Java VII. – Műveletek

Tekintsük az alábbi kódrészletet. Definiáljunk aritmetikai műveleteket ehhez az osztályhoz. Világos, hogy egészen sok módon implementálhatjuk ezeket.

package numbers;

public class Rational {

private int numerator, denominator;

/\* class invariant: denominator > 0 \*/

public Rational( int numerator, int denominator ){

if( denominator <= 0 ) throw new IllegalArgumentException();

this.numerator = numerator;

this.denominator = denominator;

}

}

# Az imperatív stílus

public class Rational {

...

/\*\*

\* Set {@code this} to {@code this} \* {@code that}.

\* @param that Non-null reference to a rational number,

\* it will not be changed in the method.

\* @throws NullPointerException When {@code that} is null.

\*/

public void multiplyWith( Rational that ){

this.numerator \*= that.numerator;

this.denominator \*= that.denominator;

}

...

}

Az imperatív stílus lényege, hogy a meglévő objektumaink belső állapotát változtatjuk, azaz objektumaink mutálhatók.

Annak érdekében, hogy egymás után többször is meghívható legyen egy művelet gyakran használjuk az alábbi idiómát:

...

public Rational multiplyWith( Rational that ){

this.numerator \*= that.numerator;

this.denominator \*= that.denominator;

return this;

} ...

Ekkor megtehetjük az alábbit:

p.multiplyWith(q).multiplyWith(q).divideBy(q);

# A túlterhelés (overloading)

A túlterhelés lehetővé teszi, hogy a kódban több azonos nevű műveletet definiáljunk.

static void m( long n ){ ... }

static void m( float n ){ ... }

public static void main( String[] args ){

m(3);

}

Ekkor a megadott paraméterre legjobban illeszkedő művelet kerül meghívásra. Ha nincs pontosan illeszkedő szignatúra, akkor automatikus konverzió történik és az átkonvertált értékre legjobban illeszkedő művelet érvényesül.

Bizonyos esetekben az is előfordulhat, hogy a fordító nem tudja kikövetkeztetni, a legjobban illeszkedő műveletet.

static void m( long n, float m ){ ... }

static void m( float m, long n ){ ... }

public static void main( String[] args ){

m(4,2);

}

Foo.java:5: error: reference to m is ambiguous

m(4,2);

^

both method m(long,float) in Foo

and method m(float,long) in Foo match

1 error

A túlterhelés akkor lehetséges, ha az azonos nevű műveletek formális paraméterei számban és/vagy típusban eltérőek. Fordítási hibához vezet az, ha nincs a hívásnak megfelelő művelet vagy, ha több ilyen művelet is van.

Gyakran előfordul, hogy olyan túlterhelt műveleteket alkotunk, melyek kevesebb paraméterrel rendelkeznek, mint a tényleges adattagok száma, a nem paraméterrel megadott adattagokat pedig valamilyen kezdőértékre akarjuk állítani. Ezt azért tesszük mert a Java-ban nincs elehetőség alapértelmezett értéket adni paramétereknek.

public class Rational {

...

public Rational( int numerator = 0, int denominator = 1 ){ //rossz

if( denominator <= 0 ) throw new IllegalArgumentException();

this.numerator = numerator;

this.denominator = denominator;

}

A konstruktorokat szintén túlterhelhetjük sőt viszonylag gyakori technika általánosabb és specifikusabb konstruktorokat is írni.

public class Rational {

...

public Rational( int numerator, int denominator ){

if( denominator <= 0 ) throw new IllegalArgumentException();

this.numerator = numerator;

this.denominator = denominator;

}

public Rational( int value ){

this(value,1); // legelső utasítás kell legyen

}

public Rational(){

this(0);

}

...

}

Fontos megjegyezni, hogy konstruktorban másik konstruktor hívása csak abban az esetben lehetséges ha azt az első utasításként végezzük.

Néha ajánlott a túlzott konstruktorhasználat helyett gyártófüggvényeket használni.

