Lezione 2

JAVA

Basi e fondamenti

R-DeV
Revolution Development
learning@r-dev.it

Argomenti della lezione 2

- Costrutti condizionali
- Costrutti iterativi
- Tipi primitivi
- Tipi reference
- La classe Object
- Wrapper

R-DeV
Revolution Development
learning@r-dev.it

Costrutti condizionali - IF ELSE

```
if ( <condizione1> ) {
    ...istruzioni
} else if ( <condizione2>) {
        ... istruzioni
    } else {
        ...istruzioni
    }
```

Costrutti condizionali - SWITCH

```
switch (<condizione>) {
    case <costante1> :
        ...istruzioni;
        break;
    case <costante2> :
        ...istruzioni;
        break;
    default :
        ...istruzioni;
        break;
}
```

Costrutti iterativi - FOR

```
for ( <init_expr>; <max_val>; <incremento>) {
    ...istruzioni;
}
```

```
for ( int i = 0; i < 100; i++) {
    ...istruzioni;
}
```

Costrutti iterativi - FOR EACH

```
for (<tipo> <nome_var> : <in>) {
    ...istruzioni;
}
```

```
for ( String s : insieme) { //dove insieme è un Set<String> ad es.
    ...istruzioni;
}
```

Costrutti iterativi - WHILE

```
while ( <condizione_uscita>) {

     ...istruzioni;

}

int i = 0;

while ( i < 10 ) {

     ...istruzioni;

     i++;

}
```

Costrutti iterativi - DO / WHILE

```
do {
     ...istruzioni;
} while ( <condizione_uscita>);


int i = 0;
do {
     ...istruzioni;
     i++;
} while ( i < 10 );
```

Tipi primitivi

I tipi primitivi sono 8:

Logici: **boolean**

Testuali: **char**

Interi: **byte**, **short**, **int**, **long**

Floating point: **float**, **double**

boolean

Il tipo boolean può assumere solo due valori : **true** e **false**

Esempio:

```
boolean isOk = true;
```

Non c'è cast tra tipi interi e boolean. Interpretare valori numerici come valori logici non è permesso in Java.

char

Java utilizza l'Unicode. I caratteri vengono gestiti tramite il corrispondente valore numerico Unicode; è per questo che, in realtà, *char* è a tutti gli effetti un tipo numerico. (Può essere visto come una versione *unsigned* del tipo *short*.) I valori letterali di tipo *char* vanno compresi tra singoli apici: ad esempio, 'a'.

Esempi:

```
'a'          → la lettera a
'\t'         → una tabulazione
'\u03A6'     → la lettera greca phi
```

```
char lettera = 'm';
```

Tipi interi

	Lunghezza	Range
byte	8 bit	$-2^7 \ldots 2^7 - 1$
short	**16 bit**	$-2^{15} \ldots 2^{15} - 1$
int	**32 bit**	$-2^{31} \ldots 2^{31} - 1$
long	**64 bit**	$-2^{63} \ldots 2^{63} - 1$

Tipi a virgola mobile

Numeri in virgola mobile in singola precisione secondo la specifica IEEE 754, utilizzando la rappresentazione segno, mantissa esponente.

float

Sono numeri in virgola mobile con 7 cifre significative compresi tra 1.4E-45 e 3.4028235E+38 . La lunghezza dell'area di memoria dedicata ad un float è 32 bit.
Il valore di default 0.0
Le costanti vanno terminate con F o f (es. `float a = 3.456F;`)

double

Sono numeri in virgola mobile in doppia precisione (15 cifre significative) compresi tra 4.9E-324 e 1.7976931348623157E+308 .
La lunghezza dell'area di memoria dedicata ad un double è 64 bit.
Illvalore di default 0.0 .
Lle costanti con virgola sono di tipo double per default possono essere terminate con D o d ma non è necessario (es. `double b = 3.456D;`)

Esempio

```
public class Assign {
    public static void main (
String args []) {
        int x , y ;
        float z = 3.1415 f ;
        double w = 3.14145;
        boolean ok = true ;
        char c ;
        c = 'A ';
        x = 6;
        y = 1000;
    }
}
```

Assegnazioni illegali:

```
y = 3.145;
w = 17,45;
ok = 1;
```

Tipi Reference

Tutti i tipi non primitivi sono tipi *reference*.

