Programmazione Java

Tipi, Variabili e scope, Operatori, Conversione e Promozione

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

davide.diruscio@univaq.it

Sommario

- Blocco e statement
- Identificatore e Letterale
- Tipi
  - Primitivi
  - Reference
- Variabili e scope
- Operatori
- Conversione e Promozione

Statements

» *Statement* una o più linee di codice terminante con un `;`

– Esempi

- `result = x * y;`
- `x = new Point();`
-
```
int x = 1 + 3 *
        6 + (12 + 5);
```

Blocchi

- Collezione di statements oppure dichiarazione locale di classi oppure dichiarazione di variabili locali racchiusi tra parentesi graffe
- Utilizzati nella definizione di classi, metodi, `if`, `while`, `for`, …
- E' possibile annidare blocchi
- Esempio

```
{
        a = b + c;
        d += 10;
}
```

Identificatore

- Sequenza di lunghezza non limitata di *lettere* e *cifre* la cui prima cifra deve essere una lettera oppure _ o `$`
- Vengono utilizzati per i nomi delle variabili, delle classi, dei metodi e sono **case-sensitive**
- Non identificatori le keyword e i letterali `true`, `false` e `null`
- Esempi
  - Prodotto
  - prodotto
  - $userName

Keyword riservate

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Letterale (1)

- Porzione di codice che rappresenta il valore di un tipo primitivo, Stringa oppure `null`
- Tipi
  - Intero
  - Floating
  - Booleano: `true` o `false`
  - Carattere
  - String
  - `null`

Letterale: Intero (2)

- Tipo è `int`
- Se utilizzato come suffisso `L` oppure `l` diventa `long`
- Espresso
  - decimali (base 10): numero
  - ottale (base 8): `0`numero
  - esadecimale (base 16): `0x`numero
- Esempi
  - Decimale `int`: `10`
  - Decimale `long`: `10L`
  - Ottale `int`: `010`
  - Esadecimale `int`: `0x10`

Letterale: Intero (3)

- Più grande letterale decimale di tipo `int` è `2147483648` (2^{31})
- Da 0 to `2147483647` un letterale `int` può apparire
- `2147483648` può apparire soltanto come numero negativo
- Più grande decimale letterale di tipo `long` è `9223372036854775808L` (2^{63})

Letterale: Intero (4)

- Più grande letterale **positivo** di tipo `int` esadecimale ed ottale sono `0x7fffffff` e `017777777777` rispettivamente (`2147483647`)
- Più grande letterale **negativo** di tipo `int` esadecimale ed ottale sono `0x80000000` e `020000000000` che rappresentano `-2147483648`
- `0xffffffff` e `037777777777` rappresentano -1 in esadecimale ed ottale rispettivamente

Letterale: Floating (5)

» Composto da

- Parte numero intero
- Punto decimale (.)
- Parte frazionaria
- Esponente: `E` oppure `e` seguito da un intero con segno
- Suffisso
  - `F` oppure `f` `float`
  - `D` oppure `d` `double` (default)

» Esempi

- Double: `1e1, 2., .3, 0.0, 3.14`
- Float: `1e1f, 2.f, .3f, 0f, 3.14f`

Letterale: Floating (6)

- Letterale positivo float più grande è `3.40282347e+38f`
- Letterale positivo float finito non zero più piccolo è `1.40239846e-45f`
- Letterale positivo double più grande è `1.79769313486231570e+308`
- Letterale positivo double finito non zero più piccolo è `4.94065645841246544e-324`

Letterale: Carattere (7)

- Espresso come un carattere o una sequenza di *escape* racchiusa tra apici singoli
- Tipo è sempre `char`
- Esempi
  - `'c'`
  - `'\\'`
  - `'\n', '\t', '\b', '\r', '\'', '\"'`

Letterale: String (8)

- Espresso come una sequenza di zero o più caratteri racchiusi tra doppi apici
- Ogni carattere può essere rappresentato con una sequenza di escape
- Esempi
  - `"welcome"`
  - `"\""`
  - `"\n"`
  - `"ciao" + "ciao"`

(Vedere Literals.java)

Tipi di dato

- Consente di esprimere la natura del dato
- Indica il modo con cui verrà interpretata la sequenza di bit che rappresenta il dato
- La stessa sequenza può rappresentare un intero o un carattere
- Determina il campo dei valori che un dato può assumere
- Specifica le operazioni possibili sui dati

Tipi di dato

- Java è un linguaggio ***fortemente tipato***
- Il tipo di ogni **variabile** o espressione può essere identificato leggendo il programma ed è già noto al momento della compilazione
- E' obbligatorio dichiarare il tipo di una variabile prima di utilizzarla
- Dopo la dichiarazione non è possibile assegnare alla variabile valori di tipo diverso

