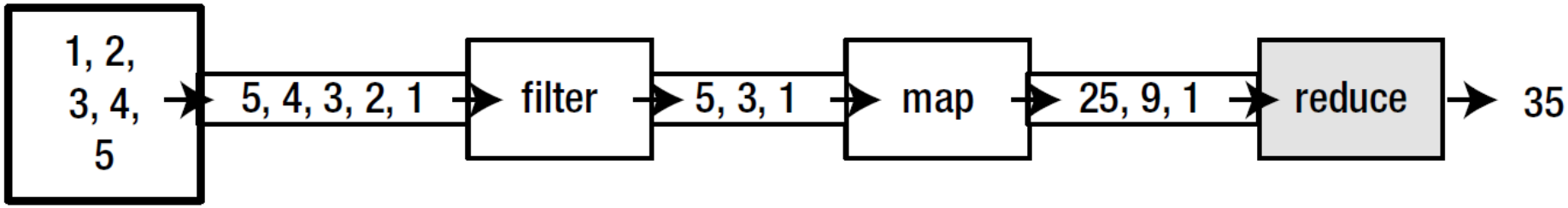
Java 8 Streams programiranje je deklarativno (što, ne kako), kao SQL

```
// 1. "klasična" obrada "klasične" kolekcije - for petlja
private static void checkBalance(List<Account> accList) {
  for (Account a : accList)
    if (a.balance() < a.threshold) a.alert();
}
// 2. primjena lambda izraza za obradu "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
// 3. primjena Streamsa za paralelizaciju "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.parallelStream().forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
```

Java 8 Streams programiranje je deklarativno (što, ne kako), kao SQL

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);
```



numbers.stream( ).filter(n -> n % 2 == 1).map(n -> n * n).reduce(0, Integer::sum)

PrimesStream


```
public class PrimesStream {
public static void main(final String[] args) {
  if (args.length != 2) {
    System.out.println("Usage: 1/0 (parallel/not parallel)");
    return;
  }
  final long number = Long.parseLong(args[0]);
  final boolean isParallel =
    (Integer.parseInt(args[1]) == 1) ? true : false;
  final long numberOfPrimes;
  final long startTime = System.nanoTime();
  if (number <= 1)
    numberOfPrimes = 0;
  else
    numberOfPrimes = countPrimes(number, isParallel);
  final long endTime = System.nanoTime();
  System.out.printf("Number of primes under %d is %d\n",
    number, numberOfPrimes);
  System.out.println("Time (seconds) taken is " +
    (endTime - startTime) / 1.0e9);
} // main
```

PrimesStream


```
protected static long countPrimes
   (final long number, final boolean isParallel) {
   long count;
   if (isParallel) {
     count = LongStream.rangeClosed(1L, number)
                       .parallel()
                       .filter(n -> isPrime(n)) // lambdaExp
                       .count();
   } else {
     count = LongStream.rangeClosed(1L, number)
                       .filter(PrimesStream::isPrime)//methodRef
                       .count();
   }
   // 1 nije prim, ali 2 je prim (a za 2 isPrime daje false),
   // pa je ukupan broj točan.
   return (count);
}

private static boolean isPrime ... // kao prije
} // class PrimesStream
```

Usporedba: Java 5/6 Executor, Java 7 ForkJoin, Java 8 Streams (i7-4702MQ 2.2 GHz; Java 1.8.0_74)


```
PrimesExecutor 10000000  1 1000  Time (seconds) taken is 8.69
PrimesExecutor 10000000  2 1000  Time (seconds) taken is 4.57
PrimesExecutor 10000000  4 1000  Time (seconds) taken is 2.51
PrimesExecutor 10000000  8 1000  Time (seconds) taken is 1.76
PrimesExecutor 10000000 16 1000  Time (seconds) taken is 1.72