Esempio:

```
public class MiaData {
    public int giorno;
    public int mese;
    public int anno;
}
```

Un riferimento ad un oggetto di tipo MiaData può essere creato in questo modo:

```
MiaData dataNascita = new MiaData();
```

new()

La parola chiave **new**, insieme al nome della classe seguito da parentesi tonde ,serve ad allocare spazio per il nuovo oggetto.

Scatena i seguenti processi:

1. Viene allocato lo spazio per il nuovo oggetto e le variabili dell'istanza sono inizializzate al loro valore di default (e.g. 0, false, null, e così via).
1. Viene eseguita ogni inizializzazione esplicita degli attributi.
2. Viene eseguito un costruttore.
3. Viene assegnato il riferimento finale all'oggetto.

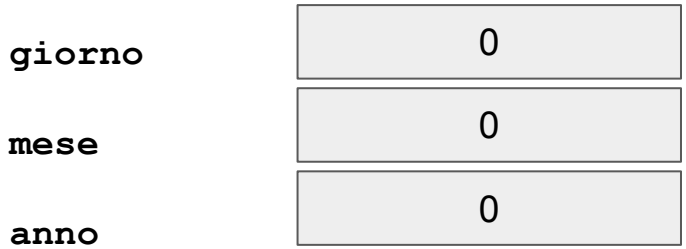
Allocazione di memoria

Nell'enunciato

```
MiaData nascita = new MiaData (19 , 6 , 1989);
```

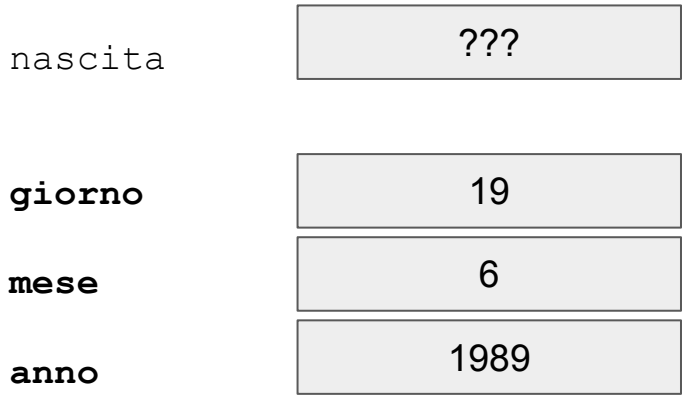
la dichiarazione `MiaData nascita` causa l'allocazione dello spazio memoria del solo riferimento (non ancora inizializzato):

l'uso della parola chiave **new**, alloca spazio per MiaData, inizializzando le variabili ai valori di default:



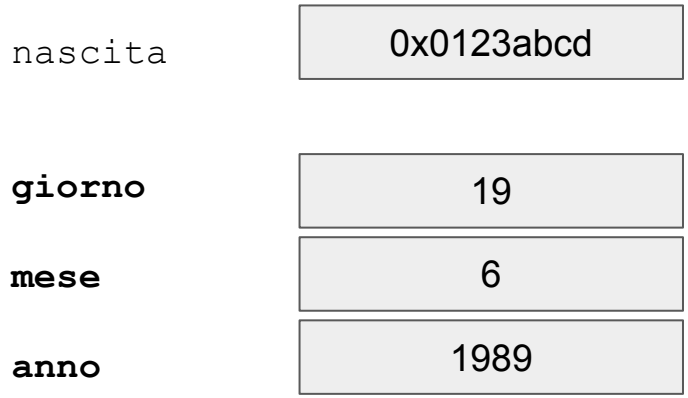
Allocazione di memoria - Esecuzione del costruttore