Tipi

» Ogni tipo di dato ha

- Un *nome*
  - `int, double, char`
- Un *insieme di valori* letterali possibili
  - `3, 3.1, 'c'`
- Un *insieme di operazioni* lecite
  - `+, *, /, %, ……`

» Java ha

- Tipi Primitivi
- Tipi Reference

Tipi primitivi

- Logici
  - `boolean`
- Numerici
  - Integrale
    - `byte`, `short`, `int`, `long` e `char`
  - Floating
    - `double` e `float`

Tipi primitivi: boolean

- Valore `boolean` rappresenta condizione di verità o falsità
- Variabile di tipo `boolean` può rappresentare un valore a due stati
  - come un interruttore che è ***acceso*** o ***spento***
- Letterali `true` e `false` sono unici valori ammessi
- Esempio
  - `boolean b = false;`

Tipi primitivi: char

- Rappresentato mediante schema di codifica Unicode
  - jdk1.1.7 utilizza 2.1
  - jdk1.4 utilizza 3.0
  - jdk5.0 utilizza 4.0
- E’ composto da 2 byte senza segno (65535)
- Primi 128 caratteri della codifica Unicode sono caratteri ASCII

Tipi primitivi: char

Caratteri	Nome	Valore Unicode
\b	Backspace	\u0008
\t	Tabulazione	\u0009
\n	Avanzamento riga	\u000a
\r	Invio a capo	\u000d
\”	Doppi apici	\u0022
\’	Apici singoli	\u0027
\\	Backslash	\u005c

Tipi primitivi: byte, short, int e long

» Rappresentazione avviene mediante notazione in complemento a due

Tipo	Dimensione	Range
byte	1 byte	-128 a 127
short	2 byte	-32768 a 32767
int	4 byte	-2147483648 a 2147483647
long	8 byte	-9223372036854775808 a 9223372036854775807

Tipi primitivi: float e double

- Definiscono numeri con parti frazionarie
- `double` vengono detti numeri a doppia precisione
- Numeri adottano le specifiche IEEE 754 che indicano anche
  - Infinito positivo e negativo (`Float.POSITIVE_INFINITY, Float.NEGATIVE_INFINITY` stessa cosa per double)
  - NaN ovvero *not a number* (`Float.NaN`)

Tipo	Spazio Memoria	Intervallo
`float`	`4 byte`	`±3,40282347E+38F` (6 o 7 cifre decimali significative)
`double`	`8 byte`	`±1,79769313486231570E+308` (15 cifre decimali significative)

Tipi Reference

- Costituiscono dei *puntatori* (reference) a degli oggetti
- Tre tipi
  - Classe
    - `Point p`
  - Interfaccia
    - `Comparable c1`
    - `Comparator c2`
  - Array
    - `int[] a1`
    - `Point[] p1`

Variabile (1)

- Una variabile è un'area di memoria ed ha associato un tipo che può essere di tipi *primitivo* o *reference*
- Viene identificata mediante l'**identificatore**
- Una variabile contiene sempre un valore che è compatibile con il suo tipo
- Tipi
  - Variabile di istanza
  - Variabile di classe (`static`)
  - **Variabile locale**
  - Componenti dell'array
  - Parametri dei metodi e dei costruttori
  - Parametro di un exception-handler: `catch(Exception e)`
  - Variabili final (`final`)

Variabile (2)


```
class Point {
  static int numPoints;     //numPoints is a class variable
  int x, y;                 // x and y are instance variables
  int[] w = new int[10];    // w[0] is an array component
  int setX(int x) {         // x is a method parameter
       int oldx = this.x;    // oldx is a local variable
       this.x = x;
       return oldx;
  }
}
```

Variabile: Variabile locale

» Variabile locale deve essere **esplicitamente** inizializzata prima di essere utilizzata, ovvero deve trovarsi alla sinistra dell'operatore di assegnamento

» Esempi

1)

```
{
  int k;
  while (true) {
       k = n;
       if (k >= 5) break;  //OK
       n = 6;
  }
  System.out.println(k);
}
```

Variabile: Variabile locale

2)

```
{
  int k;
  while (n < 4) {
       k = n;
       if (k >= 5) break;
       n = 6;
  }
  System.out.println(k); //NOT OK
}
```

3)

```
{
  int k;
  int n = 5;
  if (n > 2)
       k = 3;
  System.out.println(k); //NOT OK
}
```

(Vedere LocalVariable.java e VarInit.java)

