Statičke varijable

Statičke varijable su varijable koje pripadaju razredu a ne primjercima razreda (odnosno stvorenim objektima tog tipa). Za svaku statičku varijablu razreda u memoriji postoje samo jedan primjerak te varijable koji je također vidljiv i svim stvorenim primjercima tog razreda (tj. objektima).

Statičkim varijablama (ako to modifikator vidljivosti dozvoljava) može se pristupiti izvana sintaksom *ime_razreda točka ime_varijable*. Npr, ako razred imena `C` ima javnu statičku varijablu `brojac`, istoj se može pristupiti navođenjem `C.brojac`. Istoj je također dopušteno pristupiti i kroz referencu na bilo koji objekt tog razreda, ali prevodilac će generirati upozorenje da to nema smisla jer varijabla pripada razredu, ne objektu. Evo primjera.

```
public class C {
  public static int brojac;
  public int broj;

  public C(int broj) {
    this.broj = broj;
    brojac++;
  }
}

public static void main(String[] args) {

  System.out.printf("C.brojac = %d%n", C.brojac);

  C c1 = new C(5);
  C c2 = new C(10);

  System.out.printf("C.brojac = %d%n", C.brojac);

  System.out.printf(
    "c1.broj = %d, c2.broj = %d, c1.brojac = %d, c2.brojac = %d%n",
    c1.broj, c2.broj, c1.brojac, c2.brojac
  );

  C.brojac++;
  c1.broj += 3;
  c2.broj += 6;

  System.out.printf(
    "c1.broj = %d, c2.broj = %d, c1.brojac = %d, c2.brojac = %d%n",
    c1.broj, c2.broj, c1.brojac, c2.brojac
  );

  c1.brojac = 42;

  System.out.printf(
    "c1.broj = %d, c2.broj = %d, c1.brojac = %d, c2.brojac = %d%n",
    c1.broj, c2.broj, c1.brojac, c2.brojac
  );

}
```

Razred `C` deklariran je s jednom statičkom varijablom `brojac` (ta varijabla konceptualno pripada razredu) te s jednom nestatičkom varijablom `broj` (ona pripada primjerku razreda – svaki stvoreni primjerak imat će svoju varijablu `broj` u memoriji).

Pokretanjem metode main (nije bitno u kojem je razredu) dobili bismo sljedeći ispis.

```
C.brojac = 0
C.brojac = 2
c1.broj = 5, c2.broj = 10, c1.brojac = 2, c2.brojac = 2
c1.broj = 8, c2.broj = 16, c1.brojac = 3, c2.brojac = 3
c1.broj = 8, c2.broj = 16, c1.brojac = 42, c2.brojac = 42
```

Kako statička varijabla nije inicijalizirana, njezina je početna vrijednost `0`. Stoga je prvi ispis:

```
C.brojac = 0
```

Potom stvaramo dva primjerka razreda `C`, a svaki od njih u konstruktoru poveća sadržaj zajedničke varijable `brojac` za jedan. Stoga je sljedeći ispis:

```
C.brojac = 2
c1.broj = 5, c2.broj = 10, c1.brojac = 2, c2.brojac = 2
```

Vidimo da svaki objekt ima svoju inačicu varijable `broj` (ona nije deklarirana kao statička pa pripada primjercima razreda tj. objektima), dok oba objekta vide isti sadržaj varijable `brojac`.

Potom varijablu brojac povećamo za `1`, broj prvog objekta za `3` a drugog za `6`. Ispis je:

```
c1.broj = 8, c2.broj = 16, c1.brojac = 3, c2.brojac = 3
```

i opet oba objekta vide novi sadržaj dijeljene varijable `brojac`.