Ora è invocato il costruttore. Con esso si possono sostituire inizializzazioni personali dell'utente dell'oggetto a quelle di default previste dal progettista della classe. Si possono anche passare argomenti, così che il codice che richiede le costruzione del nuovo oggetto possa controllare l'oggetto che verrà creato:



Allocazione di memoria - Assegnazione del riferimento

L’assegnazione, infine, inserisce l’indirizzo del nuovo oggetto nella locazione del riferimento:

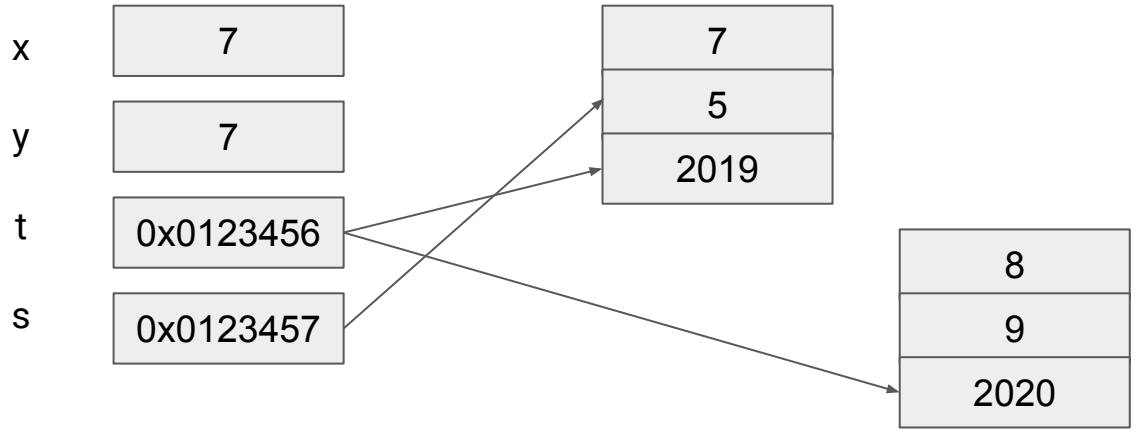


Esempio

```
int x = 7;
int y = x;

MiaData s = new MiaData(7,5,2019);
MiaData t = s;

t = new MiaData(8,9,2020)
```



Il riferimento this

La parola chiave **this** può essere usata per fare riferimento, all'interno di un metodo o di un costruttore locale, ad attributi o metodi locali.

Questa tecnica è usata per risolvere ambiguità in alcuni casi in cui una variabile locale di un metodo maschera un attributo locale dell'oggetto.

```
public class MiaData {
     public int giorno;
     public int mese;
     public int anno;

     public MiaData(int giorno, int mese, int anno){
          this.giorno = giorno;
          this.mese = mese;
          this.anno = anno;
     }
}
```

Convenzioni sul codice

Per nomi di classi e interfacce si utilizza il *CamelCase* e la prima lettera è *maiuscola*;

Per i nomi di variabili e metodi si utilizza il *CamelCase* ma la prima lettera è *minuscola*;

I nomi dei metodi sono generalmente verbi;

Le costanti vanno scritte tutte in maiuscolo utilizzando l'underscore _ come separatore;

E' fondamentale per la scrittura di un buon codice:

- indentare correttamente
- andare a capo ad ogni istruzione
- utilizzare nomi significativi
- utilizzare gli spazi
- commentare il codice quanto più possibile, soprattutto se la logica implementata non è semplice

I Package

Un package è un raggruppamento di classi (ma anche interfacce, enumerazioni e annotazioni) logicamente correlate. E' uno strumento molto utile per una buona organizzazione del codice: raggruppare le classi per funzionalità e/o responsabilità rende il codice di un'applicazione più leggibile, ordinato e riutilizzabile.
Ogni package può avere uno o più sottopackage che sul filesystem saranno delle vere e proprie cartelle.
La keyword **package** deve essere posizionata come prima istruzione, ed all'interno di ogni file può esserci un'unica dichiarazione del package.
Il nome completamente qualificato di un'entità di primo livello comprende il nome del package come prefisso ed è univoco, di conseguenza nello stesso progetto posso avere più classi omonime ma non nello stesso package.

