POGLAVLJE 2

Izrada i uklanjanje objekata

U ovom poglavlju ćemo se baviti izradom i uklanjanjem objekata: kada i kako treba napraviti objekte, a kada i kako to treba izbeći, kako se postarati da objekti budu pravovremeno uklonjeni i kako izvesti završne akcije koje prethode uklanjanju objekata.

Savet 1: Razmotrite upotrebu statičkih proizvođačkih metoda umesto konstruktora

Uobičajeno je da klasa omogućava dobijanje instance preko javnog konstruktora. Postoji još jedna, manje poznata, tehnika koju takođe treba da nauče svi programeri. Klasa može sadržati javnu *statičku proizvođačku metodu* (engl. *static factory method*), koja je zapravo statička metoda čiji je rezultat instanca te klase. Evo jednostavnog primera iz klase `Boolean` (klasa koja omotava prosti tip `boolean`). Ova statička proizvođačka metoda, dodata u verziji 1.4, prevodi prostu vrednost tipa `boolean` u referencu na objekat tipa `Boolean`:

```
public static Boolean valueOf(boolean b) {
    return (b ? Boolean.TRUE : Boolean.FALSE);
}
```

Klasa može klijentima ponuditi statičke proizvođačke metode umesto konstruktora, ili uz njih. Upotreba statičkih metoda umesto konstruktora ima i prednosti i mana.

Jedna od prednosti statičkih proizvođačkih metoda jeste to što, za razliku od konstruktora, imaju imena. Ako parametri konstruktora sami po sebi ne opisuju rezultujući objekat, statička proizvođačka metoda sa podesno odabranim imenom čini korišćenje klase jednostavnijim i klijentski program postaje čitljiviji. Na primer, konstruktor `BigInteger(int, int, Random)`, čiji je rezultat tipa `BigInteger` najverovatnije prost broj, podesnije bi bio izražen u vidu statičke metode pod imenom `BigInteger.probablePrime`. (Ova statička proizvođačka metoda dodata je u verziji 1.4 .)

Klasa može sadržati samo jedan konstruktor sa određenim potpisom. Programeri su zaobilazili to ograničenje tako što su pravili dva konstruktora čije su se liste parametara razlikovale samo u redosledu tipova parametara. To nije dobro. Korisnik takvog

API-ja nikad neće moći da razlikuje konstruktore, pa će najverovatnije slučajno pozvati pogrešan. Programski kôd s takvim konstruktorima teško je razumeti bez dokumentacije o klasama.

Pošto statičke proizvođačke metode imaju imena, za njih ne važe ograničenja koja se primenjuju na konstruktore. Kada se čini da klasa zahteva više konstruktora sa istim potpisom, razmislite o tome da jedan ili više konstruktora zamenite statičkim proizvođačkim metodama, čija pažljivo odabrana imena ukazuju na razlike među njima.

Druga prednost statičkih proizvođačkih metoda je to što, za razliku od konstruktora, one ne prave nove objekte svaki put kada ih pozovete. Na ovaj način nepromenljive klase (13. savet) mogu koristiti već napravljene instance ili čuvati instance nakon pravljenja tako da ih mogu ponovo koristiti, čime se sprečava nepotrebno dupliranje objekata. Metoda `Boolean.valueOf(boolean)` prikazuje ovu tehniku: ona nikad ne stvara objekat. Ova tehnika može znatno poboljšati performanse u slučajevima kada se ekvivalentni objekti često pozivaju, a naročito ako izrada tih objekata zahteva mnogo računarskih resursa.

Mogućnost statičke proizvođačke metode da kao odgovor na ponovljeno pozivanje daje isti objekat, takođe se može upotrebiti za strogo kontrolisanje instanci. Ovo je potrebno iz dva razloga. Kao prvo, bićete sigurni da je klasa singularna (2. savet). Drugo, osiguravate da ne postoje dve jednake instance: `a.equals(b)` ako i samo ako `a==b`. Ako klasa ovo garantuje, onda njeni klijenti mogu da upotrebe operator `==` umesto metode `equals(Object)`, što rezultuje značajnim poboljšanjem performansi. Obrazac *typesafe enum*, opisan u 21. savetu, primenjuje takvu optimizaciju, a metoda `String.intern` primenjuje je u ograničenom obliku.

Treća prednost statičkih proizvođačkih metoda je to što, nasuprot konstruktorima, njihov rezultat može biti objekat bilo kog podtipa traženog rezultata. To omogućava fleksibilan izbor klase objekta rezultata.

Jedna od primena ove pogodnosti je korišćenje zaštićenih klasa kao rezultata funkcija iz API-ja. Kada se klase sakriju na taj način, API postaje izuzetno kompaktan. Ta tehnika se koristi i u kosturima sistema (engl. *framework*) zasnovanim na interfejsima, u kojima interfejsi određuju tip rezultata statičkih proizvođačkih metoda.

Na primer, Collections Framework sadrži dvadeset pogodnih realizacija interfejsa kolekcija, uključujući nepromenljive kolekcije, sinhronizovane kolekcije i sl. Većini tih klasa pristupa se pomoću statičkih proizvođačkih metoda u jednoj klasi koja se ne može instancirati (`java.util.Collections`). Nijedna klasa rezultata nije javna.