Konačno, brojac izravno postavljamo na 42, i tu promjenu vide oba objekta što je vidljivo iz ispisa:

```
c1.broj = 8, c2.broj = 16, c1.brojac = 42, c2.brojac = 42
```

Inicijalizaciju statičkih varijabli možemo raditi izravno pri deklaraciji ili uporabom statičkog inicijalizatora (blok koda označen ključnom riječi static). Evo primjera.

```
class D {
  static int varijabla1 = 35;  // deklariram i inicijaliziram
  static int varijabla2;       // deklariram

  static {                     // statički inicijalizacijski blok
    int potencija = 1;
    for(int i = 1; i <= 3; i++) {
      potencija *= 2;
    }
    varijabla2 = potencija;    // inicijaliziram
  }
}
```

Primjer pokazuje da se u statičkom inicijalizacijskom bloku može nalaziti i dio koda koji računa vrijednost kojom će inicijalizirati statičku varijablu.

Za dodatne informacije proučiti odjeljak 8.3.1.1 na adresi:
https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.3.1.1

Statičke metode

Statičke metode su metode koje pripadaju razredu a ne primjercima razreda (odnosno stvorenim objektima tog tipa). Statičke metode u razredu se deklariraju ključnom riječi `static`. Statičke metode ne dobivaju kao skriveni parametar pokazivač na objekt (primjerak razreda) nad kojim su pozvane; štoviše, prevodilac će Vas upozoriti da je takva sintaksa poziva (iako dopuštena) čudna jer čitatelja navodi na pomisao da se metoda poziva nad objektom a ne nad njegovim razredom. Stoga statičke metode (izvana) preferirano pozivamo sintaksom *imeRazreda točka imeMetode*. Kako statičke metode ne dobivaju referencu na objekt nad kojim su pozvane, one ne vide (nemaju pristup) nestatičkim članskim varijablama razreda – mogu pristupati isključivo statičkim članskim varijablama te izravno mogu pozivati samo druge statičke metode.

Evo primjera.

```
class X {
  public static int a = 5;
  public int b = 0;

  // nestatička metoda – vidi nestatičke varijable
  public int uvecajIVratiB() {
    return ++b;
  }

  // nestatička metoda – vidi i statičke varijable
  public int uvecajIVratiA2() {
    return ++a;
  }

  // statička metoda – vidi statičke varijable
  public static int uvecajIVratiA() {
    return ++a;
  }

//  statička metoda – NE vidi nestatičke varijable
//  Ova metoda ispod se ne bi prevela: kojem primjerku pripada b?
//  public static int uvecajIVratiB2() {
//    return ++b;
//  }
}
```

```
class NekiDrugi {
  public static void main(String[] args) {
    X x1 = new X();
    X x2 = new X();

    int tmp1 = x1.uvecajIVratiB(); // vraća 1
    int tmp2 = x2.uvecajIVratiB(); // vraća 1
    // int tmpx = X.uvecajIVratiB(); // ne može! Metoda nije
                                     // statička pa tako ne može

    int tmp3 = x1.uvecajIVratiA(); // vraća 6; upozorenje...
    int tmp4 = x2.uvecajIVratiA(); // vraća 7; upozorenje...
    int tmp5 = X.uvecajIVratiA();  // vraća 8; OK, poziv preko
                                   // razreda
  }
}
```

Modifikator `final`

Koristi se (ovisno o kontekstu) kako bi zabranio daljnje promjene, nadjačavanje metode ili pak nasljeđivanje kompletnog razreda.

Primjer 1. Lokalna varijabla

```
void x() {
  final int broj = 42;
  broj++; // ovo se ne da iskompajlirati – varijabla broj
          // ne može mijenjati vrijednost
}
```

Primjer 2. Članska varijabla razreda

```
public class X {
  final int broj1 = 42;  // OK: deklarirano i inicijalizirano
  final int broj2;       // Samo deklarirano – konstruktor
                         // treba napraviti inicijalizaciju

  public X(int a, int b) {
    broj1 = 35;   // Greška: varijabla je već inicijalizirana
    broj2 = a+b;  // OK: u konstruktoru smo; on mora
                  // inicijalizirati ovu Člansku varijablu
    broj2++;      // Greška – više se ne može mijenjati!!!
  }

  public void m() {
    broj1++; // Greška – ne da se iskompajlirati
    broj2++; // Greška – ne da se iskompajlirati
  }
}
```

Članskim varijablama koje su deklarirane kao `final` vrijednost je moguće pridijeliti odmah kod deklaracije (slučaj varijable broj1) ili pak u konstruktoru. Ako `final` članska varijabla nema inicijalizaciju pri deklaraciji, onda joj konstruktor **mora** dodijeliti vrijednost.