Convenzioni: i nomi dei package sono significativi, scritti tutto in minuscolo, iniziano con un carattere e non contengono alcun carattere speciale.

La classe Object

In Java ogni classe è una classe derivata, in modo diretto o indiretto, dalla classe **Object**. La classe appartiene al package `java.util`.
Ciascuna classe eredita automaticamente i metodi di Object:

`public final Class<?> getClass()`
restituisce la classe a runtime dell'istanza sulla quale è invocato

`protected void finalize() throws Throwable`
chiamato dal Garbage Collector su un oggetto quando determina che non ci sono più riferimenti a quell'oggetto

`public final void notify()`
se qualche thread è in attesa su questo oggetto, uno di essi viene scelto per essere risvegliato. Non è ridefinibile

`public final void notifyAll()`
risveglia tutti i thread che sono in attesa di quell'oggetto

La classe Object

`public final void wait() throws InterruptedException`
`public final void wait(long timeout) throws InterruptedException`
`public final void wait(long timeout, int nanos) throws InterruptedException`
fa in modo che il thread corrente attenda fino a quando un altro thread invoca il metodo `notify()` o il metodo `notifyAll()` per questo oggetto, oppure è trascorso il periodo di tempo specificato in input. Se invocato il primo metodo, senza parametri di ingresso, si assume che l'input sia 0.

`public String toString()`
restituisce una rappresentazione umanamente leggibile dell'istanza dell'oggetto su cui è invocato. L'implementazione nella classe Object è la seguente:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

di conseguenza è fortemente raccomandato ridefinire il metodo `toString()` negli oggetti creati.

La classe Object - hashCode()

`public int hashCode()`
restituisce un codice hash per l'istanza su cui è invocato. E' un metodo di supporto alle tabelle hash come quelle fornite da HashMap.

Il contratto generale di hashCode è:

- Ogni volta che viene richiamato sullo stesso oggetto più di una volta durante l'esecuzione di un'applicazione Java, il metodo `hashCode` deve restituire costantemente lo stesso numero intero, a condizione che non venga modificata alcuna informazione utilizzata nel metodo.
- Se due oggetti sono uguali in base al metodo `equals`, la chiamata del metodo `hashCode` su ciascuno dei due oggetti deve produrre lo stesso risultato, ma non è necessario il contrario, tuttavia bisogna essere consapevoli che la produzione di risultati distinti per oggetti diversi può migliorare le prestazioni delle tabelle hash.

Se viene ridefinito il metodo `hashcode` è buona norma ridefinire anche `equals` in modo da rispettare il contratto di coerenza tra i due metodi.

La classe Object - equals()

`public boolean equals()`
restituisce true se due oggetti sono uguali, false altrimenti.
L'implementazione del metodo deve rispettare le seguenti proprietà:

- RIFLESSIVA: per ogni oggetto non nullo `x.equals(x)` deve restituire `true`
- SIMMETRIA: per ogni coppia di oggetti non nulli `x.equals(y)` restituisce `true` se e solo se `y.equals(x) = true`
- TRANSITIVA: dati tre oggetti non nulli se `x.equals(y) = true` e `y.equals(z) = true` allora `z.equals(x)` deve restituire `true`
- CONSISTENZA: per ogni coppia di oggetti non nulli le invocazioni ripetitive di `x.equals(y)` o `y.equals(x)` devono restituire il medesimo risultato a patto che durante l'esecuzione non cambino le informazioni degli oggetti
- per ogni riferimento x non nullo `x.equals(null)` deve restituire `false`

Se il metodo non viene ridefinito per comportamento standard confronterà i riferimenti, quindi restituirà true se e solo se le variabili punteranno allo stesso oggetto `(x === y)`.
Ridefinire `equals` comporta ridefinire anche `hashcode` in modo da non violare il contratto e mantenere la coerenza tra i due metodi.

