



Java

Classi wrapper e classi di servizio

Classi wrapper – Concetti di base

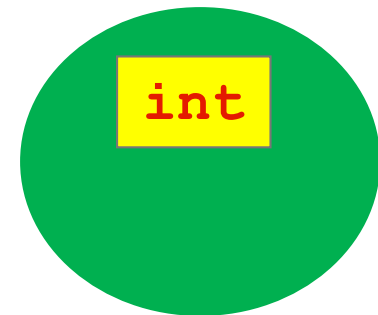
- In varie situazioni, può essere comodo poter trattare i tipi primitivi come oggetti.
- Una classe **wrapper** (involucro) incapsula una variabile di un tipo primitivo
- In qualche modo “trasforma” un tipo primitivo in un oggetto equivalente
- la classe **Boolean** incapsula un **boolean**
- la classe **Double** incapsula un **double**
- la classe **Integer** incapsula un **int**
- La classe wrapper ha nome (quasi) identico al tipo primitivo che incapsula, ma con l’iniziale maiuscola

Classi wrapper - Elenco

- Nella tabella sottostante sono riportati i tipi primitivi e le relative classi wrapper.
- Attenzione alle differenze di nome: **int/Integer** e **char/Character**

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

oggetto Integer



Classi wrapper - Funzionamento

- Ogni classe wrapper ha come stato semplicemente un attributo del tipo che incapsula: **Integer** avrà un attributo di tipo **int**, **Double** un attributo di tipo **double** e così via.
- Le classi wrapper sono state costruite per essere **immutabili**: assumono un valore al momento della creazione e non lo cambiano mai più.
- Quindi per ogni classe esiste un **costruttore** che prende come parametro una variabile del tipo incapsulato e lo memorizza nello stato.
- Esiste poi un metodo **selettore** che consente di leggere in modo protetto il valore dello stato.

Classi wrapper - Funzionamento

- Ogni classe wrapper definisce metodi per **estrarre il valore** della variabile incapsulata e viceversa. Per estrarre il valore incapsulato:
 - `Integer` fornisce il metodo `intValue()`
 - `Double` fornisce il metodo `doubleValue()`
 - `Boolean` fornisce il metodo `booleanValue()`
 - `Character` fornisce il metodo `charValue()`
 - ...
- Per **creare** un oggetto da un valore primitivo:
 - `Integer i = new Integer(valore int)`
 - `Double d = new Double(valore double)`
 - `Boolean b = new Boolean(valore boolean)`
 - `Character c = new Character(valore char)`
 - ...

Integer – Ipotesi di implementazione

- Possiamo immaginare (è un'ipotesi semplificata) che la classe Integer sia fatta così:

```
class Integer
{
    private int value;
    public Integer(int val)
    {
        value = val;
    }
    public int intValue()
    {
        return value;
    }
}
```

Integer - Esempio

- Vediamo un semplice esempio di uso della classe Integer:

```
public class EsempioWrapper
{
    public static void main(String args[])
    {
        int x = 35;
        Integer ix = new Integer(x) ;
        x = 2*ix.intValue() ;
        System.out.println("x =" + x) ;
    }
}
```

Integer – Altri metodi

- In realtà la classe Integer, come tutte le classi wrapper, è più complessa e mette a disposizione molti altri metodi
- Abbiamo innanzitutto un **secondo costruttore** che prende **come parametro una stringa** e la converte in un intero per memorizzarne il valore nello stato

```
public Integer(String s);
```

- Abbiamo poi alcuni **metodi che convertono il valore interno in un altro tipo**

- Per esempio:

```
public String toString();
```

```
public double doubleValue();
```


Classi wrapper come classi di servizio

- Le funzionalità di Integer che abbiamo visto fino ad ora sono collegate al ruolo della **classe come descrittore di istanze**.
- Le classi wrapper mettono a disposizione un gran numero di metodi **static**
- Queste funzionalità sono quindi legate all'idea di **classe come fornitore di servizi**, indipendentemente dalla creazione di istanze.
- Sono metodi che possiamo invocare senza creare un'istanza
- L'esempio più tipico è:
`public static int parseInt(String s) ;`
- Questo metodo converte una stringa in un intero **senza creare un'istanza di Integer**

Due modi per convertire stringhe in interi

- Sulla base di quanto detto finora abbiamo due modi per convertire una stringa in un intero.
 1. Costruiamo un'istanza di Integer usando il costruttore che prende come parametro una stringa e poi leggiamo il valore intero:

```
Integer in;  
int n;  
in = new Integer("23");  
n = in.intValue();
```
 2. Oppure invochiamo semplicemente il metodo parseInt() sulla classe Integer senza creare istanze:

```
int n1;  
n1 = Integer.parseInt("23");
```
 - E' evidente che il secondo metodo è più semplice e non comporta la creazione di un'istanza, e quindi è quello più usato
-  **Attenzione:** notate la differenza di utilizzo dei due metodi. **intValue()** viene invocato su un'istanza mentre **parseInt()** viene invocato sulla classe

Tipi primitivi e classi wrapper

Esempio: `toString()` nella classe `Integer`

- **versione statica:**

 - `public static String toString(int x) ;`

 - prende un valore `int` e ne produce la rappresentazione sotto forma di stringa

- **versione metodo:**

 - `public String toString() ;`

 - è implicitamente invocato su un oggetto `Integer`
 - ne recupera il valore e ne produce la rappresentazione sotto forma di stringa.

