

Java: Il sistema dei tipi

I tipi wrapper

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli "Federico II"

Corso di Linguaggi di Programmazione I

- Il linguaggio Java è **staticamente tipizzato**
- "tipizzato" vuol dire che ad ogni espressione viene assegnato un tipo, che rappresenta l'insieme di valori che l'espressione potrebbe assumere
- "staticamente" vuole dire che il tipo di ogni espressione deve essere noto al momento della compilazione
- In tal modo, il **compilatore** è in grado di controllare che tutte le operazioni (compresa, ad esempio, l'assegnazione) siano applicate ad operandi **compatibili**
- Questa fase della compilazione è chiamata appunto "*type checking*"

- I **tipi base** (o *primitivi*) sono otto:
 - boolean
 - char
 - Tipi interi: byte (8 bit), short (16), int (32), long (64)
 - Tipi in virgola mobile: float (32) e double (64)
- Inoltre, void è un tipo speciale usato solo come tipo di ritorno dai metodi
- Tra tipi base esistono le seguenti **conversioni implicite** (o **promozioni**):
 - Da byte a short, da short a int, da int a long, da long a float e da float a double
 - Da char a int
- Le conversioni implicite sono transitive
- Se esiste una conversione implicita dal tipo x al tipo y, è possibile assegnare un valore di tipo x ad una variabile di tipo y (ci torneremo con la *relazione di assegnabilità*)
- Si noti come **non** ci sono conversioni da e per il tipo boolean

Alcune conversioni implicite possono comportare una *perdita di informazione*

- Ad esempio:

```
int i = 1000000001; // un miliardo e uno
float f = i;
```

Dopo queste istruzioni, f contiene un miliardo, perché un float *non ha abbastanza bit di mantissa* per rappresentare le 10 cifre significative di i

Le conversioni a rischio sono quella da int a float e, a maggior ragione, da long a float

Inserire una dichiarazione e inizializzazione per la variabile x, in modo che il seguente ciclo sia infinito:

```
while (x == x+1) {  
    ...  
}
```

float x = 1.000.000.000

Come detto nella slide precedente, $x + 1 = 1.000.000.001$ che viene riportato ad $1.000.000.000$ per via del tipo float $\Rightarrow 1.000.000.000 = 1.000.000.001$

- A differenza di altri linguaggi orientati agli oggetti (ad es., il C++), non esistono variabili che contengono oggetti, solo *riferimenti* ad oggetti, ovvero variabili che contengono l'*indirizzo* di un oggetto
- I riferimenti Java sono quindi simili ai puntatori del linguaggio C
- Tuttavia, i riferimenti Java sono molto più restrittivi
 - Niente aritmetica dei puntatori (`p++`) e conversioni tra puntatori e numeri interi
 - Niente doppi puntatori
 - Niente puntatori a funzioni
- Quindi, rispetto ai puntatori, i riferimenti Java sono meno potenti, più facili da utilizzare e meno soggetti ad errori al run-time
- Java prevede solo il passaggio per *valore*: sia i tipi base che i riferimenti sono passati per valore
- Non è possibile passare oggetti per valore, l'unico modo di manipolare (ed in particolare, passare) oggetti è tramite i loro riferimenti (ovvero, indirizzi)

- In virtù del polimorfismo, bisogna distinguere due tipi di ogni variabile ed espressione di categoria “riferimento”:

```
Animal a = new Dog("Lilli");
```

- Animal è il tipo **dichiarato** (o statico) del riferimento “a”
- Il tipo dichiarato è noto a tempo di compilazione e rimane invariato in tutto l’ambito di validità di quel riferimento
- Dog è il tipo **effettivo** (o dinamico) del riferimento “a”, in questo momento
- Il tipo effettivo non è noto a tempo di compilazione, nel senso che il compilatore non tenta di determinarlo, perché sarebbe in generale impossibile (si pensi a un parametro di un metodo)
- Il tipo effettivo può cambiare ogni volta che si assegna un nuovo valore a quel riferimento