Scope

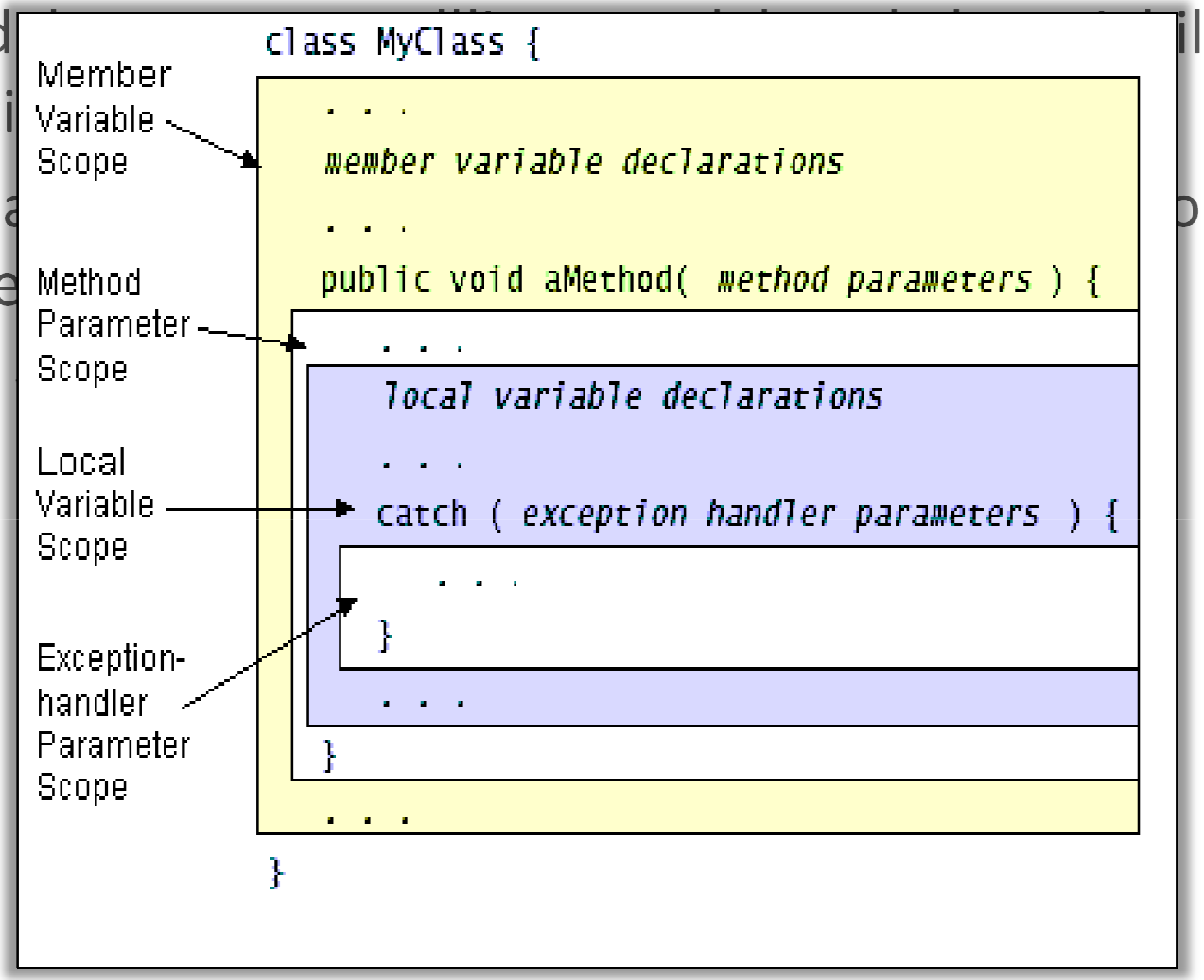
- Regione del programma all'interno del quale la variabile può essere riferita utilizzando il suo nome
- Determina quando il sistema crea e distrugge le memoria necessaria a contenere la variabile
- Scope **<>** visibilità (modificatori di accesso)

Scope

» Regione d ile può essere riferita uti

» Determina oria necessaria a contenere

» Scope <>



Operatori

- E’ una funzione che agisce su uno, due o tre operandi
- Operatori che richiedono un operando sono detti unari (es.: i++)
- Operatori che richiedono due operando sono detto binari (es.: a + b)
- Operatori che richiedono tre operandi sono detti ternari: ? : (forma particolare di if-else)
- Tipi
  - Aritmetici
  - Relazionali e condizionali
  - Shift e logici
  - Assegnamento
  - Altri

Operatori Aritmetici (1)

Operatore	Uso	Descrizione
+	op1 + op2	Addizione
-	op1 - op2	Sottrazione
*	op1 * op2	Moltiplicazione
/	op1 / op2	Divisione
%	op1 % op2	Calcola il resto

Operatori Aritmetici (2)

- Divisione tra interi restituisce un intero ovvero viene troncato
  - Esempio: 15 / 2 = 7
- Altrimenti divisione tra numeri in virgola mobile
  - Esempio: 15.0 / 2 = 7.5
- Divisione intera per zero genera eccezione (ArithmeticException)
- Divisione virgola mobile da come risultato infinito ovvero Double.POSITIVE_INFINITY (o negativo)

Operatori Aritmetici (3)