Collections Framework API je mnogo manji nego što bi bio da postoji dvadeset zasebnih javnih klasa za pogodne realizacije. Nije smanjena samo fizička veličina API-ja već i „konceptualna težina“. Korisnik zna relevantan interfejs rezultata, te nije neophodno da čita dodatnu dokumentaciju klasa. Takva upotreba statičke proizvođačke metode podrazumeva da klijent pristupa rezultatu na osnovu njegovog interfejsa umesto preko njegove klase što je, uopšteno govoreći, dobro u praksi (34. savet).

Ne samo da klasa rezultata javne statičke proizvođačke metode nije javna, već se klasa može razlikovati od poziva do poziva u zavisnosti od vrednosti parametara statičke metode. Bilo koji podtip deklarisane klase rezultata dolazi u obzir. Klasa rezultata takođe može varirati od izdanja do izdanja, što olakšava održavanje softvera.

Klasa rezultata statičke proizvođačke metode ne mora čak ni da postoji u trenutku pisanja klase koja sadrži tu statičku metodu. Takve fleksibilne statičke proizvođačke metode predstavljaju osnovu *kostura sistema za pružanje usluga* (engl. *service provider framework*), kao što je biblioteka Java Cryptography Extension (JCE). Korisnicima takvog sistema dostupno je nekoliko realizacija usluga. Postoji mehanizam za *registrovanje* ovih realizacija koji ih čini dostupnim. Klijenti strukture koriste API a da pri tome ne brinu o tome koju realizaciju koriste.

U JCE-u, administrator sistema registruje realizaciju u opštepoznatoj konfiguracionoj datoteci, dodajući ključ koji odgovara imenu klase. Klijenti koriste statičku proizvođačku metodu čiji parametar je ključ. Statička proizvođačka metoda traži objekat `Class` u mapi inicijalizovanoj u konfiguracionoj datoteci i pravi instancu klase pomoću metode `Class.newInstance`. Sledeći primer praktično prikazuje ovu tehniku:

```
// Skica okvirnog rešenja
public abstract class Foo {
    // Mapa koja povezuje ključeve i klase
    private static Map implementations = null;

    // Inicijalizacija mape realizacije posle prvog poziva
    private static synchronized void initMapIfNecessary() {
        if (implementations == null) {
            implementations = new HashMap();

            // Učitavanje imena klasa realizacije i ključeva iz
            // konfiguracione datoteke, prevođenje imena u objekte tipa
            // Class pomoću Class.forName i snimanje mape.
            ...
        }
    }

    public static Foo getInstance(String key) {
        initMapIfNecessary();
        Class c = (Class) implementations.get(key);
        if (c == null)
            return new DefaultFoo();

        try {
            return (Foo) c.newInstance();
        } catch (Exception e) {
            return new DefaultFoo();
        }
    }
}
```

Osnovna mana statičkih proizvođačkih metoda je to što je nemoguće naslediti klasu koja nema javne ili zaštićene konstruktore. Isto važi i za klase koje nisu javne a koriste se u rezultatima javnih statičkih proizvođača. Na primer, nije moguće naslediti nijednu klasu realizacije iz biblioteke Collections Framework. To ograničenje je, u suštini, korisno, jer primorava programere da upotrebljavaju kompoziciju umesto nasleđivanja (savet 2).

Drugi nedostatak statičkih proizvođačkih metoda je to što se one teško razlikuju od ostalih statičkih metoda. One nisu uočljive u dokumentaciji API-ja kao što su to konstruktori. Uz to, statička proizvođačka metoda predstavlja izuzetak od standardnih normi. Zato može biti teško saznati na osnovu dokumentacije kako instancirati klasu koja sadrži statičke proizvođačke metode umesto konstruktora. Ovaj nedostatak se može ublažiti standardizovanjem pravila imenovanja. Ova pravila i dalje evoluiraju, ali dva imena statičkih proizvođačkih metoda postaju uobičajena:

- `valueOf` – Rezultat je instanca koja, uopšteno govoreći, ima istu vrednost kao i njeni parametri. Statičke proizvođačke metode sa ovim imenom efikasni su operatori za konverziju tipova.
- `getInstance` – Rezultat je instanca koja je opisana parametrima ali nema istu vrednost kao parametri. U slučaju singularnih klasa, rezultat je jedinstvena instanca. Ovo ime se obično koristi u strukturama davalaca usluga.

Na kraju možemo reći da i statičke proizvođačke metode i javni konstruktori imaju svoje primene, i da se isplati poznavati njihove prednosti. Potrudite se da izbegnete impuls da automatski koristite konstruktore a da pre toga niste promislili da li je bolje upotrebiti statičke proizvođačke metode, što nije redak slučaj. Ako ste razmotrili upotrebu i jedne i druge opcije, a niste sigurni šta je bolje, verovatno je bolje upotrebiti konstruktor iz jednostavnog razloga što je to norma.

Savet 2: Nametnite korišćenje singularne vrednosti pomoću privatnog konstruktora

Singularna klasa (engl. *singleton*) jeste klasa koja se instancira tačno jednom [Gamma98, str. 127]. Singularne klase obično predstavljaju neku jedinstvenu sistemsku komponentu, poput monitora ili sistema datoteka.

