Type dependency in Java

Tipuri de date in Java:

1. Primitive:
   - Numere intregi: byte(8 biti), short(16 biti), int(32 biti), long(64 biti)
   - Numere reale: float, double
   - Logice: boolean
   - Character: char(16 bit Unicode)
2. Referinte (descendente din clasa Object)
   - Clase
   - Interfete
   - Array-uri

Valori implicite :

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Compatibilitate

În programarea orientată pe obiecte, compatibilitatea se referă la o relație direcționată între tipuri. Spunem că două tipuri sunt compatibile în Java dacă este posibil să se transfere date între variabile ale tipurilor. Transferul de date este posibil dacă compilatorul îl acceptă și se face prin atribuire sau transferul parametrilor.

De exemplu, short este compatibil cu int deoarece atribuirea intVariable = shortVariable este posibila. Dar boolean nu este compatibil cu int deoarece atribuirea intVariable = booleanVariable nu e posibila; compilatorul nu o va accepta.

Deoarece compatibilitatea este o relație direcționată, uneori T1 este compatibil cu T2, dar T2 nu este compatibil cu T1 sau nu în același mod. Vom

vedea acest lucru mai departe când vom discuta despre compatibilitatea explicită sau implicită.

Ceea ce contează este că compatibilitatea între tipurile referință este posibilă numai într-o ierarhie de tipuri. Toate tipurile de clase sunt compatibile cu Object, de exemplu, deoarece toate clasele moștenesc implicit de la Object.

Totuși, Integer nu este compatibil cu Float, deoarece Float nu este o super-clasă a Integer. Integer este compatibil cu Number, deoarece Number este o super-clasă (abstractă) a clasei Integer. Deoarece sunt situate în același tip de ierarhie, compilatorul acceptă atribuirea :

numberReference = integerReference;

Vorbim despre compatibilitatea *implicită* sau *explicită*, în situatiile in care compatibilitatea trebuie să fie marcată explicit sau nu. De exemplu, short este implicit compatibil cu int, dar nu si invers:

atribuirea shortVariable = intVariable nu e posibila. Cu toate acestea, short este în mod **explicit compatibil** cu int, deoarece atribuirea shortVariable = (short) intVariable este posibila (***type cast***).

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

În mod similar, pt. ***tipurile de referință***:

integerReference = numberReference // NU

integerReference = (Integer) numberReference // DA.

Prin urmare, Integer este implicit compatibil cu Number, dar Number este doar explicit compatibil cu Integer.

Dependenţă

Un tip ar putea depinde de alte tipuri. De exemplu, tipul matricei int [] depinde de tipul primitiv int. În mod similar, tipul generic ArrayList <Customer> este dependent de tipul Customer. Metodele pot fi, de asemenea, dependente de tip, în funcție de tipul parametrilor lor. De exemplu, metoda void increment (Integer i); depinde de tipul Integer. Unele metode (cum ar fi unele tipuri generice) depind de mai multe tipuri - cum ar fi metodele care au mai mult de un parametru.

Covarianța și contravarianța. Invarianța

Covarianța înseamnă că compatibilitatea a două tipuri implică compatibilitatea tipurilor dependente de acestea. Având în vedere compatibilitatea tipurilor, se presupune că tipurile dependente sunt covariante

Compatibilitatea dintre T1 și T2 implică compatibilitatea dintre A (T1) și A (T2). Tipul dependent A (T) se numește covariant; sau mai exact, A (T1) este covariant la A (T2).

In Java, daca avem Number[] `numberArray` si Integer[] `integerArray` atunci, deoarece atribuirea `numberArray` = `integerArray` este posibila (cel puțin în Java), tipurile array (matrice) Integer [] și Number [] sunt covariante.

Invarianța : tipurile sunt incompatibile la atribuie

Pentru array-uri in Java:

1. Arrays of **primitive types** are invariant in Java:

```
longArray = intArray; // type error
shortArray = (short[])intArray; // type error
```

2. Arrays of **reference types** are *implicitly covariant* and *explicitly contravariant*, however:

```
SuperType[] superArray;

SubType[] subArray;

    ...

superArray = subArray; // implicit covariant
```

```
subArray = (SubType[])superArray; // explicit contravariant
```

Exemplu:

```
Object[] objectArray; // array reference

objectArray = new String[3]; // Ok-covarianta implicita

objectArray[0] = new Integer(5); // throws ArrayStoreException
```

⇨ Eroarea este detectata doar la Runtime !

Covariant return type in Java

Semnatura unei metode = În Java, o semnătură de metodă face parte din declarația de metodă. Este combinația dintre **numele metodei** și **lista parametrilor.**

Overriding (Suprascrierea) este mecanismul de rescriere a unei metode virtuale (publica sau protected) într-o clasă derivată. Metoda din clasa derivata o suprascrie pe cea din clasa parinte daca are aceeasi semnatura . In plus, tipul de return trebuie sa fie acelasi ca al metodei sprascrise, SAU:

Covariant return type in JAVA expects that the overriding method in sub-class can have different return data type and not the same return type as that of parent class method, provided that the return type of the overriding method in the sub-class is a sub-type of Parent's return type.