Operatore	Uso	Descrizione
++	op++	Post-incremento
++	++op	Pre-incremento
--	op--	Post-decremento
--	--op	Pre-decremento

Operatori Aritmetici (4)

» Esempi

> Pre e post incremento e decremento

```
public class AutoInc {

  public static void main(String[] args) {
    int i = 1;
     System.out.println("i : " + i);
     System.out.println("++i : " + ++i); // Pre-increment
     System.out.println("i++ : " + i++); // Post-increment
     System.out.println("i : " + i);
     System.out.println("--i : " + --i); // Pre-decrement
     System.out.println("i-- : " + i--); // Post-decrement
     System.out.println("i : " + i);

  }

}
```

Output

```
"i : 1",
"++i : 2",
"i++ : 2",
"i : 3",
"--i : 2",
"i-- : 2",
"i : 1“
```

(Vedere IncDec.java)

Operatori Relazionali (1)

- Generano sempre un risultato di tipo boolean
- Applicato ai tipi primitivi tranne == e != che può essere applicato a tutti gli oggetti
  - Nel caso di oggetti viene confrontato il valore del reference della variabile
- Esempi
  - 3 > 5
  - a == b (a e b sono di tipo Point)

Operatori Relazionali (2)

Operatore	Uso	Descrizione
>	op1 > op2	op1 è maggiore di op2
>=	op1 >= op2	op1 è maggiore o uguale di op2
<	op1 < op2	op1 è minore di op2
<=	op1 <= op2	op1 è minore o uguale di op2
==	op1 == op2	op1 è uguale di op2
!=	op1 != op2	op1 è diverso di op2

Operatori Relazionali (3)


```
public class Equivalence {

    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);

    }

}
```

Operatori Relazionali (3)


```
public class Equivalence {

   public static void main(String[] args) {
      Integer n1 = new Integer(47);
      Integer n2 = new Integer(47);
      System.out.println(n1 == n2);
      System.out.println(n1 != n2);

   }

}
```

Output

```
"false"

"true“
```

(Vedere Equivalence.java e Equivalence2.java)

Operatori Logici(1)

» Generano sempre un risultato di tipo boolean

» Gli argomenti sono di tipo boolean

» Esempio

```
String s = “Welcome”;
if ( (s!=null) && (s.length()!=10)) {
   System.out.println(s + “ World”);
}
```

Operatori Logici(2)

Operatore	Uso	Descrizione
&& (short-circuit)	op1 && op2	`true` se op1 e op2 sono `true`. op2 viene valutata se op1 è `true`
|| (short-circuit)	op1 || op2	`true` se op1 oppure op2 sono `true`. op2 viene valutata se op1 è `false`
!	! op1	`true` se op1 è `false`. `false` se op1 è `true`.
&	op1 & op2	`true` se op1 e op2 sono `true`. op2 viene comunque valutata
|	op1 | op2	`true` se op1 oppure op2 sono `true`. op2 viene comunque valutata
^	op1 ^ op2	`true` se op1 oppure op2 sono `true` ma non ambedue

Operatori Logici(3)


```
import java.util.*;

public class Bool {

  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);


    System.out.println("i = " + i);
    System.out.println("j = " + j);
    System.out.println("i > j is " + (i > j));
    System.out.println("i < j is " + (i < j));
    System.out.println("i >= j is " + (i >= j));
    System.out.println("i <= j is " + (i <= j));
    System.out.println("i == j is " + (i == j));
    System.out.println("i != j is " + (i != j));
    System.out.println("(i < 10) && (j < 10) is “
       + ((i < 10) && (j < 10)) );
    System.out.println("(i < 10) || (j < 10) is “ + ((i < 10) || (j <
10)) );

  }

}
```

Operatori Logici(4): short circuit


```
public class ShortCircuit {

  static boolean test1(int val) {
   System.out.println("test1(" + val + ")");
    System.out.println("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
    System.out.println("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
  }


  public static void main(String[] args) {
    if(test1(0) && test2(2) && test3(2))
      System.out.println("expression is true");
    else
      System.out.println("expression is false");
  }
}
```

Lo short circuit permette di ottenere un potenziale miglioramento delle prestazioni qualora non sia necessario valutare tutte le parti di un'espressione logica

(Vedere ShortCircuit.java)

Operatori Bitwise (1)

- Vengono applicati a numeri di tipo intero
- Permettono l'accesso diretto alla rappresentazione binaria dei dati
- Effettuano operazioni dell'algebra booleana sulle coppie di bit corrispondenti nei due argomenti
- Esempi

> int a = 100 = 1100100

> int b = 70 = 1000110

> a & b = 1000100

> a | b = 1100110

(Vedere AndBit.java)

Operatori Bitwise (2)