- Java prevede i seguenti tipi:
 - Gli 8 tipi base, o *primitivi*
 - I tipi riferimento
 - Tra cui: i tipi array
 - Il tipo speciale “nullo”
- I tipi base sono già stati introdotti
- I tipi riferimento corrispondono a classi, interfacce, enumerazioni, o array
- I tipi array (caso particolare di riferimento) sono tipi composti, ovvero si definiscono a partire da un altro tipo, detto tipo “componente”, e si individuano sintatticamente per l'uso delle parentesi quadre
- Il tipo nullo prevede come unico valore possibile la costante “null”

Esempio di type checking

- Proviamo a simulare la fase di type checking del compilatore Java sul seguente frammento di codice:

```
int n;  
double d;  
d = (new Object()).hashCode() + n/2.0;
```

The diagram shows the type checking process for the assignment statement. It starts from the right side of the assignment operator (=) and moves left. A blue curved arrow points from the division operation (n/2.0) to the addition operation (+). Above the arrow, the text "int + double -> double" indicates the result type of the addition. Another blue curved arrow points from the addition operation to the multiplication operation (n/). Above this arrow, the text "int * double -> double" indicates the result type of the multiplication. A third blue curved arrow points from the multiplication operation to the object creation (new Object()). Above this arrow, the text "double * int -> int" indicates the result type of the multiplication. Finally, a fourth blue curved arrow points from the object creation to the variable d, with the text "int -> double" indicating the assignment type.

- Cominciamo dalla parte destra dell'assegnazione e procediamo dalle sottoespressioni più piccole via via fino all'intera parte destra
- La prima espressione base che incontriamo è "new Object()"
 - questa espressione è per definizione di tipo "riferimento alla classe Object", o per brevità, di tipo "Object"
- Passiamo poi all'espressione "(new Object()).hashCode()"
 - in questo contesto il punto denota la chiamata ad un metodo
 - quindi, il tipo dell'espressione è il **tipo di ritorno** del metodo, in questo caso "int"
- Esercizio: quali altri significati del punto ci sono in Java?

```
int n;  
double d;  
  
d = (new Object()).hashCode() + n/2.0;
```

- L'espressione "n" ha chiaramente tipo "int", mentre "2.0" è una costante di tipo "double"
- Quindi, i due operandi della divisione "n/2.0" hanno tipo diverso
 - il primo operando viene quindi promosso da int a double tramite *conversione implicita*
 - complessivamente, la divisione ha a sua volta tipo double
- Infine, per lo stesso motivo la somma avrà tipo double
- A questo punto, il compilatore verifica che il tipo calcolato per il lato destro dell'assegnazione sia compatibile (si veda dopo) con il tipo del lato sinistro (d)
- In questo caso, i due tipi coincidono esattamente
- Per completezza, anche l'intera assegnazione ha tipo "double"; questo consente di concatenare le assegnazioni, come in `a=b=c`
- Le regole di type checking sono specificate nel capitolo 15 della definizione del linguaggio Java (Java Language Specification)

- Per permettere il polimorfismo (in particolare, la capacità di un riferimento di puntare ad oggetti di tipo diverso), esiste una relazione binaria tra tipi, chiamata **relazione di sottotipo**
- La relazione di sottotipo è definita dalle seguenti regole, in cui T ed U rappresentano tipi arbitrari, *esclusi i tipi base*:
 - 1) T è sottotipo di se stesso
 - 2) T è sottotipo di Object
 - 3) Se T estende U oppure implementa U, T è sottotipo di U
 - 4) Il tipo nullo è sottotipo di T
 - 5) Se T è sottotipo di U allora T[] è sottotipo di U[]
- La relazione di sottotipo non coinvolge i tipi base
- E' facile verificare che la relazione di sottotipo è riflessiva, antisimmetrica e transitiva
 - pertanto, essa è una relazione d'ordine sull'insieme dei tipi (non base)
- Queste regole non tengono conto dei tipi parametrici introdotti da Java 1.5, di cui si parlerà successivamente