PrimesForkJoin 10000000  1 100   Time (seconds) taken is 8.89
PrimesForkJoin 10000000  2 100   Time (seconds) taken is 4.64
PrimesForkJoin 10000000  4 100   Time (seconds) taken is 2.59
PrimesForkJoin 10000000  8 100   Time (seconds) taken is 1.85
PrimesForkJoin 10000000 16 100   Time (seconds) taken is 1.79

PrimesStream   10000000 0        Time (seconds) taken is 8.92
PrimesStream   10000000 1        Time (seconds) taken is 1.92

... Number of primes under 10000000 is 664579
```

Jedan kritički pogled: A Java Fork/Join Blunder (Ed Harned)

❖ There are four major faults with the F/J framework:

1. The use of Deques/Submission queues

Deques/Submission-Queues are a feature primarily for

1. Applications that run on clusters of computers (Cilk for one.)
2. Operating systems that balance the load between CPU's.
3. A number of other environments irrelevant to this discussion

2. The use of an intermediate join()

The framework design comes from MIT's Cilk (now the property of Intel). Cilk uses a container for the applications…
JavaSE is open, no closed container. There is no way a JavaSE application can do a context-switch, or a pseudo context-switch, or anything similar. The intermediate join() in the F/J framework doesn't work because it requires a context-switch…

Jedan kritički pogled: A Java Fork/Join Blunder (Ed Harned)

3. The use of academic research standards instead of application development standards

The Fork/Join pool constructs without specifying the maximum number of threads... Since there is no guarantee the new compensation thread won't block, the framework can flood the system with hundreds/thousands of compensation threads until the entire box stalls or the JVM runs out of memory… While no timing/cancelling may be acceptable in an academic setting, it is unsuitable for professional, commercial application development. Furthermore, since there is no notification to Anyone About Anything, no one is aware of the stalling app…

4. The use of the CountedCompleter class

… Using the CountedCompleter class to emulate scatter-gather is a debacle…

Zaključak

- U zadnjih (otprilike) desetak godina sve se više govori o **multiparadigmatskom programiranju**. Pritom postoje barem dva pristupa – jedan je da koristimo više programskih jezika (objektno-orijentirani, funkcijski, logički ...), a drugi pristup kaže da je puno jednostavnije i bolje imati jedan programski jezik koji podržava različite paradigme.
- Neki OO jezici imaju podršku za funkcijsko programiranje već duže vrijeme (npr. Eiffel već 20-ak godina, C# i Scala više od 10, C++ oko 5). Java 8 priključila se tom trendu.
- OO 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.
- S funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za konkurentno i paralelno programiranje.

Literatura (dio)

- Boyarsky, J., Selikoff, S. (2015): OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809, Sybex
- Brinch Hansen, P. (1998): Java insecure Parallelism, Syracuse University, http://brinch-hansen.net/papers/1999b.pdf (svibanj 2016.)
- Göetz B. i ostali (2006): Java Concurrency in Practice, Addison-Wesley
- Harned, E. (2015): A Java Fork/Join Blunder, http://coopsoft.com/dl/Blunder.pdf (svibanj 2016.)
- Herlihy, M., Shavit, N. (2008): The Art of Multiprocessor Programming, Elsevier / Morgan Kaufmann Publishers
- Meyer, B. (1997): Object-Oriented Software Construction, Prentice Hall
- Meyer, B. (2009): Touch of Class - Learning to Program Well with Objects and Contracts, Springer
- Odersky, M., Spoon, L., Venners, B. (2010): Programming in Scala (2.izdanje), Artima Press
- Prokopec A. (2014): Learning Concurrent Programming in Scala, Packt Publishing
- Sharan, K (2014): Beginning Java 8 Language Features - Lambda Expressions, Inner Classes, Threads, I/O, Collections and Streams, Apress
- Sierra K, Bates B. (2015): OCA/OCP Java SE 7 Programmer I & II Study Guide, McGraw-Hill Education
- Subramaniam, V. (2011): Programming Concurrency on the JVM – Mastering Synchronization, STM, and Actors, The Pragmatic Bookshelf, Texas / Raleigh