Primjer 3. Konstantnost parametra metode

```
class X {
  public void m1(final int p) {
    p++; // Greška: unutar metode, argument p se tretira kao
         // konstantna vrijednost
  }
  public void m2(int x) {
    m1(x);   // Sada pozivam m1 s argumentom postavljenim na x
    m1(x+4); // Sada pozivam m1 s argumentom postavljenim na x+4
  }
}
```

Primjer pokazuje da pojedini argumenti metode mogu biti označeni kao `final`; u tom slučaju, kod

u tijelu metode vrijednost tog argumenta može samo čitati, a ne može ga koristiti kao pomoćnu lokalnu varijablu koju po potrebi može i mijenjati.

Primjer 4. Zabrana nadjačavanja

```
class X {
  public void m1() { … }
  public final void m2() { … }
}

class Y extends X {
  public void m1() { … }
  public void m2() { … } // Greška: ne mogu nadjačati m2
}
```

U ovom primjeru razred `X` definira dvije metode: `m1` i `m2` gdje je `m2` označena kao `final`. Potom razred `Y` nasljeđuje razred `X` i pokušava ponuditi nove definicije metoda `m1` i `m2`; za metodu `m1` to može ali kod `m2` prevodilac će javiti grešku – metodu `m2` nije moguće nadjačati jer je `X` ponudio njezinu konačnu verziju.

Primjer 5. Zabrana nasljeđivanja

```
class X {
  ...
}

final class Y extends X {
  ...
}

class Z extends Y {  // Greška!!!
  ...
}
```

U ovom primjeru razred `X` je osnovni razred. Razred `Y` nasljeđuje razred `X` i deklarira da je on zadnji u toj grani nasljeđivanja – nitko ne može naslijediti razred `Y`. Razred `Z` pokušava naslijediti razred `Y` i prevodilac tu javlja grešku – razred `Y` je konačan i nitko ga ne može naslijediti.

Za dodatne informacije proučiti odjeljak 8.3.2.2 na adresi:
https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.3.1.2

Primjer 6. Pri deklaraciji varijabli u petlji for

```
class X {
  public void m3() throws FileNotFoundException {
    int[] polje = {1,2,3,4,5};
    for(final int broj : polje) {
      System.out.println(broj);
    }
  }
}
```

Modifikator `final` ovdje govori da se u tijelu petlje `for` u svakoj iteraciji varijabla broj neće mijenjati od strane programera.

Enumeracije

Razmotrite sljedeći primjer.

```
package hr.fer.zemris.javatecaj;

public class Main {

	public static void main(String[] args) {
		obrada(2.7182818284590452354);
	}

	private static void obrada(double d) {
		// Kako ograničiti pozivatelja da zada nešto
		// što nije 0, 1, PI ili E?
		System.out.println("Dobio sam broj: " + d+".");
	}

}
```

Pretpostavimo da imamo metodu `obrada(broj)` koja kao argument smije primiti jednu od četiri konstante: 0, 1, e ili pi. Kako pozivatelju takve metode nametnuti takvo ograničenje?

Prikazani primjer kao argument metode deklarira varijablu tipa **`double`**. Uz taj tip argumenta, pozivatelj metodi može poslati bilo kakav decimalni broj i prevoditelj neće prijaviti pogrešku. U metodi `obrada` tada bismo morali na početku imati provjeru je li vrijednost argumenta jedna od dozvoljenih pa ako nije, nešto učiniti (baciti iznimku ili slično). Imamo li više takvih metoda, kod provjere bismo morali ponavljati (ili izlučiti u novu metodu pa pozivati) više puta. Također, pogrešku bismo mogli utvrditi tek pri pokretanju programa, a ne već pri prevođenju.