La relazione di sottotipo permette di definire precisamente il comportamento dell'operatore *instanceof*

Data un'espressione *exp* ed il nome di una classe o interfaccia *T*, l'espressione

exp instanceof T

restituisce *vero* se e solo se il tipo **effettivo** di *exp* **non è nullo ed è sottotipo di T**

Nota: il primo argomento di *instanceof* deve essere un'espressione di categoria "riferimento", pena un errore di compilazione

- La relazione di *compatibilità* (o *assegnabilità*) tra tipi stabilisce quando è possibile assegnare un valore di un certo tipo T ad una variabile di tipo U
- Si dice che T è **assegnabile** ad U se
 - T è sottotipo di U, oppure
 - T ed U sono tipi base e c'è una conversione implicita da T ad U

La relazione di assegnabilità si applica nei seguenti contesti:

- Assegnazione: $a = \text{exp}$
- Chiamata a metodo: $x.f(\text{exp})$
- Ritorno da metodo: `return exp`

- In base alla definizione di sottotipo, un array di qualunque tipo è sottotipo di “array di Object”
- Questo consente, ad esempio, di passare qualunque array ad un metodo che abbia come parametro formale un array di Object
- Consideriamo il seguente esempio:

```
String[] arr1 = new String[10];
Object[] arr2 = arr1;
arr2[0] = new Object();
String s = arr1[0];
```

```
String[] arr1 = new String[10];
Object[] arr2 = arr1;
arr2[0] = new Object();
String s = arr1[0];
```

- L'esempio risulta corretto per il compilatore
- Tuttavia, nell'ultima istruzione assegnamo alla variabile "s", dichiarata String, un oggetto di tipo effettivo "Object", che non è compatibile
- In effetti, al run-time viene sollevata un'**eccezione** (ArrayStoreException) al momento della **terza** istruzione
- Questo perché al run-time gli array "ricordano" il tipo con il quale sono stati creati
- La JVM utilizza questa informazione per controllare che gli oggetti inseriti nell'array siano sempre di tipo compatibile con quello dichiarato in origine

Ad ogni modifica ad un array di riferimenti:

```
a[i] = exp
```

la JVM controlla che:

- il tipo effettivo di `exp`
- sia sottotipo
- del tipo con cui è stato creato l'array.

In caso contrario, la JVM solleva l'eccezione *ArrayStoreException*

I cast

- Java permette alcune conversioni esplicite di tipo tramite *cast*
 - Anche dette *coercizioni di tipo*
- La sintassi è:
$$(T) \ exp$$

- Analizziamo prima i cast tra tipi primitivi
- Si può utilizzare un cast per effettuare esplicitamente una **promozione**
 - in questo caso il cast è superfluo
- Si può utilizzare un cast per effettuare una **promozione al contrario**
 - ad esempio, da double a int
 - in questi casi, è facile incorrere in *perdite di informazioni*
 - ad esempio, nel passaggio da numeri in virgola mobile a numeri interi, si può perdere **sia in precisione che in magnitudine** (ordine di grandezza)
 - i dettagli sono definiti nella sezione 5.1.3 del JLS: Narrowing Primitive Conversion
 - queste conversioni sono decisamente sconsigliate, al loro posto è opportuno utilizzare i metodi appropriati della classe Math (come Math.round)

- Sono **consentiti** dal compilatore i seguenti cast tra un tipo riferimento (o array) A ad un tipo riferimento (o array) B:

1) se B è **supertipo** di A

- si chiama “upcast”
- è superfluo, perché i valori di tipo A sono di per sé assegnabili al tipo B

2) se B è **sottotipo** di A

- si chiama “downcast”
- **al run-time, la JVM controlla che l'oggetto da convertire appartenga effettivamente ad una sottoclasse di B**
- in caso contrario, viene sollevata l'eccezione ClassCastException
- si deve cercare di evitare i downcast, perché aggirano il type checking svolto dal compilatore
- a tale scopo, i tipi parametrici introdotti da Java 1.5 possono aiutare
- se proprio si deve usare un downcast, esso andrebbe preceduto da un controllo instanceof, che assicuri la correttezza della conversione

3) se nessuno dei due è sottotipo dell'altro...vedere la prossima slide

- Consideriamo un cast dal tipo A al tipo B, ovvero:

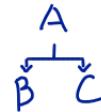
A a = ...