Esempio

```
public class EsempioWrapper {  
    public static void main(String args[]) {  
        int x = 35;  
        Integer ix = new Integer(x);  
        x = 2 * ix.intValue();  
  
        System.out.println("ix =" + ix.toString() );  
        System.out.println("x =" + Integer.toString(x) );  
    }  
}
```

Conversione esplicita da **Integer** a **String** (usando il metodo **toString()** di **Integer**)

Conversione esplicita da **int** a **String** (usando la funzione statica **toString()** di **Integer**)

```
ix = 35  
x = 70
```

Esempio

```
public class EsempioWrapper {  
    public static void main(String args[]) {  
        int x = 35;  
        Integer ix = new Integer(x)  
        x = 2 * ix.intValue();  
  
        System.out.println("ix =" + ix);  
        System.out.println("x =" + x);  
    }  
}
```

Conversione implicita da **Integer** a **String** (usando il metodo **toString()** di **Integer**)

Conversione implicita da **int** a **String** (usando la funzione statica **toString()** di **Integer**)

```
ix = 35  
x = 70
```

Classi di servizio

- Normalmente il ruolo principale di una classe è quello di creare istanze: tutte le classi che abbiamo creato nei nostri esempi (Counter, Orologio, ecc.) avevano questa impostazione
- Abbiamo però visto che nel caso dei wrapper ci troviamo di fronte a classi in cui il ruolo di creazione di istanze e quello di fornitura di servizi sono egualmente importanti
- Esistono addirittura classi che svolgono solo il ruolo di fornitori di servizi e non hanno la capacità di creare istanze
- Sono classi che:
 - **Non hanno costruttori**
 - **Hanno tutti i metodi dichiarati come static**

La classe Math (*libreria, più costanti*)

- L'esempio più evidente di classe di servizio in Java è la classe Math
- La classe Math risolve un problema di questo tipo
- Dal momento che in Java:
 - Non esistono funzioni ma solo metodi di una qualche classe
 - I numeri reali - float o double - non sono oggetti ma tipi primitivi e quindi non hanno metodi
- **Come si fa a calcolare la radice quadrata, il logaritmo, il seno o il coseno di un numero?**
- Una possibile soluzione sarebbe quella di inserire nella classe wrapper Double (e in Float) un metodo per ogni funzione matematica
- Abbiamo però visto che questo approccio è complicato e poco efficiente: ogni volta che devo calcolare una funzione matematica dovrei creare un'istanza di Double
- Si è quindi scelta una strada più semplice: **esiste una classe, denominata Math, che definisce solo metodi statici e ogni metodo corrisponde ad una funzione matematica.**

I metodi di Math

- Math contiene un gran numero di metodi
- Tutti i metodi sono dichiarati come **public static**
- Vediamoli per categorie:
 - Potenze radici: `pow()` e `sqrt()`
 - Funzioni trigonometriche: `sin()`, `cos()`, `tan()` ...
 - Logaritmo naturale ed esponenziale: `log()`, `exp()`
 - Funzioni di conversione da reali a interi: `round()`
 - Varie ed eventuali: `abs()`, `max()` ...
- Di molti metodi esistono versioni **overloaded** per gestire i vari tipi. Per esempio `max()` ha 4 definizioni:

```
public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)
```


Esempio di uso di Math

- Vediamo un esempio di utilizzo di Math
- Math è contenuta nel package **java.lang** e quindi **non è necessario usare import**

```
public class EsempioMath
{
    public static void main(String args[])
    {
        double x,y;
        x = 5;
        y = Math.log(x) ;
        System.out.println("Il logaritmo di "+x+" è "+y);
        x = Math.PI / 2;    // Pi greco/2
        y = Math.sin(x) ;
        System.out.println("Il seno di "+x+" è "+y);
    }
}
```

Math: che cos'è PI

- Nell'esempio appena visto c'è un'istruzione strana:

```
x = Math.PI / 2;
```

- Math.PI evidentemente non è un metodo (non ha le parentesi), è un attributo che ha come valore π (3.14...)
- Però c'è qualcosa che non torna:
 - Gli attributi costituiscono lo stato di un'istanza
 - Però abbiamo detto che la classe Math non genera istanze
- Che cos'è PI?

Attributi di classe - 1

- E' un **attributo statico** della classe **Math**
- La sua definizione è
public static final double PI;
- E' marcato come **static** e come tale ha un significato simile ai metodi static:
 - Non appartiene ad una istanza, ma alla classe
 - È un attributo che esiste per tutto il tempo di vita dell'applicazione
 - Per usarlo facciamo riferimento al nome della classe e non ad una variabile di tipo Math
x = Math.PI
- PI è definito anche come **final** perché è una costante: non deve essere possibile cambiare il suo valore

Attributi di classe - 2

- 💣 **Attenzione:** si tratta di un caso speciale: normalmente gli attributi si trovano nell'istanza: costituiscono lo stato dell'istanza.
- Se però marchiamo un attributo con il modificatore **static** questo attributo viene messo nella classe e non entra a far parte dello stato dell'istanza
- Gli attributi **static** vengono chiamati **attributi di classe**, in quanto non appartengono ad un'istanza ma alla classe nel suo insieme
- Si usa anche in questo caso il marcatore **static** perché questi attributi esistono per tutto il tempo di vita dell'applicazione
- Non vengono creati dinamicamente come avviene per gli attributi che costituiscono lo stato dell'istanza