Postoje dva načina da se primeni singularnost. Oba se zasnivaju na ideji da konstruktor ostane privatan i da se napravi javni statički član koji klijentu omogućava pristup jedinoj instanci klase. U jednom od pristupa, javni statički član je konstantno polje:

```
// Singularno svojstvo pomoću konstantnog polja
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    ... // Ostatak izostavljen
}
```

Privatni konstruktor se poziva samo jednom, kako bi se inicijalizovala javna statička konstanta `Elvis.INSTANCE`. Izostavljanje javnih ili zaštićenih konstruktora *garantuje* „monoelvisni“ univerzum: kada se inicijalizuje klasa `Elvis` postojaće tačno jedna instanca klase `Elvis` – ni manje, ni više. Klijent ne može da utiče na to.

Drugi pristup je primena javne statičke proizvođačke metode umesto statičkog konstantnog polja:

```
// Singularno svojstvo pomoću statičkog proizvođača
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    public static Elvis getInstance() {
        return INSTANCE;
    }

    ... // Ostatak izostavljen
}
```

Rezultat statičke metode `Elvis.getInstance` uvek je ista referenca, i nijedna druga instanca tipa `Elvis` neće biti stvorena.

Osnovna prednost prvog pristupa je to što deklaracije članova koji sačinjavaju klasu jasno ukazuju na to da je klasa singularna: javno statičko polje je konstantno, tako da će uvek sadržati istu referencu objekta. Prvi pristup može biti bolji po performansama, ali bi dobra virtuelna mašina za Javu to mogla eliminisati ugrađivanjem poziva statičke proizvođačke metode u drugom pristupu direktno u kôd (engl. *inline*).

Osnovna prednost drugog pristupa je fleksibilnije odlučivanje da li klasa treba da bude singularna a da ne morate menjati API. Rezultat statičke proizvođačke metode singularne klase jeste jedinstvena instanca klase, ali se jednostavno može promeniti da rezultat bude, recimo, jedinstvena instanca za svaki poziv metode.

Razumno bi dakle bilo da se prvi pristup koristi kada ste apsolutno sigurni da će klasa zauvek ostati singularna. Drugi pristup koristite ako želite da odložite odluku po tom pitanju za kasnije.

Da biste singularnu klasu učinili serijabilnom (poglavlje 10), nije dovoljno samo dodati `implements Serializable` u njenu deklaraciju. Da biste se osigurali po pitanju singularnosti, morate dodati i metodu `readResolve` (57. savet). U protivnom, prilikom svake deserijalizacije serijalizovane instance biće napravljena nova instanca, dovodeći – u primeru koji smo mi naveli – do neosnovanih pojavljivanja instanci `Elvis`. Da biste to sprečili, klasi `Elvis` dodajte sledeću metodu `readResolve`:

```
// Metoda readResolve čuva svojstvo singularnosti
private Object readResolve() throws ObjectStreamException {
    /*
     * Vraća jednog pravog Elvisa i prepušta sakupljaču smeća
     * da se pobrine za Elvisovog imitatora.
     */
    return INSTANCE;
}
```

Ovaj savet i 21. savet, koji opisuje obrazac *typesafe enum*, imaju sličnosti. U oba slučaja, privatni konstruktori se koriste u kombinaciji sa javnim statičkim članovima kako bi se sprečilo pravljenje nove instance relevantne klase pošto se ona inicijalizuje. U ovom savetu, stvara se samo jedna instanca klase; u 21. savetu, po jedna instanca se stvara za svakog člana numerisanog tipa. U sledećem savetu (3. savet), na taj način uvodimo još veće ograničenje: izostavljanje javnog konstruktora osigurava da se nikad ne stvori nijedna instanca klase.

Savet 3: Sprečavanje instanciranja pomoću privatnog konstruktora

Povremeno ćete poželeti da napišete klasu koja samo predstavlja grupisane statičke metode i statička polja. Takve klase su na lošem glasu jer su povremeno zloupotrebljene u svrhe pisanja proceduralnih programa u objektno orijentisanim jezicima, iako one imaju i dosta korisnih primena. One se takođe mogu upotrebiti za grupisanje srodnih metoda u radu sa prostim vrednostima ili nizovima, poput `java.lang.Math` ili `java.util.Arrays`, ili za grupisanje statičkih metoda za rad sa objektima koji primenjuju određeni interfejs, poput `java.util.Collections`. Mogu se koristiti i za grupisanje metoda za rad sa završnim klasama, u cilju proširenja klase.

Takve *pomoćne klase* (engl. *utility class*) nisu predviđene da budu instancirane: njihova instanca ne bi imala smisla. Ipak, u nedostatku eksplicitnih konstruktora, prevodilac pravi javni, *podrazumevani konstruktor* (engl. *default constructor*) bez parametara. Korisnik ne može da razlikuje ovakav konstruktor od bilo kog drugog. Nije neuobičajeno da u API-jima nađete klase koje ne bi trebalo da instancirate, ali je to ipak moguće.