B b = (B) a;

- Supponiamo che A e B non siano legati dalla relazione di sottotipo (un *sidecast*)
 - Se nessuna tra A e B è un'interfaccia, il cast dà errore di compilazione
 - Perché è impossibile che abbia successo a runtime!
 - Non esistono oggetti che sono sottotipi di A e B simultaneamente
 - Si ricordi che una classe non può estendere simultaneamente due classi diverse
- Se almeno una tra A e B è un interfaccia, il cast è lecito in compilazione
 - A runtime, può avere successo o fallire

Sapendo che sia la classe C sia la classe B estendono A, effettuare il type checking del seguente codice, evidenziando:

- errori di tipo
- cast non validi (errori di compilazione)
- cast validi ma potenzialmente pericolosi al run-time



```

boolean f(A a, B b) {
    C c = (C) a;
    A a1 = (A) b;
    Object o = a;           Superfluo, B e' sottotipo di A
    A[] arr = new A[10];
    arr[5] = (Object) a;    arr[5] e' d. tipo A
                            A = Object non e' valido
    arr[6] = b;             cast implicito, OK
    return a == c;          <- sempre vero (se non fosse per l'errore di compilazione)
}
  
```

I tipi Wrapper

- Per ogni tipo base, Java offre una corrispondente classe, che ingloba (*wraps*) un valore di quel tipo in un oggetto
- Queste classi, dette appunto wrapper, servono a trattare i valori base come se fossero oggetti
 - ad esempio, per inserirli nelle strutture dati offerte dalla Java Collection Framework (vedi lezioni seguenti)
- Le classi wrapper sono:
 - Byte, Short, Integer, Long, Float, Double
 - Boolean
 - Character
 - Void
- Come si vede, sono tutte omonime del rispettivo tipo base, tranne Integer, Character, e Void (perché formalmente void non è un tipo base)

Caratteristiche base delle classi wrapper

- Tutte le classi wrapper sono **immutabili** e **final** (come le stringhe)
I
- **Immutabile:** non è possibile modificare il contenuto di un oggetto
- **Final:** non è possibile creare sottoclassi

I

- Ogni classe wrapper ha un metodo statico **valueOf** che prende come argomento un valore del tipo base corrispondente alla classe e restituisce un oggetto wrapper che lo ingloba
- Esempio:

```
Integer n = Integer.valueOf(3);  
Double x = Double.valueOf(3.1415);
```

- Perché un metodo statico invece di un costruttore?
- A differenza di un costruttore, l'oggetto restituito da valueOf *non è necessariamente nuovo*
 - ovvero, il metodo valueOf cerca di riciclare gli oggetti già creati (*caching*)
 - questo non è un problema, perché gli oggetti wrapper sono immutabili

- Le sei classi wrapper relative ai tipi numerici estendono la **classe astratta Number**
- La classe Number prevede sei metodi, che estraggono il valore contenuto, convertendolo nel tipo base desiderato

```
public byte    byteValue()
public short   shortValue()
public int     intValue()
public long    longValue()
public float   floatValue()
public double  doubleValue()
```

- Se un oggetto wrapper viene convertito in un valore base verso il quale non c'è una conversione implicita (ad es., da double a int), l'effetto sarà lo stesso di quello di un cast

- Fino alla versione 1.4 di Java, era necessario convertire esplicitamente i valori dei tipi base in oggetti e viceversa
- A partire dalla versione 1.5, questo procedimento è stato automatizzato, introducendo l'**autoboxing** e l'**auto-unboxing**
- Grazie a queste funzionalità, il **compilatore** si occupa di inserire le istruzioni di conversione laddove queste siano necessarie