*factory method*, pl. new Rational(0) helyett Rational.zero()

public class Rational {

...

private Rational( int numerator, int denominator ){

this.numerator = numerator;

this.denominator = denominator;

}

public static Rational make( int numerator, int denominator ){

return new Rational(numerator,denominator);

}

public static Rational valueOf( int val ){return make(val,1);}

public static Rational oneOver( int den ){return make(1,den);}

public static Rational zero(){ return make(0,1); }

}

Operátorok túlterhelésére viszont nincs lehetőség.

# A funkcionális stílus

public class Rational {

...

public void multiplyWith( Rational that ){ ... }

public Rational times( Rational that ){ ... }

}

Times összeszorozza a két racionális számot és visszaadja az így létrejött új számot. (elnevezési ajánlás: ige – változás, névszó – valami kiszámlása)

Ha azt szeretnénk, hogy egy mező megváltoztathatalan legyen akkor a final kulcsszót használjuk.

public class Rational {

private final int numerator, denominator;

...

Léynegében a funkcionális stílus lényege az, hogy az objektumok belső állapotát nem változtatjuk, hanem mindig új objektumokat hozunk létre az eredeti objektumokból kiszámolt értékekkel.

# A final kulcsszó

A final kulcsszó attól függően, hogy mi előtt áll többféle dolgot jelenthet.

## Globális konstans

public static final int WIDTH = 80;

Egy osztályszintű publikusan látható konstans mező. Konvenció, hogy a teljes azonosítóját nagybetűvel írjuk.

## Módosíthatatlan mezők

public class Rational {

private final int numerator, denominator;

Olyan mezők, melyek csak egyszer kaphatnak értéket.

## Módosíthatatlan lokális változó

public class Rational {

...

public void simplify(){

final int gcd = gcd(numerator, denominator);

...

## Módosíthatatlan formális paraméter

Annak kikényszerítése, hogy ne kerüljön a paraméter módosításra (ekvivalens a const paraméterrel a C++-ból). Ez a kódrészlet például hibát okoz:

static java.math.BigInteger factorial( final int n ){

assert n > 0;

java.math.BigInteger result = java.math.BigInteger.ONE;

while( n > 1 ){

result = result.multiply(java.math.BigInteger.valueOf(n));

--n;

}

return result;

}

## Final referencia

Nem állítható át más címre.

final int[] data = new int[2];

data[0] = 3;

data[1] = 4;

data = new int[3]; // fordítási hiba

## Karaktersorozatok ábrázolása

java.lang.String módosíthatatlan (*immutable*/mutálhatatlan).

String fourtytwo = "42";

String twentyfour = fourtytwo.reverse();

String twentyfourhundredfourtytwo = twentyfour + fourtytwo;

Ha módosítható String-et szeretnénk akkor a StringBuilder osztályt használjuk (esetleg a char[]-t ha valamiért C nosztalgiánk lenne). Ha sokszor kell módosítani egy karaktersorozatot akkor a StringBuilder egy sokkal hatékonyabb megoldás.

# Procedurális stílus

Osztályszintű metódusok előnyben részesítése:

public class Rational {

private final int numerator, denominator;

public Rational( int numerator, int denominator ){ ... }

public int numerator(){ return numerator; }

public int denominator(){ return denominator; }

public static Rational times( Rational left, Rational right ){

return new Rational( left.numerator \* right.numerator,

left.denominator \* right.denominator );

}

}

Szimmetriai szempontból talán egy kicsit elegánsabb, a paraméterek egyenrangúságát egy kicsit jobban kifejező megoldás, ettől függetlenül nem ajánlott.

# Paraméterátadás

A Java nyelvben az érték szerinti (call-by-value) és a megosztás szerinti (call-by-sharing) paraméterátadás van jelen.

A megosztás szerinti paraméterátadás egy kevésbé megengedőbb változata a referencia szerinti paraméterátadásnak, azaz a paraméterben átadott objektumhoz hozzáfér az adott eljárás, de annak új értéket adni nem tud, tehát a paraméterként átadott objektum helyére sosem fog egy másik objektum.

public static void multiplyLeftWithRight( Rational left,Rational right){

left.numerator \*= right.numerator;

left.denominator \*= right.denominator;

left = new Rational(9,7);

}

Tehát attól függetlenül, hogy a left-nek új értéket adunk a paraméterként átadott objektum nem változik meg.