Operatore	Uso	Descrizione
&	op1 & op2	Effettua un AND bit a bit tra op1 e op2
|	op1 | op2	Effettua un OR bit a bit tra op1 e op2
~	~op1	Effettua un NOT bit a bit di op1
^	op1 ^ op2	Effettua uno XOR bit a bit tra op1 e op2

Operatori Scorrimento (1)

Operatore	Uso	Descrizione
>>	op1 >> op2	Effettua uno scorrimento a dx di op2 bit su op1
<<	op1 << op2	Effettua uno scorrimento a sx di op2 bit su op1
>>>	op1 >>> op2	Effettua uno scorrimento a dx di op2 bit su op1 senza considerare il bit del segno

Operatori Scorrimento (2)

Operatore	Uso	Descrizione
>>	op1 >> op2	Effettua uno scorrimento a dx di op2 bit su op1
<<	op1 << op2	Effettua uno scorrimento a sx di op2 bit su op1
>>>	op1 >>> op2	Effettua uno scorrimento a dx di op2 bit su op1 senza considerare il bit del segno

- L'operatore >> usa la proprietà di estensione del segno: se il valore da scorrere è positivo vengono inseriti degli zero nei bit più significativi, viceversa se il valore da scorrere è negativo vengono invece inseriti degli uno
- L'operatore >>> indipendentemente dal segno del numero da scorrere, inserisce degli zero nei bit più significativi

Operatori Scorrimento (3)

» -8 = 1 11111111111111111111111111111100 (k)

31 bit

» k >>> 3

536870911 = 0 00111111111111111111111111111111

31 bit

» k >> 3

-1 = 1 11111111111111111111111111111111

(Vedere Scorrimento.java)

Operatori Scorrimento (4)

» Se si fa scorrere un dato di tipo char, byte short questo sarà promosso a int prima che lo scorrimento abbia luogo e il risultato sarà un int

– In questi casi nell'operazione di scorrimento verranno usati solo i cinque bit meno significativi dell'operando di destra (in modo da non poter scorrere più del numero di bit con i quali è rappresentato un int)

» Se si fa scorrere un dato di tipo long il risultato sarà un long

– In questo caso nell'operazione di scorrimento verranno usati solo i sei bit meno significativi dell'operando di destra (in modo da non poter scorrere più del numero di bit con i quali è rappresentato un long)

Precedenza (1)

- Stabilisce come viene calcolata un'espressione in presenza di diversi operatori
- Gli operatori hanno una precedenza implicita ben definita che determina l'ordine con cui vengono valutati
  - Moltiplicazione, divisione e resto sono valutati prima di somma, sottrazione e concatenazione tra stringhe
- Gli operatori che hanno la stessa precedenza sono valutati da sinistra a destra
- Mediante le parentesi si può alterare l'ordine di precedenza

Precedenza (2)

Associatività	Operatori
Dx a Sx	`++ -- + - ~ ! (tipo di dato)`
Sx a Dx	`* / %`
Sx a Dx	`+ -`
Sx a Dx	`<< >> >>>`
Sx a Dx	`< > <= >= instanceof`
Sx a Dx	`== !=`
Sx a Dx	`&`
Sx a Dx	`^`
Sx a Dx	`|`
Sx a Dx	`&&`
Sx a Dx	`||`
Sx a Dx	`?:`
Sx a Dx	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`

Operatore Assegnamento (1)

» = è considerato un operatore

» Prende il valore che si trova alla destra (rvalue) e lo copia nell'entità di sinistra (l-value)

» l-value deve essere il nome di una variabile

> a = 4 //OK

> 4 = a //NOT OK

» Nel caso di tipi primitivi della variabile viene copiato il valore

» Tipi reference viene copiato il riferimento dell'oggetto

(Vedere Assignment.java)

Operatore Assegnamento (2)

Operatore	Uso	Equivalente a
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b

Errori comuni nell'uso degli operatori

» Derivanti dal mancato uso di parentesi quando si ha l'incertezza su come verrà valutata un'espressione

» Errore comune in C e C++

```
while (x=y) {
    …
}
```

– Il programmatore intendeva chiaramente effettuare un test di equivalenza (==), anziché un assegnamento

- In C e C++ il risultato di questa operazione sarà sempre interpretato come true, se y è diverso da zero
- In Java viene segnalato un errore in fase di compilazione. Infatti il risultato di questa operazione non è un dato di tipo boolean ma int (a meno che x e y sono boolean, in tal caso x=y è un'espressione leggittima)

(Vedere SampleError.java e PassObject.java)