- Ad esempio, l'istruzione

```
Integer n = 7;
```

viene convertita in:

```
Integer n = Integer.valueOf(7);
```

- Allo stesso modo, le istruzioni:

```
Integer n = 7;
```

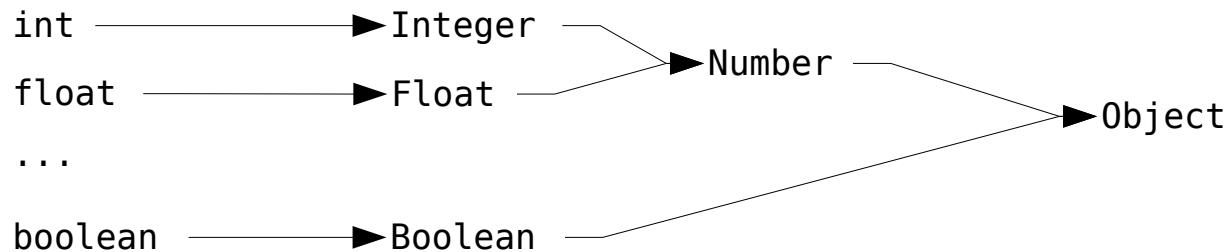
```
Integer i = n + 7;
```

vengono convertite in:

```
Integer n = Integer.valueOf(7);
```

```
Integer i = Integer.valueOf(n.intValue() + 7);
```

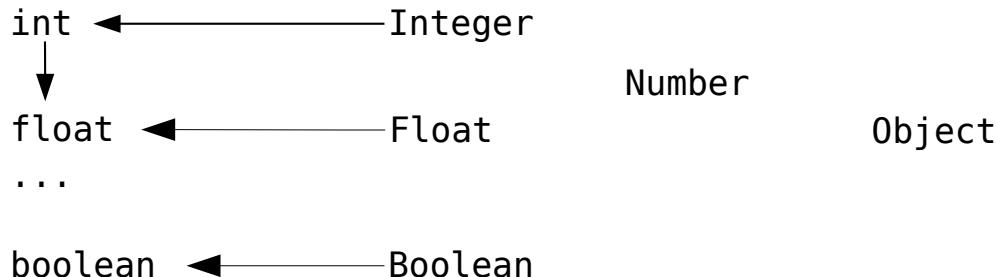
- L'autoboxing può convertire un'espressione di tipo primitivo in un oggetto del tipo wrapper corrispondente, o di un suo supertipo
- Quindi, l'autoboxing può effettuare le seguenti trasformazioni:



- L'autoboxing *non* prende in considerazione le conversioni implicite
- Ad esempio, ciascuna delle seguenti istruzioni provoca un **errore di compilazione**, perché oltre all'autoboxing richiederebbe anche una conversione implicita di tipo (promozione):

```
Double x = 1;
Double y = 1.0f;
Integer n = (byte) 1;
```

- L'auto-unboxing può essere seguito da una promozione



- Ad esempio, le seguenti istruzioni sono corrette:

```
Integer i = 7;      // autoboxing
double d = i;       // auto-unboxing seguito da promozione
```

- L'auto-unboxing di un'espressione che a runtime risulta **null** comporta il lancio di un'eccezione (verificata o non verificata?)

- L'operatore “==” può dare risultati sorprendenti se applicato ai tipi wrapper
- Infatti, è facile dimenticare che si tratta di un **confronto tra riferimenti**, come per tutti gli oggetti
- Ad esempio:

```
Integer a = new Integer(7), b = new Integer(7);  
System.out.println(a==b); // false, sono oggetti diversi
```

```
Integer a = 7, b = 7;  
System.out.println(a==b); // true, perché viene chiamato il metodo statico valueOf,  
                         che riutilizza gli oggetti corrispondenti a valori piccoli
```

```
Integer a = 700, b = 700;  
System.out.println(a==b); // false, perché il metodo valueOf riutilizza soltanto  
                         gli interi compresi tra -127 e 127
```

*Per questioni di
dimensioni della cache*

- Morale: confrontare i tipi wrapper con **equals** e non con “==”