A primitív típusok érték szerint a referencia típusok megosztás szerint adódnak át. (A referencia értéke, ami egy memóriacím viszont érték szerint adódik át.)

# Változó számú paraméter

A nyelv tartalmaz egy speciális szintaktikai elemet az ún. vargarg-ot.

static int sum( int[] nums ){

int s = 0;

for( int n: nums ){ s += n; }

return s;

}

sum( new int[]{1,2,3,4,5,6} )

Az alábbi kódrészlet a fentivel ekvivalens, mivel a fordítóprogram a fordítás egy viszonylag korai fázisban a vararg-ot automatikusan a fenti példában látható tömbös megvalósításra cseréli.

static int sum( int... nums ){

int s = 0;

for( int n: nums ){ s += n; }

return s;

}

sum(1,2,3,4,5,6)

# Aliasing

Tekintsük az alábbi hibás kódot:

public class Rational {

...

public void divideBy( Rational that ){

if( that.numerator == 0 ) throw new ArithmeticException("0 division!");

this.numerator \*= that.denominator;

this.denominator \*= that.numerator;

}

}

Látszólag működőképesnek tűnik, de mi van, ha meghívjuk a divideBy() metódust a következő paraméterekkel:

Rational p = new Rational(1,2);

p.divideBy(p);

Világos, hogy ebben az esetben nem fog helyesen működni a programunk, mert először megváltozik a nevező 2-re, majd a számláló változik meg 4-re.

További gyakori probléma az alábbi:

public void set( int[] data ){

this.data = data;

}

Hajlamosak vagyunk megfeledkezni arról, hogy egy ilyen értékadásnál csak egy alias jön létre a paraméterként megadott objektumra. Ha a paraméterként megadott objektumot szeretnénk az objektum belső állapotává tenni, ez könnyen annak kiszivárgásához vezethet. Hasonló dolog történhet, ha egy getter-en keresztül próbáljuk visszaadni egy referencia típusú adattagot.

A tanulság az, hogy ha referencia típussal konstruálunk vagy referencia típust adunk vissza mindig másoljunk. Kivételt képeznek a módosíthatatlan objektumok, mint a String.

További érdekesség, hogy a tömbelemek közt is lehet aliasing.

Rational[] rats = new Rational[2];

rats[0] = new Rational(1,2);

rats[1] = rats[0];

rats[1].setDenominator(3);

Tehát egy referenciákat tároló tömbben ugyan az az objektum akár többször is szerepelhet. Ha ez befolyásolná a működést érdemes dokumentálni:

/\*\*

...

PRE: rats != null and (i!=j => rats[i] != rats[j])

...

\*/

# Funkcionális elemek:

A -> operátorral lehetőségünk van lambda függvényeket definiálni.

int[] nats = new int[1000];

java.util.Arrays.setAll(nats, i->i);

java.util.Arrays.setAll(nats, i->(int)(100\*Math.random()));

Akár több paraméterű lambdát is írhatunk, ez például egy hossz alapú rendezés.

public static void main( String[] args ){

java.util.Arrays.sort(args);

java.util.Arrays.sort( args, (s,z) -> s.length()-z.length() );

}

Meglepően sok a Haskell-ből már megismert dologgal találkozhatunk.

int[] nums = new int[1000];

java.util.Arrays.setAll(nums, i->(int)(100\*Math.random()));

java.util.Arrays.stream(nums)

.filter( i -> i%2 == 0 )

.map( i -> i/2 )

.limit(10)

.forEach( i -> System.out.println(i) )

Még a részleges alkalmazásra is van lehetőség:

java.util.Arrays.stream(nums).forEach( i -> System.out.println(i) )

java.util.Arrays.stream(nums).forEach( System.out::println )