Jedno rješenje je definirati javni razred koji predstavlja jednu konstantu. Primjerak takvog razreda kroz konstruktor prima i u članskoj varijabli pamti vrijednost konstante te nudi getter za dohvat vrijednosti. Kako nitko ne bi mogao mijenjati ponašanje primjeraka ovog razreda nasljeđivanjem, razred čemo zaključati za nasljeđivanje (final class). Konstruktor ćemo definirati kao privatan tako da ga nitko izvana ne može pozivati a u definiciji razreda sami ćemo deklarirati nekoliko javnih statičkih konstanti. Evo koda.

```
package hr.fer.zemris.javatecaj;

public final class MathematicalConstant {

	// public constants:
	// -----------------

	/**
	 * Constant PI (the ratio of circle's circumference to its diameter).
	 */
	public static final MathematicalConstant PI =
				new MathematicalConstant(3.14159265358979323846);
	/**
	 * Constant E (base of natural logarithm).
	 */
	public static final MathematicalConstant E =
				new MathematicalConstant(2.7182818284590452354);
	/**
	 * First natural number. Neutral element for addition.
	 */
	public static final MathematicalConstant ZERO =
				new MathematicalConstant(0.0);
	/**
	 * Second natural number. Neutral element for multiplication.
	 */
	public static final MathematicalConstant ONE =
				new MathematicalConstant(1.0);
```

```
        // private final member variables:
        // -------------------------------


        /**
         * The value of this constant.
         */
        private final double value;

        /**
         * Constructor.
         * @param value constant's value
         */
        private MathematicalConstant(double value) {
                super();
                this.value = value;
        }

        /**
         * Getter for value.
         *
         * @return value of this constant
         */
        public double getValue() {
                return value;
        }

}
```

Uočite: kako je konstruktor privatan, nitko ne može stvarati primjerke ovog razreda osim koda u tom razredu. Razred definira četiri statičke konstante i inicijalizira ih stvarajući jedina četiri primjerka tog razreda koja će ikada postojati u memoriji virtualnog stroja.

Sada klijentski kod možemo prepisati tako da kažemo da kao argument prima referencu na jedan objekt razreda `MathematicalConstant`. Na taj način pozivatelj neće klijentu moći poslati ništa što nije jedna od postojećih konstanti (ili eventualno referenca `null`). Evo popravljenog koda.

```
package hr.fer.zemris.javatecaj;

public class Main {

        public static void main(String[] args) {
                obrada(MathematicalConstant.E);
        }

        private static void obrada(MathematicalConstant e) {
                System.out.println("Dobio sam broj: "+ e.getValue());
        }
}
```

Da ne bismo sami morali pisati konačne razrede i eksplicitno definirati i stvarati statičke konstante, Java za to nudi pokratu: ključnu riječ **enum** kojom se definira enumeracija.

Najjednostavniji primjer enumeracije dobivamo samo nabrajanjem imena statičkih konstanti koje bismo deklarirali pri pisanju odgovarajućeg razreda. Tako je sljedeći kod:

```
public enum X {
        A, B, C
}
```

konceptualno zamjena za:

```
public final class X {
	public static final X A = new X();
	public static final X B = new X();
	public static final X C = new X();

	private X() {}
}
```

Međutim, enumeracije jest "posebna" vrsta razreda koji uz neka ograničenja može imati gotovo sve što može i klasičan razred: može imati konstruktore (ali modifikator ili ne smije pisati, ili može biti samo **`private`**), može imati nestatičke članske varijable (trebale bi biti **`final`**), može imati javne metode (primjerice, gettere za dohvat vrijednosti nestatičkih članskih varijabli). Svaka enumeracija implicitno nasljeđuje vršni razred `java.lang.Enum` i zaključana je za daljnje nasljeđivanje. Ako imamo definiran konstruktor koji prima parametre, onda pri deklaraciji svake od konstanti odmah otvaramo zagradu i predajemo parametre koji idu konstruktoru – ne možemo eksplicitno pozivati konstruktor uporabom **`new`**.