Instanciranje nećete sprečiti tako što proglasite klasu apstraktnom. Klasa se može naslediti a potklasa može biti instancirana. Uz to, korisnik može pogrešno protumačiti da je klasa predviđena za nasleđivanje (15. savet). Uprkos svemu, postoji jednostavna tehnika kojom se sigurno sprečava instanciranje. Podrazumevani konstruktor se stvara samo ako klasa ne sadrži nijedan eksplicitan konstruktor, pa je moguće sprečiti instanciranje **klase tako što ćete napraviti eksplicitan privatni konstruktor**:

```
// Pomoćna klasa koja se ne može instancirati
public class UtilityClass {

    // Potiskivanje podrazumevanog konstruktora
    private UtilityClass() {
        // Ovaj konstruktor nikad neće biti pozvan
    }
    ...  // Ostatak izostavljen
}
```

Pošto je eksplicitan konstruktor privatan, njemu se ne može pristupiti izvan klase. Stoga smo osigurali da klasa nikad neće biti instancirana, izuzev ako konstruktor bude pozvan unutar klase. Ta tehnika nije baš intuitivna, jer se konstruktor namerno navodi da ne bi mogao biti pozvan. Stoga je preporučljivo da u komentaru opišete svrhu konstruktora.

Sporedan efekat je da tu klasu ne možete naslediti. Svi konstruktori moraju pozvati konstruktor nadređene klase kojem se može pristupiti, eksplicitno ili implicitno, a potklasi nije na raspolaganju nijedan takav konstruktor.

Savet 4: Izbegavanje duplikata objekata

Često je zgodno iznova koristiti jedan objekat umesto da se pravi objekat iste funkcionalnosti kad god se za tim ukaže potreba. Ponovna upotreba postojećeg objekta je i brža i stilski bolja. Objekat uvek može biti iznova korišćen ako je *nepromenljiv* (13. savet).

Ekstreman primer nečega što ne treba da radite ilustrovan je sledećom naredbom:

```
String s = new String("besmisleno"); // OVO NEMOJTE DA RADITE!
```

Naredba stvara novu instancu tipa `String` pri svakom izvršavanju, a nijedna od izrada tih objekata nije neophodna. Argument koji se predaje konstruktoru tipa `String` (`"besmisleno"`) instanca je klase `String`, što je po funkcionalnosti ekvivalentno svakom objektu koji konstruktor stvara. Ako se prethodna naredba nalazi unutar petlje ili unutar često pozivane metode, milioni instanci tipa `String` mogu biti stvoreni nepotrebno.

Unapređena verzija se svodi na sledeće:

```
String s = ("Nije besmisleno");
```

Ova verzija koristi jedinstvenu instancu tipa `String`, umesto da pravi novu po svakom izvršavanju. Uz to, garantovano je da će objekat biti ponovo korišćen u svim programima u istoj virtuelnoj mašini koji sadrže isti znakovni niz [JLS, 3.10.5].

Dupliranje objekata često možete izbeći upotrebom *statičkih proizvođačkih metoda* (1. savet) umesto konstruktora u nepromenljivim klasama, koje imaju i konstruktore i statičke proizvođače. Na primer, statičku proizvođačku metodu `Boolean.valueOf(String)` gotovo je uvek bolje koristiti nego konstruktor `Boolean(String)`. Konstruktor stvara nov objekat svaki put kad je pozvan, dok statička proizvođačka metoda to nikad ne čini.

Pored ponovne upotrebe nepromenljivih objekata, možete ponovno koristiti i promenljive objekte za koje ste sigurni da neće biti promenjeni. Evo jednog od finijih i uobičajenijih primera onoga što ne treba da radite. Primer se odnosi na promenljive objekte koji se nikad ne menjaju pošto se njihove vrednosti izračunaju:

```
public class Person {
    private final Date birthDate;
    // Ostala polja su izostavljena

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }
    // OVO NEMOJTE DA RADITE!
    public boolean isBabyBoomer() {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
```

```
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
               birthDate.compareTo(boomEnd)   <  0;
    }
}
```

Metoda `isBabyBoomer` nepotrebno stvara nove instance `Calendar`, `TimeZone`, i dve instance `Date` svaki put kad je pozvana. U verziji koja sledi, ovakva neefikasnost izbegava se pomoću statičkog inicijatora:

```
class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }


    /**
     * Određivanje perioda datuma.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
               birthDate.compareTo(BOOM_END)   <  0;
    }
}
```

Unapređena verzija klase `Person` stvara instance `Calendar`, `TimeZone`, i `Date` samo jednom, po inicijalizaciji, umesto da ih stvara pri svakom pozivu metode `isBabyBoomer`. Tako se znatno poboljšavaju performanse ako se metoda poziva često. Na mom računaru, milion poziva originalne metode traje 36 sekundi, dok za unapređenu verziju treba samo 370 ms, što je sto puta brže. Ne samo da su performanse poboljšane, već je i upotreba jasnija. Promena lokalnih promenljivih `boomStart` i `boomEnd` iz u konstantna statička polja čini da se ovi datumi koriste kao konstante, usled čega je

kôd značajno jasniji. Uštede zbog upotrebe ovakve optimizacije neće uvek biti tako značajne, pošto se prilikom stvaranja instance klase `Calendar` zauzima mnogo resursa.