Casting

- Ogni espressione in Java ha un tipo che può essere dedotto dalla struttura dell'espressione, dal tipo dei letterali, variabili e metodi menzionati nell'espressione
- E' possibile utilizzare un'espressione in un contesto dove il tipo non è appropriato
- In alcuni casi porta ad errore (es. if può avere soltanto espressioni con tipo boolean)
- Altri casi il contesto accetta in modo implicito l'espressione anche di tipo diverso (conversione)
- Java vi lascia convertire qualsiasi dato primitivo in qualsiasi altro tipo, eccezione fatta per il tipo boolean che non permette alcun casting
- Per quanto riguarda gli oggetti questi possono essere convertiti all'interno di una famiglia di tipi; ad esempio una *Quercia* può essere convertita in un *Albero* e viceversa, ma non in un tipo estraneo come *Sasso*

Casting

- *Conversione con riduzione*, quando si passa da un tipo di dato che può contenere più informazioni a uno che ne può contenere meno
- *Conversione con ampliamento*, quando si passa da un tipo di dato che può contenere meno informazioni a uno che ne può contenere di più
- In Java, il casting è normalmente sicuro, eccetto quando si esegue una *conversione con riduzione,* poiché si corre il rischio di perdere informazioni. In questo caso il compilatore obbliga ad eseguire una conversione segnalando che l'operazione può essere pericolosa pertanto se si vuole farla è necessario un cast esplicito

Conversione e Promozione (1)

» Esempi

```
Object obj = new Object();
String newStr = (String)obj;  // requires an extra check at runtime
```

```
String newStr = new String();
Object obj  = newStr;  // requires no extra check at runtime, but
information loss.
```

» Una conversione dal tipo primitivo int al tipo primitivo double richiede un check a runtime per il segno dell'intero a 32 e convertire l'intero valore a 64 bit

```
int intValue = 100;
double doubleValue = intValue;
```

» Una conversione da long ad int richiede un cast esplicito ed un check a runtime. Ci potrebbe essere perdita di informazione

```
long longValue = 50;
int intValue = (int)longValue;
```

Conversione e Promozione (2)

» Categorie di conversione

> Identity

> Widening Primitive

> Narrowing Primitive

> Widening e Narrowing References

> String

> ……..

Conversione e Promozione (3)

» Le espressioni si possono trovare in 5 contesti di conversione

> Assegnamento

> Invocazione del metodo

> Casting

> String

> Promozione numerica

> Si ha nell'espressioni dove compaiono operatori numerici (Es.: +, *)

> Si parla di promozioni poiché la conversione può dipendere in parte dal tipo dell'altro operando nell'espressione

» Ogni contesto permette la conversione in una delle categorie precedenti

(vedere ConvExamples.java)

Conversione Identity

» E’ permessa la conversione da un tipo allo stesso tipo

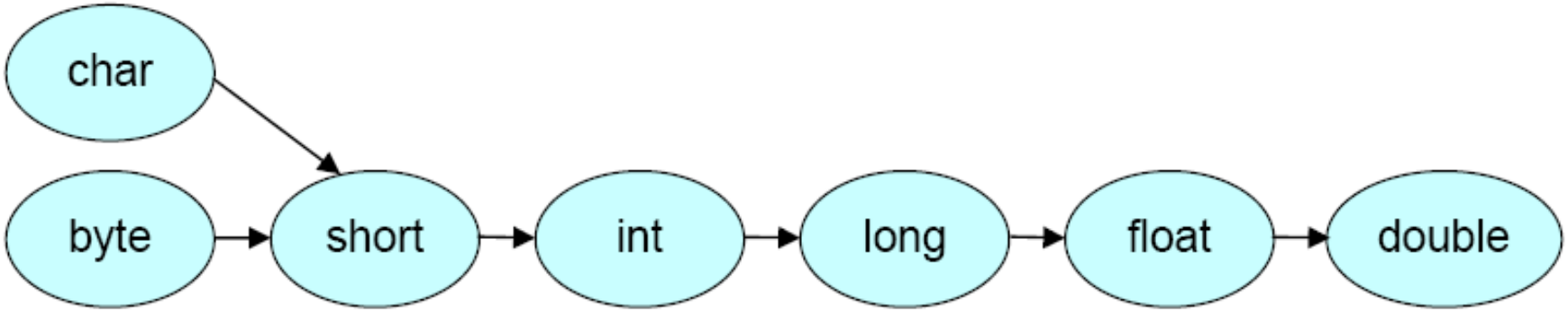
» E’ ovvia ma permette di introdurre cast superflui per aumentare la chiarezza del codice

```
boolean b1,b2=false;
b1 = (boolean)b2;
```

» Espressione tipo boolean ammette soltanto conversione Identity da boolean a boolean

> int i = (5 <= 3); //Non Ammessa

Conversione Widening Primitive (1)



» Conversioni sui tipi primitivi ammessi

> byte a short, int, long, float, o double

> short a int, long, float, o double

> char a int, long, float, o double

> int a long, float, o double

> long a float o double

> float a double

Conversione Widening Primitive (1)