Matematičke konstante iz ovog zadatka kao enumeraciju bismo definirali na sljedeći način.

```
package hr.fer.zemris.javatecaj;

public enum MathematicalConstant {

	// public constants:
	// -----------------

	/**
	 * Constant PI (the ratio of circle's circumference to its diameter).
	 */
	PI(3.14159265358979323846),
	/**
	 * Constant E (base of natural logarithm).
	 */
	E(2.7182818284590452354),
	/**
	 * First natural number. Neutral element for addition.
	 */
	ZERO(0.0),
	/**
	 * Second natural number. Neutral element for multiplication.
	 */
	ONE(1.0);

	/**
	 * The value of this constant.
	 */
	private final double value;

	/**
	 * Constructor.
	 * @param value constant's value
	 */
	private MathematicalConstant(double value) {
		this.value = value;
	}

	/**
	 * Getter for value.
	 *
	 * @return value of this constant
	 */
	public double getValue() {
		return value;
	}

}
```

Pažljivo proučite kod; uočite da smo pojedine konstante razdvojili zarezom, a popis svih konstanti od ostatka razreda (deklaracije nestatičkih članskih varijabli, konstruktora i metoda) razdvojili znakom točka-zareza. Svaka ovako definirana konstanta jedan je primjerak razreda (odnosno enumeracije) `MathematicalConstant`. Definirali smo da postoji članska varijabla `value` pa svaki primjerak ove enumeracije ima svoj vlastiti primjerak te varijable koju inicijalizira kroz konstruktor a vanjskom korisniku nudi kroz odgovarajuću metodu za dohvat (*getter*). Uočite da razini stanja u memoriji, enumeracija doista nije ništa drugo doli "posebna" vrsta razreda a definirane konstante nisu ništa dugo doli primjerci tog razreda (objekti).

Usporedite u razredu `MathematicalConstant` redak:

```
/**
 * Constant PI (the ratio of circle's circumference to its diameter).
 */
public static final MathematicalConstant PI =
                     new MathematicalConstant(3.14159265358979323846);
```

te u enumeraciji istovjetan redak:

```
/**
 * Constant PI (the ratio of circle's circumference to its diameter).
 */
PI(3.14159265358979323846),
```

Uočavate li koji su dijelovi koda ispali (odnosno što prevodilac automatski dopisuje za nas)?

Uz ovu modifikaciju, kod klijenta (metodu obrada) uopće ne treba mijenjati: ona i dalje očekuje kao argument referencu na objekt. Konstante enumeracije jesu objekti koji su stvoreni pozivom operatora `new` i možemo ih slati okolo. Vrijednost konstante PI sada bismo dobili tako da pristupimo objektu koji predstavlja tu konstantu i nad njim pozovemo getter za dohvat vrijednosti koja je pohranjena u člansku varijablu kroz konstruktor; evo koda:

```
double pi = MathematicalConstant.PI.getValue();
```

Također, konstante enumeracije ne moramo uspoređivati pozivom metode `equals` – s obzirom da za svaku konstantu u memoriji virtualnog stroja postoji upravo jedan primjerak tog razreda (ovo nam za enumeracije garantira specifikacija jezika Java), možemo ih izravno uspoređivati operatorom `==`.

Često ćemo enumeracije koristiti samo za deklariranje pobrojanih konstanti gdje pojedine konstante nemaju "dublju" strukturu. Primjerice:

```
enum Izlaz {
  EKRAN,
  DATOTEKA,
  PISAČ;
}
```

```
void m(Izlaz izlaz) {
  …
  if(izlaz == Izlaz.EKRAN) {
    …
  } else  if(izlaz == Izlaz.DATOTEKA) {
    …
  } else if(izlaz == Izlaz.PISAČ) {
    …
  }
}
```