Ako se metoda `isBabyBoomer` nikad ne pozove, unapređena verzija klase `Person` nepotrebno će inicijalizovati polja `BOOM_START` i `BOOM_END`. Nepotrebne inicijalizacije je moguće izbeći *lenjom inicijalizacijom* ovih polja (48. savet) prvi put kada se pozove metoda `isBabyBoomer`, ali to nije preporučljivo. Kao što je čest slučaj sa lenjim inicijalizacijama, realizacija se komplikuje bez znatnog poboljšanja performansi. U svim prethodnim primerima u ovom savetu, bilo je očigledno da se objekti mogu iznova koristiti jer su nepromenljivi. Postoje i druge situacije u kojima to nije toliko očigledno. Razmotrite slučaj *adaptera* [Gamma98, str. 139], poznatih i kao *prikazi* (engl. *views*). Adapter je objekat koji prenosi pozive drugom objektu stvarajući alternativni interfejs. Kako adapter ne može biti u drugačijem stanju od objekta na koji ukazuje, nema potrebe za izradom više od jedne instance adaptera za dati objekat.

Na primer, rezultat metode `keySet` interfejsa `Map` je tipa `Set` objekta tipa `Map`, koji se sastoji od ključeva svih objekata u mapi. Na prvi pogled, čini se da svaki poziv `keySet` mora da stvara novu instancu tipa `Set`, ali rezultat poziva `keySet` u datom objektu tipa `Map` uvek je isti. Iako je rezultujuća instanca tipa `Set` obično promenljiva, svi vraćeni objekti su po funkciji identični: kada se jedan od rezultujućih objekata izmeni, izmene se i ostali jer ukazuju na istu instancu tipa `Map`.

Ovaj savet ne bi trebalo da bude pogrešno protumačen: on ne sugeriše da je izrada objekata skupa i da je treba izbegavati. Naprotiv, efikasno je napraviti i obrisati manje objekte čiji konstruktori rade veoma malo posla, naročito u modernim primenama na virtuelnim mašinama. Uopšteno govoreći, izrada dodatnih objekata radi poboljšanja jasnoće, jednostavnosti ili moći programa, preporučljiva je.

Nasuprot tome, izbegavanje izrade objekata održavanjem sopstvenog *rezervoara objekata* (engl. *object pool*) nije preporučljivo, osim kada su objekti u grupi izuzetno zahtevni. Najbolji primer objekta koji opravdava postojanje rezervoara objekata jeste veza baze podataka. Cena uspostavljanja veze je dovoljno visoka da opravdava ponovnu upotrebu objekata. S druge strane, održavanje sopstvenih rezervoara objekata dosta komplikuje program, povećava zauzeće memorije, i škodi performansama. Moderne virtuelne mašine imaju izuzetno optimizovane sakupljače smeća koji sa manjim objektima rade efikasnije nego rezervoari objekata.

Protivprimer ovog saveta je 24. savet koji se odnosi na *oprezno kopiranje* (engl. *defensive copying*). Tekući savet sugeriše: „Nemojte stvarati nov objekat kada umesto njega možete koristiti postojeći," dok 32. savet kaže: „Nemojte iznova koristiti postojeći objekat kada treba napraviti nov." Obratite pažnju na to da je kazna za ponovnu upotrebu postojećeg objekta u prilikama kad je neophodno oprezno kopiranje daleko veća nego kazna za nepotrebnu izradu duplikata objekata. Neuspeh pri izradi opreznih kopija kada je to neophodno dovodi do teško uočljivih grešaka i propusta u bezbednosnom sistemu; nepotrebna izrada objekata samo utiče na stil i performanse.

Savet 5: Uklonite zastarele reference objekata

Kada, posle korišćenja jezika u kojima sami upravljate memorijom, poput jezika C ili C++, počnete da programirate na jeziku u kojem postoje tzv. skupljači smeća, posao je znatno lakši jer objekti automatski nestaju kada vam više nisu potrebni. Kada ovo prvi put doživite, čini se kao da je u pitanju magija. To vas može lako dovesti do uverenja da uopšte ne morate voditi računa o održavanju memorije, ali to nije sasvim tačno.

Pogledajte sledeću jednostavnu realizaciju steka:

```
// Možete li uočiti "curenje memorije"?
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Obezbeđivanje mesta za barem još jedan element, grubim
     * dupliranjem kapaciteta svaki put kad je potrebno uvećati niz.
     */
    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

Na prvi pogled, ovom programu ništa ne nedostaje. Možete ga testirati koliko god želite puta, i svaki test će proći sa zavidnim rezultatima, ali u njemu i dalje postoji problem. Blago rečeno, program sadrži „curenje memorije," koje smanjuje performanse

usled povećane aktivnosti skupljača smeća ili povećanog zauzimanja memorije. U ekstremnim slučajevima, takvo curenje memorije može izazvati korišćenje virtuelne memorije na disku ili čak prekid rada programa sa greškom `OutOfMemoryError`, ali su takvi slučajevi relativno retki.

Dakle, zašto curi memorija? Ako stek naraste a zatim se smanji, objekte koji su izbačeni sa steka neće pokupiti skupljač smeća, čak i ako ih program više ne koristi. To se dešava zato što stek zadržava *zastarele reference* ovih objekata. Zastarela referenca je ona koja nikad neće biti poništena. U ovom slučaju, zastarela je svaka referenca izvan „aktivnog dela“ niza elemenata. Aktivni deo se sastoji od elemenata čiji je indeks manji od vrednosti `size`.

Curenje memorije u jezicima sa sakupljačima smeća (koje se preciznije može nazvati *nenamerna zadržavanja objekata*) teško se uočava. Ako je referenca objekta nenamerno zadržana, skupljanjem smeća neće biti obuhvaćeni ni taj objekat ni svi objekti na koje taj objekat ukazuje itd. Čak i ako je svega nekoliko referenci objekata slučajno zadržano, mnogi, mnogi objekti mogu biti izostavljeni iz skupljanja smeća, a to može značajno uticati na performanse.

Rešenje tog tipa problema je jednostavno: anulirajte reference kad postanu zastarele. U slučaju klase `Stack`, referenca postaje zastarela čim je izbačena sa steka. Ispravljena verzija metode `pop` izgleda ovako:

```
public Object pop() {
    if (size==0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminisanje zastarelih referenci
    return result;
}
```

Dodatna prednost anuliranja zastarelih referenci je to što će program prekinuti rad i prijaviti grešku `NullPointerException` ako one naknadno budu greškom dereferencirane, umesto da nastavi da radi pogrešnu stvar. Uvek je korisno što pre otkriti greške u programu.

Kada programeri prvi put naiđu na problem ove vrste, često pokušavaju da ga reše anuliranjem svih referenci objekata čim završe korišćenje tih objekata. Ovo nije ni neophodno ni poželjno pošto se program nepotrebno komplikuje pa se performanse mogu znatno pogoršati. Anuliranje referenci objekata pre bi trebalo da bude izuzetak nego pravilo. Najbolji način za uklanjanje zastarelih referenci jeste ponovna upotreba promenljive koja ju je sadržala ili završavanje oblasti važenja promenljive. To se prirodno dešava ako svaku promenljivu definišete u najužoj mogućoj oblasti važenja (29. savet). Moderne virtuelne mašine uklanjaju referencu nakon završetka metode koja obuhvata oblast važenja te reference.

Kada, dakle, treba anulirati referencu? Zbog čega je klasa `Stack` podložna curenju memorije? Jednostavno govoreći, klasa `Stack` *sama upravlja memorijom*. *Skladište* (engl. *storage pool*) sastoji se od elemenata niza `items` (reference objekata, ne objekti). Elementi u aktivnom delu niza su *aktivni* (engl. *allocated*), a oni koji se nalaze u ostatku niza su *neaktivni* (engl. *free*). Sakupljač smeća to ne može znati; za njega, sve reference objekata u nizu `items` podjednako su ispravne. Samo programer zna da je neaktivni deo niza nebitan. O toj činjenici programer obaveštava sakupljača smeća ručnim anuliranjem neaktivnih elemenata niza.

Uopšteno govoreći, kad god klasa sama upravlja memorijom, programer mora obratiti pažnju na curenje memorije. Kad god neki element postane neaktivan, sve reference objekata sadržane u njemu treba anulirati.

Još jedan od uobičajenih uzroka curenja memorije jeste keš memorija. Kada postavite referencu objekta u keš memoriju, lako je možete zaboraviti, pa ona ostaje tamo i dugo pošto postane nebitna. Postoje dva rešenja ovog problema. Ako imate dovoljno sreće da je uneta vrednost relevantna tačno onoliko dugo koliko postoje reference na njen ključ izvan keš memorije, predstavite keš memoriju objektom tipa `WeakHashMap`; unete vrednosti će automatski biti uklonjene kad zastare. Češće se dešava da period tokom kog je objekat u memoriji relevantan nije jasno definisan, pa unete vrednosti s vremenom postaju manje bitne. U takvim situacijama, keš memorija bi povremeno trebalo da se očisti od vrednosti koje se više ne koriste. To mogu činiti pozadinske niti (preko API-ja `java.util.Timer`) ili se obavlja prilikom dodavanja novih vrednosti u keš memoriju, kao sporedan efekat. Klasa `java.util.LinkedHashMap`, uvedena u verziji 1.4, primenjuje drugi pomenuti pristup pomoću metode `removeEldestEntry`.

Kako se curenje memorije obično ne manifestuje kao očigledan problem, može se desiti da godinama ne primetite takav propust. Najčešće se otkriva pažljivim pregledom koda ili pomoću alata za uklanjanje grešaka koji prati rad *dinamičke memorije*. Stoga je izuzetno bitno da naučite da predvidite ovakve probleme kako biste ih predupredili.

Savet 6: Izbegavajte korišćenje završnih metoda

Završne metode (engl. *finalizers*) nepredvidljive su, često opasne i – uopšteno govoreći – nepotrebne. Njihova upotreba može izazvati nepredvidivo ponašanje, slabe performanse i probleme s prenosivošću. Završne metode imaju nekoliko opravdanih primena, o kojima ćemo govoriti kasnije u ovom savetu, ali je opšte pravilo da ih treba izbegavati.

C++ programere upozoravamo da ne smatraju završne metode analogijama destruktora iz C++-a. U C++-u, destruktori predstavljaju uobičajen način da se oslobode resursi vezani za određen objekat, čime postaju neophodna dopuna konstruktora. U programskom jeziku Java, sakupljači smeća oslobađaju memoriju koju zauzimaju nedostupni objekti, ne zahtevajući pri tom nikakav dodatni trud programera. Destruktori u programskom jeziku C++ takođe se koriste za oslobađanje ostalih nememorijskih resursa. U programskom jeziku Java, najčešće se u tu svrhu koristi blok `try-finally`.

Ne postoji nikakva garancija da će završne metode biti brzo izvršene [JLS, 12.6]. Period između trenutka kada objekat postane nedostupan i trenutka kad njegova završna metoda bude izvršena nije strogo definisan. To znači da **ništa što je vremenski ograničeno ne bi trebalo da bude izvršeno završnom metodom.** Na primer, ozbiljna greška je prepustiti završnoj metodi da zatvori otvorene datoteke jer su deskriptori otvorenih datoteka ograničen resurs. Ako veliki broj datoteka ostane otvoren zato što JVM kasni sa izvršavanjem završnih metoda, nastaće greška jer program ne može više otvarati datoteke.

Izvršavanje završne metode zavisi od algoritma za sakupljanje smeća, koji se umnogome razlikuje od jedne do druge verzije virtuelne mašine. Ponašanje programa koji zavisi od izvršavanja završnih metoda takođe može varirati. Sasvim je moguće da će takav program savršeno raditi na virtuelnoj mašini koju koristite prilikom programiranja, a da će izazivati greške na virtuelnoj mašini koju koristi vaš najznačajniji kupac.

Zakasnelo izvršavanje završnih metoda nije samo teoretski problem. Korišćenje završne metode u klasi može, u retkim okolnostima, odužiti oslobađanje memorije njenih instanci. Kolega je nedavno otklonio greške u GUI aplikaciji koja se već dugo koristi, a koja je misteriozno prekidala rad usled greške `OutOfMemoryError`. Analiza je otkrila da je u trenutku prekidanja aplikacija sadržala hiljade grafičkih objekata koji su čekali u redu da ih završna metoda obriše. Nažalost, nit završne metode bila je pokrenuta na nižem prioritetu nego što je bio prioritet druge niti u aplikaciji, pa objekti nisu bili uništavani brzinom kojom su postajali nedostupni. JLS ne garantuje koja će od niti izvršavati finalizaciju, tako da ne postoji nijedan univerzalan način da se spreči ovakav tip problema sem da se uzdržite od upotrebe završnih metoda.

Ne samo da JLS ne pruža nikakvu garanciju da će se završne metode izvršavati na vreme, već ne garantuje ni da će one uopšte biti izvršene. Sasvim je moguće, čak vrlo verovatno, da će se program završiti bez izvršavanja završnih metoda nekih objekata. Prema tome, **nikad ne treba da koristite završne metode za ažuriranje ključnog trajnog stanja**. Na primer, ako završna metoda treba da otključa trajno zaključan deljeni resurs poput baze podataka, napravićete zastoj na celom deljenom sistemu.

Neka vas ne zavedu metode poput `System.gc` i `System.runFinalization`. One mogu povećati verovatnoću da će završne metode biti izvršene, ali to ne mogu sa sigurnošću garantovati. Jedine metode koje navodno garantuju izvršavanje završnih metoda jesu `System.runFinalizersOnExit` i njena zla bliznakinja, `Runtime.runFinalizersOnExit`. Te metode sadrže nepopravljive propuste i zato se preporučuje da ih ne koristite (označene su kao zastarele).

Ukoliko još uvek niste uvereni da treba izbegavati završne metode, evo još jednog detalja vrednog razmatranja: ako se neobrađeni izuzetak pojavi tokom izvršavanja završne metode, on će biti zanemaren, a ta završna metoda će se prekinuti [JLS, 12.6]. Neobrađeni izuzeci mogu dovesti do toga da objekti ostanu u neispravnom stanju. Ako još neka nit pokuša da koristi takav neispravan objekat, rezultat nije predvidljiv. U normalnim uslovima, neobrađeni izuzetak će prekinuti nit i ispisati stek poziva metoda, ali ne ako se to bude dogodilo unutar završne metode – čak neće prijaviti ni upozorenje.

Šta onda treba raditi umesto pisanja završne metode za klasu čiji objekti zauzimaju resurse koje morate osloboditi, poput datoteka ili niti? **Napišite *metodu za eksplicitno oslobađanje resursa*** (engl. *explicit termination method*), i tražite od klijenata klase da pozovu tu metodu kada im instanca više ne bude trebala. Još jedan detalj vredan pomena je da instance moraju znati da li su resursi oslobođeni: metoda za eksplicitno oslobađanje mora zabeležiti u privatno polje da objekat više nije važeći, i da druge metode moraju proveriti to polje i prijaviti `IllegalStateException` u slučaju da su pozvane pošto su resursi oslobođeni.

Tipičan primer metode za eksplicitno oslobađanje resursa jeste metoda `close` u `InputStream` i `OutputStream`. Još jedan primer je metoda `cancel` u `java.util.Timer`, koja menja stanje kako bi se nit vezana za instancu `Timer` regularno završila. Primeri iz paketa `java.awt` obuhvataju `Graphics.dispose` i `Window.dispose`. Ove metode se često previde, pa mogu nastati ozbiljne posledice po performanse. Srodna metoda je `Image.flush`, koja oslobađa sve resurse koje je zauzela instanca klase `Image` ali ih ostavlja u stanju u kome se instanca i dalje može koristiti, ponovo zauzimajući resurse ako je to neophodno.

Metode za eksplicitno oslobađanje resursa često se koriste u kombinaciji sa konstrukcijom `try-finally` kako bi se osigurao blagovremeni završetak. Pozivanje metode za eksplicitno oslobađanje unutar odredbe `finally` osigurava njeno izvršenje čak i ako se pojavi izuzetak tokom upotrebe objekta:

```
// blok try-finally osigurava izvršenje završne metode
Foo foo = new Foo(...);
try {
    // Uradi šta mora biti urađeno sa foo
    ...
} finally {
    foo.terminate();  // Eksplicitna završna metoda
}
```

Čemu onda završne metode uopšte služe? Postoje dve opravdane upotrebe. Jedna je da se ponašaju kao „sigurnosna mreža“ u slučaju da vlasnik objekta zaboravi da pozove metodu za eksplicitno oslobađanje koju ste koristili po savetu iz prethodnog pasusa. Iako ne postoji nikakva garancija da će završna metoda biti pozvana na vreme, kritični resurs je bolje osloboditi ikad nego nikad, u (nadajmo se retkim) slučajevima kada klijent ne pozove metodu za eksplicitno oslobađanje. Tri klase spomenute u primerima (`InputStream`, `OutputStream`, `iTimer`) poseduju završne metode koje služe kao bezbednosne mreže u slučaju da ne budu pozvane metode za eksplicitno oslobađanje.

Druga opravdana upotreba završnih metoda odnosi se na objekte koji koriste *lokalne ekvivalente*. Lokalni ekvivalent (engl. *native peer*) jeste objekat lokalnog operativnog sistema (izvan virtuelne mašine), kome normalan objekat predaje dužnost preko lokalnih metoda. Pošto lokalni ekvivalent nije normalan objekat, skupljač smeća ne zna za njega i ne može ga osloboditi kada njegov ekvivalent postane nedostupan. Završna metoda je odgovarajuće rešenje za izvršavanje ovog zadatka, *pod uslovom da lokalni ekvivalent ne sadrži kritične resurse*. Ako lokalni ekvivalent sadrži resurse koji na vreme moraju biti oslobođeni, klasa treba da sadrži metodu za eksplicitno oslobađanje, kao što je prethodno opisano. Završna metoda mora da uradi šta god je neophodno kako bi se kritični resurs oslobodio. Završna metoda može biti i lokalna metoda, ili je može pozivati.

Bitno je reći da se „ulančavanje završnih metoda“ (engl. *finalizer chaining*) ne obavlja automatski. Ako klasa (bilo koja sem `Object`) sadrži završnu metodu i potklasu koja je redefiniše, završna metoda potklase mora ručno pozvati završnu metodu natklase. Resurse potklase oslobodite u bloku `try` i završnu metodu natklase pozovite u odgovarajućem bloku `finally`. Tako će završna metoda nadređene klase biti izvršena čak i ako se pojavi izuzetak pri oslobađanju resursa potklase, i obrnuto:

```
// Ručno ulančavanje završnih metoda
protected void finalize() throws Throwable {
    try {
        // Oslobađanje resursa
        ...
```

```
    } finally {
        super.finalize();
    }
}
```

Ako potklasa redefiniše završnu metodu natklase ali se pri tom zaboravi na ručno pozivanje završne metode natklase (ili se to namerno desi), završna metoda natklase nikad neće biti pozvana. Od takvih problema možete se zaštititi pravljenjem dodatnog objekta za svaki objekat koji treba da bude oslobođen. Umesto pisanja završne metode klase čije se oslobađanje traži, postavite je u anonimnu klasu (18. savet) čija je jedina svrha da oslobodi resurse instance koju obuhvata. Jedinstvena instanca anonimne klase, zvana *čuvar završne metode* (engl. *finalizer guardian*), pravi se za svaku instancu obuhvaćene klase. Obuhvaćena instanca pamti jedinstvenu referencu svog čuvara završne metode u privatnom polju kako bi čuvar završne metode bio obrisan neposredno pre obuhvaćene instance. Kada se čuvar briše, on oslobađa resurse obuhvaćene instance, baš kao da je njegova završna metoda izvršena nad obuhvaćenom klasom:

```
// Čuvar završne metode
public class Foo {
   // Jedina svrha ovog objekta je da oslobodi resurse
   // spoljašnjeg objekta
   private final Object finalizerGuardian = new Object() {
      protected void finalize() throws Throwable {
         // Oslobađanje resursa spoljašnjeg objekta
         ...
      }
   };
   ...  // Ostatak izostavljen
}
```

Obratite pažnju na to da javna klasa, `Foo`, ne sadrži završnu metodu (sem trivijalne koju nasleđuje od klase `Object`), pa nije važno da li završna metoda potklase poziva `super.finalize` ili ne. Ovu tehniku treba uzeti u obzir u slučaju svake nezavršne javne klase koja ima završnu metodu.

Ukratko rečeno, nemojte koristiti završne metode osim kao sigurnosne mreže ili u svrhu uklanjanja lokalnih resursa koji nisu ključni. U tim retkim situacijama u kojima koristite završne metode, ne zaboravite da pozovete `super.finalize`. Na kraju, ako je potrebno da povežete završnu metodu u nezavršnoj javnoj klasi (njeno nasleđivanje nije blokirano), razmotrite upotrebu čuvara završne metode kako biste se osigurali da će završna metoda biti izvršena, čak i ako završna metoda potklase ne uspe da pozove `super.finalize`.