- Non vi è perdita di informazioni ovvero il valore numerico è conservato
  - E’ ammissibile convertire una variabile di tipo int in double, la conversione è possibile perche un double ha più bit dell’int
- Conversione da int o long a float oppure da long a double potrebbe comportare perdita di precisione ovvero si potrebbero perdita dei bit più significativi
  - Una variabile di tipo int è rappresentata in 32 bit come anche un float. Comunque il float usa una parte di questi bit per rappresentare l’esponente
- Non vengono mai generate delle eccezioni

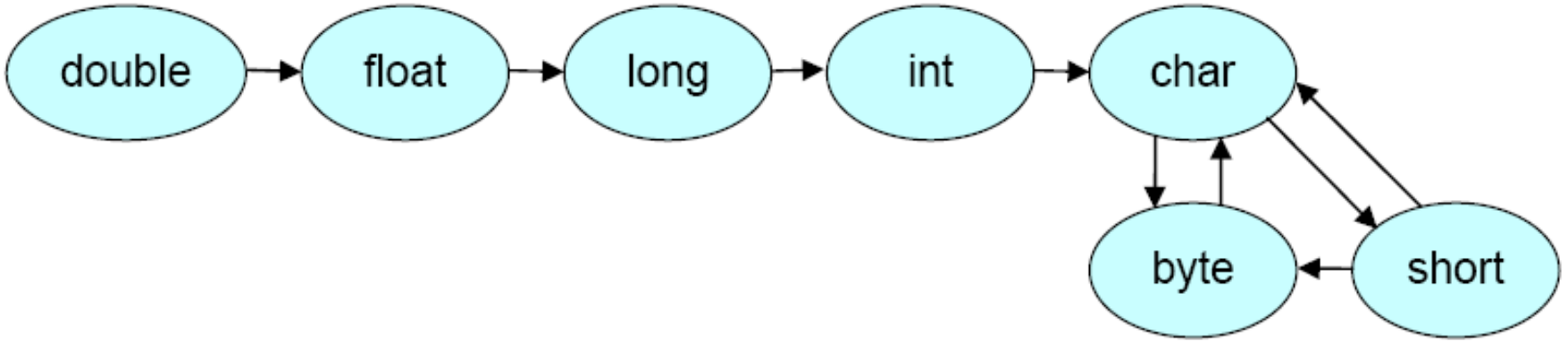
Conversione Widening Primitive (2)


```
class Test {
 public static void main(String[] args) {
   int big = 1234567890;
   float approx = big;
   System.out.println(big - (int)approx);
 }
}
```

» Output: -46

» Perdita di informazioni poiché il valore float non è preciso alle nove cifre

Conversione Narrowing Primitive (1)



» Conversioni sui tipi primitivi ammessi

> short a byte o char

> char a byte o short

> int a byte, short, o char

> long a byte, short, char, o int

> float a byte, short, char, int, o long

> double a byte, short, char, int, long, o float

Conversione Narrowing Primitive (2)

- Vi è perdita di informazione
- Vengono presi i bit meno significativi relativi al tipo destinatario (quindi vi può essere il cambio di segno)
- Non vengono mai generate delle eccezioni

(Vedere Conv.java)

Conversione Narrowing Primitive (3)


```
class Test {

  public static void main(String[] args) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.println("long: " + (long)fmin +
                           ".." + (long)fmax);
        System.out.println("int: " + (int)fmin +
                           ".." + (int)fmax);
        System.out.println("short: " + (short)fmin +
                           ".." + (short)fmax);
        System.out.println("char: " + (int)(char)fmin +
                           ".." + (int)(char)fmax);
        System.out.println("byte: " + (byte)fmin +
                           ".." + (byte)fmax);
    }
  }
```

Output

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

Conversione Widening of instance references

» Esempi di conversioni widening references

> Da una class Apple ad una classe Fruit, ammesso che Apple è sottoclasse di Fruit

> Da una classe di tipo Apple ad una interfaccia Fruit, ammesso che Apple implementa Fruit

> Da un interfaccia Vehicle ad un'interfaccia Car, ammesso che Vehicle è sotto interfaccia di Car

```
// From a class to another class, providing that String is a
// subclass of Object
String source = new String("source");
Object destination = source;

// From a class to an interface, providing that the class implements
// the interface
Thread myThread = new Thread();
Runnable run = myThread;
```

» Tale conversione non richiede nessun check a runtime

» **Nessun errore in fase di compilazione o eccezione a runtime**

Conversione Narrowing of instance references

» Esempi di conversioni narrowing references

> Da una class A ad una qualsiasi classe B, ammesso che A è superclasse di B

```
// Hashtable is a superclass of Properties
Hashtable english = new Hashtable();
Properties prop   = (Properties)english;

// Narrow to an interface from a non-final class
Object myObject   = new Object();
Runnable runnable = (Runnable)myObject;

// Narrow to an interface from a final class String will not compile
String aString = new String();
Runnable runnableString = (Runnable)aString;
```

» Tale conversione richiede un esplicito cast

» Anche se la compilazione ha successo una **ClassCastException può essere sollevata a runtime**

Conversione String

» E’ possibile convertire qualsiasi tipo al tipo String incluso il tipo null

(Vedere ConvExamples.java)

Conversioni non ammesse

- Da un tipo reference a un tipo primitivo e viceversa (eccetto per la conversione String)
- Da null a un tipo primitivo
- Tipo classe S a una classe di tipo T se S non è sottoclasse di T e viceversa
- ……
- ……..
- ……..