La classe Object - clone()

`protected Object clone()  throws CloneNotSupportedException`
Crea e restituisce una copia dell'oggetto su cui è invocato. Il significato preciso di "copia" dipende dall'implementazione. L'intento generale è che, per qualsiasi oggetto x, l'espressione:
`x.clone() != x` deve essere true, ovvero che l'oggetto creato deve avere un riferimento diverso dall'oggetto copiato.

Per convenzione, l'oggetto restituito da questo metodo dovrebbe essere completamente indipendente dall'oggetto copiato. Per ottenere questa indipendenza, potrebbe essere necessario modificare uno o più campi dell'oggetto restituito da `super.clone` prima di restituirlo effettuando una copia profonda dei campi, ovvero invocando il metodo clone su qualsiasi attributo di tipo non primitivo presente tra gli attributi. Se una classe contiene solo campi primitivi o riferimenti a oggetti immutabili si può evitare di ridefinire il metodo `clone`.

I wrapper

Dato un tipo primitivo (i cui nomi iniziano tutti rigorosamente con la prima lettera minuscola) si ottiene la corrispondente classe wrapper (o Data Object) sostanzialmente capitalizzando il nome come mostrato nella tabella seguente:

Tipo primitivo	Classe wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

I wrapper

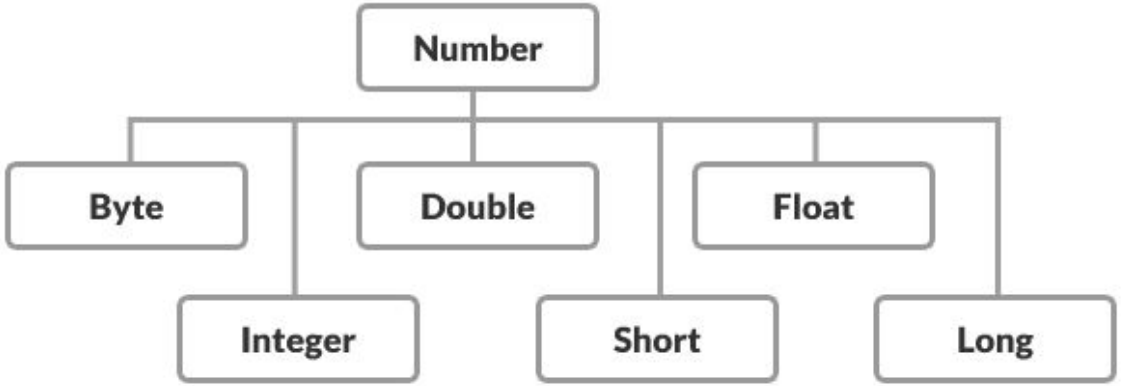
Anche se ad un primo sguardo può sembrare che ci sia poca differenza tra un tipo primitivo e la sua controparte ‘wrapped’ (anche e spesso detta ‘boxed’) tra le due c’è una fondamentale distinzione: i tipi primitivi non sono oggetti e non hanno associata alcuna classe e quindi devono essere trattati in modo diverso rispetto agli altri tipi (ad esempio non è possibile utilizzarli nelle collezioni, che saranno argomento di future lezioni) e non possono avere metodi.

Per ovviare a questa distinzione Java mette dunque a disposizione delle classi preconfezionate per contenere, “wrappare” i tipi primitivi. Possiamo infatti pensare ad una classe wrapper esattamente come un involucro (wrap) che ha l’unico scopo di contenere un valore primitivo rendendolo da un lato un oggetto e dall’altro “ornandolo” con metodi che altrimenti non avrebbero una loro naturale collocazione.

I wrapper

Tutte le classi wrapper sono definite nel package *java.lang* e sono qualificate come **final**, perciò non è possibile derivare da loro.

Mentre **Boolean** e **Character** derivano direttamente da **Object**, tutti i Data Object di tipo numerico derivano da **Number**, che a sua volta è un discendente diretto di Object.