Contesto conversione: Assegnamento

- Si verifica quando il valore di un'espressione è assegnato ad una variabile ed il tipo dell'espressione è convertito al tipo della variabile
- Conversioni ammesse
  - Identity
  - Widening primitive e reference
  - Narrowing primitive se tutte le seguenti condizioni sono verificate
    - Espressione è una costante di tipo byte short char o int
    - Tipo della variabile è byte short o char
    - Valore dell'espressione (conosciuto a tempo di compilazione) è rappresentabile all'interno del tipo della variabile
- Si verifica un errore in compilazione se non è possibile convertire il tipo dell'espressione al tipo della variabile
- Esempio
  - byte theAnswer = 42; //OK per ultimo punto
  - short s3 = 0x1ffff; //NON VALIDO

Contesto conversione: Invocazione di un metodo (1)

- E’ applicata ad ogni valore di un parametro in una invocazione di un metodo o di un costruttore
- Tipo dell’espressione deve essere convertito al corrispondente tipo del parametro
- Conversioni ammesse
  - Identity
  - Widening primitive e reference
- Non vengono incluse conversioni narrowing di costanti intere

Contesto conversione: Invocazione di un metodo (2)


```
class Test {

  static int m(byte a, byte b) { return a+b; }

  static int m(short a, short b) { return a-b; }


  public static void main(String[] args) {

    System.out.println(m(12, 2));
  }

}
```

» Errore in compilazione poiché 12 e 2 sono letterali di tipo int

(Vedere ConvMethod.java)

Contesto conversione: String

- Si applica soltanto agli operandi dell'operatore binario + dove uno degli argomenti è una stringa
- Esempio
- System.out.println(12 + " Hello");

Contesto conversione: Casting (1)

- Viene applicato all'operando di cast ()
- Tipo dell'espressione viene convertito al tipo esplicito indicato nell'operatore di cast
- Conversioni ammesse
  - Identity
  - Widening o narrowing primitive
  - Widening o narrowing reference
- Alcuni cast producono errore in compilazione
  - Casting tra tipo primitivo e reference e vicevesa
- E' ammesso il casting tra qualsiasi tipo primitivo
- Esempio di casting tra tipo classe S a un tipo classe T
  - S e T devono essere in relazione ovvero stessa classe, oppure S sotto-classe di T o viceversa altrimenti si verifica un errore in compilazione
- Altri casi presenti (vedere libro della specifica)

Contesto conversione: Casting (2)


```
class Point {
  int x, y;

}

class ColoredPoint extends Point {
  int color;
  public void setColor(int c) {
       color = c;
  }

}

class Test {
  public static void main(String[] args) {
    Point p = new Point();
    ColoredPoint cp = new ColoredPoint();
    cp = (ColoredPoint)p;       //Errore a run-time
    Long l = (Long)p;           //Errore in compilazione
    int i = (int) p;            //Errore in compilazione
  }

                                                        }
```

(Vedere Casting.java)

Promozione numerica

- Applicata agli operandi di un operatore aritmetico
- Conversioni ammesse
  - Identity
  - Widening primitive
- Viene utilizzata per convertire gli operandi dell'operatore ad un tipo comune prima di eseguire l'operazione
- Tipi di promozione
  - Unaria
  - Binaria

Promozione numerica Unaria (1)

- Applicata agli operatori con un singolo operando
- Se l'operando è di tipo byte, short o char l'operando viene promosso ad int
- Altrimenti non viene convertito
- Applicata alle seguenti espressioni
  - > Dimensione nella creazione di un array
  - > Indice in un accesso ad un array
  - > Operando dell'operatore unario + e –
  - > Operando dell'operatore complemento ~
  - > Ogni singolo operando degli operatori di shift >>, >>> e <<

Promozione numerica Binaria (1)

- Applicata ad una coppia di operandi dove ognuno denota un valore numerico
  - Se uno dei due operandi è double l'altro è convertito a double
  - Altrimenti se uno dei due operandi è float l'altro è convertito a float
  - Altrimenti se uno dei due operandi è long l'altro è convertito a long
  - Altrimenti entrambi gli operandi sono convertiti a int
- Operatori dove si può applicare la conversione
  - *, / e %
  - + e -
  - <, <=, >, e >=
  - == e !=
  - Operatori bit a bit &, ^, e |