Autoboxing e Unboxing

A partire dalla versione di Java 1.5 è stata introdotta la possibilità di poter convertire automaticamente (**autoboxing**) i valori dei tipi base in oggetti e viceversa (**unboxing**). Questi processi trovano grande utilità con i tipi wrapper, in particolare è possibile scrivere senza avere nessun errore di compilazione istruzioni come

`Integer n = 7;` → **Autoboxing**

Quest'istruzione assegna un tipo numerico ad un oggetto dello stesso tipo, viene eseguito in automatico l'incapsulamento del valore in un oggetto. Quest'istruzione equivale:

```
Integer n = new Integer(7);
```

E' stato automatizzato anche il processo inverso, per cui è possibile scrivere :

`int m = n;` e l'istruzione verrà tradotta con:

`int m = n.intValue();` → **Unboxing**

Operatori algebrici

Operatore	Uso	Significato
+	a+b	somma a e b
-	a-b	sottra b da a
*	a*b	moltiplica a per b
/	a/b	divide a per b
%	a%b	calcola il resto della divisione intera di a per b (solo con int, long e byte)

Operatori relazionali

Operatore	Uso	Significato
==	a == b	vero se a è uguale a b
!=	a != b	vero se a è diverso da b
>	a > b	vero se a è maggiore di b
<	a < b	vero se a è minore di b
>=	a >= b	vero se a è maggiore o uguale di b
<=	a <= b	vero se a è minore o uguale di b

Operatori logici

Operatore	Uso	Significato
&&	a && b	vero se a e b sono entrambi true, falso in tutti gli altri casi
||	a || b	false solo se sono entrambi false

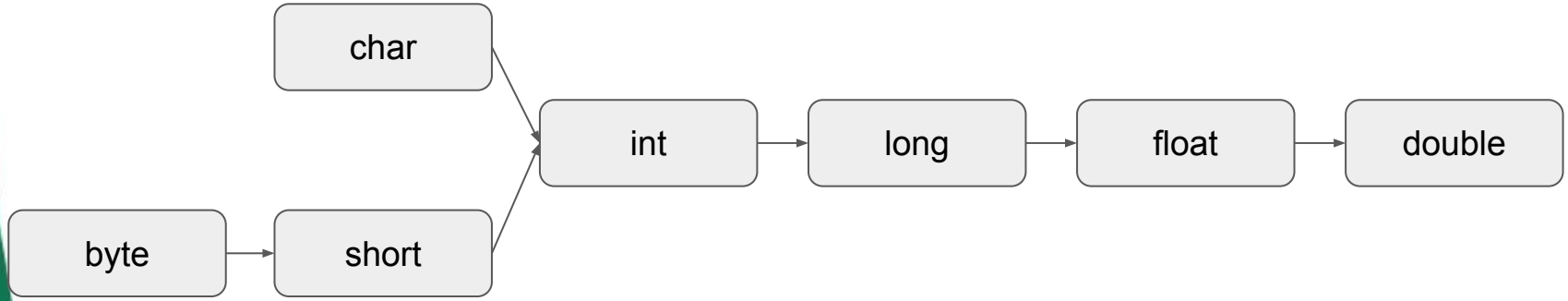
Esempi:

```
true && true → true
true && false → false
false && false → false
```

```
true || true → true
true || false → true
false || false → false
```

Casting tra primitivi - Conversioni implicite

In Java esistono le conversioni di tipo che vengono fatte automaticamente ed altre che richiedono un cast esplicito. Con il termine cast si intende l'operazione di passaggio di una variabile da un tipo di dato ad un altro.



Il tipo `boolean` non può essere convertito **MAI**!

Casting tra primitivi - Conversioni esplicite

SE è possibile la perdita di informazioni in una assegnazione o in un passaggio di parametri, il programmatore **DEVE** confermare l'assegnazione con un "cast" esplicito.

In un cast esplicito, il valore in eccesso viene troncato.

Per esempio, l'assegnazione da long a int richiede un cast esplicito:

```
long x = 77L;
int y = (int)x;
```

Fine lezione 2

R-DeV
Revolution Development
learning@r-dev.it